

САМОЕ ПОЛНОЕ РУКОВОДСТВО ПО РАЗРАБОТКЕ

ПРОГРАММИРОВАНИЕ
ОТ ЭКСПЕРТОВ

PYTHON

в примерах от сообщества
Stack Overflow

RESCUER

ПРОГРАММИРОВАНИЕ
ОТ ЭКСПЕРТОВ

САМОЕ ПОЛНОЕ
РУКОВОДСТВО
ПО РАЗРАБОТКЕ

PYTHON

в примерах от сообщества
Stack Overflow



рекомендовано
Библиотекой программиста

rescuer

МОСКВА
ИЗДАТЕЛЬСТВО АСТ

УДК 004.42
ББК 32.973.26-018.2
С17

Последнюю версию этой книги на английском языке можно скачать с сайта: <https://GoalKicker.com/PythonBook>. Пожалуйста, не стесняйтесь поделиться этим PDF-файлом с кем угодно бесплатно.

*Книга Python® Notes for Professionals составлена на основе документации Stack Overflow (<https://archive.org/details/documentation-dump.7z>), содержание написано прекрасными людьми из Stack Overflow. **Текстовое содержимое представлено на условиях Creative Commons BY-SA.** В конце книги указаны авторы, внесшие вклад в создание различных глав. Авторские права на изображения принадлежат их соответствующим владельцам, если не указано иное.*

Самое полное руководство по разработке на Python в примерах
C17 от сообщества Stack Overflow. — Москва : Издательство АСТ, 2024. —
672 с. : ил. — (Программирование от экспертов).
ISBN 978-5-17-160252-9.

Данное руководство по программированию на одном из широко распространенных языков — Python — основано на практических примерах кодов, написанных специалистами и экспертами сообщества Stack Overflow, в котором лучшие разработчики программного обеспечения со всего мира делятся своими знаниями и опытом, отвечая на многие технические вопросы. Опытные Python-программисты найдут в книге множество примеров кода с подробными комментариями, что поможет им усовершенствовать свои навыки и достичь новых высот в отрасли. Однако данное издание будет полезно и начинающим специалистам с минимальным опытом и уровнем знаний, так как содержит исчерпывающее объяснение важнейших концепций Python с примерами, которые позволят избежать погружения в сухую теорию и помогут быстро повысить уровень своих компетенций. Читатели найдут здесь мощный и универсальный инструмент для профессиональной работы в самых разных областях применения: с базами данных, веб-фреймворком Flask, XML и JSON, звуковыми данными, синтаксическим анализатором Lex-Yacc, а также при сетевом программировании, визуализации данных, многопоточности и многопроцессорности, программировании «интернета вещей». Кроме того, в книге представлена информация о применении Python в сфере науки, например, в математике, химии и криптографии. Отдельные главы посвящены секретам повышения скорости работы Python-кода и оптимизирования его производительности.

УДК 004.42
ББК 32.973.26-018.2

ISBN 978-5-17-160252-9

Перевод на русский язык: ООО «Интеджер».
Издание на русском языке: ООО «Издательство АСТ».

Содержание

- Глава 1. Начало работы с языком Python27
 - 1.1. Начало работы27
 - 1.2. Создание переменных и присвоение им значений.....31
 - 1.3. Отступы блоков.....34
 - 1.4. Типы данных.....36
 - 1.5. Типы коллекций.....39
 - 1.6. IDLE – графический интерфейс Python43
 - 1.7. Ввод данных пользователем44
 - 1.8. Встроенные модули и функции.....45
 - 1.9. Создание модуля.....47
 - 1.10. Установка Python 2.7.x и 3.x.....48
 - 1.11. Строковые функции – str() и repr()50
 - 1.12. Установка внешних модулей с помощью pip51
 - 1.13. Справочная утилита.....52
- Глава 2. Типы данных в Python.....53
 - 2.1. Строковый тип данных.....53
 - 2.2. Множества (set и frozenset)53
 - 2.3. Числовые типы данных54
 - 2.4. Тип данных “список” (list).....54
 - 2.5. Тип данных “словарь” (dic).....54
 - 2.6. Тип данных “кортеж” (tuple)54
- Глава 3. Отступы55
 - 3.1. Простой пример.....55
 - 3.2. Как происходит разбор отступов.....55
 - 3.3. Ошибки отступа.....56
- Глава 4. Комментарии и документация.....56
 - 4.1. Однострочные, строчные и многострочные комментарии.....56
 - 4.2. Программный доступ к строкам документации (docstrings)57
 - 4.3. Написание документации с использованием docstrings58
- Глава 5. Дата и время60
 - 5.1. Разбор строки в объект даты и времени (datetime) с учетом часового пояса60
 - 5.2. Построение временных интервалов с учетом временных зон60
 - 5.3. Вычисление разницы во времени.....62
 - 5.4. Использование базовых объектов datetime62
 - 5.5. Переключение между часовыми поясами.....63
 - 5.6. Простые арифметические действия с датами.....63
 - 5.7. Преобразование временной метки (timestamp) в объект datetime64
 - 5.8. Точное вычитание месяцев из даты64
 - 5.9. Разбор (парсинг) произвольной временной метки ISO 8601 с минимальным использованием библиотек64
 - 5.10. Получение временной метки ISO 860165
 - 5.11. Разбор строки с коротким именем часового пояса в объект datetime с учетом часового пояса.....65
 - 5.12. Нечеткий синтаксический анализ времени (извлечение даты и времени из текста)66
 - 5.13. Итерация по датам.....66
- Глава 6. Форматирование даты67
 - 6.1. Время между двумя датами.....67
 - 6.2. Вывод объекта datetime в строку.....67
 - 6.3. Парсинг строки в объект datetime.....67

Глава 7. Модуль Enum	68
7.1. Создание перечисления (Python 2.4–3.3)	68
7.2. Итерация	68
Глава 8. Множества	68
8.1. Операции над множествами	68
8.2. Получение уникальных элементов списка	69
8.3. Множество множеств	70
8.4. Операции над множествами с использованием методов и встроенных модулей	70
8.5. Множества и мультимножества	72
Глава 9. Простые математические операторы	72
9.1. Деление	73
9.2. Сложение	74
9.3. Возведение в степень	74
9.4. Тригонометрические функции	75
9.5. Операторы присваивания (“операции на месте”)	76
9.6. Вычитание	76
9.7. Умножение	77
9.8. Логарифмы	77
9.9. Остаток от деления	77
Глава 10. Побитовые операторы	78
10.1. Побитовое НЕ	78
10.2. Побитовое XOR (исключающее ИЛИ)	80
10.3. Побитовое И	80
10.4. Побитовое ИЛИ	80
10.5. Побитовый сдвиг влево	81
10.6. Побитовый сдвиг вправо	81
10.7. Операторы присваивания (“операции на месте”)	81
Глава 11. Логические операции	82
11.1. “and” и “or” не гарантируют возврата булевого значения	82
11.2. Простой пример	82
11.3. Вычисления по короткой схеме (“короткое замыкание”)	82
11.4. Оператор “and” (“и”)	83
11.5. Оператор “or” (“или”)	83
11.6. Оператор “not” (“не”)	84
Глава 12. Приоритет операторов	84
12.1. Примеры приоритета операторов в Python	84
Глава 13. Область видимости и привязка переменных	85
13.1. Нелокальные переменные	85
13.2. Глобальные переменные	86
13.3. Локальные переменные	87
13.4. Команда “del”	87
13.5. Функции пропускают область видимости класса при поиске имен	88
13.6. Локальные и глобальные области	89
13.7. Возникновение привязки	92
Глава 14. Условные выражения	92
14.1. Условное выражение (“тернарный оператор”)	92
14.2. if, elif и else	92
14.3. Значения истинности	93
14.4. Выражения булевой логики	93
14.5. Использование функции “cmp” для получения результата сравнения двух объектов	95

14.6. Оператор "else"	95
14.7. Проверка принадлежности объекта к None и его присвоение	95
14.8. Оператор "if"	96
Глава 15. Сравнение.....	96
15.1. Цепное сравнение	96
15.2. Сравнение по принципу "is" и "=="	97
15.3. Больше или меньше.....	98
15.4. Не равно.....	98
15.5. Равно	99
15.6. Сравнение объектов	99
Глава 16. Циклы	100
16.1. Перерыв и продолжение в циклах	100
16.2. Циклы For	102
16.3. Итерирование по спискам	102
16.4. Циклы с "else"	103
16.5. Заявление о прохождении	105
16.6. Итерация в словарях	105
16.7. "Полуцикл" do-while	106
16.8. Циклирование и распаковка	106
16.9. Итерирование части списка с разным размером шага	107
16.10. Цикл While	108
Глава 17. Массивы.....	108
17.1. Доступ к отдельным элементам через индексы.....	109
17.2. Введение в массивы.....	109
17.3. Добавление произвольного значения в массив с помощью метода append().....	110
17.4. Вставка значения в массив с помощью метода insert()	110
17.5. Расширение массива с помощью метода extend()	110
17.6. Добавление элементов из списка в массив с помощью метода fromlist()	110
17.7. Удаление любого элемента массива с помощью метода remove()	110
17.8. Удаление последнего элемента массива с помощью метода pop()	110
17.9. Получение любого элемента по его индексу с помощью метода index()	111
17.10. Обратное преобразование массива с помощью метода reverse()	111
17.11. Получение информации о буфере массива с помощью метода buffer_info ().....	111
17.12. Проверка количества вхождений элемента с помощью метода count ().....	111
17.13. Преобразование массива в строку с помощью метода tostring ()	111
17.14. Преобразование массива в список с одинаковыми элементами с использованием метода tolist ()	111
17.15. Добавление строки в массив char, используя метод fromstring ()	112
Глава 18. Многомерные массивы.....	112
18.1. Списки в списках.....	112
18.2. Списки в списках в списках в	113
Глава 19. Словарь	113
19.1. Введение в словарь	113
19.2. Избегание исключений KeyError	114
19.3. Итерация по словарю	115
19.4. Словарь со значениями по умолчанию	115
19.5. Слияние словарей	116
19.6. Доступ к ключам и значениям.....	116
19.7. Доступ к значениям словаря	117
19.8. Создание словаря	117
19.9. Создание упорядоченного словаря	118
19.10. Распаковка словарей с помощью оператора **.....	118

19.11. Запятая в конце строки.....	119
19.12. Конструктор dict()	119
19.13. Пример словарей	119
19.14. Все комбинации значений словаря	119
Глава 20. Список	120
20.1. Методы перечисления и поддерживаемые операторы	120
20.2. Доступ к значениям списка	125
20.3. Проверка списка на пустоту.....	126
20.4. Итерирование по списку	126
20.5. Проверка наличия элемента в списке	126
20.6. Функции “any” и “all”	127
20.7. Реверсирование элементов списка	127
20.8. Конкатенация и слияние списков	128
20.9. Длина списка	129
20.10. Удаление дублирующихся значений в списке	129
20.11. Сравнение списков.....	129
20.12. Доступ к значениям во вложенном списке	129
20.13. Инициализация списка с фиксированным числом элементов	130
Глава 21. Генератор списков	131
21.1. Списковые вычисления	131
21.2. Условные генераторы списков.....	133
21.3. Избегание повторных и ресурсоемких операций с помощью условного предложения	134
21.4. Генератор словаря (словарь включений).....	136
21.5. Генераторы списков с вложенными циклами	137
21.6. Генераторные выражения	138
21.7. Генераторы наборов	140
21.8. Рефакторинг функций filter и map в генераторы списков	140
21.9. Генераторы с использованием кортежей	141
21.10. Подсчет вхождений при использовании генераторов.....	142
21.11. Изменение типов в списке	142
21.12. Вложенные генераторы списков.....	142
21.13. Итерация двух и более списков одновременно внутри генератора списка.....	143
Глава 22. Срезы списков (выделение частей списков).....	143
22.1. Использование третьего аргумента “шаг”	143
22.2. Выбор подсписка из списка	144
22.3. Реверсирование списка с помощью среза	144
22.4. Смещение списка с помощью среза	144
Глава 23. Метод groupby().....	145
23.1. Пример	145
23.2. Пример 2	146
23.3. Пример 3	146
Глава 24. Связные списки	147
24.1. Пример односвязного списка	147
Глава 25. Узел связного списка	150
25.1. Написание простого узла связного списка на языке Python	150
Глава 26. Фильтр	151
26.1. Основное использование фильтра	151
26.2. Фильтр без функции.....	151
26.3. Фильтр для проверки на “короткое замыкание” (вычисления по короткой схеме)	152

26.4. Дополняющая функция: <code>filterfalse</code> , <code>ifilterfalse</code>	152
Глава 27. Модуль "Heapq"	153
27.1. Самые большие и самые маленькие предметы в коллекции	153
27.2. Наименьший элемент в коллекции	154
Глава 28. Кортеж (tuple)	154
28.1. Кортеж	154
28.2. Кортежи являются неизменяемыми	155
28.3. Упаковка и распаковка кортежей	155
28.4. Встроенные функции кортежей	156
28.5. Кортежи являются поэлементно хешируемыми и сравнимыми	157
28.6. Индексирование кортежей	157
28.7. Реверсирование элементов	158
Глава 29. Основы ввода и вывода данных	158
29.1. Использование функции вывода "print"	158
29.2. Ввод из файла	158
29.3. Чтение из <code>stdin</code>	160
29.4. Использование функций <code>input()</code> и <code>raw_input()</code>	160
29.5. Функция запроса числа у пользователя	161
29.6. Печать строки без новой строки в конце	161
Глава 30. Ввод/вывод файлов и папок	162
30.1. Режимы работы с файлами	162
30.2. Построчное чтение <code>file</code>	164
30.3. Итерация файлов (рекурсивно)	164
30.4. Получение полного содержимого файла	165
30.5. Запись в файл	165
30.6. Проверка наличия файла или пути	166
30.7. Произвольный доступ к файлам с помощью <code>mmap</code>	167
30.8. Замена текста в файле	167
30.9. Проверка того, что файл пуст	167
30.10. Чтение файла в диапазоне строк	168
30.11. Копирование дерева каталогов	168
30.12. Копирование содержимого файла в другой файл	168
Глава 31. Модуль <code>os.path</code>	168
31.1. Объединение путей	168
31.2. Управление компонентами пути	169
31.3. Получение родительского каталога	169
31.4. Проверка существования пути	169
31.5. Проверка того, является ли данный путь каталогом, файлом, символической ссылкой, точкой монтирования	169
31.6. Абсолютный путь из относительного пути	169
Глава 32. Итерируемые типы данных и итераторы	170
32.1. Итератор (iterator), итерируемый объект (iterable) и генератор (generator)	170
32.2. Извлечение значений по одному	171
32.3. Итерирование по всему итерируемому	171
32.4. Проверка только одного элемента в итерируемом	171
32.5. Что может быть итерируемым	171
32.6. В итератор нельзя входить повторно!	172
Глава 33. Функции	172
33.1. Создание и вызов простых функций	172
33.2. Определение функции с произвольным числом аргументов	174

33.3. Лямбда-функции (встроенные/анонимные)	176
33.4. Создание функции с необязательными аргументами	178
33.5. Определение функции с необязательными изменяемыми аргументами	178
33.6. Передача аргументов и изменяемость	179
33.7. Возврат значений из функций	180
33.8. Замыкание (closure).....	181
33.9. Принудительное использование именованных параметров.....	182
33.10. Вложенные функции.....	182
33.11. Предел рекурсии	183
33.12. Рекурсивная лямбда-функция с использованием присваиваемой переменной.....	183
33.13. Рекурсивные функции.....	184
33.14. Определение функции с аргументами.....	185
33.15. Распаковка итерируемых объектов и словарей.....	185
33.16. Определение функции с несколькими аргументами.....	187
Глава 34. Создание функций со списочными аргументами	187
34.1. Функция и вызов	187
Глава 35. Функциональное программирование в Python	188
35.1. Лямбда-функция.....	188
35.2. Функция map	188
35.3. Функция reduce	188
35.4. Функция filter	188
Глава 36. Частичные функции	189
36.1. Возведение в степень.....	189
Глава 37. Декораторы	189
37.1. Функция-декоратор	190
37.2. Класс декоратора.....	191
37.3. Декоратор с аргументами (фабрика декораторов).....	192
37.4. Приведение декоратора к виду декорируемой функции.....	193
37.5. Использование декоратора для определения времени выполнения функции	194
37.6. Создание класса-синглтона (одиночки) с помощью декоратора	194
Глава 38. Классы.....	195
38.1. Введение в классы.....	195
38.2. Связанные, несвязанные и статические методы.....	197
38.3. Базовое наследование.....	199
38.4. Манкипатчинг (Monkey Patching).....	200
38.5. Классы нового и старого стиля.....	201
38.6. Методы класса: альтернативные инициализаторы.....	202
38.7. Множественное наследование.....	203
38.8. Свойства	204
38.9. Значения по умолчанию для переменных экземпляра	206
38.10. Переменные класса и экземпляра	207
38.11. Композиция классов	208
38.12. Перечисление всех членов класса	209
38.13. Класс-синглтон	209
38.14. Дескрипторы и точечный поиск	211
Глава 39. Метаклассы.....	211
39.1. Базовые метаклассы	211
39.2. Синглтоны, использующие метаклассы	212
39.3. Использование метакласса	212
39.4. Введение в метаклассы	213

39.5. Пользовательская функциональность в метаклассах.....	214
39.6. Метакласс по умолчанию.....	214
Глава 40. Форматирование строк.....	215
40.1. Основы форматирования строк.....	215
40.2. Выравнивание и заполнение.....	216
40.3. Форматные литералы (f-строка).....	217
40.4. Форматирование чисел с плавающей точкой.....	217
40.5. Именованные заполнители.....	218
40.6. Форматирование строк с использованием типа datetime.....	219
40.7. Форматирование числовых значений.....	219
40.8. Вложенное форматирование.....	219
40.9. Форматирование с помощью функций Getitem и Getattr.....	220
40.10. Комбинированное заполнение и усечение строк.....	220
40.11. Пользовательское форматирование класса.....	221
Глава 41. Строковые методы.....	222
41.1. Изменение регистра букв в строке.....	222
41.2. str.translate: замена символов в строке.....	223
41.3. str.format и f-strings: форматирование значений в строку.....	223
41.4. Полезные константы модуля string.....	224
41.5. Удаление ненужных ведущих и завершающих символов из строки.....	225
41.6. Реверсирование строки.....	226
41.7. Разбиение строки по разделителю на список строк.....	226
41.8. Замена всех вхождений одной подстроки на другую подстроку.....	227
41.9. Проверка содержимого строк.....	228
41.10. Проверка содержания подстроки в строке.....	230
41.11. Объединение списка строк в одну строку.....	230
41.12. Подсчет количества вхождений подстроки в строке.....	230
41.13. Сравнение строк без учета регистра.....	230
41.14. Выравнивание строк.....	232
41.15. Проверка начальных и конечных символов строки.....	232
41.16. Преобразование между строковыми или байтовыми данными и символами Unicode.....	233
Глава 42. Использование циклов внутри функций.....	234
42.1. Оператор возврата внутри цикла в функции.....	234
Глава 43. Импорт модулей.....	235
43.1. Импорт модуля.....	235
43.2. Специальная переменная __all__.....	236
43.3. Импорт модулей из произвольного места файловой системы.....	237
43.4. Импорт всех имен из модуля.....	237
43.5. Программный импорт.....	237
43.6. Правила PEP8 для импорта.....	238
43.7. Импорт конкретных имен из модуля.....	238
43.8. Импорт submodule.....	239
43.9. Повторный импорт модуля.....	239
43.10. Функция __import__.....	240
Глава 44. Разница между модулем и пакетом.....	240
44.1. Модули.....	240
44.2. Пакеты.....	240
Глава 45. Математический модуль math.....	241
45.1. Округление: round, floor, ceil, trunc.....	241
45.2. Тригонометрия.....	242

45.3. Функция pow для быстрого возведения в степень	243
45.4. Бесконечность и NaN ("не число")	244
45.5. Логарифмы	246
45.6. Константы	246
45.7. Мнимые числа	247
45.8. Копирование знаков	247
45.9. Комплексные числа и модуль cmath	247
Глава 46. Комплексная математика	250
46.1. Расширенная комплексная арифметика	250
46.2. Основы комплексной арифметики	250
Глава 47. Модуль collections	251
47.1. collections.Counter	251
47.2. collections.OrderedDict	252
47.3. collections.defaultdict	253
47.4. collections.namedtuple	254
47.5. collections.deque	254
47.6. collections.ChainMap	256
Глава 48. Модуль operator	257
48.1. Функция itemgetter	257
48.2. operator как альтернатива инфиксному оператору	257
48.3. Methodcaller	257
Глава 49. Модуль JSON	258
49.1. Хранение данных в файле	258
49.2. Получение данных из файла	258
49.3. Форматирование вывода JSON	258
49.4. "load" и "loads", "dump" и "dumps"	259
49.5. Вызов "json.tool" из командной строки для эстетичного вывода JSON	260
49.6. JSON-кодирование пользовательских объектов	260
49.7. Создание JSON из словаря Python	261
49.8. Создание словаря Python из JSON	261
Глава 50. Модуль Sqlite3	261
50.1. Sqlite3 не требует отдельного серверного процесса	261
50.2. Получение значений из базы данных и обработка ошибок	261
Глава 51. Модуль os	262
51.1. mkdir - recursive creation of directories	262
51.2. Создание каталога	263
51.3. Получение текущего каталога	263
51.4. Определение операционной системы	263
51.5. Удаление каталога	263
51.6. Следование по симлинку (POSIX)	263
51.7. Изменение разрешений файла	264
Глава 52. Модуль locale	264
52.1. Форматирование валюты в долларах США с использованием модуля locale	264
Глава 53. Модуль itertools	264
53.1. Метод комбинаций в модуле itertools	264
53.2. itertools.dropwhile	265
53.3. Использование zip_longest для двух итераторов до тех пор, пока они оба не будут исчерпаны	265
53.4. Получение среза генератора	265

53.5. Группировка элементов из итерируемого объекта с помощью функции	266
53.6. <code>itertools.takewhile</code>	267
53.7. <code>itertools.permutations</code>	267
53.8. <code>itertools.repeat</code>	268
53.9. Получение накопленной суммы чисел в итерируемом объекте	268
53.10. Циклический переход по элементам итератора	268
53.11. <code>itertools.product</code>	268
53.12. <code>itertools.count</code>	269
53.13. Объединение нескольких итераторов в цепочку	270
Глава 54. Модуль <code>Asyncio</code>	270
54.1. Синтаксис корутин (сопрограмм) и делегирования	270
54.2. Асинхронные исполнители (<code>Executors</code>)	271
54.3. Использование <code>UVLoop</code>	272
54.4. Примитив синхронизации: <code>Event</code>	272
54.5. Простой <code>Websocket</code>	273
54.6. Распространенные заблуждения, связанные с модулем <code>asyncio</code>	274
Глава 55. Модуль <code>Random</code>	274
55.1. Создание случайного пароля пользователя	274
55.2. Создание криптографически защищенных случайных чисел	275
55.3. Случайности и последовательности: перемешивание, выбор и выборка	275
55.4. Создание чисел типов <code>int</code> и <code>float</code> : <code>randint</code> , <code>randrange</code> , <code>random</code> и <code>uniform</code>	276
55.5. Воспроизводимые случайные числа: методы <code>Seed</code> и <code>State</code>	277
55.6. Случайное двоичное решение	277
Глава 56. Модуль <code>Functools</code>	278
56.1. Функция <code>partial</code>	278
56.2. <code>cmp_to_key</code>	278
56.3. <code>@lru_cache</code>	278
56.4. <code>@total_ordering</code>	279
56.5. <code>reduce</code>	279
Глава 57. Модуль <code>dis</code>	280
57.1. Что такое байт-код Python?	280
57.2. Константы в модуле <code>dis</code>	280
57.3. Демонтаж модулей	280
Глава 58. Модуль <code>base64</code>	281
58.1. Кодирование и декодирование <code>Base64</code>	282
58.2. Кодирование и декодирование <code>Base32</code>	283
58.3. Кодирование и декодирование <code>Base16</code>	284
58.4. Кодирование и декодирование <code>ASCII85</code>	284
58.5. Кодирование и декодирование <code>Base85</code>	285
Глава 59. Модуль <code>Queue</code>	285
59.1. Простой пример	285
Глава 60. Модуль <code>Deque</code>	286
60.1. Базовое использование <code>deque</code>	286
60.2. Доступные методы в <code>deque</code>	286
60.3. Ограничение размера <code>deque</code>	287
60.4. Поиск по ширине (<code>Breadth First Search</code>)	287
Глава 61. Модуль <code>Webbrowser</code>	288
61.1. Открытие URL-адреса браузером по умолчанию	288
61.2. Открытие URL-адреса с помощью различных браузеров	289

Глава 62. tkinter.....	289
62.1. Менеджеры геометрии	289
62.2. Минимальное приложение tkinter	291
Глава 63. Модуль ruautogui	291
63.1. Функции мыши.....	291
63.2. Функции клавиатуры	292
63.3. Распознавание скриншотов и изображений.....	292
Глава 64. Индексация и нарезка.....	292
64.1. Базовая нарезка	292
64.2. Реверсирование объекта	293
64.3. Назначение среза.....	293
64.4. Создание неглубокой копии массива.....	294
64.5. Индексация пользовательских классов: <code>__getitem__</code> , <code>__setitem__</code> и <code>__delitem__</code>	294
64.6. Базовая индексация	295
Глава 65. Построение графиков с помощью Matplotlib	296
65.1. Графики с общей осью x, но разными осями y: использование <code>twinx()</code>	296
65.2. Графики с общей осью y, но разными осями x: использование функции <code>twiny()</code> ...	297
65.3. Простой график в Matplotlib	299
65.4. Добавление дополнительных функций к простому графику: названия осей, заголовок, метки осей, сетка и легенда	300
65.5. Создание нескольких кривых на одном графике путем наложения, аналогично MATLAB	301
65.6. Создание нескольких кривых на одном графике с помощью наложения с использованием отдельных команд <code>plot</code>	302
Глава 66. graph-tool	303
66.1. PyDotPlus.....	303
66.2. PyGraphviz	303
Глава 67. Генераторы	304
67.1. Введение.....	304
67.2. Бесконечные последовательности.....	306
67.3. Отправка объектов генератору.....	307
67.4. Предоставление всех значений из другого итерируемого объекта.....	308
67.5. Итерация	308
67.6. Функция <code>next()</code>	308
67.7. Корутины (сопрограммы)	309
67.8. Рефакторинг кода построения списков	309
67.9. Использование оператора <code>yield</code> с рекурсией: рекурсивное перечисление всех файлов в каталоге	310
67.10. Генераторные выражения	310
67.11. Использование генератора для получения чисел Фибоначчи	311
67.12. Поиск	311
67.13. Параллельный итерационный перебор генераторов	311
Глава 68. Функция <code>reduce</code>	312
68.1. Обзор	312
68.2. Использование <code>reduce</code>	312
68.3. Накопительный продукт	313
68.4. Вариант без “короткого замыкания” функций <code>any/all</code>	313
Глава 69. Функция <code>map</code>	313
69.1. Базовое использование <code>map</code> , <code>itertools.imap</code> и <code>future_builtins.map</code>	313
69.2. Сопоставление каждого значения в итерируемом объекте	314

69.3. Отображение (mapping) значений различных итерируемых объектов	315
69.4. Транспонирование с помощью map: использование "None" в качестве аргумента функции (только для Python 2.x)	316
69.5. Последовательное и параллельное отображение	317
Глава 70. Возведение в степень	318
70.1. Возведение в степень с использованием встроенных модулей: ** и pow()	318
70.2. Квадратный корень: math.sqrt() и cmath.sqrt	319
70.3. Возведение в степень по модулю: функция pow() с тремя аргументами	319
70.4. Вычисление больших целых корней	320
70.5. Возведение в степень с использованием модуля math: math.pow()	321
70.6. Экспоненциальная функция: math.exp() и cmath.exp()	321
70.7. Экспоненциальная функция -1: math.expm1()	321
70.8. Магические методы и возведение в степень: builtin, math и cmath	322
70.9. Корень n-й степени с дробным показателем степени	323
Глава 71. Поиск	324
71.1. Поиск элемента	324
71.2. Поиск в пользовательских классах: __contains__ и __iter__	324
71.3. Получение индекса для строк: str.index(), str.rindex() и str.find(), str.rfind()	325
71.4. Получение индексов в списках и кортежах: list.index(), tuple.index()	325
71.5. Поиск ключа или ключей для значения в словаре	326
71.6. Получение индекса в отсортированных последовательностях: bisect.bisect_left()	326
71.7. Поиск во вложенных последовательностях	327
Глава 72. Сортировка, минимум и максимум	328
72.1. Упорядоченность в пользовательских классах	328
72.2. Особый случай: словари	330
72.3. Использование ключевого аргумента	331
72.4. Аргумент по умолчанию для max, min	331
72.5. Получение отсортированной последовательности	331
72.6. Извлечение N наибольших или N наименьших элементов из итерируемого объекта	332
72.7. Получение минимального или максимального из нескольких значений	332
72.8. Минимум и максимум последовательности	333
Глава 73. Подсчет	333
73.1. Подсчет всех вхождений всех элементов в итерируемом объекте: collections.Counter	333
73.2. Получение наиболее часто встречающегося значения: collections.Counter.most_common()	334
73.3. Подсчет количества вхождений элемента в последовательность: list.count() и tuple.count()	334
73.4. Подсчет вхождений подстроки в строку: str.count()	334
73.5. Подсчет вхождений в numpy-массиве	335
Глава 74. Функция print	335
74.1. Основы вывода на экран	335
74.2. Параметры функции print	336
Глава 75. Регулярные выражения (Regex)	337
75.1. Сопоставление начала строки	337
75.2. Поиск	338
75.3. Предварительно скомпилированные шаблоны (precompiled patterns)	339
75.4. Флаги	339
75.5. Замена	340

75.6. Поиск всех непересекающихся совпадений.....	340
75.7. Проверка на разрешенные символы.....	341
75.8. Разбиение строки с помощью регулярных выражений.....	341
75.9. Группировка	341
75.10. Исключение специальных символов.....	342
75.11. Сопоставление выражения только в определенных местах.....	343
75.12. Итерация по совпадениям с помощью re.finditer.....	343
Глава 76. Копирование данных.....	344
76.1. Копирование словаря.....	344
76.2. Неглубокое копирование.....	344
76.3. Глубокое копирование.....	344
76.4. Неглубокое копирование списка.....	345
76.5. Копирование набора.....	345
Глава 77. Менеджеры контекста (инструкция with)	345
77.1. Введение в менеджеры контекста и инструкцию with.....	345
77.2. Написание собственного менеджера контекста	346
77.3. Написание собственного менеджера контекста с использованием синтаксиса генератора	347
77.4. Использование нескольких менеджеров контекста	348
77.5. Назначение цели	348
77.6. Управление ресурсами.....	348
Глава 78. Специальная переменная __name__	349
78.1. Проверка __name__ == '__main__'.....	349
78.2. Использование в протоколировании	349
78.3. function_class_or_module.__name__.....	349
Глава 79. Проверка существования пути и прав доступа	351
79.1. Выполнение проверки с помощью os.access	351
Глава 80. Создание пакетов Python.....	351
80.1. Введение	351
80.2. Загрузка в каталог пакетов PyPI	352
80.3. Создание исполняемого пакета	354
Глава 81. Использование модуля "pip": Менеджер пакетов PyPI.....	354
81.1. Пример использования команд	354
81.2. Обработка исключения ImportError.....	355
81.3. Принудительная установка	355
Глава 82. pip: менеджер пакетов PyPI.....	356
82.1. Установка пакетов	356
82.2. Получение списка пакетов, установленных с помощью `pip`	356
82.3. Обновление пакетов	356
82.4. Удаление пакетов.....	357
82.5. Обновление всех устаревших пакетов в Linux	357
82.6. Обновление всех устаревших пакетов под Windows	357
82.7. Создание файла requirements.txt file всех пакетов в системе	357
82.8. Использование определенной версии Python с помощью pip	357
82.9. Создание файла requirements.txt file пакетов только текущей virtualenv	358
82.10. Установка пакетов, еще не включенных в pip в качестве wheels.....	358
Глава 83. Разбор аргументов командной строки	361
83.1. Hello world в argparse	361
83.2. Использование аргументов командной строки с помощью argv	362

83.3. Задание взаимоисключающих аргументов с помощью argparse.....	363
83.4. Базовый пример с использованием docopt	363
83.5. Пользовательское сообщение об ошибке синтаксического анализатора с помощью argparse	364
83.6. Концептуальная группировка аргументов с помощью argparse.add_argument_group()	364
83.7. Расширенный пример с docopt и docopt_dispatch	365
Глава 84. Библиотека подпроцессов.....	366
84.1. Больше возможностей с помощью Popen	366
84.2. Вызов внешних команд	367
84.3. Как создать аргумент списка команд	368
Глава 85. Файл setup.py.....	368
85.1. Назначение файла setup.py	368
85.2. Использование метаданных системы управления версиями в файле setup.py	369
85.3. Добавление скриптов командной строки в пакет Python	369
85.4. Добавление параметров установки	370
Глава 86. Рекурсия.....	370
86.1. Что, как и когда делать с рекурсией	370
86.2. Задача “исследования дерева” с помощью рекурсии	373
86.3. Сумма чисел от 1 до n	374
86.4. Увеличение максимальной глубины рекурсии	375
86.5. Хвостовая рекурсия – плохая практика	375
86.6. Оптимизация хвостовой рекурсии с помощью интроспекции стека	376
Глава 87. Подсказки типов	377
87.1. Добавление типов в функцию	377
87.2. Функция NamedTuple	378
87.3. Общие типы	378
87.4. Переменные и атрибуты	378
87.5. Члены и методы классов.....	379
87.6. Подсказки типов для именованных аргументов	379
Глава 88. Исключения.....	379
88.1. Перехват исключений	379
88.2. Не перехватывайте все подряд!	380
88.3. Повторный вызов исключений	380
88.4. Перехват множественных исключений	381
88.5. Иерархия исключений	381
88.6. Else	383
88.7. Вызов исключений.....	384
88.8. Создание пользовательских типов исключений	384
88.9. Практические примеры обработки исключений.....	384
88.10. Исключения – это тоже объекты	385
88.11. Выполнение кода очистки с помощью finally	386
88.12. Создание цепочек исключений с использованием raise from.....	386
Глава 89. Вызов пользовательских ошибок и исключений.....	387
89.1. Пользовательское исключение.....	387
89.2. Перехват пользовательского исключения.....	387
Глава 90. Исключения сообщества	387
90.1. Прочие ошибки	387
90.2. Ошибка “NameError. name ‘?’ is not defined”	388
90.3. Ошибки типов данных	389

90.4. Синтаксическая ошибка в хорошем коде.....	391
90.5. Ошибки отступа (или ошибки синтаксиса отступов)	391
Глава 91. urllib	392
91.1. HTTP GET.....	392
91.2. HTTP POST	393
91.3. Декодирование полученных байтов в соответствии с кодировкой типа содержимого	394
Глава 92. Веб-скрапинг с помощью Python.....	394
92.1. Скрапинг с использованием фреймворка Scrapy	394
92.2. Скрапинг с использованием Selenium WebDriver	395
92.3. Базовый пример использования запросов и lxml для скрапинга некоторых данных.....	395
92.4. Поддержание сессии веб-скрапинга с помощью запросов	396
92.5. Скрапинг с использованием BeautifulSoup4	396
92.6. Простое скачивание веб-контента с помощью urllib.request	396
92.7. Модификация пользовательского агента Scrapy	397
92.8. Скрапинг с использованием инструмента командной строки cURL	397
Глава 93. Парсинг HTML	397
93.1. Использование CSS-селекторов в библиотеке BeautifulSoup	397
93.2. Библиотека PyQuery	398
93.3. Нахождение текста после элемента в библиотеке BeautifulSoup.....	399
Глава 94. Работа с XML	399
94.1. Открытие и чтение с помощью библиотеки ElementTree.....	399
94.2. Создание и построение XML-документов.....	399
94.3. Изменение XML-файла	400
94.4. Поиск в XML с помощью XPath.....	400
94.5. Открытие и чтение больших XML-файлов с помощью инкрементного парсинга	401
Глава 95. Python Requests Post	402
95.1. Простая операция Post.....	402
95.2. Данные, закодированные в форме	403
95.3. Загрузка файлов	403
95.4. Ответы.....	403
95.5. Аутентификация.....	404
95.6. Прокси-серверы.....	405
Глава 96. Распространение кода	405
96.1. py2app	405
96.2. cx_Freeze	406
Глава 97. Объекты свойств.....	407
97.1. Использование декоратора @property для свойств чтения и записи	407
97.2. Еще об использовании декоратора @property	407
97.3. Переопределение только функций getter, setter или deleter объекта свойства	408
97.4. Использование свойств без декораторов.....	408
Глава 98. Перегрузка.....	410
98.1. Перегрузка операторов.....	410
98.2. "Магические" методы, или Dunder-методы	412
98.3. Типы контейнеров и последовательностей.....	413
98.4. Вызываемые типы	413
98.5. Обработка нереализованного поведения.....	414

Глава 99. Полиморфизм	415
99.1. "Утиная типизация"	415
99.2. Базовый полиморфизм	415
Глава 100. Переопределение методов	418
100.1. Переопределение базовых методов	418
Глава 101. Пользовательские методы	418
101.1. Создание пользовательских объектов метода	418
101.2. Пример с использованием Turtle Graphics	420
Глава 102. Строковые представления экземпляров классов: <code>__str__</code> и <code>__repr__</code> методы	420
102.1. Мотивация	420
102.2. Реализация обоих методов, <code>__repr__()</code> в стиле eval-round-trip	424
Глава 103. Отладка	424
103.1. При помощи IPython и ipdb	424
103.2. Отладчик Python. Пошаговая отладка с помощью <code>_pdb_</code>	425
103.3. Удаленный отладчик	426
Глава 104. Чтение и запись в формате CSV	426
104.1. Использование <code>pandas</code>	426
104.2. Запись файла TSV	427
Глава 105. Запись в формат CSV из строки или списка	427
105.1. Пример базовой записи	427
105.2. Добавление строки в качестве новой строки в CSV-файл	428
Глава 106. Динамическое выполнение кода с помощью функций <code>exec</code> и <code>eval</code>	428
106.1. Выполнение кода, предоставленного недоверенным пользователем, с использованием <code>exec</code> , <code>eval</code> или <code>ast.literal_eval</code>	428
106.2. Оценка строки, содержащей литерал Python, с помощью <code>ast.literal_eval</code>	429
106.3. Оценка утверждений с помощью функции <code>exec</code>	429
106.4. Оценка выражения с помощью функции <code>eval</code>	429
106.5. Предварительная компиляция выражения для его многократной оценки	429
106.6. Оценка выражения с помощью <code>eval</code> с использованием пользовательских глобальных переменных	430
Глава 107. PyInstaller – распространение кода Python	430
107.1. Установка и настройка	430
107.2. Использование Pyinstaller	431
107.3. Объединение в одну папку	431
107.4. Объединение в один файл	431
Глава 108. Визуализация данных с помощью Python	432
108.1. Seaborn	432
108.2. Matplotlib	434
108.3. Plotly	435
108.4. MayaVI	436
Глава 109. Интерпретатор (консоль командной строки)	437
109.1. Получение общей помощи	437
109.2. Ссылка на последнее выражение	437
109.3. Открытие консоли Python	438
109.4. Переменная <code>PYTHONSTARTUP</code>	438
109.5. Аргументы командной строки	438
109.6. Получение справки об объекте	439

Глава 110. *args и **kwargs.....	440
110.1. Использование **kwargs при написании функций.....	440
110.2. Использование *args при написании функций.....	441
110.3. Заполнение значений kwarg с помощью словаря.....	441
110.4. Аргументы только для ключевых слов (keyword-only arguments) и аргументы, требующие ключевых слов (keyword-required arguments).....	441
110.5. Использование **kwargs при вызове функций.....	441
110.6. **kwargs и значения по умолчанию.....	442
110.7. Использование *args при вызове функций.....	442
Глава 111. Сборка мусора	442
111.1. Повторное использование примитивов	442
111.2. Последствия команды del	443
111.3. Подсчет ссылок	443
111.4. Сборщик мусора для ссылочных циклов	444
111.5. Принудительное удаление объектов	445
111.6. Просмотр счетчика ссылок (refcount) объекта.....	445
111.7. Не ждите, пока сборщик мусора наведет порядок	446
111.8. Управление сборкой мусора	446
Глава 112. Сериализация данных при помощи модуля Pickle.....	447
112.1. Использование модуля Pickle для сериализации и десериализации объекта.....	447
112.2. Данные для Pickle	448
Глава 113. Двоичные данные.....	449
113.1. Форматирование списка значений в байтовый объект.....	449
113.2. Распаковка байтового объекта в соответствии со строкой формата.....	449
113.3. Упаковка структуры.....	449
Глава 114. Идиомы.....	450
114.1. Инициализация ключей словаря	450
114.2. Переключение переменных.....	450
114.3. Использование проверки значения истинности.....	451
114.4. Тест на функцию __main__, чтобы избежать неожиданного выполнения кода	451
Глава 115. Сериализация данных.....	452
115.1. Сериализация с использованием формата JSON	452
115.2. Сериализация с использованием модуля Pickle.....	452
Глава 116. Многопроцессорная обработка.....	453
116.1. Запуск двух простых процессов.....	453
116.2. Использование класса Pool and функции pool.map	454
Глава 117. Многопоточность	454
117.1. Основы многопоточности	454
117.2. Общение между потоками.....	455
117.3. Создание пула процессов (workers).....	456
117.4. Продвинутое использование многопоточности.....	457
117.5. Останавливаемый поток с циклом while	458
Глава 118. Процессы и потоки	459
118.1. Глобальная блокировка интерпретатора.....	459
118.2. Работа с несколькими потоками.....	460
118.3. Работа с несколькими процессами.....	461
118.4. Совместное использование состояния потоками	461
118.5. Совместное использование состояния процессами.....	462
Глава 119. Параллелизм в Python.....	462
119.1. Многопроцессорный модуль	462

119.2. Модуль потокообразования (threading)	463
119.3. Передача данных между многопроцессорными процессами.....	464
Глава 120. Параллельные вычисления	465
120.1. Использование модуля многопроцессорной обработки для распараллеливания задач.....	465
120.2. Использование C-расширения для распараллеливания задач.....	465
120.3. Использование родительских и дочерних скриптов для параллельного выполнения кода	465
120.4. Использование модуля PyPar для распараллеливания	466
Глава 121. Сокеты.....	466
121.1. Сырые сокеты (Raw Sockets) в Linux	467
121.2. Передача данных по протоколу UDP	467
121.3. Получение данных по протоколу UDP	467
121.4. Отправка данных по протоколу TCP.....	468
121.5. Многопоточный сервер TCP Socket Server.....	468
Глава 122. Веб-сокеты.....	470
122.1. Простое эхо с помощью aiohttp.....	470
122.2. Класс-обертка с aiohttp	470
122.3. Использование пакета Autobahn в качестве фабрики вебсокетов.....	471
Глава 123. Сокеты и шифрование/дешифрование сообщений между клиентом и сервером.....	472
123.1. Реализация на стороне сервера	473
123.2. Реализация на стороне клиента.....	474
Глава 124. Сетевое взаимодействие на языке Python.....	476
124.1. Создание простого Http-сервера.....	476
124.2. Создание TCP-сервера	477
124.3. Создание UDP-сервера.....	477
124.4. Запуск простого HttpServer в потоке и открытие браузера.....	478
124.5. Простейший пример клиент-сервер сокетов на Python	478
Глава 125. Python HTTP Server	479
125.1. Запуск простого HTTP-сервера.....	479
125.2. Обслуживание файлов	479
125.3. Базовая обработка GET, POST, PUT с помощью BaseHTTPRequestHandler.....	480
125.4. Программное API SimpleHTTPServer.....	481
Глава 126. Микровебфреймворк Flask.....	482
126.1. Файлы и шаблоны	482
126.2. Основы	483
126.3. Маршрутизация URL-адресов	483
126.4. HTTP-методы.....	484
126.5. Jinja Templating	485
126.6. Объект запроса.....	485
Глава 127. Использование библиотеки AMQPStorm в брокере сообщений RabbitMQ	486
127.1. Как получать сообщения из RabbitMQ.....	486
127.2. Как публиковать сообщения в RabbitMQ	487
127.3. Как создать очередь с задержкой в RabbitMQ	488
Глава 128. Дескриптор.....	489
128.1. Простой дескриптор	489
128.2. Двусторонние преобразования.....	490

Глава 129. Временный файл NamedTemporaryFile.....	491
129.1. Создание известного постоянного временного файла и запись в него	491
Глава 130. Ввод, подмножество и вывод внешних файлов данных с помощью библиотеки Pandas.....	492
130.1. Базовый код для импорта, подмножества и записи внешних файлов данных с помощью библиотеки Pandas.....	492
Глава 131. Распаковка файлов	493
131.1. Использование Python ZipFile.extractall() для распаковки ZIP-файла	494
131.2. Использование Python TarFile.extractall() для распаковки tarball-архива	494
Глава 132. Работа с ZIP-архивами	494
132.1. Изучение содержимого ZIP-файла	494
132.2. Открытие ZIP-файлов	494
132.3. Извлечение содержимого ZIP-файла в каталог	495
132.4. Создание новых архивов.....	495
Глава 133. Работа с GZip.....	495
133.1. Чтение и запись файлов GNU zip.....	496
Глава 134. Стек.....	496
134.1. Создание класса Stack с объектом List.....	496
134.2. Разбор (парсинг) круглых скобок.....	497
Глава 135. Работа в обход глобальной блокировки интерпретатора (GIL).....	498
135.1. Multiprocessing.Pool	498
135.2. Использование Cython nogil	499
Глава 136. Развертывание.....	499
136.1. Загрузка пакета Conda	499
Глава 137. Модуль logging	500
137.1. Введение в использование модуля logging.....	500
137.2. Протоколирование исключений.....	501
Глава 138. Стандарт Web Server Gateway Interface (WSGI).....	503
138.1. Объект (метод) сервера	503
Глава 139. События, посылаемые сервером (SSE).....	504
139.1. Flask SSE.....	504
139.2. Asyncio SSE	505
Глава 140. Альтернативы оператору switch из других языков.....	505
140.1. Используйте то, что предлагает Python: конструкция if/else	505
140.2. Использование словаря функций	505
140.3. Использование интроспекции классов	506
140.4. Использование менеджера контекста.....	507
Глава 141. Деструктуризация списка, упаковка и распаковка	508
141.1. Присваивание деструктуризации	508
141.2. Аргументы функции упаковки	509
141.3. Распаковка аргументов функции	511
Глава 142. Доступ к исходному коду Python и байт-коду.....	511
142.1. Отображение байт-кода функции	511
142.2. Отображение исходного кода объекта.....	512
142.3. Исследование кодового объекта функции.....	512

Глава 143. Миксины	513
143.1. Миксин	513
143.2. Переопределение методов в миксинах	514
Глава 144. Доступ к атрибутам.....	515
144.1. Базовый доступ к атрибутам с использованием точечной нотации.....	515
144.2. Методы setter, getter и свойства	515
Глава 145. Пакет ArcPy.....	516
145.1. Использование createDissolvedGDB для создания gdb-файла в рабочей области	516
145.2. Печать значения одного поля для всех строк класса признаков в файле geodatabase с помощью функции SearchCursor	516
Глава 146. Абстрактные базовые классы (abc)	517
146.1. Установка метакласса ABCMeta	517
146.2. Зачем и как использовать ABCMeta и @abstractmethod	517
Глава 147. Плагины и расширения	519
147.1. Миксины	519
147.2. Плагины с пользовательскими классами	520
Глава 148. Неизменяемые типы данных (int, float, str, tuple и frozenset).....	521
148.1. Отдельные символы строк не подлежат присвоению	521
148.2. Индивидуальные члены кортежа не могут быть присвоены.....	521
148.3. Frozenset является неизменяемым и не подлежит присваиванию.....	521
Глава 149. Несовместимости при переходе с Python 2 на Python 3	521
149.1. Целочисленное деление	521
149.2. Распаковка итерируемых объектов.....	523
149.3. Строки: байты и Unicode	524
149.4. Оператор Print и функция Print.....	526
149.5. Отличия между функциями range и xrange.....	526
149.6. Возникновение и обработка исключений	528
149.7. Утечка переменных в генераторе списка	529
149.8. True, False и None	530
149.9. Ввод данных пользователем	530
149.10. Сравнение разных типов	531
149.11. Переименование метода .next() для итераторов	531
149.12. Функции filter(), map() и zip() возвращают итераторы вместо последовательностей	532
149.13. Переименованные модули.....	532
149.14. Удалены операторы <> и `, синонимичные операторам != и repr().....	533
149.15. Типы данных long и int.....	533
149.16. Все классы в Python 3 являются “классами нового стиля”	534
149.17. Функция reduce больше не является встроенной	535
149.18. Абсолютный и относительный импорт.....	535
149.19. Функция map()	536
149.20. Функция round() – “тай-брейкинг” и возвращаемый тип	537
149.21. Файловый ввод/вывод	538
149.22. Функция str отсутствует в Python 3	539
149.23. Восьмеричные константы	539
149.24. Возвращаемое значение при записи в файловый объект	539
149.25. В Python 3 hex является функцией	539
149.26. Кодирование в шестнадцатеричный формат и декодирование из него больше не доступно	540
149.27. Изменения в методах словарей	541

149.28. Булево значение класса	541
149.29. Ошибка функции <code>hasattr</code> в Python 2.....	542
Глава 150. Инструмент 2to3	542
150.1. Базовое использование.....	543
Глава 151. Неофициальные реализации Python	544
151.1. IronPython.....	544
151.2. Jython.....	545
151.3. Transcrypt	545
Глава 152. Абстрактное синтаксическое дерево.....	548
152.1. Анализ функций в скрипте Python.....	548
Глава 153. Unicode и bytes	549
153.1. Обработка ошибок кодирования и декодирования	549
153.2. Файловый ввод/вывод	550
153.3. Основы	550
Глава 154. Последовательное соединение в Python (pyserial)	551
154.1. Инициализация последовательного устройства	551
154.2. Чтение из последовательного порта	551
154.3. Проверка доступности последовательных портов.....	552
Глава 155. Работа с Neo4j и Cypher с использованием библиотеки Py2Neo.....	552
155.1. Добавление узлов в граф в графовой СУБД Neo4j.....	552
155.2. Импорт и аутентификация	552
155.3. Добавление отношений в граф Neo4j.....	552
155.4. Запрос 1: автозаполнение заголовков новостей.....	553
155.5. Запрос 2: получение новостных статей по местоположению на определенную дату	553
155.6. Образцы запросов языка Cypher.....	553
Глава 156. Основы работы с библиотекой Curses в Python	554
156.1. Вспомогательная функция <code>wrapper()</code>	554
156.2. Пример базового обращения.....	554
Глава 157. Шаблоны в Python	555
157.1. Простая программа вывода данных с использованием шаблона	555
157.2. Изменение разделителя	555
Глава 158. Библиотека Pillow	555
158.1. Чтение файла изображения	555
158.2. Преобразование файла в формат JPEG.....	555
Глава 159. Оператор <code>pass</code>	556
159.1. Игнорирование исключения	556
159.2. Создание нового исключения, которое может быть перехвачено.....	556
Глава 160. Подкоманды CLI (Command Line Interface) для аккуратного вывода справочной информации	556
160.1. Способ без использования библиотек.....	556
160.2. Использование <code>argparse</code> (средство форматирования справки по умолчанию)...	557
160.3. Использование <code>argparse</code> (создание пользовательского средства форматирования справки).....	557
Глава 161. Доступ к базам данных.....	559
161.1. SQLite	559

161.2. Доступ к базе данных MySQL с помощью MySQLdb	563
161.3. Соединение	564
161.4. Доступ к базе данных PostgreSQL с помощью psycopg2	564
161.5. Работа с базами данных Oracle	565
161.6. Использование библиотеки sqlalchemy	567
Глава 162. Подключение Python к SQL Server	567
162.1. Подключение к серверу, создание таблицы, запрос данных	567
Глава 163. Реляционная база данных PostgreSQL	568
163.1. Начало работы	568
Глава 164. Python и Excel	569
164.1. Чтение данных Excel с помощью модуля xlrd	569
164.2. Форматирование файлов Excel с помощью модуля xlswriter	570
164.3. Помещение списковых данных в файл Excel	570
164.4. Модуль OpenPyXL	571
164.5. Создание диаграмм Excel с помощью xlswriter	571
Глава 165. Turtle Graphics ("Черепашня графика")	573
165.1. Создание Ninja Twist при помощи Turtle Graphics	573
Глава 166. Продолжающиеся действия (Persistence) в Python	574
166.1. Продолжающиеся действия в Python	574
166.2. Функциональная утилита для сохранения и загрузки	575
Глава 167. Шаблоны проектирования	575
167.1. Введение в паттерны проектирования и паттерн синглтон	576
167.2. Стратегический шаблон	577
167.3. Прокси-объект	578
Глава 168. Модуль hashlib	580
168.1. MD5-хеш строки	580
168.2. Алгоритм, предоставляемый OpenSSL	581
Глава 169. Создание службы Windows с помощью Python	581
169.1. Сценарий на языке Python, который может быть запущен как служба	582
169.2. Запуск веб-приложения Flask в качестве сервиса	582
Глава 170. Изменяемые, неизменяемые и хешируемые в Python	583
170.1. Изменяемые и неизменяемые типы	583
170.2. Изменяемые и неизменяемые типы как аргументы	585
Глава 171. Модуль configparser	586
171.1. Создание конфигурационного файла программным способом	586
171.2. Базовое использование	587
Глава 172. Оптическое распознавание символов (OCR)	587
172.1. PyTesseract	587
172.2. PyOCR	588
Глава 173. Виртуальные среды	589
173.1. Создание и использование виртуальной среды	589
173.2. Указание версии Python для использования в скрипте на Unix/Linux	590
173.3. Создание виртуальной среды для различных версий Python	591
173.4. Создание виртуальных сред с помощью Anaconda	591
173.5. Управление несколькими виртуальными средами с помощью утилиты virtualenvwrapper	592

173.6. Установка пакетов в виртуальной среде	593
173.7. Определение используемой виртуальной среды.....	594
173.8. Проверка на работу в виртуальной среде	594
173.9. Использование виртуальной среды с оболочкой Fish shell	594
Глава 174. Виртуальная среда Python – virtualenv	595
174.1. Установка	595
174.2. Использование	595
174.3. Установка пакета в виртуальной среде	595
174.4. Другие полезные команды virtualenv	595
Глава 175. Создание виртуальной среды с помощью надстройки virtualenvwrapper	596
Глава 176. Создание виртуальной среды с помощью virtualenvwrapper в Windows.....	597
Глава 177. Модуль sys.....	597
177.1. Аргументы командной строки.....	598
177.2. Имя скрипта	598
177.3. Ошибки и стандартный вывод.....	598
177.4. Преждевременное завершение процесса и возврат кода выхода.....	598
Глава 178. Пакет ChemPy	598
178.1. Разбор формул.....	599
178.2. Стехиометрия химических реакций.....	599
178.3. Уравнение химической реакции.....	599
178.4. Химическое равновесие	599
178.5. Ионная сила	600
178.6. Химическая кинетика (система обыкновенных дифференциальных уравнений).....	600
Глава 179. Библиотека Pygame	601
179.1. Модуль mixer в Pygame	601
179.2. Установка Pygame.....	602
Глава 180. Модуль Pyglet.....	602
180.1. Установка Pyglet	602
180.2. “Hello World” в Pyglet	603
180.3. Воспроизведение звука в Pyglet.....	603
180.4. Использование Pyglet для OpenGL.....	603
180.5. Рисование точек с помощью Pyglet и OpenGL	603
Глава 181. Работа со звуком.....	603
181.1. Работа с файлами в формате WAV.....	603
181.2. Преобразование любого звукового файла с помощью Python и ffmpeg	604
181.3. Воспроизведение звуковых сигналов Windows	604
181.4. Воспроизведение звука с помощью Pyglet	604
Глава 182. Pyaudio.....	605
182.1. Режим обратного вызова звукового ввода/вывода	605
182.2. Режим блокировки звукового ввода/вывода	606
Глава 183. Модуль Shelve	607
183.1. Использование Shelve	607
183.2. Пример кода для Shelve	608
183.3. Обобщение информации о интерфейсе	608
183.4. Обратная запись (writeback).....	608

Глава 184. Программирование “интернета вещей” (IoT) с помощью Python и Raspberry Pi	610
184.1. Пример – датчик температуры.....	610
Глава 185. Kivy – кроссплатформенный Python-фреймворк для разработки естественных пользовательских интерфейсов (NUI).....	611
185.1. Первое приложение.....	612
Глава 186. Использование Pandas transform: предварительное выполнение операций с группами и конкатенация результатов.....	613
186.1. Простое преобразование.....	613
186.2. Несколько результатов для одной группы.....	614
Глава 187. Сходство в синтаксисе, различия в значении: Python и JavaScript.....	615
187.1. `in` со списками	615
Глава 188. Вызов Python из C#.....	615
188.1. Python-скрипт, который вызывается C#-приложением	615
188.2. Код на C#, вызывающий Python-скрипт.....	615
Глава 189. Библиотека ctypes.....	617
189.1. Массивы ctypes	617
189.2. Обертывающие функции для ctypes	617
189.3. Базовое использование.....	618
189.4. Основные “подводные камни”	618
189.5. Базовый объект ctypes.....	619
189.6. Комплексное использование	619
Глава 190. Написание расширений	620
190.1. “Hello World” расширение на C	620
190.2. Расширение на языке C с помощью C++ и библиотеки Boost	621
190.3. Передача открытого файла в C-расширения	622
Глава 191. Python Lex-Yacc	622
191.1. Начало работы с Python Lex-Yacc	622
191.2. “Hello, World!” от PLY – простой калькулятор.....	623
191.3. Часть 1. Токенизация входных данных с помощью Lex	624
191.4. Часть 2. Парсинг токенизированного ввода с помощью Yacc.....	627
Глава 192. Модульное тестирование.....	630
192.1. Использование методов setUp и tearDown при работе с unittest.TestCase.....	630
192.2. Утверждения при исключениях.....	631
192.3. Тестирование исключений	632
192.4. Выбор утверждений в рамках модуля unittests	632
192.5. Модульные тесты с помощью pytest	633
192.6. Подражание функциям при помощи unittest.mock.create_autospec	635
Глава 193. Библиотека py.test.....	636
193.1. Настройка py.test.....	636
193.2. Введение в тестовые фикстуры (Fixtures)	637
193.3. Неудачные испытания.....	640
Глава 194. Профилирование.....	640
194.1. %timeit и %timeit в IPython	640
194.2. Использование cProfile (предпочтительный инструмент)	641
194.3. Функция timeit()	641
194.4. Командная строка timeit.....	641
194.5. Использование модуля line_profiler и скрипта kernprof в командной строке.....	642

Глава 195. Скорость работы программы на языке Python	642
195.1. Deque-операции	642
195.2. Алгоритмические нотации	643
195.3. Нотация Big-O	644
195.4. Операции со списком	644
195.5. Операции с множеством	644
Глава 196. Оптимизация производительности	645
196.1. Профилирование кода	645
Глава 197. Безопасность и криптография	646
197.1. Безопасное хеширование паролей	646
197.2. Вычисление дайджеста сообщения	647
197.3. Доступные алгоритмы хеширования	647
197.4. Хеширование файлов	647
197.5. Генерация RSA-подписей с помощью библиотеки Pycrypto	648
197.6. Асимметричное RSA-шифрование с помощью Pycrypto	649
197.7. Симметричное шифрование с использованием Pycrypto	649
Глава 198. SSH-протокол в Python	650
198.1. SSH-соединение	650
Глава 199. Антипаттерны программирования в Python	651
199.1. Чрезмерное использование except	651
199.2. Процессороемкие функции	651
Глава 200. Общие ошибки	652
200.1. Умножение списков и общие ссылки	652
200.2. Изменяемый аргумент по умолчанию	656
200.3. Изменение последовательности, над которой выполняется итерация	657
200.4. Целочисленные и строковые тождества	659
200.5. Словари неупорядочены	660
200.6. Утечка переменных в генераторах списков и циклах for	661
200.7. Цепочка операторов "or"	661
200.8. sys.argv[0] – имя исполняемого файла	662
200.9. Доступ к атрибутам целочисленных литералов	662
200.10. Глобальная блокировка интерпретатора (GIL) и блокировка потоков	663
200.11. Многократное использование оператора return	663
200.12. JSON-ключи в Python	664
Глава 201. Скрытые возможности	664
201.1. Перегрузка операторов	664
Благодарности	666

Глава 1. Начало работы с языком Python

Python 3.x

Дата выпуска версии

3.11	2022-10-24
3.10	2021-10-04
3.9	2020-10-05
3.8	2020-04-29
3.7	2018-06-27
3.6	2016-12-23
3.5	2015-09-13
3.4	2014-03-17
3.3	2012-09-29
3.2	2011-02-20
3.1	2009-06-26
3.0	2008-12-03

Python 2.x

Дата выпуска версии

2.7	2010-07-03
2.6	2008-10-02
2.5	2006-09-19
2.4	2004-11-30
2.3	2003-07-29
2.2	2001-12-21
2.1	2001-04-15
2.0	2000-10-16

1.1. Начало работы

Python – широко распространенный язык программирования высокого уровня для программирования общего назначения, созданный Гвидо ван Россумом и впервые выпущенный в 1991 году. Python имеет динамическую систему типов и автоматическое управление памятью, поддерживает множество парадигм программирования, включая объектно-ориентированное, императивное, функциональное программирование и процедурные стили. Он имеет большую и обширную стандартную библиотеку.

В настоящее время активно используются две основные версии Python:

- Python 3.x является текущей версией и находится в стадии активной разработки.
- Python 2.x является устаревшей версией и до 2020 года будет получать только обновления безопасности. Никаких новых возможностей реализовано не будет.

Отметим, что многие проекты по-прежнему используют Python 2, хотя переход на Python 3 становится все проще.

Подробнее об отличиях версий можно узнать в главе 149 “Несовместимости при переходе с Python 2 на Python 3”. Кроме того, некоторые сторонние разработчики выпускают перепакованные версии Python, в которые добавлены часто используемые библиотеки и другие возможности, облегчающие настройку для обычных областей использования, таких как математика, анализ данных или научные исследования.

Проверьте, установлен ли Python

Чтобы убедиться в том, что Python был установлен правильно, можно выполнить следующую команду в терминале (если вы используете ОС Windows, то перед использованием в командной строке необходимо добавить путь к python в переменную окружения):

```
$ python --version
```

Версия Python 3.x ≥ 3.0:

Если у вас установлен Python 3 и это версия по умолчанию (подробнее см. главу 1.6. “IDLE – графический интерфейс Python”), то вы должны увидеть нечто подобное:

```
$ python --version Python 3.6.0
```

Версия Python 2.x ≤ 2.7:

Если у вас установлен Python 2 и это версия по умолчанию (подробнее см. главу 1.6. “IDLE – графический интерфейс Python”), то вы должны увидеть нечто подобное:

```
$ python --version Python 2.7.13
```

Если у вас установлен Python 3, но команда `$ python --version` выводит версию Python 2, значит, у вас также установлен Python 2. Это часто встречается в MacOS и многих дистрибутивах Linux. Для явного использования интерпретатора Python 3 используйте команду `$ python3`.

Hello, World на языке Python с использованием IDLE

IDLE – это простой редактор для Python, поставляемый в комплекте с Python.

- Откройте IDLE на выбранной вами системе.
 - В старых версиях Windows его можно найти в разделе “Все программы” меню Windows.
 - В Windows 8+ выполните поиск IDLE или найдите его в приложениях, присутствующих в системе.
 - В системах на базе Unix (включая Mac) его можно открыть из оболочки, набрав `$ idle python_file.py`.
- Откроется оболочка с опциями в верхней части.

В оболочке имеется подсказка из трех прямых угловых скобок:

```
>>>
```

Теперь напишите после подсказки следующий код:

```
>>> print("Hello, World")
```

Нажмите Enter.

```
>>> print("Hello, World")
```

```
Hello, World
```

Файл “Hello, World” на Python

Создайте новый файл `hello.py`, содержащий следующую строку:

Версия Python 3.x ≥ 3.0:

```
print('Hello, World')
```

Версия Python 2.x ≥ 2.6:

Вы можете использовать функцию `print` Python 3 в Python 2 с помощью следующего оператора `import`:

```
from __future__ import print_function
```

Python 2 имеет ряд функциональных возможностей, которые могут быть опционально импортированы из Python 3 с помощью модуля `__future__`.

Версия Python 2.x ≤ 2.7:

Если используется Python 2, то можно также набрать строку, представленную ниже. Обратите внимание, что в Python 3 это не работает и, следовательно, не рекомендуется, так как снижает кросс-версионную совместимость кода.

```
print 'Hello, World'
```

В терминале перейдите в каталог, содержащий файл `hello.py`. Введите `python hello.py`, а затем нажмите клавишу “Enter”.

```
$ python hello.py
Hello, World
```

В консоли должно появиться сообщение `Hello, World`.

Вы также можете заменить `hello.py` на путь к вашему файлу. Например, если файл находится в домашнем каталоге, а ваш пользователь в Linux это “user”, то вы можете набрать `python/home/user/hello.py`.

Запуск интерактивной оболочки Python

При запуске команды `python` в терминале вы получаете интерактивную оболочку Python. Она также известна как интерпретатор Python или REPL (от английского Read Evaluate Print Loop).

```
$ python
Python 2.7.12 (по умолчанию, Jun 28 2016, 08:46:01)
[GCC 6.1.1 20160602] на linux
Для получения дополнительной информации введите “help”, “copyright”, “credits” или “license”.
>>> print 'Hello, World'
Hello, World
>>>
```

Если вы хотите запустить Python 3 из терминала, выполните команду `python3`.

```
$ python3
Python 3.6.0 (по умолчанию, 13 января 2017 г., 00:00:00)
[GCC 6.1.1 20160602] на linux
Для получения дополнительной информации введите “help”, “copyright”, “credits” или “license”.
>>> print('Hello, World') Hello, World
>>>
```

В качестве альтернативы запустите интерактивное приглашение и загрузите файл с помощью команды `python -i <file.py>`. В командной строке выполните команду:

```
$ python -i hello.py
“Hello World”
>>>
```

Существует несколько способов закрыть оболочку Python:

```
>>> exit()
или
>>> quit()
```

В качестве альтернативы комбинация клавиш “CTRL + D” закроет оболочку и вернет вас в командную строку вашего терминала.

Если вы хотите отменить команду, которую вы вводите, и вернуться в чистую командную строку, оставаясь внутри оболочки интерпретатора, нажмите “CTRL + C”.

Другие онлайн-оболочки

Различные сайты предоставляют онлайн-доступ к оболочкам Python. Онлайн-оболочки могут быть полезны для следующих целей.

- Запуск небольшого фрагмента кода с устройств, на которых не установлен Python (смартфоны, планшеты и т. д.).
- Изучение или преподавание основ языка Python.
- Для работы с онлайн-судьями.

Примеры:

Отказ от ответственности: автор(ы) документации не связан(ы) с какими-либо ресурсами, перечисленными ниже.

- <https://www.python.org/shell/> – Онлайн-оболочка Python, размещенная на официальном сайте Python.
- <https://ideone.com/> – Ресурс, широко используемый в сети для иллюстрации поведения фрагментов кода.
- <https://repl.it/languages/python3> – Мощный и простой онлайн-компилятор, IDE и интерпретатор. Кодировать, компилировать и выполнять код на языке Python.
- https://www.tutorialspoint.com/execute_python_online.php – Полнофункциональная UNIX-оболочка и удобный проводник проектов.
- http://rextester.com/l/python3_online_compiler – Простая и удобная в использовании IDE, показывающая время выполнения.

Выполнение команд в виде строки

В оболочке Python можно передавать произвольный код в виде строки:

```
$ python -c 'print("Hello, World")'
Hello, World
```

Это может быть полезно при конкатенации результатов работы скриптов в оболочке.

Оболочки и не только

Управление пакетами. Рекомендуются инструментом PyPA для установки пакетов Python является PIP. Для установки выполните в командной строке команду `pip install <имя пакета>`. Например, `pip install numpy`. (**Примечание:** в Windows необходимо добавить `pip` в переменные окружения PATH. Чтобы избежать этого, используйте команду `python -m pip install <имя пакета>`).

Оболочки. До сих пор мы обсуждали различные способы выполнения кода с помощью встроенной интерактивной оболочки Python. Оболочки используют интерпретирующие возможности Python для экспериментов с кодом в реальном времени. Альтернативными оболочками являются IDLE – готовый графический интерфейс, IPython, известный своей возможностью расширения интерактивных возможностей, и др.

Программы. Для долговременного хранения можно сохранять содержимое в файлы с расширением `.py` и редактировать или исполнять их как скрипты или программы с помощью внешних инструментов, например, оболочек, IDE (например, PyCharm), Jupyter notebooks и т. д. Продвинутые пользователи могут использовать эти инструменты, однако рассмотренные здесь методы являются достаточными для начала работы.

Python tutor позволяет пошагово просмотреть код Python, чтобы наглядно представить, как будет работать программа, и помогает понять, где в программе допущены ошибки.

PEP8 содержит рекомендации по форматированию кода Python. Правильное форматирование кода важно для того, чтобы можно было быстро прочитать, что он делает.

1.2. Создание переменных и присвоение им значений

Чтобы создать переменную в Python, достаточно указать имя переменной, а затем присвоить ей значение.

```
<имя переменной> = <значение>
```

В Python для присвоения значений переменным используется символ `=`. Нет необходимости объявлять переменную заранее (или присваивать ей тип данных), присвоение значения переменной само по себе объявляет и инициализирует переменную этим значением. Невозможно объявить переменную, не присвоив ей начального значения.

```
# Целое число
a = 2
print(a)
# Выходное значение: 2

# Целое число
b = 9223372036854775807
print(b)
# Выходное значение: 9223372036854775807

# Числа с плавающей точкой
pi = 3.14
print(pi)
# Выходное значение: 3.14

# Строка
c = 'A'
print(c)
# Выходное значение: A

# Строка
name = 'John Doe'
print(name)
# Выходное значение: John Doe

# Логическое значение
q = True
print(q)
# Выходное значение: True

# Пустое значение или тип данных null
x = None
print(x)
# Выходное значение: None
```

Присвоение переменных работает слева направо. Поэтому в следующем примере будет допущена синтаксическая ошибка:

```
0 = x
=> Output: SyntaxError: can't assign to literal
```

Вы не можете использовать ключевые слова Python в качестве допустимого имени переменной. Список ключевых слов можно посмотреть при помощи:

```
import keyword
print(keyword.kwlist)
```

Правила именования переменных:

1. Имена переменных должны начинаться с буквы или символа подчеркивания.

```
x = True          # допустимо
_y = True         # допустимо
```

```
9x = False      # начинается с цифры
=> SyntaxError: invalid syntax
```

```
$y = False      # начинается с символа
=> SyntaxError: invalid syntax
```

2. Остальная часть имени переменной может состоять из букв, цифр и знаков подчеркивания.

```
has_0_in_it = "Still Valid"
```

3. Имена чувствительны к регистру.

```
x = 9
y = X*5
=> NameError: name 'X' is not defined.
```

Несмотря на то, что при объявлении переменной в Python нет необходимости указывать тип данных, при выделении необходимой области в памяти под переменную интерпретатор Python автоматически выбирает для нее наиболее подходящий встроенный тип:

```
a = 2
print(type(a))
# Выходное значение: <type 'int'>
```

```
b = 9223372036854775807
print(type(b))
# Выходное значение: <type 'int'>
```

```
pi = 3,14
print(type(pi))
# Выходное значение: <type 'float'>
```

```
c = 'A'
print(type(c))
# Выходное значение: <type 'str'>
```

```
name = 'John Doe'
print(type(name))
# Выходное значение: <type 'str'>
```

```
q = True
print(type(q))
# Выходное значение: <type 'bool'>
```

```
x = None
print(type(x))
# Выходное значение: <type 'NoneType'>
```

Теперь, когда вы знаете основы присваивания, давайте разберемся с тонкостями присваивания в Python.

Когда вы используете символ = для выполнения операции присваивания, то, что находится слева от =, является **именем** расположенного справа **объекта**. В конечном итоге символ = присваивает имени, расположенному слева, ссылку на объект, находящийся справа.

То есть:

```
a_name = an_object      # "a_name" теперь является именем для ссылки на объект "an_object"
```

Так, если из множества приведенных выше примеров мы выберем `pi = 3,14`, то `pi` – это **имя** (**не название**, поскольку у объекта может быть несколько имен) для объекта 3,14. Если вы что-то не поняли, вернитесь к этому пункту и прочитайте еще раз!

В одной строке можно присвоить несколько значений нескольким переменным. Обратите внимание, что справа и слева от оператора = должно быть одинаковое количество аргументов:

```
a, b, c = 1, 2, 3
print(a, b, c)
# Выходное значение: 1 2 3

a, b, c = 1, 2
=> Traceback (most recent call last):
=>   Файл "name.py", строка N, in <module>
=>       a, b, c = 1, 2
=> ValueError: need more than 2 values to unpack

a, b = 1, 2, 3
=> Traceback (most recent call last):
=>   File "name.py", line N, in <module>
=>       a, b = 1, 2, 3
=> ValueError: too many values to unpack
```

Ошибку в последнем примере можно устранить, присвоив оставшиеся значения равному количеству произвольных переменных. Эта фиктивная переменная может иметь любое имя, но для присвоения ненужных значений принято использовать знак подчеркивания (_):

```
a, b, _ = 1, 2, 3
print(a, b)
# Выходное значение: 1, 2
```

Обратите внимание, что количество знаков подчеркивания () и количество оставшихся значений должны быть равны. В противном случае будет выдана ошибка 'too many values to unpack':

```
a, b, _ = 1, 2, 3, 4
=> Traceback (most recent call last):
=> File "name.py", line N, in <module>
=> a, b, _ = 1, 2, 3, 4
=> ValueError: too many values to unpack (expected 3)
```

Также можно присвоить одно значение нескольким переменным одновременно.

```
a = b = c = 1
print(a, b, c)
# Выходное значение: 1 1 1
```

При использовании такого каскадного присваивания важно отметить, что все три переменные a, b и c ссылаются на **один и тот же объект** в памяти – объект int со значением 1. Другими словами, a, b и c – это три разных имени, присвоенных одному и тому же объекту типа int. Присвоение впоследствии одному из них другого объекта не меняет остальных, как и ожидалось:

```
a = b = c = 1      # все три имени – a, b и c – ссылаются на один и тот же объект int
                   # со значением 1
print(a, b, c)
# Выходное значение: 1 1 1
b = 2              # b теперь ссылается на другой объект int, имеющий значение 2
print(a, b, c)
# Выходное значение: 1 2 1      # теперь выходное значение соответствует ожиданиям.
```

Сказанное выше справедливо для изменяемых типов (таких как list, dict и т. д.) так же, как и для неизменяемых (таких как int, string, tuple и т. д.):

```
x = y = [7, 8, 9]  # x и y ссылаются на один и тот же только что созданный объект списка [7, 8, 9]
x = [13, 8, 9]     # x теперь ссылается на другой только что созданный объект списка [13, 8, 9].
```

```
print(y)      # y по-прежнему ссылается на список, которому он был присвоен в первый раз
# Выходное значение: [7, 8, 9]
```

Пока все хорошо. Несколько иначе обстоит дело с **изменением** объекта (в отличие от **присвоения** имени другому объекту, что мы делали выше), когда каскадное присваивание используется для изменяемых типов. Посмотрите пример ниже:

```
x = y = [7, 8, 9]  # x и y - это два разных имени для одного и того же только что созданного
# объекта списка [7, 8, 9]
x[0] = 13          # мы обновляем значение списка [7, 8, 9] через одно из его имен, в данном
# случае x
print(y)          # печать значения списка с использованием его другого имени
# Выходные данные: [13, 8, 9]      # следовательно, изменение отражается
```

Вложенные списки также допустимы в Python. Это означает, что список может содержать в качестве элемента другой список.

```
x = [1, 2, [3, 4, 5], 6, 7]  # это вложенный список
print x[2]
# Выходное значение: [3, 4, 5]
print x[2][1]
# Выходное значение: 4
```

Напоследок отметим, что переменные в Python не обязательно должны оставаться того же типа, в котором они были впервые определены – можно просто использовать `=`, чтобы присвоить переменной новое значение, даже если это значение имеет другой тип.

```
a = 2
print(a)
# Выходное значение: 2

a = "New Value"
print(a)
# Выходное значение: New Value
```

Если вас это смущает, подумайте о том, что то, что находится слева от `=`, это просто имя объекта. Сначала вы называете объект `int` со значением 2 именем `a`, затем передумываете и решаете присвоить имя `a` объекту `string`, имеющему значение "New Value". Просто, не так ли?

1.3. Отступы блоков

Python использует отступы для выделения управляющих и циклических конструкций. Это способствует повышению удобочитаемости Python, однако требует от программиста пристального внимания к использованию пробельных символов. Таким образом, неправильная настройка редактора может привести к тому, что код будет вести себя неожиданным образом.

В Python для указания места начала и конца блоков кода используется символ двоеточия (`:`) и отступ (если вы пришли из другого языка, не путайте это с тем, что это как-то связано с тернарным оператором). То есть блоки в Python, такие как функции, циклы, предложения `if` и другие конструкции не имеют концевых идентификаторов. Все блоки начинаются с двоеточия, а затем содержат расположенные под ним строки с отступами.

Например:

```
def my_function():      # Это определение функции. Обратите внимание на двоеточие (:)
    a = 2               # Эта строка принадлежит функции, так как имеет отступ
    return a            # Эта строка также принадлежит той же функции
print(my_function())    # Эта строка находится ВНЕ блока функций

или

if a > b:               # блок if начинается здесь
    print(a)            # Это часть блока if
```



```
else:                # else должен находиться на том же уровне, что и if
    print(b)         # Эта строка является частью блока else
```

Блоки, содержащие ровно одно однострочное утверждение, могут быть помещены на одну строку, хотя такая форма обычно не считается хорошим стилем в программировании:

```
if a > b: print(a)
else: print(b)
```

Попытка сделать это с помощью более чем одного оператора **не даст результата**:

```
if x > y: y = x
    print(y) # IndentationError: неожиданный отступ
```

```
if x > y: while y != z: y -= 1 # SyntaxError: неверный синтаксис
```

Пустой блок вызывает ошибку `IndentationError` (ошибку отступа). Используйте команду `pass` (команда, которая ничего не делает), если у вас есть блок без содержимого:

```
def will_be_implemented_later():
    pass
```

Пробелы и табуляции

Вкратце: **всегда** используйте **четыре** пробела для отступа.

Использование только табуляции возможно, но в PEP 8, руководстве по стилю кода Python, говорится, что пробелы предпочтительнее.

Версия Python 3.x ≥ 3.0:

В Python 3 запрещено смешивать использование табуляции и пробелов для отступов. В этом случае возникает ошибка компиляции, связанная с непоследовательным использованием табуляций и пробелов в отступах, в результате чего программа не запускается.

Версия Python 2.x ≤ 2.7:

В Python 2 допускается смешивание символов табуляции и пробела в отступах, но делать это категорически не рекомендуется. Символ табуляции дополняет предыдущий отступ до значения, **кратного 8 пробелам**. Поскольку обычно редакторы настроены на отображение символов табуляции как кратных 4 пробелам, это может привести к возникновению трудноуловимых ошибок.

Процитируем руководство PEP 8:

При вызове интерпретатора командной строки Python 2 с опцией `-t` он выдает предупреждения о коде, в котором неправомерно смешиваются табуляции и пробелы. При использовании опции `-tt` эти предупреждения превращаются в ошибки. Эти возможности настоятельно рекомендуются!

Многие редакторы имеют настройку перевода табуляций в пробелы. При настройке редактора следует различать символ табуляции (`'\t'`) и клавишу `"Tab"`.

- **Символ** табуляции должен быть преобразован в 8 пробелов, чтобы соответствовать семантике языка – по крайней мере, в тех случаях, когда возможен (случайный) смешанный отступ. Редакторы могут также автоматически преобразовывать символ табуляции в пробелы.
- Однако, возможно, вам может оказаться полезным настроить редактор таким образом, чтобы нажатие на клавишу `"Tab"` вместо вставки символа табуляции добавляло бы четыре пробела.

Исходный код Python, написанный с использованием смеси табуляций и пробелов или с нестандартным количеством отступов, может быть приведен в соответствие с pep8 с помощью инструмента `autoper8` (менее мощная альтернатива поставляется с большинством инсталляций Python: это `reindent.py`).

1.4. Типы данных

Встроенные типы

Логические значения (булева логика)

`bool`: это значение булевой логики, содержащее либо `True`, либо `False`. Логические операции типа `and`, `or`, `not` могут быть выполнены при помощи булевой логики.

```
x or y  # если x – False, то y, иначе x
x and y # если x – False, то x, иначе y
not x   # если x – True, то False, иначе True
```

В Python 2.x и Python 3.x логические значения также являются `int`. Тип `bool` является подклассом типа `int`, а `True` и `False` – единственные его экземпляры:

```
issubclass(bool, int) # True
```

```
isinstance(True, bool) # True
isinstance(False, bool) # True
```

Если в арифметических операциях используются логические значения, то их целочисленные значения (1 и 0 для `True` и `False` соответственно) будут использоваться для возврата целочисленного результата:

```
True + False == 1      # 1 + 0 == 1
True * True == 1       # 1 * 1 == 1
```

Числа

- `int`: Целое число

```
a = 2
b = 100
c = 123456789
d = 38563846326424324
```

Целочисленные числа в Python имеют произвольный размер.

Примечание: в старых версиях Python был доступен тип `long`, который отличался от `int`. Эти два типа были объединены.

- `float`: число с плавающей точкой; точность зависит от реализации и архитектуры системы, для CPython тип данных `float` соответствует типу `double` в C.

```
a = 2.0
b = 100.e0
c = 123456789.e1
```

- `complex`: комплексные числа

```
a = 2 + 1j
b = 100 + 10j
```

Операторы `<`, `<=`, `>` и `>=` вызывают исключение `TypeError`, если любой из операндов является комплексным числом.

Строки

Версия Python 3.x ≥ 3.0:

- `str`: строка в кодировке Unicode ('hello')
- `bytes`: массив байтов (b'hello')

Версия Python 2.x ≤ 2.7:

- `str`: байтовая строка ('hello')

- bytes: то же, что и str
- unicode: строка в формате Unicode (Тип u'hello')

Последовательности и коллекции

В Python различают упорядоченные последовательности и неупорядоченные коллекции (такие как set и dict).

- строки (str, bytes, unicode) – это последовательности.
- reversed: обратный порядок str с функцией reversed.

```
a = reversed('hello')
```

- tuple: упорядоченная коллекция из n значений любого типа (n >= 0) (“кортеж”).

```
a = (1, 2, 3)
```

```
b = ('a', 1, 'python', (1, 2))
```

```
b[2] = 'something else' # возвращает ошибку типа TypeError
```

Поддерживает индексацию; неизменяемый; хешируемый, если все его члены хешируемы.

- list: Упорядоченная коллекция из n значений (n >= 0).

```
a = [1, 2, 3]
```

```
b = ['a', 1, 'python', (1, 2), [1, 2]]
```

```
b[2] = 'something else' # такое присвоение разрешено
```

Не хешируемый; изменяемый.

- set: неупорядоченная коллекция уникальных значений. Элементы должны быть хешируемыми.

```
a = {1, 2, 'a'}
```

- dict: неупорядоченная коллекция уникальных пар “ключ-значение”; ключи должны быть хешируемыми.

```
a = {1: 'one',  
     2: 'two'}
```

```
b = {'a': [1, 2, 3],  
     'b': 'a string'}
```

Объект является хешируемым, если он имеет хеш-значение, которое никогда не меняется в течение его жизни (для этого нужен метод `__hash__()`), и может сравниваться с другими объектами (для этого нужен метод `__eq__()`). Хешируемые объекты, которые сравниваются на равенство, должны иметь одинаковое хеш-значение.

Встроенные константы

Вместе со встроенными типами данных во встроенном пространстве имен имеется небольшое количество встроенных констант.

- True: истинное значение встроенного типа bool.
- False: ложное значение встроенного типа bool.
- None: синглтон-объект, который сигнализирует об отсутствии значения.
- Ellipsis или ...: используется в ядре Python3+ везде и ограничено используется в Python2.7+ как часть нотации массивов; numpy и родственные пакеты используют это в качестве ссылки “включить все” в массивы.
- NotImplemented: синглтон, используемый для указания Python на то, что специальный метод не поддерживает указанные аргументы, и Python будет пробовать альтернативные варианты, если они доступны.

```
a = None # Значение не присваивается. Позже может быть присвоен любой допустимый  
        # тип данных
```

Версия Python 3.x ≥ 3.0:

None не имеет естественного упорядочения. Использование операторов сравнения упорядочивания (<, <=, >=, >) больше не поддерживается и приведет к ошибке TypeError.

Версия Python 2.x ≤ 2.7:

None всегда меньше любого числа (None < -32 оценивается как True).

Тестирование типа переменных

В Python мы можем проверить тип данных объекта с помощью встроенной функции type.

```
a = '123'
print(type(a))
# Результат: <class 'str'>
b = 123
print(type(b))
# Результат: <class 'int'>
```

В условных операторах можно проверять тип данных с помощью функции isinstance. Однако полагаться на тип переменной обычно не рекомендуется.

```
i = 7
if isinstance(i, int):
    i += 1
elif isinstance(i, str):
    i = int(i)
    i += 1
```

Чтобы проверить, является ли что-либо типом NoneType:

```
x = None
if x is None:
    print('Ничего удивительного, я просто определил x как None.')
```

Преобразование между типами данных

Можно выполнить явное преобразование типов данных.

Например, '123' имеет тип str и может быть преобразован в целое число с помощью функции int.

```
a = '123'
b = int(a)
```

Преобразование из float строки, например '123.456', можно выполнить с помощью функции float.

```
a = '123.456'
b = float(a)
c = int(a)      # ValueError: недопустимый литерал для int() с основанием 10: '123.456'
d = int(b)      # 123
```

Можно также преобразовывать типы последовательностей или коллекций

```
a = 'hello'
list(a)  # ['h', 'e', 'l', 'l', 'o']
set(a)   # {'o', 'e', 'l', 'h'}
tuple(a) # ('h', 'e', 'l', 'l', 'o')
```

Явный строковый тип при определении литералов

С помощью однобуквенных меток, расположенных непосредственно перед кавычками, можно определить тип строки, которую требуется определить.

- b'foo bar': результатом будет bytes в Python 3, str в Python 2
- u'foo bar': результатом будет str в Python 3, unicode в Python 2

- 'foo bar': результатом будет str
- r'foo bar': результатом будет так называемая сырая строка, где все символы будут интерпретироваться как есть (т. е. без обработки специальных символов, таких как обратные слешы)

```
normal = 'foo\nbar'      # foo
                        # bar
escaped = 'foo\\nbar'    # foo\nbar
raw     = r'foo\nbar'    # foo\nbar
```

Изменяемые и неизменяемые типы данных

Объект называется изменяемым, если он может изменять свое состояние или содержимое. Например, при передаче списка в некоторую функцию список может быть изменен:

```
def f(m):
    m.append(3)  # добавляет число в список. Это изменение.
```

```
x = [1, 2]
f(x)
x == [1, 2]      # теперь это ложно, так как в список был добавлен элемент
```

Объект называется неизменяемым, если он не может быть изменен никаким образом. Например, целые числа являются неизменяемыми, поскольку их невозможно изменить:

```
def bar():
    x = (1, 2)
    g(x)
    x == (1, 2)    # всегда будет правдивым, так как ни одна функция не может изменить объект (1, 2)
```

Заметим, что сами переменные являются изменяемыми, поэтому мы можем переназначить переменную x, но это не изменит объект, на который ранее указывала x. Это только заставит x указывать на новый объект.

Типы данных, экземпляры которых являются изменяемыми, называются **изменяемыми типами данных**; аналогично для неизменяемых объектов и типов данных.

Примеры неизменяемых типов данных:

- int, long, float, complex
- str
- bytes
- tuple
- frozenset

Примеры изменяемых типов данных:

- bytearray
- list
- set
- dict

1.5. Типы коллекций

В Python существует несколько типов коллекций. В то время как такие типы как int и str хранят одно значение, типы коллекций хранят несколько значений.

Списки (lists)

Тип list является, пожалуй, наиболее часто используемым типом коллекции в Python. Несмотря на свое название, список больше похож на массив в других языках, особенно в JavaScript.

В Python list (список) – это просто упорядоченная коллекция допустимых значений Python. Список можно создать, заключив значения, разделенные запятыми, в квадратные скобки:

```
int_list = [1, 2, 3]
string_list = ['abc', 'defghi']
```

Список может быть пустым:

```
empty_list = []
```

Элементы списка не ограничены одним типом данных, что вполне логично, учитывая, что Python – динамический язык:

```
mixed_list = [1, 'abc', True, 2.34, None]
```

Список может содержать в качестве своего элемента другой список:

```
nested_list = [['a', 'b', 'c'], [1, 2, 3]].
```

Доступ к элементам списка может осуществляться через индекс, или числовое представление их положения. Списки в Python имеют нулевой индекс, это означает, что первый элемент в списке имеет индекс 0, второй элемент имеет индекс 1 и так далее:

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric'].
print(names[0]) # Alice
print(names[2]) # Craig
```

Индексы могут быть и отрицательными, что означает отсчет от конца списка (-1 – индекс последнего элемента). Таким образом, используя список из приведенного выше примера:

```
print(names[-1]) # Eric
print(names[-4]) # Bob
```

Списки являются изменяемыми, значения в них можно менять:

```
names[0] = 'Ann'
print(names)
# Выходное значение ['Ann', 'Bob', 'Craig', 'Diana', 'Eric'].
```

Кроме того, можно добавлять и/или удалять элементы из списка.

Добавить объект в конец списка можно с помощью команды `L.append(object)`, возвращающей `None`.

```
names = ['Alice', 'Bob', 'Craig', 'Diana', 'Eric']
names.append("Sia")
print(names)
# Выходное значение ['Alice', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia'].
```

Добавить новый элемент в список по заданному индексу можно при помощи команды `L.insert(index, object)`.

```
names.insert(1, "Nikki")
print(names)
# Выходное значение ['Alice', 'Nikki', 'Bob', 'Craig', 'Diana', 'Eric', 'Sia'].
```

Удалить первое вхождение значения можно с помощью команды `L.remove(value)`, которая возвращает `None`.

```
names.remove("Bob")
print(names) # Выходное значение ['Alice', 'Nikki', 'Craig', 'Diana', 'Eric', 'Sia'].
```

Получить индекс в списке первого элемента, значение которого равно `x` (при отсутствии такого элемента выдаст ошибку):

```
name.index("Alice")
0
```

Подсчитать длину списка:

```
len(names)
6
```

Подсчет количества вхождений любого элемента в списке:

```
a = [1, 1, 1, 2, 3, 4]
a.count(1)
3
```

Обратить последовательность в списке:

```
a.reverse()
[4, 3, 2, 1, 1, 1]
# или
a[::-1]
[4, 3, 2, 1, 1, 1]
```

Удаление и возврат элемента по индексу (по умолчанию последний элемент) с помощью `L.pop([index])` возвращает элемент:

```
names.pop()    # Выведет 'Sia'
```

Вы можете перебирать элементы списка, как показано ниже:

```
for element in my_list:
    print (element)
```

Кортежи (tuples)

tuple похож на список, за исключением того, что он имеет фиксированную длину и является неизменяемым. Таким образом, значения в кортеже не могут быть изменены, а также не могут быть добавлены в кортеж или удалены из него. Кортежи обычно используются для небольших коллекций значений, которые не должны меняться, например IP-адрес и порт. В кортежах используются круглые скобки вместо квадратных:

```
ip_address = ('10.20.30.40', 8080)
```

Правила индексирования списков применяются и к кортежам. Кортежи также могут быть вложенными, а их значения могут быть любыми допустимыми в Python значениями.

Кортеж, имеющий только один элемент, должен быть определен (обратите внимание на запятую) таким образом:

```
one_member_tuple = ('Only member',)
или
one_member_tuple = 'Only member',      # Без скобок
или просто использовать синтаксис кортежей:
one_member_tuple = tuple(['Only member'])
```

Словари (dictionaries)

Словарь в Python – это набор пар “ключ-значение”. Словарь окружен фигурными скобками. Каждая пара разделяется запятой, а ключ и значение – двоеточием. Приведем пример:

```
state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Чтобы получить значение, обратитесь к нему по ключу:

```
ca_capital = state_capitals['California']
```


Можно также получить все ключи в словаре и затем перебирать их:

```
for k in state_capitals.keys():
    print('{} является столицей {}'.format(state_capitals[k], k))
```

Словари напоминают синтаксис JSON. Для преобразования между JSON и словарями можно использовать собственный модуль `json` стандартной библиотеки Python.

Set (набор)

Set – это совокупность элементов без повторов и порядка вставки, но с отсортированным порядком. Они используются в ситуациях, когда важно только то, что некоторые элементы сгруппированы вместе, а не то, в каком порядке они были включены. Для больших групп данных гораздо быстрее проверить, входит ли элемент в `set` (набор), чем сделать то же самое для `list` (списка).

Определение набора очень похоже на определение словаря:

```
first_names = {'Adam', 'Beth', 'Charlie'}
```

Или можно создать `set`, используя существующий `list`:

```
my_list = [1,2,3]
my_set = set(my_list)
```

Проверить принадлежность к набору можно с помощью функции `in`:

```
if name in first_names:
    print(name)
```

Вы можете перебирать элементы набора точно так же, как элементы списка, но помните: значения будут располагаться в произвольном, определяемом реализацией порядке.

defaultdict

`defaultdict` – это словарь со значением по умолчанию для ключей, так что к ключам, для которых не было явно определено значение, можно обращаться без ошибок. `defaultdict` особенно полезен, когда значения в словаре представляют собой коллекции (списки, словари и т. д.) в том смысле, что его не нужно инициализировать каждый раз, когда используется новый ключ.

В `defaultdict` никогда не возникает ошибка `KeyError`. Для любого несуществующего ключа возвращается значение по умолчанию. Например, рассмотрим следующий словарь:

```
>>> state_capitals = {
    'Arkansas': 'Little Rock',
    'Colorado': 'Denver',
    'California': 'Sacramento',
    'Georgia': 'Atlanta'
}
```

Если мы попытаемся получить доступ к несуществующему ключу, Python выдаст нам ошибку, как показано ниже:

```
>>> state_capitals['Alabama']
Traceback (most recent call last):
```

```
File "<ipython-input-61-236329695e6f>", строка 1, in <module>
    state_capitals['Alabama']
```

```
KeyError: 'Alabama'
```

Попробуем использовать `defaultdict`. Его можно найти в модуле `collections`.

```
>>> from collections import defaultdict
>>> state_capitals = defaultdict(lambda: 'Boston')
```

Здесь мы задали значение по умолчанию (**Boston**) на случай, если ключ не существует. Теперь заполните dict, как и раньше:

```
>>> state_capitals['Arkansas'] = 'Little Rock'
>>> state_capitals['California'] = 'Sacramento'
>>> state_capitals['Colorado'] = 'Denver'
>>> state_capitals['Georgia'] = 'Atlanta'
```

Если мы попытаемся обратиться к dict с несуществующим ключом, Python вернет нам значение по умолчанию, т.е. Boston:

```
>>> state_capitals['Alabama']
'Boston'
```

и возвращает созданные значения для существующего ключа также, как и обычный словарь

```
>>> state_capitals['Arkansas']
'Little Rock'
```

1.6. IDLE – графический интерфейс Python

IDLE – это интегрированная среда разработки и обучения Python, являющаяся альтернативой командной строке. Как следует из названия, IDLE очень полезна для разработки нового кода или изучения языка Python. В Windows она поставляется вместе с интерпретатором Python, но в других операционных системах может потребоваться ее установка через менеджер пакетов.

Среди возможностей IDLE следует выделить:

- многооконный текстовый редактор с подсветкой синтаксиса, автозаполнением и интеллектуальным отступом;
- оболочку Python с подсветкой синтаксиса;
- встроенный отладчик с пошаговым выполнением, постоянными точками останова и видимостью стека вызовов;
- автоматические отступы (полезно для начинающих изучать отступы в Python);
- сохранив программу на языке Python в формате .py, в дальнейшем с помощью IDLE ее можно запускать и редактировать в любом месте.

В IDLE для запуска интерпретатора нажмите F5 или запустите оболочку Python. Использование IDLE может быть более удобным для начинающих пользователей, поскольку код интерпретируется по мере написания.

Заметим также, что существует множество альтернатив IDLE.

Поиск и устранение неисправностей (Troubleshooting)

- Windows

Если вы работаете под Windows, то по умолчанию используется команда python. Если вы получаете ошибку “python’ is not recognized”, то, скорее всего, причина в том, что местоположение Python не указано в переменной окружения PATH вашей системы. Доступ к ней можно получить, щелкнув правой кнопкой мыши на “Мой компьютер” и выбрав “Свойства” или перейдя в “Система” через “Панель управления”. Выберите пункт “Дополнительные параметры системы”, а затем “Переменные среды...”. Отредактируйте переменную PATH, указав в ней каталог установки Python, а также папку Script (обычно это C:\Python27; C:\Python27\Scripts). Это требует прав администратора; также может понадобиться перезагрузка.

При использовании нескольких версий Python на одном устройстве возможным решением является переименование одного из файлов python.exe. Например, если назвать одну из версий python27.exe, то python27 станет командой запуска Python для этой версии.

Можно также использовать Python Launcher for Windows, который доступен через программу установки и поставляется по умолчанию. Он позволяет выбрать версию Python для запуска с помощью команды py -[x.y] вместо python[x.y]. Вы можете использовать последнюю версию Python 2, запуская скрипты с py -2, и последнюю версию Python 3, запуская скрипты с py -3.

Debian/Ubuntu/MacOS

В данном разделе предполагается, что местоположение исполняемого файла python было добавлено в переменную окружения PATH.

Если вы работаете на Debian/Ubuntu/MacOS, откройте терминал и введите python для Python 2.x или python3 для Python 3.x.

Введите which python, чтобы узнать, какой интерпретатор Python будет использоваться.

Arch Linux

По умолчанию в Arch Linux (и его потомках) используется Python 3, поэтому используйте python или python3 для Python 3.x и python2 для Python 2.x.

Другие системы

Python 3 иногда связывают с python вместо python3. Для использования Python 2 на таких системах, где он установлен, можно использовать команду python2.

1.7. Ввод данных пользователем

Интерактивный ввод

Чтобы получить ввод от пользователя, используйте функцию input (**примечание:** в Python 2.x эта функция называется raw_input, хотя в Python 2.x есть своя, совершенно отличная версия input):

Версия Python 2.x ≥ 2.3:

```
name = raw_input("What is your name? ")
# Вывод: What is your name? _
```

Замечание по безопасности. Не используйте функцию input() в Python 2 – введенный текст будет оценен как выражение в Python (эквивалентно eval(input()) в Python 3), что может легко стать уязвимостью.

Версия Python 3.x ≥ 3.0:

```
name = input("What is your name? ")
# Вывод: What is your name? _
```

В остальной части этого примера будет использоваться синтаксис Python 3.

Функция принимает строковый аргумент, который отображает это в виде подсказки и возвращает строку. Приведенный код выдает подсказку, ожидая ввода пользователя.

```
name = input("What is your name? ")
# Вывод: What is your name?
```

Если пользователь наберет "Bob" и нажмет клавишу Enter, то переменной name будет присвоена строка "Bob":

```
name = input("What is your name? ")
# Вывод: What is your name? Bob
print(name)
# Вывод: Bob
```

Обратите внимание, что input всегда имеет тип str, что важно, если вы хотите, чтобы пользователь вводил числа. Поэтому необходимо преобразовать str, прежде чем пытаться использовать его в качестве числа:

```
x = input("Write a number:")
# Вывод: Write a number: 10
x / 2
# Вывод: TypeError: unsupported operand type(s) for /: 'str' and 'int'
float(x) / 2
# Вывод: 5.0
```

NB: При работе с пользовательским вводом рекомендуется использовать блоки try/except для перехвата исключений. Например, если ваш код хочет привести raw_input к int, а то, что пишет пользователь, оказывается неприводимым (некастируемым), то возникает ошибка ValueError.

1.8. Встроенные модули и функции

Модуль – это файл, содержащий определения и операторы языка Python. Функция – это фрагмент кода, выполняющий некоторую логику.

```
>>> pow(2,3) #8
```

Для проверки встроенной функции в python мы можем использовать dir(). Если она вызывается без аргумента, то возвращает имена в текущей области видимости. В противном случае возвращается алфавитный список имен, содержащих (некоторые) атрибуты данного объекта, а также атрибуты, доступные из него.

```
>>> dir(__builtin__)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    'BaseException',
    'BufferError',
    'BytesWarning',
    'DeprecationWarning',
    'EOFError',
    'Ellipsis',
    'EnvironmentError',
    'Exception',
    'False',
    'FloatingPointError',
    'FutureWarning',
    'GeneratorExit',
    'IOError',
    'ImportError',
    'ImportWarning',
    'IndentationError',
    'IndexError',
    'KeyError',
    'KeyboardInterrupt',
    'LookupError',
    'MemoryError',
    'NameError',
    'None',
    'NotImplemented',
    'NotImplementedError',
    'OSError',
    'OverflowError',
    'PendingDeprecationWarning',
    'ReferenceError',
    'RuntimeError',
    'RuntimeWarning',
    'StandardError',
    'StopIteration',
    'SyntaxError',
    'SyntaxWarning',
    'SystemError',
    'SystemExit',
    'TabError',
    'True',
    'TypeError',
    'UnboundLocalError',
    'UnicodeDecodeError',
    'UnicodeEncodeError',
    'UnicodeError',
    'UnicodeTranslateError',
    'UnicodeWarning',
    'UserWarning',
    'ValueError',
    'Warning',
    'ZeroDivisionError',
    '__debug__',
    '__doc__',
    '__import__',
    '__name__',
    '__package__',
    'abs',
    'all',
    'any',
    'apply',
    'basestring',
    'bin',
    'bool',
    'buffer',
    'bytearray',
    'bytes',
    'callable',
    'chr',
    'classmethod',
    'cmp',
    'coerce',
    'compile',
    'complex',
    'copyright',
    'credits',
```

```

'delattr',
'dict',
'dir',
'divmod',
'enumerate',
'eval',
'execfile',
'exit',
'file',
'filter',
'float',
'format',
'frozenset',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'intern',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
'locals',
'long',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'quit',
'range',
'raw_input',
'reduce',
'reload',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'unichr',
'unicode',
'vars',
'xrange',
'zip'

```

Для того чтобы узнать функциональность какой-либо функции, можно воспользоваться встроенной функцией `help`.

```

>>> help(max)
Help on built-in function max in module __builtin__:
max(...)
    max(iterable[, key=func]) -> value
    max(a, b, c, ...[, key=func]) -> value
    With a single iterable argument, return its largest item.
    With two or more arguments, return the largest argument.
    ("При наличии одного итерируемого аргумента возвращает наибольшее значение.
    При наличии двух или более аргументов возвращает наибольший аргумент".)

```

Встроенные модули содержат дополнительные функциональные возможности. Например, для получения квадратного корня из числа необходимо включить модуль `math`.

```

>>> import math
>>> math.sqrt(16) # 4.0

```

Чтобы узнать все функции в модуле, можно присвоить переменной список функций, а затем вывести его на печать.

```

>>> import math
>>> dir(math)

['_doc_', '__name__', '__package__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',

```

```
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc'].
```

при помощи doc можно получить некоторую документацию по функциям:

```
>>> math.__doc__
'This module is always available. It provides access to the\nmathematical functions defined by the C
standard.'
(“Этот модуль всегда доступен. Он обеспечивает доступ к \nматематическим функциям,
определенным стандартом C”.)
```

Помимо функций документация может быть представлена и в модулях. Так, если у вас есть модуль с именем helloWorld.py, как в примере:

```
"""This is the module docstring."""
```

```
def sayHello():
    """This is the function docstring."""
    return 'Hello World'
```

Получить доступ к его документам можно следующим образом:

```
>>> import helloWorld
>>> helloWorld.__doc__
'This is the module docstring.'
>>> helloWorld.sayHello.__doc__
'This is the function docstring.'
```

- Для любого заданного пользователем типа с помощью dir() можно получить его атрибуты, атрибуты его класса и рекурсивно атрибуты базовых классов этого класса.

```
>>> class MyClassObject(object):
    pass
```

```
>>> dir(MyClassObject)
['_class_', '_delattr_', '_dict_', '_doc_', '_format_', '_getattrattribute_', '_hash_', '_init_',
'_module_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_setattr_', '_sizeof_', '_str_',
'_subclasshook_', '_weakref_']
```

Любой тип данных может быть просто преобразован в строку с помощью встроенной функции str. Эта функция вызывается по умолчанию, когда тип данных передается в print

```
>>> str(123) # "123"
```

1.9. Создание модуля

Модуль – это импортируемый файл, содержащий определения и выражения. Модуль может быть сделан путем создания файла .py.

```
# hello.py
def say_hello():
    print("Hello!")
```

Функции модуля могут быть использованы путем импорта модуля.

Созданные модули должны находиться в том же каталоге, что и файл, в который вы их импортируете. (Их можно поместить и в каталог библиотек Python lib вместе с предустановленными модулями, но по возможности этого следует избегать.)

```
$ python
>>> import hello
```

```
>>> hello.say_hello()
=> "Hello!"
```

Модули могут быть импортированы другими модулями.

```
# greet.py
import hello
hello.say_hello()
```

Импортировать можно конкретные функции модуля.

```
# greet.py
from hello import say_hello
say_hello()
```

Модули могут быть псевдонимами.

```
# greet.py
import hello as ai
ai.say_hello()
```

Модуль может представлять собой отдельный запускаемый скрипт.

```
# run_hello.py
if __name__ == 'main':
    from hello import say_hello
    say_hello()
```

Запускайте!

```
$ python run_hello.py
=> "Hello!"
```

Если модуль находится в каталоге и должен быть обнаружен Python'ом, то в каталоге должен находиться файл с именем `__init__.py`.

1.10. Установка Python 2.7.x и 3.x

Примечание: следующие инструкции написаны для Python 2.7 (если это не оговорено особо). Инструкции для Python 3.x аналогичны.

Windows

Сначала загрузите последнюю версию Python 2.7 с официального сайта (<https://www.python.org/downloads/>). Версия предоставляется в виде пакета MSI. Чтобы установить ее вручную, просто дважды щелкните на файле.

По умолчанию Python устанавливается в каталог:

```
C:\Python27\
```

Внимание: установка не приводит к автоматическому изменению переменной окружения PATH. Предполагая, что ваша установка Python находится в каталоге C:\Python27, добавьте в PATH:

```
C:\Python27\; C:\Python27\Scripts\
```

Теперь для проверки правильности установки Python напишите в cmd:

```
python --version
```

Python 2.x и 3.x вместе

Для установки и использования Python 2.x и 3.x на компьютере под управлением Windows:

1. Установите Python 2.x с помощью программы установки MSI.

- Убедитесь, что Python установлен для всех пользователей.
- По желанию: добавьте Python в PATH, чтобы Python 2.x можно было вызывать из командной строки с помощью команды python.

2. Установите Python 3.x с помощью соответствующего инсталлятора.

- Снова убедитесь, что Python установлен для всех пользователей.
- По желанию: добавьте Python в PATH, чтобы сделать Python 3.x доступным для вызова из командной строки с помощью команды python. Это может отменить настройки PATH для Python 2.x, поэтому дважды проверьте PATH и убедитесь, что он настроен в соответствии с вашими предпочтениями.
- Убедитесь, что py launcher установлен для всех пользователей.

В Python 3 будет установлена программа Python launcher, с помощью которой из командной строки можно запускать и Python 2.x, и Python 3.x:

```
P:\>py -3
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
("Для получения дополнительной информации введите "help", "copyright", "credits" или "license").
>>>
```

```
C:\>py -2
Python 2.7.13 (v2.7.13:a06454b1afa1, Dec 17 2016, 20:42:59) [MSC v.1500 32 Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
("Для получения дополнительной информации введите "help", "copyright", "credits" или "license").
>>>
```

Чтобы использовать соответствующую версию pip для конкретной версии Python, используйте:

```
C:\>py -3 -m pip -V
pip 9.0.1 from C:\Python36\lib\site-packages (python 3.6)
```

```
C:\>py -2 -m pip -V
pip 9.0.1 from C:\Python27\lib\site-packages (python 2.7)
```

Linux

Последние версии CentOS, Fedora, Red Hat Enterprise (RHEL) и Ubuntu поставляются с установленным Python 2.7. Чтобы установить Python 2.7 на linux вручную, просто выполните следующие действия в терминале:

```
wget --no-check-certificate https://www.python.org/ftp/python/2.7.X/Python-2.7.X.tgz
tar -xzf Python-2.7.X.tgz
cd Python-2.7.X
./configure
make
sudo make install
```

Также добавьте путь к новому Python в переменную окружения PATH. Если новый Python установлен в каталог /root/python-2.7.X, то выполните команду export PATH = \$PATH:/root/python-2.7.X

Теперь для проверки правильности установки Python напишите в терминале:

```
python --version
```

Ubuntu

Если вам нужен Python 3.6, вы можете установить его приведенным ниже способом (для Ubuntu 16.10 и 17.04 версия 3.6 находится в универсальном репозитории). Для Ubuntu 16.04 и более низких версий необходимо выполнить следующие шаги:

```

sudo apt install build-essential checkinstall
sudo apt install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev
libc6-dev libbz2-dev
wget https://www.python.org/ftp/python/3.6.1/Python-3.6.1.tar.xz
tar xvf Python-3.6.1.tar.xz
cd Python-3.6.1/
./configure --enable-optimizations
sudo make altinstall

```

macOS

В настоящее время на macOS устанавливается Python 2.7.10, но эта версия устарела и немного отличается от обычного Python.

Версия Python, поставляемая с OS X, отлично подходит для обучения, но не очень годится для разработки. Версия, поставляемая с OS X, может быть устаревшей по сравнению с текущим выпуском Python, который считается стабильной производственной версией.

Установите Homebrew при помощи:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Установите Python 2.7 при помощи:

```
brew install python
```

Для Python 3.x вместо этого используйте команду `brew install python3`.

1.11. Строковые функции – `str()` и `repr()`

Для получения читаемого представления объекта можно использовать две функции.

`repr(x)` вызывает `x.__repr__()`: представление `x`. `eval` обычно преобразует результат этой функции обратно в исходный объект.

`str(x)` вызывает `x.__str__()`: читабельная для человека строка, описывающая объект. При этом могут быть не учтены некоторые технические детали.

`repr()`

Для многих типов эта функция пытается вернуть строку, которая при передаче в `eval()` будет возвращать объект с тем же значением. В противном случае представление представляет собой строку, заключенную в угловые скобки, которая содержит имя типа объекта вместе с дополнительной информацией. Часто это включает имя и адрес объекта.

`str()`

Для строк эта функция возвращает саму строку. Разница между ней и `repr(object)` заключается в том, что `str(object)` не всегда пытается вернуть строку, приемлемую для `eval()`. Скорее, ее целью является возвращение печатаемой или “читабельной для человека” строки. Если аргумент не указан, то функция возвращает пустую строку ‘’.

Пример 1:

```

s = """w'o'w"""
repr(s)           # Вывод: '\w\\\'o\'w\'
str(s)            # Вывод: 'w\'o\'w'
eval(str(s)) == s  # Выдает SyntaxError (ошибку синтаксиса)
eval(repr(s)) == s # Вывод: True

```

Пример 2:

```

import datetime
today = datetime.datetime.now()
str(today)      # Вывод: '2016-09-15 06:58:46.915000'
repr(today)     # Вывод: 'datetime.datetime(2016, 9, 15, 6, 58, 46, 915000)'

```

При написании класса вы можете переопределить эти методы, чтобы сделать то, что вам нужно:

класс Represent(object):

```
def __init__(self, x, y):
    self.x, self.y = x, y
```

```
def __repr__(self):
    return "Represent(x={},y=\"{}\").format(self.x, self.y)
```

```
def __str__(self):
    return "Representing x as {} and y as {}".format(self.x, self.y)
```

Используя приведенный выше класс, мы можем увидеть результаты:

```
r = Represent(1, "Hopper")
print(r)           # печатает str
print(r.__repr__)  # печатает __repr__: '< связанный метод Represent. repr of Represent(x=1,y="Hopper")>'
rep = r.repr()     # устанавливает выполнение repr в новую переменную
print(rep)         # печатает 'Represent(x=1,y="Hopper")'
r2 = eval(rep)     # оценивает rep
print(r2)          # печатает __str__ из нового объекта
print(r2 == r)     # выводит 'False', так как это разные объекты
```

1.12. Установка внешних модулей с помощью pip

Вам пригодится pip, когда понадобится установить любой пакет из множества доступных в индексе пакетов Python (PyPI). pip уже установлен, если вы используете Python 2 версии выше 2.7.9 или Python 3 версии выше 3.4, загруженные с сайта python.org. На компьютерах, работающих под управлением Linux или другой *nix операционной системы с собственным менеджером пакетов, pip часто приходится устанавливать вручную.

На устройствах, на которых установлен и Python 2, и Python 3, программа pip часто ссылается на Python 2, а pip3 – на Python 3. pip будет устанавливать только пакеты для Python 2, а pip3 – только пакеты для Python 3.

Поиск / установка пакета

Поиск пакета осуществляется просто – достаточно набрать

```
$ pip search <query>
```

Поиск пакетов, имя или краткое содержание которых содержит <запрос>.

Установить пакет очень просто: наберите (в терминале / командной строке, а не в интерпретаторе Python)

```
$ pip install [имя_пакета]           # последняя версия пакета
$ pip install [имя_пакета]==x.x.x   # конкретная версия пакета
$ pip install [имя_пакета]>=x.x.x'   # минимальная версия пакета
```

где x.x.x – номер версии пакета, который вы хотите установить.

Если ваш сервер находится за прокси-сервером, вы можете установить пакет с помощью следующей команды:

```
$ pip --proxy http://<адрес сервера>:<порт> install
```

Обновление установленных пакетов

При появлении новых версий установленных пакетов они не устанавливаются в систему автоматически. Чтобы узнать, какие из установленных пакетов устарели, выполните команду:

```
$ pip list --outdated
```

Для обновления конкретного пакета используйте

```
$ pip install [имя_пакета] --upgrade
```

Обновление всех устаревших пакетов не является стандартной функциональностью pip.

Повышение версии pip

Обновить существующую установку pip можно с помощью следующих команд.

- В Linux или macOS X:

```
$ pip install -U pip
```

В некоторых системах Linux может потребоваться использование sudo вместе с pip.

- В операционной системе Windows:

```
py -m pip install -U pip
```

или

```
python -m pip install -U pip
```

1.13. Справочная утилита

В Python есть несколько функций, встроенных в интерпретатор. Если вы хотите получить информацию о ключевых словах, встроенных функциях, модулях или темах, откройте консоль Python и введите:

```
>>> help()
```

Вы будете получать информацию, вводя ключевые слова напрямую:

```
>>> help(help)
```

или внутри утилиты:

```
help> help
```

в результате чего появится пояснение:

```
Help on _Helper in module _sitebuiltins object:
```

```
class _Helper(builtins.object)
```

```
Define the builtin 'help'.
```

```
This is a wrapper around pydoc.help that provides a helpful message
when 'help' is typed at the Python interactive prompt.
```

```
Calling help() at the Python prompt starts an interactive help session.
```

```
Calling help(thing) prints help for the python object 'thing'.
```

```
Methods defined here:
```

```
__call__(self, *args, **kwargs)
```

```
__repr__(self)
```

```
Data descriptors defined here:
```

```
__dict__
```

```
dictionary for instance variables (if defined)
```

```
__weakref__
```

```
list of weak references to the object (if defined)
```

Можно также запрашивать подклассы модулей:

```
help pymysql.connections)
```

Вы можете использовать `help` для доступа к документам различных импортированных модулей, например, попробуйте сделать следующее:

```
>>> help(math)
и вы получите ошибку

>>> import math
>>> help(math)
```

Теперь вы получите список доступных методов в модуле, но только ПОСЛЕ того, как вы его импортировали. Закрывать справочную утилиту можно с помощью команды `quit`.

Глава 2. Типы данных в Python

Типы данных – это не что иное как переменные, которые используются для резервирования некоторого пространства в памяти. Для резервирования места в памяти переменные Python не требуют явного объявления. Объявление происходит автоматически при присвоении переменной значения.

2.1. Строковый тип данных

Строка идентифицируется как непрерывный набор символов, заключенных в кавычки. Python допускает использование пар одинарных или двойных кавычек. Строки являются неизменяемым последовательным типом данных, т. е. при каждом изменении строки создается совершенно новый строковый объект.

```
a_str = 'Hello World'
print(a_str)           #вывод будет представлять собой целую строку. Hello World
print(a_str[0])        #выводом будет первый символ. H
print(a_str[0:5])      #вывод будет состоять из первых пяти символов. Hello
```

2.2. Множества (set и frozenset)

Множества – это неупорядоченные коллекции уникальных объектов. Существует два их типа:

1. Sets – изменяемы, новые элементы могут быть добавлены после того, как множества определены.

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
print(basket)         # дубликаты будут удалены
> {'orange', 'banana', 'pear', 'apple'}
a = set('abracadabra')
print(a)              # выведет уникальные буквы в a
> {'a', 'r', 'b', 'c', 'd'}
a.add('z')
print(a)
> {'a', 'c', 'r', 'b', 'z', 'd'}
```

2. Frozensets – неизменяемы, новые элементы не могут быть добавлены после определения.

```
b = frozenset('asdfagsa')
print(b)
> frozenset({'f', 'g', 'd', 'a', 's'})
города = frozenset(['Frankfurt', 'Basel', 'Freiburg'])
print(cities)
> frozenset({'Frankfurt', 'Basel', 'Freiburg'})
```

2.3. Числовые типы данных

Числа в Python имеют четыре типа: int, float, complex и long.

```
int_num = 10          #значение int (целочисленное)
float_num = 10.2      # значение float (число с плавающей точкой)
complex_num = 3.14j   # значение complex (комплексное число)
long_num = 1234567L   # значение long ("длинное" целое число неограниченного размера)
```

2.4. Тип данных "список" (list)

Список содержит элементы, разделенные запятыми и заключенные в квадратные скобки []. Списки практически аналогичны массивам в языке C. Отличие состоит в том, что все элементы, входящие в список, могут иметь различный тип данных.

```
list = [123,'abcd',10.2,'d'] #может быть массивом данных любого типа или одного типа данных
list1 = ['hello','world']
print(list)                  #выведет весь список. [123, 'abcd', 10.2, 'd']
print(list[0:2])             #выведет первые два элемента списка [123, 'abcd']
print(list1 * 2)              #выдаст list1 два раза.['hello', 'world', 'hello', 'world']
print(list + list1)          #выведет конкатенацию (соединение) обоих списков. [123, 'abcd',
#10.2, 'd', 'hello','world']
```

2.5. Тип данных "словарь" (dic)

Словарь состоит из пар "ключ-значение". Он заключен в фигурные скобки {}, а для присвоения значений и доступа к ним используются квадратные скобки [].

```
dic={'name':'red', 'age':10}
print(dic)                  #выведет все пары "ключ-значение" {'name':'red', 'age':10}
print(dic['name'])           #выведет только значение с ключом 'name'. 'red'
print(dic.values())          #выведет список значений в dic ['red',10]
print(dic.keys())            #выведет список ключей ['name','age']
```

2.6. Тип данных "кортеж" (tuple)

Списки заключаются в скобки [], и их элементы и размер могут быть изменены, в то время как кортежи заключаются в круглые скобки () и не могут быть обновлены. Кортежи являются неизменяемыми.

```
tuple = (123,'hello')
tuple1 = ('world')
print(tuple)                #выведем весь кортеж (123,'hello')
print(tuple[0])              #выведем первое значение (123)
print(tuple + tuple1)        #выведем (123,'hello','world')
tuple[1]='update'            #это приведет к ошибке.
```

Глава 3. Отступы

3.1. Простой пример

При создании языка Python Гвидо ван Россум использовал группировку выражений по отступам. Двоеточие используется для *объявления блока кода с отступом*, например:

```
class ExampleClass:
    #Каждая функция, принадлежащая классу, должна иметь одинаковый отступ
    def __init__(self):
        name = "example"

    def someFunction(self, a):
        #Обратите внимание, что все, принадлежащее функции, должно быть выделено отступом
        if a > 5:
            return True
        else:
            return False

#Если функция не имеет отступа одного уровня, она не будет рассматриваться как часть
родительского класса
def separateFunction(b):
    for i in b:
        #Циклы также имеют отступ, а вложенные условия начинаются с нового отступа
        if i == 1:
            return True
    return False

separateFunction([2,3,5,6,1])
```

Пробелы или табуляции?

Рекомендуемый отступ – 4 пробела, но можно использовать и табуляции, и пробелы согласованно. **Не смешивайте табуляции и пробелы**, так как это приведет к ошибке в Python 3 и может вызвать ошибки в Python 2.

3.2. Как происходит разбор отступов

Пробельные символы перед разбором обрабатываются лексическим анализатором.

Для хранения уровней отступов лексический анализатор использует стек. В начале стек содержит только значение 0, которое является крайней левой позицией. Каждый раз, когда начинается вложенный блок, новый уровень отступа заносится в стек, а в поток лексем, передаваемый синтаксическому анализатору, вставляется токен “INDENT” (“отступ”). В строке не может быть более одного токена “INDENT” (иначе последует ошибка отступа – `IndentationError`).

Когда встречается строка с меньшим уровнем отступа, значения выгружаются из стека до тех пор, пока на вершине не окажется значение, равное новому уровню отступа (если таковое не найдено, то возникает синтаксическая ошибка). Для каждого извлеченного значения генерируется токен “DEDENT” (“впадина”). Очевидно, что токенов “DEDENT” может быть несколько в строке.

Лексический анализатор пропускает пустые строки (содержащие только пробельные символы и, возможно, комментарии) и никогда не генерирует для них токены “INDENT” или “DEDENT”. В конце исходного кода для каждого оставшегося в стеке уровня отступа генерируются токены “DEDENT”, пока не останется только 0. Например:

```
if foo:
    if bar:
        x = 42
    else:
        print foo
```


анализируется как:

<code><if> <foo> <:></code>	<code>[0]</code>
<code><INDENT> <if> <bar> <:></code>	<code>[0, 4]</code>
<code><INDENT> <x> <=> <42></code>	<code>[0, 4, 8]</code>
<code><DEDENT> <DEDENT> <else> <:></code>	<code>[0]</code>
<code><INDENT> <print> <foo></code>	<code>[0, 2]</code>
<code><DEDENT></code>	

Синтаксический анализатор затем использует токены “INDENT” и “DEDENT” в качестве разделителей блоков.

3.3. Ошибки отступа

Интервалы должны быть равномерными и одинаковыми. Неправильный отступ может вызвать ошибку `IndentationError` или привести к неожиданным действиям программы. Следующий пример приводит к ошибке `IndentationError`:

```
a = 7
if a > 5:
    print "foo"
else:
    print "bar"
    print "done"
```

Или если строка, следующая за двоеточием, не имеет отступа, то также последует ошибка `IndentationError`:

```
if True:
    print "true"
```

Если добавить отступы там, где они не должны быть, результатом также станет ошибка `IndentationError`:

```
if True:
    a = 6
    b = 5
```

Если забыть снять отступ, то функциональность может быть потеряна. В данном примере вместо ожидаемого значения `False` возвращается `None`:

```
def isEven(a):
    if a%2 ==0:
        return True
        #следующая строка должна быть вровень с if
        return False
print isEven(7)
```

Глава 4. Комментарии и документация

4.1. Однострочные, строчные и многострочные комментарии

Комментарии используются для пояснения кода, если основной код может быть непонятен сам по себе.

Python игнорирует комментарии, поэтому не будет выполнять содержащийся в них код или выдавать синтаксические ошибки для обычных предложений. Однострочные комментарии начинаются с символа хеша (#) и завершаются концом строки.

- Однострочный комментарий:

```
# Это однострочный комментарий в Python
```

- Встроенный комментарий:

```
print("Hello World")      # Эта строка печатает "Hello World"
```

- Комментарии, охватывающие несколько строк, имеют с обеих сторон символы `"""` или `'''`.

Этот тип комментария занимает несколько строк.

В основном они используются для документации в функциях, классах и модулях.

4.2. Программный доступ к строкам документации (docstrings)

Docstrings – в отличие от обычных комментариев – хранятся как атрибут функции, которую они документируют, что означает возможность программного доступа к ним.

Пример функции

```
def func():
    """Это функция, которая вообще ничего не делает"""
    return
```

Доступ к строке документации можно получить с помощью атрибута `__doc__` attribute:

```
print(func.__doc__)
```

```
    Это функция, которая вообще ничего не делает
```

```
help(func)
```

```
Help on function func in module __main__:
```

```
func()
```

```
    Это функция, которая вообще ничего не делает
```

Еще один пример функции

`function.__doc__` — это просто фактическая строка документации в виде строки, а функция `help` предоставляет общую информацию о функции, включая строку `docstring`. Вот более полезный пример:

```
def greet(name, greeting="Hello"):
    """Выведите приветствие пользователю `name`.
```

Дополнительный параметр ``greeting`` может изменить то, с чем они будут поздравлены."""

```
print("{} {}".format(greeting, name))
```

```
help(greet)
```

```
Help on function greet in module __main__:
```

```
greet(name, greeting=>Hello)
```

```
    Выведите приветствие пользователю `name`.
```

```
    Дополнительный параметр `greeting` может изменить то, с чем они будут
    поздравлены.
```

Преимущества docstrings перед обычными комментариями

Отсутствие в функции `docstring` или обычного комментария делает ее гораздо менее полезной.

```
def greet(name, greeting="Hello"):
```

```
    # Вывести приветствие пользователю `name`.
```

```
    # Необязательный параметр `greeting` позволяет изменять то, как они будут приветствоваться
```

```
print("{} {}".format(greeting, name))
print(greet.__doc__)

None

help(greet)

Help on function greet in module main:
greet(name, greeting='Hello')
```

4.3. Написание документации с использованием docstrings

docstring – это многострочный комментарий, используемый для документирования модулей, классов, функций и методов. Он должен быть первым высказыванием компонента, который он описывает.

```
def hello():
    """Поздоровайтесь с кем-нибудь."""

    Вывести приветствие ("Hello") для человека с заданным именем.

    print("Hello "+name)
```

```
class Greeter:
    """Объект, используемый для приветствия людей.
    Он содержит несколько функций приветствия для нескольких языков и времени суток.
```

Значение docstring может быть доступно внутри программы и используется, например, командой help.

Синтаксические соглашения в PEP 257

257 определяет стандарт синтаксиса для комментариев docstrings. В основном он допускает два типа:

- Однострочные docstrings:

Согласно PEP 257, их следует использовать с короткими и простыми функциями. Все помещается в одну строку, например

```
def hello():
    """Поприветствуй друзей."""
    print("Привет, друзья!")
```

docstring должна заканчиваться точкой, глагол должен быть в императивной форме.

- Многострочные docstrings:

Многострочные docstring следует использовать для более длинных и сложных функций, модулей или классов.

```
def hello(name, language="en"):
    """Поздороваться с человеком.
    Аргументы:
    name: имя человека
    language: язык, на котором следует приветствовать собеседника

    print(greeting[language]+" "+name)
```

Они начинаются с краткого резюме (эквивалентного содержанию однострочной docstring), которое может находиться в одной строке с кавычками или на следующей строке, содержат дополнительные сведения и перечисляют параметры и возвращаемые значения.

Примечание: PEP 257 определяет, какая информация должна быть представлена в docstring, но не определяет, в каком формате она должна быть представлена. Это послужило причиной того, что другие стороны и средства разбора документации определили свои собственные стандарты для документации, некоторые из которых перечислены ниже.

Синтаксические соглашения в Sphinx

Sphinx – это инструмент для генерации HTML-документации для проектов на языке Python на основе docstrings. Используемый язык разметки – это reStructuredText. В Sphinx существуют собственные стандарты документации, на сайте pythonhosted.org есть очень хорошее их описание. Формат Sphinx используется, например, в pyCharm IDE.

В таком виде функция будет документироваться в формате Sphinx/reStructuredText:

```
def hello(name, language="en"):
    """Поздороваться с человеком.

    param name: имя человека
    :type name: str
    :param language: язык, на котором следует приветствовать собеседника
    :type language: str
    :return: число
    :rtype: int

    print(greeting[language]+" "+name)
    return 4
```

Руководство по стилю Google Python

Компания Google опубликовала руководство по стилю Google Python Style Guide, которое определяет правила кодирования на языке Python, включая комментарии к документации. По сравнению с Sphinx/reST многие отмечают, что документация, составленная в соответствии с рекомендациями Google, лучше читается человеком.

На упомянутой выше странице pythonhosted.org также приведены некоторые примеры хорошей документации в соответствии с руководством по стилю Google.

Используя плагин Napoleon, Sphinx может также анализировать документацию в формате, соответствующем Google Style Guide. В формате Google Style Guide функция будет документирована следующим образом:

```
def hello(name, language="en"):
    """Поздороваться с человеком.

    Args:
        name: имя человека в виде строки
        language: строка кода языка

    Возвращает:
        Число.

    print(greeting[language]+" "+name)
    return 4
```

Глава 5. Дата и время

5.1. Разбор строки в объект даты и времени (datetime) с учетом часового пояса

В Python 3.2+ появилась поддержка формата %z при разборе строки в объект datetime.

Смещение UTC (UTC offset) задается в виде +ННММ или -ННММ (пустая строка используется, если объект не содержит информации о временной зоне).

Версия Python 3.x ≥ 3.2:

```
import datetime
dt = datetime.datetime.strptime("2016-04-15T08:27:18-0500", "%Y-%m-%dT%H:%M:%S%z")
```

Для других версий Python можно использовать внешнюю библиотеку, например dateutil, которая позволяет быстро разобрать строку с часовым поясом в объект datetime.

```
import dateutil.parser
dt = dateutil.parser.parse("2016-04-15T08:27:18-0500")
```

Переменная dt теперь является объектом datetime со следующим значением:

```
datetime.datetime(2016, 4, 15, 8, 27, 18, tzinfo=tzoffset(None, -18000))
```

5.2. Построение временных интервалов с учетом временных зон

По умолчанию все объекты datetime не относятся ни к одному часовому поясу. Чтобы сделать привязку на часовые пояса, необходимо подключить объект tzinfo, который привязывает функцию даты и времени к стандарту UTC.

Часовые пояса с фиксированным смещением

В Python 3.2+ для часовых поясов, расположенных со смещением от UTC фиксированно, модуль datetime предоставляет класс timezone, конкретную реализацию tzinfo, который принимает timedelta и необязательный параметр name:

Версия Python 3.x ≥ 3.2:

```
from datetime import datetime, timedelta, timezone
JST = timezone(timedelta(hours=+9))

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00

print(dt.tzname())
# UTC+09:00

dt = datetime(2015, 1, 1, 12, 0, 0, tzinfo=timezone(timedelta(hours=9), 'JST'))
print(dt.tzname())
# 'JST'
```

Для версий Python до 3.2 необходимо использовать сторонние библиотеки, такие как dateutil. Эта библиотека предоставляет эквивалентный класс tzoffset, который (начиная с версии 2.5.3) принимает аргументы вида dateutil.tz.tzoffset(tzname, offset), где смещение (offset) задается в секундах:

Версия Python 3.x < 3.2:

Версия Python 2.x < 2.7:

```
from datetime import datetime, timedelta
from dateutil import tz
```

```
JST = tz.tzoffset('JST', 9 * 3600) # 3600 секунд в часе
dt = datetime(2015, 1, 1, 12, 0, tzinfo=JST)
print(dt)
# 2015-01-01 12:00:00+09:00
print(dt.tzname)
# 'JST'
```

Часовые пояса с переходом на летнее время

Для часовых поясов с летним временем стандартные библиотеки python не предоставляют стандартного класса, поэтому необходимо использовать сторонние библиотеки. Популярные библиотеки, предоставляющие классы для работы с часовыми поясами – pytz и dateutil.

В дополнение к статическим часовым поясам в dateutil есть классы часовых поясов, использующих летнее время (см. документацию к модулю tz). С помощью метода tz.gettz() можно получить объект часового пояса, который затем можно передать непосредственно в конструктор datetime:

```
from datetime import datetime
from dateutil import tz
local = tz.gettz() # Местное время
PT = tz.gettz('US/Pacific') # Тихоокеанское время

dt_l = datetime(2015, 1, 1, 12, tzinfo=local) # Я нахожусь в EST (Североамериканском
# восточном времени)
dt_pst = datetime(2015, 1, 1, 12, tzinfo=PT)
dt_pdt = datetime(2015, 7, 1, 12, tzinfo=PT) # DST (летнее время) обрабатывается автоматически
print(dt_l)
# 2015-01-01 12:00:00-05:00
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-07-01 12:00:00-07:00
```

ВНИМАНИЕ: Начиная с версии 2.5.3 dateutil некорректно обрабатывает неоднозначные даты, и по умолчанию всегда выбирается *более поздняя* дата. В dateutil нет возможности создать объект с текущим часовым поясом, например 2015-11-01 1:30 EDT-4, так как это происходит во время перехода на летнее время.

Все крайние случаи при использовании pytz обрабатываются правильно, однако часовые пояса pytz не следует напрямую присоединять через конструктор. Вместо этого часовой пояс pytz должен быть присоединен с помощью метода localize:

```
from datetime import datetime, timedelta
import pytz

PT = pytz.timezone('US/Pacific')
dt_pst = PT.localize(datetime(2015, 1, 1, 12))
dt_pdt = PT.localize(datetime(2015, 11, 1, 0, 30))
print(dt_pst)
# 2015-01-01 12:00:00-08:00
print(dt_pdt)
# 2015-11-01 00:30:00-07:00
```

Следует иметь в виду, что если вы выполняете вычисление времени в часовом поясе, поддерживаемом pytz, то вычисления должны выполняться либо в UTC (если вы хотите получить абсолютное истекшее время), либо необходимо вызвать функцию normalize() :

```
dt_new = dt_pdt + timedelta(hours=3) # Это должно быть 2:30 AM PST
print(dt_new)
# 2015-11-01 03:30:00-07:00
dt_corrected = PT.normalize(dt_new)
print(dt_corrected)
# 2015-11-01 02:30:00-08:00
```

5.3. Вычисление разницы во времени

Для вычисления разницы между временами удобно использовать модуль `timedelta`:

```
from datetime import datetime, timedelta
now = datetime.now()
then = datetime(2016, 5, 23)      # datetime.datetime(2016, 05, 23, 0, 0, 0)
```

При создании нового объекта `datetime` указание времени необязательно.

```
delta = now - then
```

`delta` имеет тип `timedelta`.

```
print(delta.days)
# 60
print(delta.seconds)
# 40826
```

Для получения даты `n` дней после и `n` дней до мы можем использовать:

`n` дней после даты:

```
def get_n_days_after_date(date_format="%d %B %Y", add_days=120):

    date_n_days_after = datetime.datetime.now() + timedelta(days=add_days)
    return date_n_days_after.strftime(date_format)
```

`n` дней до даты:

```
def get_n_days_before_date(self, date_format="%d %B %Y", days_before=120):
    date_n_days_ago = datetime.datetime.now() - timedelta(days=days_before)
    return date_n_days_ago.strftime(date_format)
```

5.4. Использование базовых объектов `datetime`

Модуль `datetime` содержит три основных типа объектов – дата, время и дата вместе со временем.

```
import datetime

# Date object (объект даты)
today = datetime.date.today()
new_year = datetime.date(2017, 01, 01) #datetime.date(2017, 1, 1)

# Time object (объект времени)
noon = datetime.time(12, 0, 0) #datetime.time(12, 0)

# Current datetime (текущие дата и время)
now = datetime.datetime.now()

# Datetime object (объект даты и времени)
millenium_turn = datetime.datetime(2000, 1, 1, 0, 0, 0) #datetime.datetime(2000, 1, 1, 0, 0)
```

Арифметические операции для этих объектов поддерживаются только в пределах одного типа данных. Выполнение простых арифметических операций с экземплярами разных типов приведет к ошибке `TypeError`.

```
# вычитание полудня из сегодняшнего дня
noon - today
```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.date'
```

```
# Вместо этого сделайте следующее:
print('Время с момента наступления тысячелетия в полночь: ',
      datetime.datetime(today.year, today.month, today.day) - millenium_turn)
```

```
# Или так
print('Время с момента наступления тысячелетия в полдень: ',
      datetime.datetime.combine(today, noon) - millenium_turn)
```

5.5. Переключение между часовыми поясами

Для переключения между часовыми поясами необходимы объекты `datetime`, которые учитывают часовые пояса.

```
from datetime import datetime
from dateutil import tz

utc = tz.tzutc()
local = tz.tzlocal()

utc_now = datetime.utcnow()
utc_now      # Без учета часового пояса.

utc_now = utc_now.replace(tzinfo=utc)
utc_now      # С учетом часового пояса.

local_now = utc_now.astimezone(local)
local_now    # Преобразовано в местное время.
```

5.6. Простые арифметические действия с датами

Даты не существуют сами по себе. Часто возникает необходимость определить промежуток времени между датами или определить, какая дата будет завтра. Это можно сделать с помощью объектов `timedelta`:

```
import datetime

today = datetime.date.today()
print('Сегодня:', today)

yesterday = today - datetime.timedelta(days=1)
print('Вчера:', yesterday)

tomorrow = today + datetime.timedelta(days=1)
print('Завтра:', tomorrow)

print('Время между завтра и вчера:', tomorrow - yesterday)
```

В результате будут получены результаты:

```
Сегодня: 2016-04-15
Вчера: 2016-04-14
Завтра: 2016-04-16
Время между завтра и вчера: 2 дня, 0:00:00
```

5.7. Преобразование временной метки (timestamp) в объект datetime

Модуль `datetime` может преобразовывать временную метку POSIX (timestamp) в объект `ITC datetime`. Эпохой (epoch, т. н. “эпоха Unix”) является полночь 1 января 1970 года.

```
import time
from datetime import datetime
seconds_since_epoch=time.time() #1469182681.709

utc_date=datetime.utcfromtimestamp(seconds_since_epoch) #datetime.datetime(2016, 7, 22, 10, 18, 1, 709000)
```

5.8. Точное вычитание месяцев из даты

Используем модуль `calendar`:

```
import calendar
from datetime import date

def monthdelta(date, delta):
    m, y = (date.month+delta) % 12, date.year + ((date.month)+delta-1) // 12
    if not m: m = 12
    d = min(date.day, calendar.monthrange(y, m)[1])
    return date.replace(day=d,month=m, year=y)

next_month = monthdelta(date.today(), 1) #datetime.date(2016, 10, 23)
```

Использование модуля `dateutils`:

```
import datetime
import dateutil.relativedelta

d = datetime.datetime.strptime("2013-03-31", "%Y-%m-%d")
d2 = d - dateutil.relativedelta.relativedelta(months=1) #datetime.datetime(2013, 2, 28, 0, 0)
```

5.9. Разбор (парсинг) произвольной временной метки ISO 8601 с минимальным использованием библиотек

Python имеет лишь ограниченную поддержку разбора временных меток ISO 8601. Для работы с `strptime` необходимо точно знать, какой формат используется. В качестве усложнения можно привести строковое преобразование времени даты в метку времени ISO 8601 с пробелом в качестве разделителя и 6-значной дробью:

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 555555))
# '2016-07-22 09:25:59.555555'
```

но если дробная часть равна 0, то она не выводится

```
str(datetime.datetime(2016, 7, 22, 9, 25, 59, 0))
# '2016-07-22 09:25:59'
```

Но эти две формы требуют *разного* формата для `strptime`. Более того, `strptime` вообще не поддерживает разбор часовых поясов, где часы и минуты разделяются двоеточием, поэтому `2016-07-22 09:25:59+0300` может быть проанализирован (разобран), а стандартный формат `2016-07-22 09:25:59+03:00` – нет. Существует библиотека из одного файла `iso8601`, которая правильно анализирует временные метки ISO 8601 и только их. Она поддерживает дроби, часовые пояса и разделитель `T`, и все это с помощью одной функции:

```
import iso8601
iso8601.parse_date('2016-07-22 09:25:59')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

```
iso8601.parse_date('2016-07-22 09:25:59+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<FixedOffset '+03:00' ...>)
iso8601.parse_date('2016-07-22 09:25:59Z')
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
iso8601.parse_date('2016-07-22T09:25:59.000111+03:00')
# datetime.datetime(2016, 7, 22, 9, 25, 59, 111, tzinfo=<FixedOffset '+03:00' ...>)
```

Если часовой пояс не задан, то `iso8601.parse_date` по умолчанию принимает значение UTC. Зона по умолчанию может быть изменена с помощью ключевого аргумента `default_zone`. Примечательно, что если вместо значения по умолчанию указано `None`, то те значения даты и времени, у которых нет определенного значения часового пояса, устанавливаются как простые значения даты и времени:

```
iso8601.parse_date('2016-07-22T09:25:59', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59)
iso8601.parse_date('2016-07-22T09:25:59Z', default_timezone=None)
# datetime.datetime(2016, 7, 22, 9, 25, 59, tzinfo=<iso8601.Utc>)
```

5.10. Получение временной метки ISO 8601

Без часового пояса, с микросекундами

```
from datetime import datetime
```

```
datetime.now().isoformat()
# Вывод: '2016-07-31T23:08:20.886783'
```

С часовым поясом, с микросекундами

```
from datetime import datetime
from dateutil.tz import tzlocal
```

```
datetime.now(tzlocal()).isoformat()
# Вывод: '2016-07-31T23:09:43.535074-07:00'
```

С часовым поясом, без микросекунд

```
from datetime import datetime
from dateutil.tz import tzlocal
```

```
datetime.now(tzlocal()).replace(microsecond=0).isoformat()
# Вывод: '2016-07-31T23:10:30-07:00'
```

5.11. Разбор строки с коротким именем часового пояса в объект `datetime` с учетом часового пояса

Используя библиотеку `dateutil` можно разбирать временные метки с особым коротким буквенным названием часового пояса.

Для дат с короткими названиями зон или аббревиатурами, которые, как правило, неоднозначны (например, CST – может быть Центральным стандартным временем, Стандартным временем в Китае, Стандартным временем Кубы и т. д.) или недоступны в стандартной базе данных, необходимо задать соответствие между аббревиатурой часового пояса и объектом `tzinfo`.

```
from dateutil import tz
from dateutil.parser import parse
```

```

ET = tz.gettz('US/Eastern')
CT = tz.gettz('US/Central')
MT = tz.gettz('US/Mountain')
PT = tz.gettz('US/Pacific')

us_tzinfos = {'CST': CT, 'CDT': CT,
             'EST': ET, 'EDT': ET,
             'MST': MT, 'MDT': MT,
             'PST': PT, 'PDT': PT}

dt_est = parse('2014-01-02 04:00:00 EST', tzinfos=us_tzinfos)
dt_pst = parse('2016-03-11 16:00:00 PST', tzinfos=us_tzinfos)

```

После выполнения этого:

```

dt_est
# datetime.datetime(2014, 1, 2, 4, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Eastern'))
dt_pst
# datetime.datetime(2016, 3, 11, 16, 0, tzinfo=tzfile('/usr/share/zoneinfo/US/Pacific'))

```

Следует отметить, что при использовании в этом методе часового пояса `pytz` локализация будет неправильной:

```

from dateutil.parser import parse
import pytz

```

```

EST = pytz.timezone('America/New_York')
dt = parse('2014-02-03 09:17:00 EST', tzinfos={'EST': EST})

```

Таким способом можно присоединить временную зону `pytz` к `datetime`:

```

dt.tzinfo # Будет в местном среднем времени
# <DstTzInfo 'America/New_York' LMT-1 день, 19:04:00 STD>

```

При использовании этого метода, вероятно, следует повторно использовать `localize` для простых значений даты и времени::

```

dt_fixed = dt.tzinfo.localize(dt.replace(tzinfo=None))
dt_fixed.tzinfo # Теперь это в EST.
# <DstTzInfo 'America/New_York' EST-1 день, 19:00:00 STD>

```

5.12. Нечеткий синтаксический анализ времени (извлечение даты и времени из текста)

Извлечь дату из текста можно с помощью `dateutil.parser` в “нечетком” (fuzzy) режиме, в котором компоненты строки, не распознанные как часть даты, игнорируются.

```

from dateutil.parser import parse

```

```

dt = parse("Сегодня 1 января 2047 года в 8:21:00AM", fuzzy=True)
print(dt)

```

Теперь `dt` – это объект `datetime`, и на экране появится `datetime.datetime(2047, 1, 1, 8, 21)`.

5.13. Итерация по датам

Иногда требуется выполнить итерацию по диапазону дат от начальной даты до конечной. Для этого можно использовать библиотеку `datetime` и объект `timedelta`:

```

import datetime

```

```

# Размер каждого шага в днях
day_delta = datetime.timedelta(days=1)

```

```
start_date = datetime.date.today()
end_date = start_date + 7*day_delta

for i in range((end_date - start_date).days):
    print(start_date + i*day_delta)
    В результате чего образуются:
2016-07-21
2016-07-22
2016-07-23
2016-07-24
2016-07-25
2016-07-26
2016-07-27
```

Глава 6. Форматирование даты

6.1. Время между двумя датами

```
from datetime import datetime
a = datetime(2016,10,06,0,0,0)
b = datetime(2016,10,01,23,59,59)
a-b
# datetime.timedelta(4, 1)
(a-b).дни
# 4
(a-b).total_seconds()
# 518399.0
```

6.2. Вывод объекта datetime в строку

Используются стандартные коды формата C:

```
from datetime import datetime
datetime_for_string = datetime(2016,10,1,0,0)
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_for_string,datetime_string_format)
# Oct 01 2016, 00:00:00
```

6.3. Парсинг строки в объект datetime

Используются стандартные коды формата C:

```
from datetime import datetime
datetime_string = 'Oct 1 2016, 00:00:00'
datetime_string_format = '%b %d %Y, %H:%M:%S'
datetime.strptime(datetime_string, datetime_string_format)
# datetime.datetime(2016, 10, 1, 0, 0)
```

Глава 7. Модуль Enum

7.1. Создание перечисления (Python 2.4–3.3)

Перечисления были перенесены с Python 3.4 на Python 2.4 и Python 3.3. Вы можете получить бэкпорт enum34 из PyPI.

```
pip install enum34
```

Создание перечисления идентично тому, как это происходит в Python 3.4+

```
from enum import Enum
```

```
class Color(Enum):
```

```
    red = 1
```

```
    green = 2
```

```
    blue = 3
```

```
print(Color.red) # Color.red
```

```
print(Color(1)) # Color.red
```

```
print(Color['red']) # Color.red
```

7.2. Итерация

Перечисления являются итерируемыми:

```
class Color(Enum):
```

```
    red = 1
```

```
    green = 2
```

```
    blue = 3
```

```
[c for c in Color] # [<Color.red: 1>, <Color.green: 2>, <Color.blue: 3>]
```

Глава 8. Множества

8.1. Операции над множествами

с другими множествами

```
# Пересечение
```

```
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6}) # {3, 4, 5}
```

```
{1, 2, 3, 4, 5} & {3, 4, 5, 6} # {3, 4, 5}
```

```
# Объединение
```

```
{1, 2, 3, 4, 5}.union({3, 4, 5, 6}) # {1, 2, 3, 4, 5, 6}
```

```
{1, 2, 3, 4, 5} | {3, 4, 5, 6} # {1, 2, 3, 4, 5, 6}
```

```
# Разница
```

```
{1, 2, 3, 4}.difference({2, 3, 5}) # {1, 4}
```

```
{1, 2, 3, 4} - {2, 3, 5} # {1, 4}
```

```
# Симметричная разность
```

```
{1, 2, 3, 4}.symmetric_difference({2, 3, 5}) # {1, 4, 5}
```

```
{1, 2, 3, 4} ^ {2, 3, 5} # {1, 4, 5}
```

```
# Проверка суперпозиции (надмножества)
{1, 2}.issuperset({1, 2, 3})      # False
{1, 2} >= {1, 2, 3}              # False

# Проверка подмножества
{1, 2}.issubset({1, 2, 3})       # True
{1, 2} <= {1, 2, 3}             # True

# Проверка непересечения
{1, 2}.isdisjoint({3, 4})        # True
{1, 2}.isdisjoint({1, 4})        # False
```

с отдельными элементами

```
# проверка наличия элемента в множестве
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Добавление и удаление из множества
s = {1,2,3}
s.add(4)          # s == {1,2,3,4}

s.discard(3)      # s == {1,2,4}
s.discard(5)      # s == {1,2,4}

s.remove(2)       # s == {1,4}
s.remove(2)       # Вывод: KeyError
```

Операции над множествами возвращают новые множества, но имеют соответствующие комбинированные операторы присваивания:

Метод	Операция на месте	Метод на месте
union (объединение)	$s \mid= t$	update
intersection (пересечение)	$s \&= t$	intersection_update
(difference) разница	$s -= t$	difference_update
symmetric_difference (симметрическая разница)	$s \wedge= t$	symmetric_difference_update

Например:

```
s = {1, 2}
s.update({3, 4}) # s == {1, 2, 3, 4}
```

8.2. Получение уникальных элементов списка

Допустим, у вас есть список ресторанов – возможно, вы читаете его из файла. Вас интересуют уникальные рестораны в этом списке. Лучший способ получить уникальные элементы из списка – превратить его в множество:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
unique_restaurants = set(restaurants)
print(unique_restaurants)
# выводит {'Chicken Chicken', 'McDonald's', 'Burger King'}
```

Обратите внимание на то, что набор расположен не в том же порядке, что и исходный список; это связано с тем, что наборы (sets) неупорядочены, как и dicts.

Этот список можно легко преобразовать обратно в List с помощью встроенной в Python функции list, получив другой список, который является тем же списком, что и исходный, но без дубликатов:

```
list(unique_restaurants)
# ['Chicken Chicken', 'McDonald's', 'Burger King']
```

Также часто встречается вариант с одной строкой:

```
# Удаляет все дубликаты и возвращает другой список
list(set(restaurants))
```

Теперь все операции, которые можно было выполнить над исходным списком, могут быть произведены снова.

8.3. Множество множеств

```
{{1,2}, {3,4}}
```

приводит к ошибке:

```
TypeError: unhashable type: 'set'
```

Вместо этого следует использовать frozenset:

```
{frozenset({1, 2}), frozenset({3, 4})}
```

8.4. Операции над множествами с использованием методов и встроенных модулей

Определим два множества (набора) a и b

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

Примечание: {1} создает множество из одного элемента, а {} создает пустой dict. Правильным способом создания пустого множества является set().

Пересечение

a.intersection(b) возвращает новое множество, элементы которого присутствуют как в a, так и в b

```
>>> a.intersection(b)
{3, 4}
```

Объединение

a.union(b) возвращает новый набор с элементами, присутствующими в любом a и b

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Разница

a.difference(b) возвращает новый набор с элементами, присутствующими в a, но не в b

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Симметричная разница

`a.symmetric_difference(b)` возвращает новый набор с элементами, присутствующими в любом `a` или `b`, но не в обоих множествах

```
>>> a.symmetric_difference(b)
{1, 2, 5}
>>> b.symmetric_difference(a)
{1, 2, 5}
```

Примечание: `a.symmetric_difference(b) == b.symmetric_difference(a)`

Подмножество и надмножество

`c.issubset(a)` проверяет, соответствует ли каждый элемент `c` в `a`

`a.issuperset(c)` проверяет, является ли каждый элемент `c` находится в `a`.

```
>>> c = {1, 2}
>>> c.issubset(a)
True
```

```
>>> a.issuperset(c)
True
```

Последние операции имеют эквивалентные операторы, как показано ниже:

Метод	Оператор
<code>a.intersection(b)</code>	<code>a & b</code>
<code>a.union(b)</code>	<code>a b</code>
<code>a.difference(b)</code>	<code>a - b</code>
<code>a.symmetric_difference(b)</code>	<code>a ^ b</code>
<code>a.issubset(b)</code>	<code>a <= b</code>
<code>a.issuperset(b)</code>	<code>a >= b</code>

Множества `a` и `d` не пересекаются, если ни один элемент `a` не находится также в `d`, и наоборот.

```
>>> d = {5, 6}
>>> a.isdisjoint(b)      # {2, 3, 4} в обоих множествах
False
>>> a.isdisjoint(d)
True
```

эквивалентная проверка, но менее эффективная

```
>>> len(a & d) == 0
True
```

еще менее эффективная

```
>>> a & d == set()
True
```

Тестирование вхождения

Встроенное ключевое слово `in` ищет вхождения:

```
>>> 1 in a
True
>>> 6 in a
False
```

длина

Встроенная функция len() возвращает количество элементов в множестве

```
>>> len(a)
4
>>> len(b)
3
```

8.5. Множества и мультимножества

Множества (наборы) – это неупорядоченные коллекции отдельных элементов. Но иногда мы хотим работать с неупорядоченными коллекциями элементов, которые не обязательно различны и отслеживают множественность элементов.

Рассмотрим пример:

```
setA = {'a','b','b','c'}
setA
set(['a', 'c', 'b'])
```

При сохранении строк 'a', 'b', 'b', 'c' в структуру данных набора (set) мы потеряли информацию о том, что 'b' встречается дважды. Конечно, сохранение элементов в списке (list) сохранит эту информацию.

```
listA = ['a','b','b','c']
listA
['a', 'b', 'b', 'c']
```

Но структура данных списка вводит дополнительный ненужный порядок, который замедляет наши вычисления.

Для реализации мультимножеств Python предоставляет класс Counter из модуля collections (начиная с версии 2.7):

Версия Python 2.x ≥ 2.7:

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
Counter({'b': 2, 'a': 1, 'c': 1})
```

Counter представляет собой словарь, где элементы хранятся в виде ключей словаря и их счетчики хранятся в виде значений словаря. И, как и все словари, он является неупорядоченной коллекцией.

Глава 9. Простые математические операторы

Числовые типы и их метаклассы

Модуль numbers содержит абстрактные метаклассы для числовых типов:

subclasses	numbers.Number	numbers.Integral	numbers.Rational	numbers.Real	numbers.Complex
bool	+	+	+	+	+
int	+	+	+	+	+
fractions.Fraction	+	—	+	+	+
float	+	—	—	+	+
complex	+	—	—	—	+
decimal.Decimal	+	—	—	—	—

В Python реализованы общие математические операции, включая целочисленное деление, деление с плавающей точкой, умножение, возведение в степень, сложение и вычитание. Математический модуль (включен во все стандартные версии Python) содержит расширенную функциональность – тригонометрические функции, извлечение корня, логарифмы и многое другое.

9.1. Деление

Python выполняет целочисленное деление, когда оба операнда являются целыми числами. Поведение операторов деления Python отличается в различных версиях

a, b, c, d, e = 3, 2, 2.0, -3, 10

В Python 2 результат использования оператора “/” зависит от типа числителя и знаменателя.

```
a / b    # = 1
a / c    # = 1,5
d / b    # = -2
b / a    # = 0
d / e    # = -1
```

Обратите внимание, что поскольку a и b являются int (целочисленными), то и результат является int. Результат всегда округляется до нуля. Поскольку c является float (числом с плавающей точкой), результатом a / c является float. Вы также можете использовать модуль оператора.

```
import operator          # Модуль оператора предоставляет арифметические функции с двумя
                          # аргументами
operator.div(a, b)        # = 1
operator.__div__(a, b)    # = 1
```

Что делать, если вы хотите осуществить деление чисел с плавающей точкой:
Рекомендуемый способ:

```
from __future__ import division    # применяет к модулю деление в стиле Python 3
a / b          # = 1.5
a // b         # = 1
```

Если вы не хотите применять ко всему модулю:

```
a / (b * 1.0)    # = 1.5
1.0 * a / b      # = 1.5
a / b * 1.0      # = 1.0 (будьте осторожны с последовательностью операций)
```

```
from operator import truediv
truediv(a, b)     # = 1.5
```

Нерекомендуемый способ (может вызывать TypeEggor, например, если аргумент – комплексное число):

```
float(a) / b      # = 1.5
a / float(b)      # = 1.5
```

Версия Python 2.x ≥ 2.2:

Оператор “//” в Python 2 вызывает деление с округлением к меньшему.

```
a // b           # = 1
a // c           # = 1.0
```

В Python 3 оператор “/” выполняет “истинное” деление независимо от типа аргументов. Оператор “//” выполняет деление с округлением к меньшему и сохраняет тип.

```
a / b      # = 1.5
e / b      # = 5.0
a // b     # = 1
a // c     # = 1.0
```

```
import operator      # модуль operator предоставляет 2-аргументные арифметические функции
operator.truediv(a, b) # = 1.5
operator.floordiv(a, b) # = 1
operator.floordiv(a, c) # = 1.0
```

Возможные комбинации (для встроенных типов):

- int и int (дает int в Python 2 и float в Python 3)
- int и float (дает float)
- int и complex (дает complex)
- float и float (дает float)
- float и complex (дает complex)
- complex и complex (дает complex)

См. PEP 238 для получения дополнительной информации.

9.2. Сложение

```
a, b = 1, 2
# Используется оператор "+":
a + b      # = 3
```

Используется оператор "+" для сложения и присваивания:

```
a += b      # a = 3 (эквивалентно a = a + b)
```

```
import operator      # поддерживает арифметические функции с двумя аргументами
```

```
operator.add(a, b)    # = 5 так как a присвоено значение 3 перед этой строкой
```

Оператор "+" эквивалентен:

```
a = operator.iadd(a, b) # a = 5 так как a присвоено значение 3 перед этой строкой
```

Возможные комбинации (встроенные типы):

- int и int (дает int)
- int и float (дает float)
- int и complex (дает complex)
- float и float (дает float)
- float и complex (дает complex)
- complex и complex (дает complex)

Примечание: оператор "+" также используется для конкатенации строк, списков и кортежей:

```
"first string" + "second string"      # 'first string second string'
```

```
[1, 2, 3] + [4, 5, 6]                  # = [1, 2, 3, 4, 5, 6]
```

9.3. Возведение в степень

```
a, b = 2, 3
```

```
(a ** b)      # = 8
pow(a, b)     # = 8
```

```
import math
math.pow(a, b) # = 8.0 (всегда число с плавающей точкой; комплексные числа не разрешены)
```

```
import operator
operator.pow(a, b)      # = 8
```

Еще одно различие между встроенной функцией `pow` и `math.pow` – то, что встроенная `pow` может принимать три аргумента:

```
a, b, c = 2, 3, 2
pow(2, 3, 2)          # 0, вычисляет (2 ** 3) % 2, но, по утверждению документации Python,
                      # делает это более эффективно
```

Специальные функции

Функция `math.sqrt(x)` вычисляет квадратный корень из x .

```
import math
import cmath
c = 4
math.sqrt(c)          # = 2.0 (всегда число с плавающей точкой; комплексные числа
                      # не разрешены)
cmath.sqrt(c)         # = (2+0j) (всегда комплексное число)
```

Для вычисления других корней, например кубического корня, необходимо возвести число в степень, обратную степени корня. Это можно сделать с помощью любой экспоненциальной функции или оператора.

```
import math
x = 8
math.pow(x, 1/3)       # оценивается как 2.0
x**(1/3)               # оценивается как 2.0
```

Функция `math.exp(x)` вычисляет $e ** x$.

```
math.exp(0) # 1.0
math.exp(1) # 2.718281828459045 (e)
```

Функция `math.expm1(x)` вычисляет $e ** x - 1$. При малых значениях x это дает существенно лучшую точность, чем `math.exp(x) - 1`.

```
math.expm1(0)          # 0.0

math.exp(1e-6) - 1     # 1.0000004999621837e-06
math.expm1(1e-6)       # 1.0000005000001665e-06
# точный результат     # 1.000000500000166666708333341666...
```

9.4. Тригонометрические функции

$a, b = 1, 2$

```
import math
```

```
math.sin(a) # возвращает синус 'a' в радианах
# Вывод: 0.8414709848078965
```

```
math.cosh(b) # возвращает обратный гиперболический косинус от 'b' в радианах
# Вывод: 3.7621956910836314
```

```
math.atan(math.pi) # возвращает арктангенс от числа  $\pi$  в радианах
# Вывод: 1.2626272556789115
```

```
math.hypot(a, b) # возвращает евклидову норму, аналогично math.sqrt(a*a + b*b)
# Вывод: 2.23606797749979
```

Заметим, что `math.hypot(x, y)` – это также длина вектора (или евклидово расстояние) от начала координат (0, 0) в точку (x, y).

Для вычисления евклидова расстояния между двумя точками (x1, y1) и (x2, y2) можно воспользоваться функцией `math.hypot` следующим образом:

```
math.hypot(x2-x1, y2-y1)
```

Для преобразования радиан в градусы и градусов в радианы соответственно используются `math.degrees` и `math.radians`.

```
math.degrees(a)
```

```
# Вывод: 57.29577951308232
```

```
math.radians(57.29577951308232)
```

```
# Вывод: 1.0
```

9.5. Операторы присваивания (“операции на месте”)

В приложениях часто возникает необходимость в подобном коде:

```
a = a + 1
```

или

```
a = a * 2
```

Существует эффективное сокращение для этих операций:

```
a += 1
```

```
# и
```

```
a *= 2
```

Перед символом `=` может быть использован любой математический оператор для выполнения операции присваивания:

- `--` декремент переменной на месте
- `+=` инкремент (приращение) переменной на месте
- `*=` умножение переменной на месте
- `/=` деление переменной на месте
- `//=` деление с округлением к меньшему на месте (в Python 3)
- `%=` возвращает остаток от деления переменной на месте
- `**=` возведение переменной на месте

Для побитовых операторов существуют другие операторы на месте (`^`, `|` и т. д.).

9.6. Вычитание

```
a, b = 1, 2
```

```
# Использование оператора "-":
```

```
b - a          # = 1
```

```
import operator
```

```
operator.sub(b, a)
```

```
# содержит арифметические функции с 2 аргументами
```

```
# = 1
```

Возможные комбинации (для встроенных типов):

- `int` и `int` (дает `int`)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

9.7. Умножение

`a, b = 2, 3`

```
a * b          # = 6
import operator
operator.mul(a, b) # = 6
```

Возможные комбинации (для встроенных типов):

- `int` и `int` (дает `int`)
- `int` и `float` (дает `float`)
- `int` и `complex` (дает `complex`)
- `float` и `float` (дает `float`)
- `float` и `complex` (дает `complex`)
- `complex` и `complex` (дает `complex`)

Примечание: оператор `*` также используется для повторной конкатенации строк, списков и кортежей:

```
3 * 'ab'          # = 'ababab'
3 * ('a', 'b')    # = ('a', 'b', 'a', 'b', 'a', 'b')
```

9.8. Логарифмы

По умолчанию функция `math.log` вычисляет логарифм числа с основанием `e`. При желании можно указать основание в качестве второго аргумента.

```
import math
import cmath
```

```
math.log(5)          # = 1.6094379124341003
# Необязательный аргумент основания. По умолчанию is math.e
math.log(5, math.e)  # = 1.6094379124341003
cmath.log(5)         # = (1.6094379124341003+0j)
math.log(1000, 10)   # 3.0 (всегда возвращает float)
cmath.log(1000, 10)  # (3+0j)
```

Для различных оснований существуют специальные варианты функции `math.log`.

```
# Логарифм по основанию e - 1 (более высокая точность для малых значений)
math.log1p(5)      # = 1.791759469228055
```

```
# Логарифм по основанию 2
math.log2(8)       # = 3.0
```

```
# Логарифм по основанию 10
math.log10(100)    # = 2.0
cmath.log10(100)   # = (2+0j)
```

9.9. Остаток от деления

Как и во многих других языках, в Python для вычисления остатка от деления используется оператор `%`.

```
3 % 4   # 3
10 % 2  # 0
6 % 4   # 2
```

Также можно использовать оператор module:

```
import operator
operator.mod(3, 4)      # 3
operator.mod(10, 2)     # 0
operator.mod(6, 4)      # 2
```

Можно проводить вычисления и с отрицательными числами.

```
-9 % 7  # 5
9 % -7  # -5
-9 % -7 # -2
```

Если необходимо вывести результат целочисленного деления и остаток, для сокращения можно использовать функцию divmod:

```
quotient, remainder = divmod(9, 4)
# частное = 2, остаток = 1, так как 4 * 2 + 1 = 9
```

Глава 10. Побитовые операторы

Побитовые операции изменяют двоичные строки на уровне битов. Эти операции поддерживаются непосредственно процессором. Эти немногочисленные операции необходимы при работе с драйверами устройств, низкоуровневой графикой, криптографией и сетевыми коммуникациями. В данном разделе представлены полезные знания и примеры побитовых операторов Python.

10.1. Побитовое НЕ

Оператор ~ “переворачивает” все биты в числе. Поскольку компьютеры используют знаковое представление чисел – в частности, дополнительный код (“дополнение двойки”, англ. two’s complement) для кодирования отрицательных двоичных чисел, где отрицательные числа записываются с ведущей единицей (1) вместо ведущего нуля (0).

Это означает, что при использовании 8 бит для представления чисел с дополнительным кодом (двухкомпонентных чисел) используются шаблоны от 0000 0000 до 0111 1111 для представления чисел от 0 до 127 и резерв 1xxx xxxx для представления отрицательных чисел.

Восьмибитные двухкомпонентные числа

Биты	Дополнение	Двухкомпонентное значение
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

По сути это означает, что если 1010 0110 имеет дополнение 166 (полученное сложением $(128 * 1) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 0) + (4 * 1) + (2 * 1) + (1 * 0)$), то его двухкомпонентное значение -90 (получено путем сложения $(128 * 1) - (64 * 0) - (32 * 1) - (16 * 0) - (8 * 0) - (4 * 1) - (2 * 1) - (1 * 0)$ и дополнения полученного значения). Таким образом, диапазон отрицательных чисел простирается до -128 (1000 0000). Ноль (0) представляется как 0000 0000, а минус один (-1) - как 1111 1111. В общем случае это означает, что $\sim n = -n - 1$.

```
# 0 = 0b0000 0000
~0
# Вывод: -1
# -1 = 0b1111 1111
```

```
# 1 = 0b0000 0001
~1
# Вывод: -2
# -2 = 1111 1110
```

```
# 2 = 0b0000 0010
~2
# Вывод: -3
# -3 = 0b1111 1101
```

```
# 123 = 0b0111 1011
~123
# Вывод: -124
# -124 = 0b1000 0100
```

Заметим, что общий результат этой операции при применении к положительным числам можно объединить:

```
~n -> -|n+1|
```

И тогда в применении к отрицательным числам получается:

```
~n -> |n-1|
```

Это последнее правило иллюстрируется следующими примерами:

```
# -0 = 0b0000 0000
~-0
# Вывод: -1
# -1 = 0b1111 1111
```

0 является очевидным исключением из этого правила, так как $-0 == 0$ всегда

```
# -1 = 0b1000 0001
~-1
# Вывод: 0
# 0 = 0b0000 0000
```

```
# -2 = 0b1111 1110
~-2
# Вывод: 1
# 1 = 0b0000 0001
```

```
# -123 = 0b1111 1011
~-123
# Вывод: 122
# 122 = 0b0111 1010
```

10.2. Побитовое XOR (исключающее ИЛИ)

Оператор \wedge выполняет двоичное **XOR**, в котором двоичная 1 копируется тогда и только тогда, когда она является значением **одного** операнда. По-другому это можно выразить так: результат равен 1 только в том случае, если операнды различны. Примеры:

```
# 0 ^ 0 = 0
```

```
# 0 ^ 1 = 1
```

```
# 1 ^ 0 = 1
```

```
# 1 ^ 1 = 0
```

```
# 60 = 0b111100
```

```
# 30 = 0b011110
```

```
60 ^ 30
```

```
# Вывод: 34
```

```
# 34 = 0b100010
```

```
bin(60 ^ 30)
```

```
# Вывод: 0b100010
```

10.3. Побитовое И

Оператор $\&$ выполняет двоичное **И**, при котором бит копируется, если он есть в **обоих** операндах. Это означает:

```
# 0 & 0 = 0
```

```
# 0 & 1 = 0
```

```
# 1 & 0 = 0
```

```
# 1 & 1 = 1
```

```
# 60 = 0b111100
```

```
# 30 = 0b011110
```

```
60 & 30
```

```
# Вывод: 28
```

```
# 28 = 0b11100
```

```
bin(60 & 30)
```

```
# Вывод: 0b11100
```

10.4. Побитовое ИЛИ

Оператор $|$ выполняет двоичное **ИЛИ**, при котором бит копируется, если он есть в **любом** из операндов. Это означает:

```
# 0 | 0 = 0
```

```
# 0 | 1 = 1
```

```
# 1 | 0 = 1
```

```
# 1 | 1 = 1
```

```
# 60 = 0b111100
```

```
# 30 = 0b011110
```

```
60 | 30
```

```
# Вывод: 62
```

```
# 62 = 0b111110
```

```
bin(60 | 30)
```

```
# Вывод: 0b111110
```

10.5. Побитовый сдвиг влево

Оператор `<<` выполняет побитовый сдвиг влево, при котором значение левого операнда сдвигается влево на количество разрядов, заданное правым операндом.

```
# 2 = 0b10
2 << 2
# Вывод: 8
# 8 = 0b1000
```

```
bin(2 << 2)
# Вывод: 0b1000
```

Выполнение сдвига левого бита на 1 эквивалентно умножению на 2:

```
7 << 1
# Вывод: 14
```

Выполнение сдвига влево на n бит эквивалентно умножению на $2^{**}n$:

```
3 << 4
# Вывод: 48
```

10.6. Побитовый сдвиг вправо

Оператор `>>` выполняет побитовый “сдвиг вправо”, при котором значение левого операнда сдвигается вправо на количество битов, заданное правым операндом.

```
# 8 = 0b1000
8 >> 2
# Вывод: 2
# 2 = 0b10
bin(8 >> 2)
# Вывод: 0b10
```

Выполнение сдвига правого бита на 1 эквивалентно целочисленному делению на 2:

```
36 >> 1
# Вывод: 18
15 >> 1
# Вывод: 7
```

Выполнение правого битового сдвига на n эквивалентно целочисленному делению на $2^{**}n$:

```
48 >> 4
# Вывод: 3
59 >> 3
# Вывод: 7
```

10.7. Операторы присваивания (“операции на месте”)

Все битовые операторы (кроме `~`) имеют свои собственные варианты присваивания:

```
a = 0b001
a &= 0b010
# a = 0b000
a = 0b001
a |= 0b010
# a = 0b011
a = 0b001
a <<= 2
```

```
# a = 0b100
a = 0b100
a >>= 2
# a = 0b001
a = 0b101
a ^= 0b011
# a = 0b110
```

Глава 11. Логические операции

11.1. “and” и “or” не гарантируют возврата булевого значения

При использовании `or` либо возвращается первое значение выражения, если оно истинно, либо вслепую возвращается второе значение. Т. е. `or` эквивалентно:

```
def or_(a, b):
    if a:
        return a
    else:
        return b
```

Для `and` возвращается его первое значение, если оно ложно, в противном случае возвращается последнее значение:

```
def and_(a, b):
    if not a:
        return a
    else:
        return b
```

11.2. Простой пример

В Python для сравнения одного элемента можно использовать два бинарных оператора – по одному с каждой стороны:

```
if 3.14 < x < 3.142:
    print("x примерно равно pi")
```

Во многих (большинстве?) языках программирования эта оценка будет противоречить обычной математике: $(3,14 < x) < 3,142$, но в Python она рассматривается как $3,14 < x$ and $x < 3,142$, как и ожидало бы большинство непрограммистов.

11.3. Вычисления по короткой схеме (“короткое замыкание”)

Python упрощает вычисление логических операций.

```
>>> def true_func():
...     print("true_func()")
...     return True
...
>>> def false_func():
...     print("false_func()")
...     return False
...
>>> true_func() or false_func()
true_func()
True
>>> false_func() or true_func()
```

```

false_func()
true_func()
True
>>> true_func() and false_func()
true_func()
false_func()
False
>>> false_func() and false_func()
false_func()
False

```

11.4. Оператор "and" ("и")

Оценивает второй аргумент тогда и только тогда, когда оба аргумента истинны. В противном случае результатом вычисления является первый ложный аргумент.

```

x = True
y = True
z = x and y      # z = True

```

```

x = True
y = False
z = x and y      # z = False

```

```

x = False
y = True
z = x and y      # z = False

```

```

x = False
y = False
z = x and y      # z = False

```

```

x = 1
y = 1
z = x and y      # z = y, поэтому z = 1, `and` и `or` не являются гарантированно операторами
                  # булевой логики

```

```

x = 0
y = 1
z = x and y      # z = x, поэтому z = 0 (см. выше)

```

```

x = 1
y = 0
z = x and y      # z = y, поэтому z = 0 (см. выше)

```

```

x = 0
y = 0
z = x and y      # z = x, поэтому z = 0 (см. выше)

```

В приведенном примере единица может быть заменена на любое истинное значение, а 0 – на любое ложное.

11.5. Оператор "or" ("или")

Оценивает первый истинный аргумент, если любой из аргументов истинный. Если оба аргумента ложные, то вычисляется второй аргумент.

```

x = True
y = True
z = x or y       # z = True

```



```
x = True
y = False
z = x or y      # z = True

x = False
y = True
z = x or y      # z = True

x = False
y = False
z = x or y      # z = False

x = 1
y = 1
z = x or y      # z = x, поэтому z = 1, `and` и `or` не являются гарантированно операторами
                  # Булевой логики

x = 1
y = 0
z = x or y      # z = x, поэтому z = 1 (см. выше)

x = 0
y = 1
z = x or y      # z = y, поэтому z = 1 (см. выше))

x = 0
y = 0
z = x or y      # z = y, поэтому z = 0 (см. выше)
```

В приведенном примере единица может быть заменена на любое истинное значение, а 0 – на любое ложное.

11.6. Оператор “not” (“не”)

Возвращает противоположное утверждение:

```
x = True
y = not x       # y = False

x = False
y = not x       # y = True
```

Глава 12. Приоритет операторов

Операторы Python имеют установленный **порядок старшинства**, который определяет, какие операторы будут вычислены первыми в потенциально неоднозначном выражении. Например, в выражении $3 * 2 + 7$ сначала 3 умножается на 2, а затем полученный результат прибавляется к 7, в результате чего получается 13.

Ниже приведены список операторов по старшинству и краткое описание того, что они (обычно) делают.

12.1. Примеры приоритета операторов в Python

Python следует правилу **PEMDAS**, что расшифровывается как Parentheses, Exponents, Multiplication and Division, Addition and Subtraction (скобки, возведение в степень, умножение и деление, сложение и вычитание).

Пример:

```
>>> a, b, c, d = 2, 3, 5, 7
>>> a ** (b + c)    # скобки
256
>>> a * b ** c      # возведение в степень: то же, что `a * (b ** c)`
7776
>>> a + b * c / d    # умножение / деление: то же, что `a + (b * c / d)`
4.142857142857142
```

Дополнительно: математические правила действуют, но не всегда:

```
>>> 300 / 300 * 200
200.0
>>> 300 * 200 / 300
200.0
>>> 1e300 / 1e300 * 1e200
1e+200
>>> 1e300 * 1e200 / 1e300
inf
```

Глава 13. Область видимости и привязка переменных

13.1. Нелокальные переменные

Версия Python 3.x ≥ 3.0:

В Python 3 добавлено новое ключевое слово **nonlocal**. Ключевое слово `nonlocal` добавляет переопределение области видимости к внутренней области видимости. Об этом можно прочитать в PEP 3104. Лучше всего это проиллюстрировать на нескольких примерах кода. Один из наиболее распространенных примеров – функция, которая может создавать инкремент:

```
def counter():
    num = 0
    def incrementer():
        num += 1
    return num
return incrementer
```

Если попытаться выполнить этот код, то будет выдана ошибка **UnboundLocalError**, поскольку ссылка на переменную `num` дана до того, как она была присвоена во внутренней функции. Добавим сюда нелокальность:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
    return num
return incrementer
```

```
c = counter()
c()    # = 1
c()    # = 2
c()    # = 3
```

В принципе, ключевое слово `nonlocal` позволяет присваивать переменные во внешней области видимости, но не в глобальной. Поэтому вы не можете использовать `nonlocal` в нашей функции счетчика, так как в этом случае она попытается присвоить переменную глобальной области видимости. Попробуйте это сделать, и вы быстро получите ошибку `SyntaxError`. Поэтому необходимо использовать `nonlocal` во вложенной функции.

(Заметим, что представленную здесь функциональность лучше реализовать с помощью генераторов.)

13.2. Глобальные переменные

В Python переменные внутри функций считаются локальными тогда и только тогда, когда они встречаются в левой части выражения присваивания или в каком-либо другом случае привязки; в противном случае такая привязка ищется в окружающих функциях, вплоть до глобальной области видимости. Это справедливо даже в том случае, если оператор присваивания никогда не выполняется.

```
x = 'Hi'
```

```
def read_x():
    print(x)          # переменная x считается глобальной, на нее только ссылаются
```

```
read_x()             # выводит Hi
```

```
def read_y():
    print(y)          # переменная y считается глобальной, на нее только ссылаются
read_y()              # NameError: глобальное имя 'y' не определено
```

```
def read_y():
    y = 'Hey'         # переменная y фигурирует в присваивании, поэтому она локальная
    print(y)          # найдет локальную y
```

```
read_y()             # выводит Hey
```

```
def read_x_local_fail():
    if False:
        x = 'Hey'     # переменная x фигурирует в присваивании, поэтому она локальная
        print(x)      # будет искать локальную z, которая не присвоена и не будет найдена
```

```
read_x_local_fail()  # UnboundLocalError: на локальную переменную 'x' ссылаются до
                    # присваивания
```

Обычно присваивание внутри области видимости “затеняет” все внешние переменные с тем же именем:

```
x = 'Hi'
```

```
def change_local_x():
    x = 'Bye'
    print(x)
change_local_x() # выводит Bye
print(x)        # выводит Hi
```

Объявление имени глобальным (`global`) означает, что для остальной части области видимости все присваивания имени будут происходить на верхнем уровне модуля:

```
x = 'Hi'
```

```
def change_global_x():
    global x
```

```
x = 'Bye'
print(x)
```

```
change_global_x()      # выводит Bye
print(x)               # выводит Bye
```

Ключевое слово `global` означает, что присваивания будут происходить на верхнем уровне модуля, а не на верхнем уровне программы. Другим модулям по-прежнему будет необходим обычный точечный доступ к переменным внутри модуля.

Подведем итог: чтобы узнать, является ли переменная `x` локальной для функции, необходимо прочитать всю функцию.

1. Если вы нашли `global x`, то `x` является **глобальной** переменной.

2. Если вы нашли `nonlocal x`, то `x` принадлежит вещающей функции и не является ни локальным, ни глобальным.

3. Если вы нашли `x = 5`, или `for x in range(3)`, или какую-то другую привязку, то `x` является **локальной** переменной.

4. В противном случае `x` принадлежит некоторой охватывающей области (области функций, глобальной области или встроенных элементов).

13.3. Локальные переменные

Если имя связано внутри функции, то по умолчанию оно доступно только внутри этой функции:

```
def foo():
    a = 5
    print(a)      # ok

print(a)         # NameError: имя 'a' не определено
```

Управляющие конструкции потока не влияют на область видимости (за исключением `except`), но обращение к переменной, которая еще не была присвоена, является ошибкой:

```
def foo():
    if True:
        a = 5
    print(a)      # ok

b = 3
def bar():
    if False:
        b = 5
print(b)         # UnboundLocalError: локальная переменная 'b' упоминается до присвоения
```

Общими операциями связывания являются присваивания, циклы `for` и дополненные присваивания, такие как `a += 5`.

13.4. Команда “del”

Эта команда имеет несколько родственных, но различных форм.

```
del v
```

Если `v` – переменная, то команда `del v` удаляет переменную из области видимости. Например:

```
x = 5
print(x) # результат: 5
del x
print(x) # NameError: имя 'x' не определено
```

Заметим, что `del` является *обязательным вхождением*, а это означает, что если явно не указано иное (с помощью `nonlocal` или `global`), то `del v` сделает `v` локальным для текущей области видимости. Если вы собираетесь удалить `v` во внешней области видимости, используйте `nonlocal v` или `global v` в той же области видимости, что и оператор `del v`.

Во всех приведенных ниже случаях намерение команды является поведением по умолчанию, но не навязывается языком. Класс может быть написан таким образом, что это намерение не выполняется.

```
del v.name
```

Эта команда вызывает `v.__delattr__(name)`.

Это делается для того, чтобы сделать атрибут недоступным. Например:

```
class A:
    pass
```

```
a = A()
a.x = 7
print(a.x)          # результат: 7
del a.x
print(a.x)          # error: AttributeError: объект 'A' не имеет атрибута 'x'
del v[item]
```

Эта команда вызывает `v.__delitem__(item)`.

Имеется в виду, что `item` не будет принадлежать соответствию, реализуемому объектом `v`.

Например:

```
x = {'a': 1, 'b': 2}
del x['a']
print(x)            # результат: {'b': 2}
print(x['a'])        # error: KeyError: 'a'
del v[a:b]
```

Фактически это вызывает `v.__delslice__(a, b)`.

Замысел аналогичен описанному выше, но с использованием срезов (slices) – диапазонов элементов вместо одного элемента. Например:

```
x = [0, 1, 2, 3, 4]
del x[1:3]
print(x)            # результат: [0, 3, 4]
```

13.5. Функции пропускают область видимости класса при поиске имен

Классы имеют локальную область видимости при определении, но функции внутри класса не используют эту область при поиске имен. Поскольку лямбда-выражения являются функциями, а генераторы реализуются с использованием области видимости функции, это может привести к неожиданному поведению.

```
a = 'global'
```

```
class Fred:
    a = 'class'          # область видимости класса
    b = (a for i in range(10)) # область видимости функции
    c = [a for i in range(10)] # область видимости функции
    d = a                # область видимости класса
    e = lambda: a         # область видимости функции
    f = lambda a=a: a      # аргумент по умолчанию использует область видимости класса
```



```
print(foo) # выводит 2
```

глобальная переменная foo все еще существует, без изменений:

```
print(globals()['foo']) # выводит 1
print(locals()['foo'])  # выводит 2
```

Для изменения глобальной переменной используйте ключевое слово `global`:

```
foo = 1
```

```
def func():
    global foo
    foo = 2          # это изменяет глобальную переменную foo, а не создает локальную переменную
```

Область видимости определяется для всего тела функции!

Это означает, что переменная никогда не будет глобальной для половины функции и локальной для другой половины.

```
foo = 1
```

```
def func():
    # Эта функция имеет локальную переменную foo, поскольку она определена ниже.
    # Таким образом, с этого момента foo является локальной. Глобальная foo скрыта.

    print(foo) # вызывает ошибку UnboundLocalError, ведь локальная foo еще не инициализирована
    foo = 7
    print(foo)
```

Аналогично, наоборот:

```
foo = 1
```

```
def func():
    # В этой функции foo с самого начала является глобальной переменной.
```

```
foo = 7 # глобальная foo изменена
```

```
print(foo)          # 7
print(globals()['foo']) # 7
```

```
global foo          # это может быть в любом месте в функции
print(foo)          # 7
```

Функции внутри функций

Может существовать много уровней функций, вложенных в функции, но в рамках одной функции существует только одна локальная область видимости для этой функции и глобальная область видимости. Промежуточных областей видимости не существует.

```
foo = 1
```

```
def f1():
    bar = 1
```

```
def f2():
    baz = 2
    # здесь foo – глобальная переменная, baz – локальная переменная
    # bar не входит ни в одну из областей видимости
    print(locals().keys()) # ['baz']
```

```

print('bar' in locals()) # False
print('bar' in globals()) # False

def f3():
    baz = 3
    print(bar          # на bar из f1 есть ссылка, поэтому он попадает в локальную область
                  видимости f3 (закрытие)
    print(locals().keys()) # ['bar', 'baz']
    print('bar' in locals()) # True
    print('bar' in globals()) # False

def f4():
    bar = 4 # новая локальная bar, которая скрывает bar из локальной области f1
    baz = 4
    print(bar)
    print(locals().keys()) # ['bar', 'baz']
    print('bar' in locals()) # True
    print('bar' in globals()) # False

```

global и nonlocal (только Python 3)

Оба этих ключевых слова используются для получения доступа на запись к переменным, не являющимся локальными для текущей функции. Ключевое слово `global` объявляет, что имя должно рассматриваться как глобальная переменная.

```

foo = 0 # глобальная foo

def f1():
    foo = 1 # новая foo, локальная в f1

def f2():
    foo = 2 # новая foo, локальная в f2

    def f3():
        foo = 3 # новая foo, локальная в f3
        print(foo) # 3
        foo = 30 # изменяет локальную foo только в f3

    def f4():
        глобальная foo
        print(foo) # 0
        foo = 100 # изменяет глобальную foo

```

С другой стороны, функция `nonlocal` (см. раздел “Нелокальные переменные”), появившаяся в Python 3, переносит *локальную* переменную из объемлющей области видимости в локальную область видимости текущей функции. Из документации по Python о `nonlocal`:

Оператор `nonlocal` заставляет перечисленные идентификаторы ссылаться на ранее связанные переменные в ближайшей объемлющей области, исключая глобальные.

Python 3.x ≥ 3.0

```

def f1():
    def f2():
        foo = 2 # новая переменная foo, локальная в f2
    def f3():
        nonlocal foo # foo из f2, которая является ближайшей объемлющей областью
        print(foo) # 2
        foo = 20 # изменяет foo из f2!

```


13.7. Возникновение привязки

```
x = 5
x += 7
for x in iterable: pass
```

Каждое из приведенных выше утверждений – это возникновение привязки; `x` становится связанным с объектом, обозначенным цифрой 5. Если это утверждение встречается внутри функции, то `x` по умолчанию будет функционально-локальным. Список операторов связывания приведен в разделе “Синтаксис”.

Глава 14. Условные выражения

Условные выражения, включающие такие ключевые слова как `if`, `elif` и `else`, предоставляют программам на языке Python возможность выполнять различные действия в зависимости от логического условия `True` или `False`. В этом разделе рассматривается использование условных выражений Python, логики булевых операций и тернарных операторов.

14.1. Условное выражение (“тернарный оператор”)

Тернарный оператор используется для встраивания условных выражений. Его лучше всего использовать в простых, лаконичных операциях, которые легко читаются.

- Порядок аргументов отличается от многих других языков (таких как C, Ruby, Java и др.), что может привести к ошибкам при использовании Python людьми, незнакомыми с его “удивительным” поведением (они могут изменить порядок аргументов).
- Некоторые считают его “неуклюжим”, поскольку он противоречит обычному способу мышления (сначала думают об условии, а затем о его последствиях).

```
n = 5
"Greater than 2" if n > 2 else "Smaller than or equal to 2"
# Вывод: 'Greater than 2'
```

Результат этого выражения будет таким, как он читается на английском языке – если условное выражение является `True`, тогда он будет соответствовать выражению в левой части, в противном случае – правой части.

Тернарные операции также могут быть вложенными, как здесь:

```
n = 5
"Hello" if n > 10 else "Goodbye" if n > 5 else "Good day"
```

Они также предоставляют способ включения условий в лямбда-функции.

14.2. `if`, `elif` и `else`

В Python можно определить серию условий, используя `if` для первого условия, `elif` для остальных, вплоть до последнего (необязательного), `else` для всего, что не было “поймано” другими условиями.

```
number = 5

if number > 2:
    print("Число больше 2.")
elif number < 2: # Необязательное условие (может быть несколько elif)
    print("Число меньше 2.")
```

```
else:          # Необязательное условие (может быть только одно else)
    print("Число равно 2.")
```

Выведет "Число больше 2."

Использование `else if` вместо `elif` приведет к синтаксической ошибке и не допускается.

14.3. Значения истинности

Следующие значения считаются ложными, поскольку при применении к операторам булевой логики они оцениваются как `False`.

- `None`
- `False`
- `0`, или любое числовое значение, эквивалентное нулю, например `0L`, `0.0`, `0j`
- Пустые последовательности: `'`, `"`, `()`, `[]`
- Пустые отображения: `{}`
- Пользовательские типы, где `__bool__` или `__len__` методы возвращают `0` или `False`.

Все остальные значения в Python оцениваются как `True`.

Примечание: распространенной ошибкой является простая проверка на ложность операции, которая возвращает различные ложные значения, и различие важно. Например, использование `if foo()` вместо более явного `if foo() is None`.

14.4. Выражения булевой логики

Булевы логические выражения, помимо оценки `True` или `False`, возвращают *значение*, которое было интерпретировано как `True` или `False`. Это характерный для Python способ представления логики, которая в противном случае могла бы потребовать проверки типа `if-else`.

Оператор "and" ("и")

Оператор `"and"` оценивает все выражения и возвращает последнее выражение, если все выражения оцениваются как `True`. В противном случае возвращается первое значение, которое оценивается как `False`:

```
>>> 1 and 2
2

>>> 1 and 0
0

>>> 1 and "Hello World"
"Hello World"

>>> "" and "Pancakes"
""
```

Оператор "or" ("или")

Этот оператор оценивает выражения слева направо и возвращает первое значение, равное `True`, или последнее значение (если ни одно из них не равно `True`).

```
>>> 1 or 2
1

>>> None or 1
1

>>> 0 or []
[]
```

“Ленивые” вычисления

При использовании этого подхода следует помнить, что вычисления являются “ленивыми”. Выражения, которые не нужно оценивать для определения результата, не оцениваются. Например:

```
>>> def print_me():
    print('Я здесь!')
>>> 0 и print_me()
0
```

В приведенном выше примере `print_me` никогда не выполняется, поскольку Python может определить, что все выражение равно `False`, когда он встречается `0 (False)`. Помните об этом, если `print_me` необходимо выполнить для логики вашей программы.

Проверка на несколько условий

Распространенной ошибкой при проверке на наличие нескольких условий является неправильное применение логики.

В примере проверяется, не больше ли каждая из двух переменных 2. Утверждение оценивается как `if (a) and (b > 2)`. Это дает неожиданный результат, поскольку `bool(a)` оценивается как `True`, если `a` не равно нулю.

```
>>> a = 1
>>> b = 6
>>> if a and b > 2:
...     print('yes')
... else:
...     print('no')
yes
```

Каждую переменную необходимо сравнивать отдельно.

```
>>> if a > 2 and b > 2:
...     print('yes')
... else:
...     print('no')
no
```

Другая подобная ошибка допускается при проверке принадлежности переменной к нескольким значениям. Утверждение в этом примере оценивается как `if (a == 3) or (4) or (6)`. Это приводит к неожиданному результату, поскольку `bool(4)`, и `bool(6)` являются `True`.

```
>>> a = 1
>>> if a == 3 or 4 or 6:
...     print('yes')
... else:
...     print('no')
yes
```

Каждое сравнение должно проводиться отдельно:

```
>>> if a == 3 or a == 4 or a == 6:
...     print('yes')
... else:
...     print('no')
no
```

Общепринятым способом является использование оператора `in`:

```
>>> if a in (3, 4, 6):
...     print('yes')
... else:
...     print('no')
no
```

14.5. Использование функции "cmp" для получения результата сравнения двух объектов

В Python 2 есть функция `cmp`, которая позволяет определить, меньше, равен или больше один объект другого. Эта функция может быть использована для выбора варианта из списка на основе одного из этих трех вариантов.

Предположим, что необходимо вывести "больше, чем" ("greater than"), если $x > y$, "меньше, чем" ("less than"), если $x < y$, и "равно" ("equal"), если $x == y$.

```
[ 'equal', 'greater than', 'less than', ][cmp(x,y)]
```

```
# x,y = 1,1 результат: 'equal'
# x,y = 1,2 результат: 'less than'
# x,y = 2,1 результат: 'greater than'
```

`cmp(x,y)` возвращает следующие значения:

Результат сравнения

```
x < y    -1
x == y   0
x > y     1
```

В Python 3 эта функция отсутствует. Для преобразования старых функций сравнения в ключевые можно использовать вспомогательную функцию `cmp_to_key(func)` в `functools` в Python 3.

14.6. Оператор "else"

```
if condition:
    body
else:
    body
```

`else` выполнит свое тело только в том случае, если все предшествующие условные операторы будут иметь значение `False`.

```
if True:
    print "It is true!"
else:
    print "This won't get printed..."
# Вывод: It is true!
```

```
if False:
    print "This won't get printed..."
else:
    print "It is false!"
```

```
# Вывод: It is false!
```

14.7. Проверка принадлежности объекта к `None` и его присвоение

Часто возникает необходимость присвоить объекту какое-либо значение, если оно равно `None`, что означает, что объект не был присвоен. Мы будем использовать `aDate`. Самый простой способ сделать это – использовать проверку `is None`.

```
if aDate is None:
    aDate=datetime.date.today()
```

(Заметим, что для Python правильнее использовать `is None`, а не `== None`).

Это можно немного оптимизировать, используя представление о том, что в выражении булевой логики `not None` будет оцениваться как `True`. Следующий код эквивалентен:

```
if not aDate:
    aDate=datetime.date.today()
```

Но есть и более “питоновский” способ. Следующий код также равнозначен:

```
aDate=aDate or datetime.date.today()
```

При этом выполняется вычисление по короткой схеме. Если `aDate` инициализируется и она `not None`, то она присваивается самой себе без какого-либо эффекта. Если же она `is None`, то `aDate` присваивается значение `datetime.date.today()`.

14.8. Оператор “if”

```
if condition:
    body
```

Оператор `if` проверяет условие. Если его значение равно `True`, то выполняется тело оператора `if`. Если значение равно `False`, то тело пропускается.

```
if True:
    print “It is true!”
>> It is true!
```

```
if False:
    print “This won’t get printed..”
```

Условие может быть любым допустимым выражением:

```
if 2 + 2 == 4:
    print “I know math!”
>> I know math!
```

Глава 15. Сравнение

Параметр	Подробности
<code>x</code>	Первый сравниваемый элемент
<code>y</code>	Второй сравниваемый элемент

15.1. Цепное сравнение

С помощью цепного сравнения можно сравнивать несколько элементов с помощью нескольких операторов сравнения. Например:

```
x > y > z
```

это просто краткая форма для нижеприведенной записи:

```
x > y and y > z
```

Это значение будет равно `True` только в том случае, если оба сравнения равны `True`. В общем виде это выглядит так:

```
a OP b OP c OP d ...
```

где `OP` представляет одну из нескольких операций сравнения, которые можно использовать, а буквы – произвольные допустимые выражения.

Заметим, что `0 != 1 != 0` имеет значение `True`, хотя `0 != 0` – это `False`, в отличие от общепринятой математической нотации, в которой `x != y != z` означает, что `x`, `y` и `z` имеют разные значения.

Цепочка операций `==` в большинстве случаев имеет естественный смысл, поскольку равенство в общем случае является транзитивным.

Стиль

Теоретически не существует ограничений на количество элементов и операций сравнения при условии правильного использования синтаксиса:

```
1 > -1 < 2 > 0.5 < 100 != 24
```

Вышеприведенное возвращает `True`, если каждое сравнение возвращает `True`. Однако использование запутанной цепочки не является хорошим стилем. Хорошая цепочка должна быть "направленной", не сложнее, чем

```
1 > x > -4 > y != 8
```

Побочные эффекты

Как только одно из сравнений возвращает значение `False`, выражение сразу же оценивается в `False`, пропуская все оставшиеся сравнения. Заметим, что выражение `exp в a > exp > b` будет вычислено только один раз, тогда как в случае

```
a > exp и exp > b
```

`exp` будет вычисляться дважды, если `a > exp` истинно.

15.2. Сравнение по принципу "is" и "=="

Распространенной ошибкой является путаница операторов сравнения равенства `is` и `==`.

`a == b` сравнивает значения `a` и `b`.

`a is b` сравнивает *тождественность* `a` и `b`.

Пример:

```
a = 'Python is fun!'
```

```
b = 'Python is fun!'
```

```
a == b # возвращает True
```

```
a is b # возвращает False
```

```
a = [1, 2, 3, 4, 5]
```

```
b = a # b ссылается на a
```

```
a == b # True
```

```
a есть b # True
```

```
b = a[:] # b теперь ссылается на копию a
```

```
a == b # True
```

```
a есть b # False [!!!]
```

В принципе, это можно рассматривать как сокращение для `id(a) == id(b)`.

Кроме того, существуют причуды среды выполнения, которые еще больше усложняют ситуацию. Короткие строки и небольшие целые числа будут возвращать `True` при сравнении с использованием `is`, что связано с попыткой Python использовать меньше памяти для одинаковых объектов.

```
a = 'short'
```

```
b = 'short'
```

```
c = 5
```

```
d = 5
```

```
a is b # True
```

```
c is d # True
```

Однако более длинные строки и большие целые числа будут храниться отдельно.

```
a = 'not so short'
```

```
b = 'not so short'
```

```
c = 1000
```

```
d = 1000
```

```
a is b  # False
c is d  # False
```

Для проверки на None следует использовать is:

Один из вариантов использования is — проверка на наличие “сентинела” (т.е. уникального объекта).

```
sentinel = object()
def myfunc(var=sentinel):
    if var is sentinel:
        # значение не было предоставлено
        pass
    else:
        # значение было предоставлено
        pass
```

15.3. Больше или меньше

```
x > y
x < y
```

Эти операторы сравнивают два типа значений, это операторы “больше” и “меньше”. Для чисел это просто сравнение числовых значений с целью определения, какое из них больше:

```
12 > 4
# True
12 < 4
# False
1 < 4
# True
```

Для строк они будут сравнивать лексикографически, что похоже на алфавитный порядок, но не совсем то же самое.

```
"alpha" < "beta"
# True
"gamma" > "beta"
# True
"gamma" < "OMEGA"
# False
```

В этих сравнениях строчные буквы считаются “больше” прописных, поэтому “gamma” < “OMEGA” является ложным. Если бы все они были прописными, то был бы получен ожидаемый результат упорядочивания по алфавиту:

```
"GAMMA" < "OMEGA"
# True
```

Каждый тип по-разному определяет свои вычисления с помощью операторов < и >, поэтому перед их использованием необходимо выяснить, что означают эти операторы для данного типа.

15.4. Не равно

```
x != y
```

При этом возвращается True, если x и y не равны, а в противном случае возвращается False.

```
12 != 1
# True
12 != '12'
# True
'12' != '12'
# False
```

15.5. Равно

Это выражение проверяет, одно ли и то же значение у `x` и `y`, и возвращает результат в виде логического значения. Как правило, тип и значение должны совпадать, поэтому `int 12` не является тем же самым, что и строка `'12'`.

```
12 == 12
# True
12 == 1
# False
'12' == '12'
# True
'spam' == 'spam'
# True
'spam' == 'spam '
# False
'12' == 12
# False
```

Обратите внимание, что для каждого типа необходимо определить функцию, которая будет использоваться для оценки того, являются ли два значения одинаковыми. Для встроенных типов эти функции ведут себя так, как вы ожидаете, и просто оценивают на основе того, одно ли это и то же значение. Однако пользовательские типы могут определять проверку равенства как угодно, в том числе всегда возвращать `True` или всегда возвращать `False`.

15.6. Сравнение объектов

Для сравнения равенства пользовательских классов можно переопределить `==` и `!=`, создав методы `__eq__` и `__ne__`. Также можно переопределить `lt (<)`, `le (<=)`, `gt (>)` и `ge (>=)`. Обратите внимание, что переопределять нужно только два метода сравнения, с остальным справится Python (`==` - это то же самое, что не `<` и не `>`, и т. д.).

```
class Foo(object):
    def __init__(self, item):
        self.my_item = item
    def __eq__(self, other):
        return self.my_item == other.my_item
```

```
a = Foo(5)
b = Foo(5)
a == b # True
a != b # False
a is b # False
```

Обратите внимание: это простое сравнение предполагает, что `other` (объект, с которым производится сравнение) – это объект того же типа. Сравнение с другим типом приведет к ошибке:

```
class Bar(object):
    def __init__(self, item):
        self.other_item = item
    def __eq__(self, other):
        return self.other_item == other.other_item
    def __ne__(self, other):
        return self.other_item != other.other_item
```

```
c = Bar(5)
a == c # AttributeError: у объекта 'Foo' нет атрибута 'other_item'
```

Проверка `isinstance()` или аналогичная поможет при желании избежать этого.

Глава 16. Циклы

Параметр	Подробности
логическое выражение	выражение, которое может быть оценено в логическом контексте, например $x < 10$
переменная	имя переменной для текущего элемента итерируемого объекта
итерируемый объект (iterable)	все, что реализует итерации

Являясь одной из самых базовых функций в программировании, циклы представляют собой важную часть практически любого языка программирования. Циклы позволяют разработчикам задавать определенным участкам кода повторения на несколько циклов, которые называются итерациями. Далее рассматриваются использование нескольких типов циклов и их применение в Python.

16.1. Перерыв и продолжение в циклах

Оператор `break`

Когда оператор `break` выполняется внутри цикла, поток управления “вырывается” из цикла немедленно:

```
i = 0
while i < 7:
    print(i)
    if i == 4:
        print("Breaking from loop")
        break
    i += 1
```

Условие цикла не будет оцениваться после выполнения инструкции `break`. Обратите внимание, что инструкции `break` допускаются только внутри циклов, синтаксически. Оператор `break` внутри функции не может использоваться для завершения циклов, которые вызывали эту функцию.

Выполнение кода выводит числа до 4, когда встречается оператор `break` и цикл останавливается:

```
0
1
2
3
4
Breaking from loop
```

Операторы `break` также могут использоваться внутри циклов `for`, другой конструкции циклов в Python:

```
for i in (0, 1, 2, 3, 4):
    print(i)
    if i == 2:
        break
```

При выполнении этого цикла будет выведено:

```
0
1
2
```

Обратите внимание: 3 и 4 не печатаются, так как цикл закончился.

Если цикл имеет предложение `else`, то оно не выполняется, когда цикл завершается через оператор `break`.

Оператор continue

Оператор continue пропустит следующую итерацию цикла, минуя остальную часть текущего блока, но продолжая цикл. Как и при break, continue может использоваться только внутри циклов:

```
for i in (0, 1, 2, 3, 4, 5):
    if i == 2 or i == 4:
        continue
    print(i)

0
1
3
5
```

Обратите внимание: 2 и 4 не печатаются, потому что continue переходит к следующей итерации вместо продолжения print(i) при $i = 2$ или $i = 4$.

Вложенные петли

Операторы break и continue работают только на одном уровне цикла. В следующем примере будет выполнен выход только из внутреннего цикла for, а не внешнего while:

```
while True:
    for i in range(1,5):
        if i == 2:
            break          # Выйдет только из внутреннего цикла!
```

В Python не предусмотрена возможность одновременного выхода из циклов нескольких уровней – если это необходимо, могут понадобиться рефакторинг одной или нескольких петель в функцию и замена break на return.

Использование в функции оператора return в качестве оператора break

Оператор return завершает работу функции, не выполняя следующий за ним код.

Если у вас есть цикл внутри функции, использование return внутри этого цикла эквивалентно использованию оператора break, поскольку остальная часть кода цикла не выполняется (*обратите внимание, что любой код после цикла также не выполняется*):

```
def break_loop():
    for i in range(1, 5):
        if (i == 2):
            return(i)
        print(i)
    return(5)
```

Если у вас есть вложенные циклы, оператор return будет прерывать все циклы:

```
def break_all():
    for j in range(1, 5):
        for i in range(1,4):
            if i*j == 6:
                return(i)
            print(i*j)
```

выведет:

```
1 # 1*1
2 # 1*2
3 # 1*3
4 # 1*4
2 # 2*1
4 # 2*2
# Возврат, так как 2*3 = 6, и оставшиеся итерации обоих циклов не выполняются
```

16.2. Циклы For

Циклы `for` выполняют итерации по коллекциям элементов, например `list` или `dict`, и запускают блок кода с каждым элементом из коллекции.

```
for i in [0, 1, 2, 3, 4]:  
    print(i)
```

В приведенном выше примере в цикле `for` выполняется итерация по списку чисел.

На каждой итерации значение `i` устанавливается на следующий элемент списка. Так, сначала это будет 0, затем 1, затем 2 и т. д. На выходе получится следующее:

```
0  
1  
2  
3  
4
```

`range` – это функция, которая возвращает последовательность чисел в итерируемой форме, поэтому ее можно использовать для циклов `for`:

```
for i in range(5):  
    print(i)
```

Это дает такой же результат, что и первый цикл `for`. Обратите внимание, что 5 не выводится, так как диапазон здесь – это первые пять чисел, считая с 0.

Итерируемые объекты и итераторы

Цикл `for` может выполнять итерацию по любому итерируемому объекту, который является объектом, определяющим функцию `__getitem__` или `__iter__`. Функция `__iter__` возвращает итератор, который является объектом с функцией `next`, используемой для доступа к следующему элементу итерируемого объекта.

16.3. Итерирование по спискам

Чтобы перебрать список, вы можете использовать `for`:

```
for x in ['one', 'two', 'three', 'four']:  
    print(x)
```

В результате будут выведены элементы списка:

```
one  
two  
three  
four
```

Функция `range` генерирует числа, которые также часто используются в цикле `for`.

```
for x in range(1, 6):  
    print(x)
```

Результатом будет специальный тип последовательности диапазона в Python версии `>= 3` и список (`list`) в Python `<= 2`. Обе последовательности можно перебрать с помощью цикла `for`.

```
1  
2  
3  
4  
5
```

Если необходимо перебрать все элементы списка *и при этом* присвоить для элементов индексы, можно использовать функцию Python `enumerate`:

```
for index, item in enumerate(['one', 'two', 'three', 'four']):  
    print(index, '...', item)
```

Функция `enumerate` будет генерировать кортежи, которые распаковываются в `index` (целое число) и `item` (фактическое значение из списка). Вышеприведенный цикл выведет:

```
(0, '.', 'one')
(1, '.', 'two')
(2, '.', 'three')
(3, '.', 'four')
```

Переберем список, манипулируя значениями, с использованием `map` и `lambda`, т. е. применим лямбда-функцию для каждого элемента в списке:

```
x = map(lambda e : e.upper(), ['one', 'two', 'three', 'four'])
print(x)
```

Результат:

```
['ONE', 'TWO', 'THREE', 'FOUR'] # Python 2.x
```

Примечание: в Python 3.x `map` возвращает итератор, а не список. Поэтому в случае, если вам нужен список, вы должны использовать `print(list(x))`.

16.4. Циклы с "else"

Составные операторы (циклы) `for` и `while` могут опционально содержать условие `else` (на практике такое использование встречается довольно редко).

Предложение `else` выполняется только после того, как цикл `for` завершается путем итерации до завершения или после того, как цикл `while` завершается, поскольку его условное выражение становится ложным.

```
for i in range(3):
    print(i)
else:
    print('done')
```

```
i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('done')
```

результат:

```
0
1
2
done
```

Предложение `else` *не выполняется*, если цикл завершается каким-либо другим способом (через оператор `break` или путем выдачи исключения):

```
for i in range(2):
    print(i)
    if i == 1:
        break
else:
    print('done')
```

результат:

```
0
1
```

В большинстве других языков программирования нет этого необязательного предложения `else`. Использование ключевого слова `else`, в частности, часто считается вносящим путаницу.

Первоначальная концепция такого предложения принадлежит Дональду Кнуту. Смысл ключевого слова `else` становится ясным, если переписать цикл с использованием операторов `if` и `goto` из ранних языков программирования (до распространения структурированного программирования) или с низкоуровневого языка программирования.

Например:

```
while loop_condition():
```

```
    if break_condition():
        break
```

это эквивалентно:

псевдокод

```
<<start>>:
```

```
if loop_condition():
```

```
    if break_condition():
        goto <<end>>
```

```
    goto <<start>>
```

```
<<end>>:
```

Они останутся эквивалентными, если к каждому из них присоединить условие `else`.

Например:

```
while loop_condition():
```

```
    if break_condition():
        break
```

```
else:
```

```
    print('done')
```

что равнозначно:

псевдокод

```
<<start>>:
```

```
if loop_condition():
```

```
    if break_condition():
        goto <<end>>
```

```
    goto <<start>>
```

```
else:
```

```
    print('done')
```

```
<<end>>:
```

Аналогичным образом можно понимать цикл `for` с предложением `else`. Концептуально, есть условие цикла, которое остается `True` до тех пор, пока в итерируемом объекте или последовательности остаются какие-то элементы.

Зачем нужна эта странная конструкция?

Основным вариантом использования конструкции `for...else` является краткая реализация поиска, например:

```

a = [1, 2, 3, 4]
for i in a:
    if type(i) is not int:
        print(i)
        break
else:
    print("no exception")

```

Чтобы правильно понять else в этой конструкции, можно читать ее как *“если не прерывается”* или *“если не найдено”*.

16.5. Заявление о прохождении

pass – это нулевой оператор, который используется в тех случаях, когда по синтаксису Python требуется выполнить оператор (например, в теле цикла for или while), но никаких действий программист пока не требует. Это может быть полезно в качестве заполнителя для кода, который еще не написан.

```

for x in range(10):
    pass #мы не хотим или не готовы ничего делать здесь, поэтому pass

```

В этом примере ничего не произойдет. Цикл for будет завершен без ошибок, но никакие команды или код не будут выполнены. pass позволяет нам успешно запускать наш код без полного выполнения всех команд и действий.

Аналогично pass может быть использован в циклах while, а также в выделениях, определениях функций и т. д.

```

while x == y:
    pass

```

16.6. Итерация в словарях

Рассмотрим следующий словарь:

```
d = {"a": 1, "b": 2, "c": 3}
```

Для итерации по ключам можно использовать:

```

for key in d:
    print(key)

```

Результат:

```

"a"
"b"
"c"

```

что эквивалентно:

```

for key in d.keys():
    print(key)

```

или в Python 2:

```

for key in d.iterkeys():
    print(key)

```

Для перебора его значений используйте:

```

for value in d.values():
    print(value)

```

Результат:

```

1
2
3

```

Для итерации по ключам и значениям используйте:

```

for key, value in d.items():
    print(key, "...", value)

```

Результат:

```
a :: 1
b :: 2
c :: 3
```

Обратите внимание, что в Python 2 функции `.keys()`, `.values()` и `.items()` возвращают объект списка – list. Если необходимо просто выполнить итерацию по результату, можно использовать эквивалентные `.iterkeys()`, `.itervalues()` и `.iteritems()`.

Разница между `.keys()` и `.iterkeys()`, `.values()` и `.itervalues()`, `.items()` и `.iteritems()` заключается в том, что методы `iter*` являются генераторами. Таким образом, элементы словаря выдаются один за другим по мере их вычисления. При возврате объекта list все элементы упаковываются в список и затем возвращаются для дальнейшей оценки.

Обратите также внимание на то, что в Python 3 порядок элементов, напечатанных указанным выше способом, не соответствует какому-либо порядку.

16.7. “Полуцикл” do-while

В отличие от других языков, в Python нет конструкции `do-until` или `do-while` (позволяющей выполнить код один раз до проверки условия). Однако для достижения той же цели можно комбинировать `while True` с `break`.

```
a = 10
while True:
    a = a-1
    print(a)
    if a<7:
        break
print('Done.')
```

В результате будет выведено:

```
9
8
7
6
Done.
```

16.8. Циклирование и распаковка

Если, к примеру, требуется выполнить цикл по списку кортежей:

```
collection = [('a', 'b', 'c'), ('x', 'y', 'z'), ('1', '2', '3')]
```

вместо того, чтобы делать что-то наподобие:

```
for item in collection:
    i1 = item[0]
    i2 = item[1]
    i3 = item[2]
    # logic
```

или чего-то в этом роде:

```
for item in collection:
    i1, i2, i3 = item
    # logic
```

вы можете просто использовать:

```
for i1, i2, i3 in collection:
    # logic
```

Это будет работать для большинства итерируемых типов данных, не только для кортежей.

16.9. Итерирование части списка с разным размером шага

Предположим, у вас есть длинный список элементов, и вас интересует только каждый второй элемент этого списка. Возможно, вы хотите изучить только первый или последний элемент, или определенный диапазон записей в списке. Python обладает сильными встроенными возможностями индексирования. Приведем несколько примеров того, как можно реализовать эти сценарии.

Вот простой список, который будет использоваться во всех примерах:

```
lst = ['alpha', 'bravo', 'charlie', 'delta', 'echo']
```

Итерация по всему списку

Для перебора по каждому элементу в списке может использоваться цикл `for`, как показано ниже:

```
for s in lst:
    print s[:1] # выведет первую букву
```

Цикл `for` присваивает `s` каждому элементу `lst`. В результате будет выведено:

```
a
b
c
d
e
```

Часто вам нужен как элемент, так и индекс этого элемента. Ключевое слово `enumerate` выполняет эту задачу.

```
for idx, s in enumerate(lst):
    print("%s has an index of %d" % (s, idx))
```

Индекс `idx` будет начинаться с нуля и увеличиваться для каждой итерации, в то время как `s` будет содержать обрабатываемый элемент. В предыдущем фрагменте будет получен результат:

```
alpha has an index of 0
bravo has an index of 1
charlie has an index of 2
delta has an index of 3
echo has an index of 4
```

Итерация по части списка

Если мы хотим выполнить итерацию по диапазону (помня, что в Python используется нулевой индекс), то следует использовать ключевое слово `range`.

```
for i in range(2,4):
    print("lst at %d contains %s" % (i, lst[i]))
```

Это приведет к выводу:

```
lst at 2 contains charlie
lst at 3 contains delta
```

Список также может быть нарезан. В следующей нотации нарезка идет от элемента с индексом 1 до конца с шагом 2. Два цикла `for` дают одинаковый результат.

```
for s in lst[1::2]:
    print(s)
```

```
for i in range(1, len(lst), 2):
    print(lst[i])
```


Вышеприведенный фрагмент кода выводит:

```
bravo
delta
```

Индексирование и нарезка рассмотрены в отдельной теме.

16.10. Цикл While

В цикле `while` операторы цикла будут выполняться до тех пор, пока условие цикла не станет ложным. Следующий код выполнит операторы цикла в общей сложности 4 раза.

```
i = 0
while i < 4:
    #условие цикла
    i = i + 1
```

Хотя приведенный выше цикл можно легко преобразовать в более элегантный цикл `for`, циклы `while` полезны для проверки выполнения некоторого условия. Следующий цикл будет выполняться до тех пор, пока объект `myObject` не будет готов.

```
myObject = anObject()
while myObject.isNotReady():
    myObject.tryToGetReady()
```

Циклы `while` также могут работать без условия, используя числа (комплексные или вещественные) или `True`:

```
import cmath

complex_num = cmath.sqrt(-1)
while complex_num:                # Вы также можете заменить complex_num на любое число,
    print(complex_num)            # True или значение любого типа
                                # Выводит 1j вечно
```

Если условие всегда верно, цикл `while` будет выполняться вечно (бесконечный цикл), если его не прервет оператор `break`, `return` или исключение.

```
while True:
    print "Infinite loop"
# Infinite loop
# Infinite loop
# Infinite loop
# ...
```

Глава 17. Массивы

Параметр	Подробности
<code>b</code>	представляет собой знаковое целое число размером 1 байт
<code>B</code>	беззнаковое целое число размером 1 байт
<code>c</code>	символ размером 1 байт
<code>u</code>	символ Unicode размером 2 байта
<code>h</code>	знаковое целое число размером 2 байта
<code>H</code>	беззнаковое целое число размером 2 байта
<code>i</code>	знаковое целое число размером 2 байта
<code>I</code>	беззнаковое целое число размером 2 байта

w	символ Unicode размером 4 байта
I	знаковое целое число размером 4 байта
L	беззнаковое целое число размером 4 байта
f	число с плавающей точкой размером 4 байта
d	число с плавающей точкой размером 8 байт

“Массивы” в Python – это не массивы в обычных языках программирования, таких как C и Java, а ближе к спискам. Список может быть коллекцией как однородных, так и неоднородных элементов и может содержать целые числа, строки или другие списки.

17.1. Доступ к отдельным элементам через индексы

Доступ к отдельным элементам можно получить с помощью индексов. Массивы Python имеют нулевую индексацию. Приведем пример:

```
my_array = array('i', [1,2,3,4,5])
print(my_array[1])
# 2
print(my_array[2])
# 3
print(my_array[0])
# 1
```

17.2. Введение в массивы

Массив – это структура данных, в которой хранятся значения одного и того же типа. В Python это основное различие между массивами и списками.

Хотя списки Python могут содержать значения, соответствующие различным типам данных, массивы в этом языке могут содержать только значения, соответствующие одному типу данных.

Чтобы использовать массивы на языке Python, вам нужно импортировать стандартный модуль `array`. Это связано с тем, что массив не является фундаментальным типом данных, как строки, целые числа и т. д. Импортировать модуль `array` можно следующим образом:

```
from array import *
```

Когда модуль `array` импортирован, вы можете объявить массив следующим образом:

```
arrayIdentifierName = array(typecode, [Initializers])
```

В этом объявлении `arrayIdentifierName` – это имя массива, `typecode` позволяет Python узнать тип массива, а `Initializers` – это значения, с которыми инициализируется массив.

Типовые коды – это коды, которые используются для определения типа значений массива или типа массива. В приведенной в начале главы таблице приведены возможные значения, которые можно использовать при объявлении массива и его типа.

Вот пример реального объявления массива:

```
my_array = array('i', [1,2,3,4])
```

В приведенном выше примере используется типовой код `i`. Этот типовой код представляет собой знаковое целое число, размер которого равен 2 байтам.

Вот простой пример массива, содержащего 5 целых чисел:

```
from array import *
my_array = array('i', [1,2,3,4,5])
for i in my_array:
    print(i)
# 1
# 2
# 3
# 4
# 5
```

17.3. Добавление произвольного значения в массив с помощью метода `append()`

```
my_array = array('i', [1,2,3,4,5]) my_array.append(6)
# array('i', [1, 2, 3, 4, 5, 6])
```

Обратите внимание, что значение 6 было добавлено к существующим значениям массива.

17.4. Вставка значения в массив с помощью метода `insert()`

Мы можем использовать метод `insert()` для вставки значения в любой индекс массива. Приведем пример :

```
my_array = array('i', [1,2,3,4,5]) my_array.insert(0,0)
#array('i', [0, 1, 2, 3, 4, 5])
```

В приведенном примере в индекс 0 было вставлено значение 0. Обратите внимание, что первый аргумент – это индекс, а второй аргумент – значение.

17.5. Расширение массива с помощью метода `extend()`

Массив в Python может быть расширен более чем одним значением с помощью метода `extend()`. Вот пример :

```
my_array = array('i', [1,2,3,4,5]) my_extnd_array = array('i', [7,8,9,10])
my_array.extend(my_extnd_array)
# array('i', [1, 2, 3, 4, 5, 7, 8, 9, 10])
```

Мы видим, что массив `my_array` был расширен значениями из `my_extnd_array`.

17.6. Добавление элементов из списка в массив с помощью метода `fromlist()`

Приведем пример:

```
my_array = array('i', [1,2,3,4,5]) c=[11,12,13]
my_array.fromlist(c)
# array('i', [1, 2, 3, 4, 5, 11, 12, 13])
```

Таким образом, мы видим, что из списка `c` в массив `my_array` были добавлены значения 11, 12 и 13.

17.7. Удаление любого элемента массива с помощью метода `remove()`

Приведем пример:

```
my_array = array('i', [1,2,3,4,5]) my_array.remove(4)
# array('i', [1, 2, 3, 5])
```

Мы видим, что из массива был удален элемент 4.

17.8. Удаление последнего элемента массива с помощью метода `pop()`

Метод `pop` удаляет из массива последний элемент. Приведем пример :

```
my_array = array('i', [1,2,3,4,5]) my_array.pop()
# array('i', [1, 2, 3, 4])
```

Таким образом, мы видим, что последний элемент (5) “выскочил” (“pop”) из массива.

17.9. Получение любого элемента по его индексу с помощью метода index()

Метод index() возвращает первый индекс совпадающего значения. Помните, что массивы имеют нулевую индексацию.

```
my_array = array('i', [1,2,3,4,5])
print(my_array.index(5))
# 5
my_array = array('i', [1,2,3,3,5])
print(my_array.index(3))
# 3
```

Обратите внимание, что во втором примере был возвращен только один индекс, хотя значение существует в массиве дважды.

17.10. Обратное преобразование массива с помощью метода reverse()

Метод reverse() делает то, о чем говорит его название, – переворачивает массив. Вот пример:

```
my_array = array('i', [1,2,3,4,5]) my_array.reverse()
# array('i', [5, 4, 3, 2, 1])
```

17.11. Получение информации о буфере массива с помощью метода buffer_info()

Этот метод предоставляет вам начальный адрес буфера массива в памяти и количество элементов в массиве. Вот пример:

```
my_array = array('i', [1,2,3,4,5])
my_array.buffer_info()
(33881712, 5)
```

17.12. Проверка количества вхождений элемента с помощью метода count()

Метод count() возвращает количество повторений элемента в массиве. В следующем примере мы видим, что значение 3 встречается дважды.

```
my_array = array('i', [1,2,3,3,5])
my_array.count(3)
# 2
```

17.13. Преобразование массива в строку с помощью метода tostring()

Метод tostring() преобразует массив в строку.

```
my_char_array = array('c', ['g','e','e','k'])
# array('c', 'geek')
print(my_char_array.tostring())
# geek
```

17.14. Преобразование массива в список с одинаковыми элементами с использованием метода tolist()

Когда вам нужен объект списка list, вы можете использовать метод tolist() для преобразования вашего массива в список.

```
my_array = array('i', [1,2,3,4,5])
c = my_array.tolist()
# [1, 2, 3, 4, 5]
```

17.15. Добавление строки в массив char, используя метод fromstring()

Вы можете добавить строку в массив символов, используя fromstring():

```
my_char_array = array('c', ['g','e','e','k'])
my_char_array.fromstring("stuff")
print(my_char_array)
#array('c', 'geekstuff')
```

Глава 18. Многомерные массивы

18.1. Списки в списках

Хорошим способом визуализации двумерного массива является представление его в виде списка списков. Что-то вроде этого:

```
lst=[[1,2,3],[4,5,6],[7,8,9]]
```

Здесь внешний список lst содержит три элемента. Каждый из них – список: первый – [1,2,3], второй – [4,5,6], третий – [7,8,9]. К этим спискам можно обращаться так же, как и к другим элементам списка, например:

```
print (lst[0])
#результат: [1, 2, 3]
```

```
print (lst[1])
#результат: [4, 5, 6]
```

```
print (lst[2])
#результат: [7, 8, 9]
```

Затем можно обращаться к различным элементам в каждом из этих списков таким же образом:

```
print (lst[0][0])
#результат: 1
```

```
print (lst[0][1])
#результат: 2
```

Здесь первое число внутри квадратных скобок ([]) означает получение списка в этой позиции. В приведенном выше примере мы использовали число 0, чтобы получить список в 0-й позиции, который равен [1,2,3]. Второй набор квадратных скобок означает получение элемента в этой позиции из внутреннего списка. В этом случае мы использовали как 0 и 1. В 0-й позиции списка мы получили число 1, а в 1-й позиции – 2.

Аналогичным образом можно задавать значения внутри этих списков:

```
lst[0]=[10,11,12]
```

Теперь список имеет вид [[10,11,12],[4,5,6],[7,8,9]]. В этом примере мы изменили весь первый список на совершенно новый список.

```
lst[1][2]=15
```

Теперь список имеет вид [[10,11,12],[4,5,15],[7,8,9]]. В этом примере мы изменили один элемент внутри одного из внутренних списков. Сначала мы вошли в список в позиции 1 и изменили элемент внутри него в позиции 2, который был 6, а стал 15.

18.2. Списки в списках в списках в ...

Вот трехмерный массив:

```
[[[111,112,113],[121,122,123],[131,132,133]],[[211,212,213],[221,222,223],[231,232,233]],[[311,312,313],[321,322,323],[331,332,333]]]
```

Очевидно, что это становится немного трудным для чтения. Используйте обратную косую черту, чтобы разбить различные измерения:

```
[[[111,112,113],[121,122,123],[131,132,133]],\
 [[211,212,213],[221,222,223],[231,232,233]],\
 [[311,312,313],[321,322,323],[331,332,333]]]
```

Вложенные списки, подобные этому, можно расширить до произвольно больших размеров. Доступ аналогичен доступу к двумерным массивам:

```
print(myarray)
print(myarray[1])
print(myarray[2][1])
print(myarray[1][0][2])
etc.
```

И редактирование тоже похоже:

```
myarray[1]=new_n-1_d_list
myarray[2][1]=new_n-2_d_list
myarray[1][0][2]=new_n-3_d_list    #или одно число, если вы имеете дело с трехмерными
                                     # массивами
etc.
```

Глава 19. Словарь

Параметр	Подробности
ключ (key)	желаемый ключ для поиска
значение (value)	значение, которое необходимо установить или вернуть

19.1. Введение в словарь

Словарь является примером *хранилища “ключ-значение”* также известного как *Mapping* в Python. Он позволяет хранить и извлекать элементы, ссылаясь на ключ. Поскольку словари ссылаются на ключ, в них очень быстро выполняется поиск. Поскольку они используются в основном для обращения к элементам по ключу, они не сортируются.

Создание словаря

Словари могут быть инициализированы различными способами:

С помощью литерала

```
d = {}                # пустой словарь
d = {"key": "value"}  # словарь с начальными значениями
```

Версия Python 3.x ≥ 3.5:

```
# Также возможна распаковка одного или нескольких словарей с использованием
# литерального синтаксиса
# создает поверхностную копию (shallow copy) otherdict
d = {**otherdict}
# также обновляет поверхностную копию содержимым yetanotherdict.
d = {**otherdict, **yetanotherdict}
```

Генератор словаря

```
d = {k:v for k,v in [('key', 'value'),]}
```

см. также главу 21.4. “Генератор словаря”.

Встроенный класс: dict()

```
d = dict() # пустой dict
d = dict(key='value') # явные аргументы ключевых слов
d = dict([('key', 'value')]) # передача списка пар ключ/значение
# сделать поверхностную копию другого dict (возможно только если ключи – строки!)
d = dict(**otherdict)
```

Модификация словаря

Чтобы добавить элементы в словарь, нужно создать новый ключ со значением:

```
d['newkey'] = 42
```

Также можно добавить list и словарь в качестве значения:

```
d['new_list'] = [1, 2, 3]
d['new_dict'] = {'nested_dict': 1}
```

Чтобы удалить элемент, нужно удалить ключ из словаря:

```
del d['newkey']
```

19.2. Избегание исключений KeyError

Одна из распространенных ошибок при использовании словарей – обращение к несуществующему ключу. Это обычно приводит к исключению KeyError.

```
mydict = {}
mydict['not there']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'not there'
```

Один из способов избежать ошибки с ключами — использовать метод dict.get, позволяющий вам указать значение по умолчанию, которое будет возвращаться в случае отсутствия ключа.

```
value = mydict.get(key, default_value)
```

Метод возвращает mydict[key], если он существует, а в противном случае возвращает default_value. Обратите внимание, что это не добавляет ключ к mydict. Поэтому, если вы хотите сохранить эту пару “ключ-значение”, вы должны использовать метод mydict.setdefault(key, default_value), который действительно сохраняет пару “ключ-значение”.

```
mydict = {}
print(mydict)
# {}
print(mydict.get("foo", "bar"))
# bar
print(mydict)
# {}
print(mydict.setdefault("foo", "bar"))
# bar
print(mydict)
# {'foo': 'bar'}
```

Другой способ избавиться от ошибки — перехватить исключение:

```
try: value = mydict[key]except KeyError: value = default_value
```

Вы также можете проверить, есть ли ключ в словаре:

```
if key in mydict: value = mydict[key]else: value = default_value
```

Обратите внимание, что в многопоточных средах ключ может быть удален из словаря после проверки, создавая состояние, при котором все еще может быть выдано исключение.

В качестве другого варианта можно использовать подкласс `dict` – `collection.defaultdict`, который имеет значение `default_factory` для создания новых записей в словаре при задании ключа `new_key`.

19.3. Итерация по словарю

Если использовать словарь в качестве итератора (например, в операторе `for`), то он обходит *ключи* словаря. Например:

```
d = {'a': 1, 'b': 2, 'c':3}
for key in d:
    print(key, d[key])
# c 3
# b 2
# a 1
```

То же самое верно и при использовании:

```
print([key for key in d])
# ['c', 'b', 'a']
```

Версия Python 3.x ≥ 3.0:

Метод `items()` можно использовать для *одновременного заикливания ключа и значения*:

```
for key, value in d.items():
    print(key, value)
# c 3
# b 2
# a 1
```

Метод `values()` можно использовать для перебора только значений:

```
for key, value in d.values():
    print(key, value)
# 3
# 2
# 1
```

Версия Python 2.x ≥ 2.2:

Здесь методы `keys()`, `values()` и `items()` возвращают списки, а три дополнительных метода `iterkeys()`, `itervalues()` и `iteritems()` возвращают итераторы.

19.4. Словарь со значениями по умолчанию

Доступен в стандартной библиотеке как `defaultdict`.

```
from collections import defaultdict
d = defaultdict(int)
d['key']          # 0
d['key'] = 5
d['key']          # 5

d = defaultdict(lambda: 'empty')
d['key']          # 'empty'
d['key'] = 'full'
d['key']          # 'full'
```


В качестве альтернативы, если необходимо использовать встроенный класс `dict`, использование `dict.setdefault()` позволит создавать ключ по умолчанию при каждом обращении к ключу, которого раньше не существовало*:

```
>>> d = {}
{}
>>> d.setdefault('Another_key', []).append("This worked!")
>>> d
{'Another_key': ['This worked!']}
```

Следует помнить, что при большом количестве добавляемых значений `dict.setdefault()` при каждом вызове будет создавать новый экземпляр начального значения (в данном примере `[]`), что может создать ненужную нагрузку.

19.5. Слияние словарей

Рассмотрим следующие словари:

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
```

Версия Python 3.5+:

```
>>> fishdog = {**fish, **dog}
>>> fishdog
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

В этом примере дубликаты ключей соответствуют последнему значению (например, “Клиффорд” переопределяет “Немо”).

Версия Python 3.3+:

```
>>> from collections import ChainMap
>>> dict(ChainMap(fish, dog))
{'hands': 'fins', 'color': 'red', 'special': 'gills', 'name': 'Nemo'}
```

Здесь первостепенное значение отдается конкретному ключу, а не последнему (“Клиффорд” выбрасывается в пользу “Немо”).

Версия Python 2.x, 3.x:

```
>>> from itertools import chain
>>> dict(chain(fish.items(), dog.items()))
{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

При этом используется последнее значение, как и в случае с техникой объединения на основе `**` (“Клиффорд” переопределяет “Немо”).

```
>>> fish.update(dog)
>>> fish
{'color': 'red', 'hands': 'paws', 'name': 'Clifford', 'special': 'gills'}
```

`dict.update` использует последний словарь для перезаписи предыдущего.

19.6. Доступ к ключам и значениям

При работе со словарями часто возникает необходимость получить доступ ко всем ключам и значениям словаря либо в цикле `for`, либо в генераторе списка (`list comprehension`), либо просто как обычный список. Дается словарь вида:

* Поваренная книга Python, 3-е издание, авторы Дэвид Бизли и Брайан К. Джонс (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

```
mydict = {
    'a': '1',
    'b': '2'
}
```

Вы можете получить список ключей, используя метод `keys()`:

```
print(mydict.keys())
# Python2: ['a', 'b']
# Python3: dict_keys(['b', 'a'])
```

Если вместо этого вы хотите получить список значений, используйте метод `values()`:

```
print(mydict.values())
# Python2: ['1', '2']
# Python3: dict_values(['2', '1'])
```

Если вы хотите работать как с ключом, так и с соответствующим значением, вы можете использовать метод `items()`:

```
print(mydict.items())
# Python2: [('a', '1'), ('b', '2')]
# Python3: dict_items([('b', '2'), ('a', '1')])
```

Примечание: поскольку `dict` является несортированным, `keys()`, `values()` и `items()` не имеют порядка сортировки. Используйте `sort()`, `sorted()` или `OrderedDict`, если вам важен порядок, который возвращают эти методы.

Отличие Python 2 и 3: в Python 3 эти методы возвращают специальные итерируемые объекты, а не списки, и являются эквивалентом методов Python 2 `iterkeys()`, `itervalues()` и `iteritems()`. Эти объекты в большинстве случаев можно использовать как списки, хотя есть и некоторые отличия. Более подробную информацию см. в PEP 3106.

19.7. Доступ к значениям словаря

```
dictionary = {"Hello": 1234, "World": 5678}
print(dictionary["Hello"])
```

Приведенный выше код выведет 1234.

Строка "Hello" в этом примере называется ключом. Она используется для поиска значения в `dict` путем заключения ключа в квадратные скобки. После соответствующего двоеточия в определении словаря встречается число 1234. Это и называется *значением*, которому сопоставляется "Hello" в данном словаре.

Поиск такого значения по несуществующему ключу вызовет исключение `KeyError`, которое, если не будет поймано, остановит выполнение. Если мы хотим получить доступ к значению без риска возникновения ошибки `KeyError`, мы можем воспользоваться методом `dictionary.get()`. По умолчанию, если ключ не существует, метод возвращает `None`. Мы можем передать ему второе значение, которое будет возвращено вместо `None` в случае неудачного поиска.

```
w = dictionary.get("whatever")
x = dictionary.get("whatever", "nuh-uh")
```

В этом примере `w` получит значение `None`, а `x` – значение "nuh-uh".

19.8. Создание словаря

Правила создания словаря:

- Каждый ключ должен быть уникальным (иначе он будет переопределен).
- Каждый ключ должен быть хешируемым (для хеширования можно использовать функцию `hash`, в противном случае будет выдана ошибка `TypeError`).
- Определенного порядка следования ключей нет.

```
# Создание словаря и заполнение его значениями
stock = {'eggs': 5, 'milk': 2}

# или создание пустого словаря
dictionary = {}

# и заполнение его после
dictionary['eggs'] = 5
dictionary['milk'] = 2

# Значения также могут быть списками
mydict = {'a': [1, 2, 3], 'b': ['1', '2', '3']}

# Используйте метод list.append() для добавления новых элементов в список значений
mydict['a'].append(4) # => {'a': [1, 2, 3, 4], 'b': ['1', '2', '3']}
mydict['b'].append('four') # => {'a': [1, 2, 3, 4], 'b': ['1', '2', '3', '4']}

# Мы также можем создать словарь, используя список кортежей из двух элементов
iterable = [('eggs', 5), ('milk', 2)] dictionary = dict(iterables)

# или с использованием аргумента keyword:
dictionary = dict(eggs=5, milk=2)

# Другим способом будет использование dict.fromkeys:
dictionary = dict.fromkeys((milk, eggs)) # => {'milk': None, 'eggs': None}
dictionary = dict.fromkeys((milk, eggs), (2, 5)) # => {'молоко': 2, 'яйца': 5}
```

19.9. Создание упорядоченного словаря

Вы можете создать упорядоченный словарь, который будет следовать определенному порядку при итерации по ключам в словаре. Используйте `OrderedDict` из модуля `collections`. При итерации он всегда будет возвращать элементы словаря в исходном порядке вставки.

```
from collections import OrderedDict

d = OrderedDict()
d['first'] = 1
d['second'] = 2
d['third'] = 3
d['last'] = 4

# Выведет "first 1", "second 2", "third 3", "last 4"
for key in d:
    print(key, d[key])
```

19.10. Распаковка словарей с помощью оператора **

Для доставки пар “ключ-значение” из словаря в аргументы функции можно использовать оператор распаковки аргументов по ключевому слову `**`. Упрощенный пример из официальной документации:

```
>>>
>>> def parrot(voltage, state, action):
    print("This parrot wouldn't", action, end=' ')
    print("If you put", voltage, "volts through it.", end=' ')
    print("E's", state, "!")

>>> d = {'voltage': "four million", 'state': "bleedin' demised", 'action': "VOOM"}
>>> parrot(**d)
This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

Начиная с Python 3.5 этот синтаксис можно использовать и для объединения произвольного числа объектов dict.

```
>>> fish = {'name': "Nemo", 'hands': "fins", 'special': "gills"}
>>> dog = {'name': "Clifford", 'hands': "paws", 'color': "red"}
>>> fishdog = {**fish, **dog}
>>> fishdog

{'hands': 'paws', 'color': 'red', 'name': 'Clifford', 'special': 'gills'}
```

Как видно из этого примера, дублирующиеся ключи отображаются на свое последнее значение (например, "Clifford" перекрывает "Nemo").

19.11. Запятая в конце строки

Как и в случае со списками и кортежами, в словаре можно ставить закрывающую запятую.

```
role = {"By day": "A typical programmer",
        "By night": "Still a typical programmer", }
```

PEP 8 предписывает оставлять пробел между запятой и закрывающей фигурной скобкой.

19.12. Конструктор dict()

Конструктор dict() может быть использован для создания словарей из аргументов-ключей, или из одной итерированной пары "ключ-значение", или из одного словаря и аргументов-ключей.

```
dict(a=1, b=2, c=3) # {'a': 1, 'b': 2, 'c': 3}
dict([('d', 4), ('e', 5), ('f', 6)]) # {'d': 4, 'e': 5, 'f': 6}
dict([('a', 1)], b=2, c=3) # {'a': 1, 'b': 2, 'c': 3}
dict({'a': 1, 'b': 2}, c=3) # {'a': 1, 'b': 2, 'c': 3}
```

19.13. Пример словарей

Словари сопоставляют ключи со значениями

```
car = {}
car["wheels"] = 4
car["color"] = "Red"
car["model"] = "Corvette"
```

Доступ к значениям словаря осуществляется по их ключам

```
print "Little " + car["color"] + " " + car["model"] + "!"
# Этот код напечатает "Little Red Corvette!"
```

Словари также могут быть созданы в стиле JSON:

```
car = {"wheels": 4, "color": "Red", "model": "Corvette"}
```

Значения словаря можно перебирать:

```
for key in car:
    print key + ": " + car[key]

# wheels: 4
# color: Red
# model: Corvette
```

19.14. Все комбинации значений словаря

```
options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}
```

Словарь в этом примере выше содержит список, представляющий набор значений для поиска соответствующего ключа. Предположим, вы хотите найти "x" = "a" с "y" = 10, затем "x" = "b" с "y" = 10 и т. д., пока вы не переберете все возможные комбинации.

Вы можете создать список, который возвращает все такие комбинации значений, используя следующий код:

```
import itertools

options = {
    "x": ["a", "b"],
    "y": [10, 20, 30]
}

keys = options.keys()
values = (options[key] for key in keys)
combinations = [dict(zip(keys, combination)) for combination in itertools.product(*values)]
print(combinations)
```

Это дает нам следующий список, хранящийся в переменной combinations:

```
#[{ 'x': 'a', 'y': 10},
# { 'x': 'b', 'y': 10},
# { 'x': 'a', 'y': 20},
# { 'x': 'b', 'y': 20},
# { 'x': 'a', 'y': 30},
# { 'x': 'b', 'y': 30}]
```

Глава 20. Список

В Python список (**List**) – это общая структура данных, широко используемая в программах. В других языках их часто называют *динамическими массивами*. Они являются одновременно *изменяемым* типом данных и вместе с тем имеют тип данных *последовательности*, что позволяет их *индексировать* и создавать *срезы*. Список может содержать объекты различных типов, в том числе и другие объекты списка.

20.1. Методы перечисления и поддерживаемые операторы

Начиная с заданного списка a:

```
a = [1, 2, 3, 4, 5]
```

1. `append(value)` – добавляет новый элемент в конец списка.

```
# Добавить в список значения 6, 7 и 7
a.append(6)
a.append(7)
a.append(7)
# a: [1, 2, 3, 4, 5, 6, 7, 7]
```

```
# Добавить еще один список
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
```

```
# Добавить элемент другого типа, так как элементы списка не обязательно должны иметь
одинаковый тип
my_string = "hello world"
a.append(my_string)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9], "hello world"]
```

Обратите внимание, что метод `append()` добавляет только один новый элемент в конец списка. Если вы добавляете список в другой список, то добавляемый список становится единственным элементом в конце первого списка.

```
# Добавление списка к другому списку
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9]
a.append(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, [8, 9]]
a[8]
# Возвращает: [8,9]
```

2. `extend(enumerable)` – этот метод расширяет список, добавляя элементы из другого перечисляемого.

```
a = [1, 2, 3, 4, 5, 6, 7, 7]
b = [8, 9, 10]

# Расширить список, добавив в него все элементы из b
a.extend(b)
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]

# Расширение списка элементами несписочного перечисления:
a.extend(range(3))
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10, 0, 1, 2]
```

Списки также можно объединять (конкатенировать) при помощи оператора `+`. Обратите внимание, что это не изменяет ни один из исходных списков:

```
a = [1, 2, 3, 4, 5, 6] + [7, 7] + b
# a: [1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

3. Метод `index(value, [startIndex])` – получает индекс первого вхождения входного значения. Если входное значение отсутствует в списке, возникает исключение `ValueError`. Если указан второй аргумент, поиск начинается с указанного индекса.

```
a.index(7)
# Возвращает: 6

a.index(49) # ValueError, так как 49 нет в a.

a.index(7, 7)
# Возвращает: 7

a.index(7, 8) # ValueError, так как нет 7, начинающейся с индекса 8
```

4. Метод `insert(index, value)` – вставляет `value` непосредственно перед указанным `index`. Таким образом, после вставки новый элемент занимает позицию `index`.

```
a.insert(0, 0)      # вставить 0 в позицию 0
a.insert(2, 5)      # вставить 5 в позицию 2
# a: [0, 1, 5, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
```

5. Метод `pop([index])` – удаляет и возвращает элемент по `index`. Без аргумента он удаляет и возвращает последний элемент списка.

```
a.pop(2)
# Возвращает: 5
# a: [0, 1, 2, 3, 4, 5, 6, 7, 7, 8, 9, 10]
a.pop(8)
# Возвращает: 7
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# Без аргумента:
a.pop()
# Возвращает: 10
# a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

6. Метод `remove(value)` – удаляет первое вхождение указанного значения. Если это значение не может быть найдено, выдается `ValueError`.

```
a.remove(0)
a.remove(9)
# a: [1, 2, 3, 4, 5, 6, 7, 8]
a.remove(10)
# ValueError, так как 10 не находится в a
```

7. Метод `reverse()` – инвертирует список на месте и возвращает `None`.

```
a.reverse()
# a: [8, 7, 6, 5, 4, 3, 2, 1]
```

Существуют и другие способы реверсирования списков.

8. Метод `count(value)` – подсчитывает количество вхождений некоторого значения в списке.

```
Метод a.count(7)
# Возвращает: 2
```

9. Метод `sort()` – сортирует список в числовом и лексикографическом порядке и возвращает `None`.

```
a.sort()
# a = [1, 2, 3, 4, 5, 6, 7, 8]
# Сортирует список в числовом порядке
```

Списки также могут быть реверсированы при сортировке с помощью параметра `reverse=True` в методе `sort()`.

```
a.sort(reverse=True)
# a = [8, 7, 6, 5, 4, 3, 2, 1]
```

Если требуется сортировка по атрибутам элементов, можно использовать аргумент `key`:

```
import datetime
```

```
class Person(object):
    def __init__(self, name, birthday, height):
        self.name = name
        self.birthday = birthday
        self.height = height
```

```
def __repr__(self):
    return self.name
```

```
l = [Person("John Cena", datetime.date(1992, 9, 12), 175),
     Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
     Person("Jon Skeet", datetime.date(1991, 7, 6), 185)]
```

```
l.sort(key=lambda item: item.name)
# l: [Chuck Norris, John Cena, Jon Skeet]
```

```
l.sort(key=lambda item: item.birthday)
# l: [Chuck Norris, Jon Skeet, John Cena]
```

```
l.sort(key=lambda item: item.height)
# l: [John Cena, Chuck Norris, Jon Skeet]
```

В случае списка словарей используется та же концепция:

```
import datetime

l = [{'name': 'John Cena', 'birthday': datetime.date(1992, 9, 12), 'height': 175},
     {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28), 'height': 180},
     {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'height': 185}]

l.sort(key=lambda item: item['name'])
# l: [Chuck Norris, John Cena, Jon Skeet]

l.sort(key=lambda item: item['birthday'])
# l: [Chuck Norris, Jon Skeet, John Cena]

l.sort(key=lambda item: item['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Сортировка по подписку:

```
import datetime

l = [{'name': 'John Cena', 'birthday': datetime.date(1992, 9, 12), 'size': {'height': 175, 'weight': 100}},
     {'name': 'Chuck Norris', 'birthday': datetime.date(1990, 8, 28), 'size': {'height': 180, 'weight': 90}},
     {'name': 'Jon Skeet', 'birthday': datetime.date(1991, 7, 6), 'size': {'height': 185, 'weight': 110}}]

l.sort(key=lambda item: item['size']['height'])
# l: [John Cena, Chuck Norris, Jon Skeet]
```

Улучшенный способ сортировки с использованием attrgetter и itemgetter

Списки также можно сортировать с помощью функций attrgetter и itemgetter из модуля operator. Это может помочь улучшить читаемость и удобство повторного использования. Вот несколько примеров:

```
from operator import itemgetter, attrgetter

people = [{'name': 'chandan', 'age': 20, 'salary': 2000},
          {'name': 'chetan', 'age': 18, 'salary': 5000},
          {'name': 'guru', 'age': 30, 'salary': 3000}]
by_age = itemgetter('age')
by_salary = itemgetter('salary')

people.sort(key=by_age) #сортировка на месте по возрасту
people.sort(key=by_salary) #сортировка на месте по зарплате
```

Функции itemgetter также может быть присвоен индекс. Это полезно, если вы хотите сортировать по индексам кортежа.

```
list_of_tuples = [(1, 2), (3, 4), (5, 0)]
list_of_tuples.sort(key=itemgetter(1))
print(list_of_tuples) #[(5, 0), (1, 2), (3, 4)]
```

Используйте функцию attrgetter, если вы хотите сортировать по атрибутам объекта:

```
persons = [Person("John Cena", datetime.date(1992, 9, 12), 175),
           Person("Chuck Norris", datetime.date(1990, 8, 28), 180),
           Person("Jon Skeet", datetime.date(1991, 7, 6), 185)] # повторное использование класса Person
                                                         # из примера выше

person.sort(key=attrgetter('name')) #sort by name
by_birthday = attrgetter('birthday')
person.sort(key=by_birthday) #sort by birthday
```


10. Метод `clear()` – удаляет все элементы из списка:

```
a.clear()
# a = []
```

11. **Репликация** – умножение существующего списка на целое число приведет к созданию большего списка, состоящего из множества копий оригинала. Это может быть полезно, например, для инициализации списка:

```
b = ["blah"] * 3
# b = ["blah", "blah", "blah"]
b = [1, 3, 5] * 5
# [1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5, 1, 3, 5]
```

Будьте осторожны, если ваш список содержит ссылки на объекты (например, список списков), см. главу “Общие ошибки” – “Умножение списков и общие ссылки”.

12. **Удаление элемента** – можно удалить несколько элементов в списке, используя ключевое слово `del` и нарезку:

```
a = list(range(10))
del a[:2]
# a = [1, 3, 5, 7, 9]
del a[-1]
# a = [1, 3, 5, 7]
del a[:]
# a = []
```

13. Копирование

Назначение по умолчанию, т. е. “=”, присваивает новому имени ссылку на исходный список. То есть исходное имя и новое имя указывают на один и тот же объект списка. Изменения, сделанные в одном из них, будут отражены в другом. Зачастую это совсем не то, что вы задумали.

```
b = a
a.append(6)
# b: [1, 2, 3, 4, 5, 6]
```

Если вы хотите создать копию списка, то у вас есть несколько вариантов.

Вы можете нарезать его:

```
new_list = old_list[:]
```

Вы можете использовать встроенную функцию `list ()`:

```
new_list = list(old_list)
```

Вы можете использовать `copy.copy ()`:

```
import copy
new_list = copy.copy(old_list)      # вставляет ссылки на объекты, найденные в оригинале.
```

Это немного медленнее, чем `list ()`, потому что сначала нужно выяснить тип данных у `old_list`.

Если список содержит объекты и вы хотите их скопировать, используйте `copy.deepcopy ()`:

```
import copy
new_list = copy.deepcopy(old_list) #inserts copies of the objects found in the original.
```

Этот метод самый медленный и требующий много памяти, но иногда его использование неизбежно.

Версия Python 3.x ≥ 3.0:

`copy()` – возвращает неглубокую (поверхностную) копию списка:

```
aa = a.copy()
# aa = [1, 2, 3, 4, 5]
```

20.2. Доступ к значениям списка

Списки Python имеют нулевую индексацию и ведут себя подобно массивам в других языках.

```
lst = [1, 2, 3, 4]
lst[0] # 1
lst[1] # 2
```

Попытка доступа к индексу за пределами списка приведет к ошибке `IndexError`.

```
lst[4] # IndexError: list index out of range
```

Отрицательные индексы интерпретируются как считанные с *конца* списка.

```
lst[-1] # 4
lst[-2] # 3
lst[-5] # IndexError: list index out of range
```

Это функционально эквивалентно

```
lst[len(lst)-1] # 4
```

Списки позволяют использовать *нотацию срезов* (slice notation) в виде `lst[start:end:step]`. Вывод нотации среза – это новый список, содержащий элементы от индекса `start` до `end-1`. Если параметры опущены, то `start` по умолчанию присваивается началу списка, `end` – концу списка, а шаг (`step`) – 1:

```
lst[1:]      # [2, 3, 4]
lst[:3]      # [1, 2, 3]
lst[::2]     # [1, 3]
lst[::-1]    # [4, 3, 2, 1]
lst[-1:0:-1] # [4, 3, 2]
lst[5:8]     # [] так как начальный индекс больше длины lst, возвращает пустой список
lst[1:10]    # [2, 3, 4] то же, что и отсутствие конечного индекса
```

Учитывая это, можно вывести обратную версию списка, вызвав команду

```
lst[::-1] # [4, 3, 2, 1]
```

При использовании с отрицательной длиной начальный индекс должен быть больше конечного, иначе результатом будет пустой список.

```
lst[3:1:-1] # [4, 3]
```

Использование отрицательных шаговых индексов эквивалентно следующему коду:

```
reversed(lst)[0:2] # 0 = 1 -1
                  # 2 = 3 -1
```

Используемые индексы на 1 меньше, чем те, которые используются при отрицательной индексации, и являются обратными.

Расширенная нарезка

При нарезке списков вызывается метод `__getitem__()` объекта списка с объектом `slice`. Python имеет встроенный метод среза для создания объектов среза. Можно использовать это для хранения среза и последующего его использования следующим образом:

```
data = 'chandan purohit 22 2000' #при условии, что поля данных имеют фиксированную длину
name_slice = slice(0,19)
age_slice = slice(19,21)
salary_slice = slice(22,None)
```

```
#теперь мы можем получить более читаемые срезы
print(data[name_slice]) #chandan purohit
print(data[age_slice]) #22
print(data[salary_slice]) #2000'
```

Это может быть очень полезно для обеспечения функциональности нарезки наших объектов, переопределяя `__getitem__` в нашем классе.

20.3. Проверка списка на пустоту

Пустота списка связана со значением булевой логики False, поэтому можно не проверять `len(lst) == 0`, а просто `lst` или `not lst`:

```
lst = []
if not lst:
    print("список пуст")
# Вывод: список пуст
```

20.4. Итерирование по списку

Python поддерживает использование цикла `for` непосредственно в списке:

```
my_list = ['foo', 'bar', 'baz']
for item in my_list:
    print(item)
```

```
# Вывод: foo
# Вывод: bar
# Вывод: baz
```

Вы также можете получить информацию о положении каждого элемента:

```
for (index, item) in enumerate(my_list):
    print('The item in position {} is: {}'.format(index, item))
```

```
# Вывод: The item in position 0 is: foo
# Вывод: The item in position 1 is: bar
# Вывод: The item in position 2 is: baz
```

Другой способ итерации списка на основе значения индекса:

```
for i in range(0, len(my_list)):
    print(my_list[i])
```

```
# Вывод:
```

```
>>>
foo
bar
baz
```

Обратите внимание, что изменение элементов списка во время итерации по нему может привести к неожиданным результатам:

```
for item in my_list:
    if item == 'foo':
        del my_list[0]
    print(item)
```

```
# Вывод: foo
# Вывод: baz
```

В последнем примере мы удалили первый элемент на первой итерации, но это привело к тому, что элемент `bar` был пропущен.

20.5. Проверка наличия элемента в списке

Python очень просто проверить, находится ли элемент в списке. Для этого достаточно воспользоваться оператором `in`.

```
lst = ['test', 'twest', 'twest', 'treast']
```

```
'test' in lst
# Результат: True
```

```
'toast' in lst
# Результат: False
```

Примечание: оператор `in` на множествах работает асимптотически быстрее, чем в списках. Если вам нужно использовать его много раз в потенциально больших списках, возможно, будет целесообразно преобразовать `list` в `set` и проверить наличие элементов в `set`.

```
slst = set(lst)
'test' in slst
# Результат: True
```

20.6. Функции “any” и “all”

С помощью функции `all()` можно определить, все ли значения в итерируемом объекте имеют значение `True`.

```
nums = [1, 1, 0, 1]
all(nums)
# False
chars = ['a', 'b', 'c', 'd']
all(chars)
# True
```

Аналогично функция `any()` определяет, имеет ли одно или несколько значений в итерируемой последовательности значение `True`.

```
nums = [1, 1, 0, 1]
any(nums)
# True
vals = [None, None, None, False]
any(vals)
# False
```

Хотя в этом примере используется список, важно отметить, что эти встроенные работы работают с любым итерируемым типом данных, включая генераторы.

```
vals = [1, 2, 3, 4]
any(val > 12 for val in vals)
# False
any((val * 2) > 6 for val in vals)
# True
```

20.7. Реверсирование элементов списка

Функция `reversed` возвращает итератор к перевернутому списку:

```
In [3]: rev = reversed(numbers)

In [4]: rev
Out[4]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Обратите внимание, что список “numbers” остается неизменным для этой операции и остается в том же порядке, что и был изначально. Для реверсирования на месте можно также использовать метод `reverse`.

Вы также можете обратить список (фактически получить копию, исходный список при этом не изменяется), используя синтаксис нарезки, задав в качестве третьего аргумента (шаг) значение `-1`:

```
In [1]: numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

In [2]: numbers[::-1]
Out[2]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

20.8. Конкатенация и слияние списков

1. Простейший способ конкатенации list1 и list2:

```
merged = list1 + list2
```

2. Функция `zip` **возвращает список кортежей**, где *i*-й кортеж содержит *i*-й элемент из каждой из последовательностей аргументов или итерируемых объектов:

```
alist = ['a1', 'a2', 'a3']
```

```
blist = ['b1', 'b2', 'b3']
```

```
for a, b in zip(alist, blist):  
    print(a, b)
```

Результат:

```
# a1 b1
```

```
# a2 b2
```

```
# a3 b3
```

Если списки имеют разную длину, тогда результат будет содержать лишь столько элементов, сколько кратчайший:

```
alist = ['a1', 'a2', 'a3']
```

```
blist = ['b1', 'b2', 'b3', 'b4']
```

```
for a, b in zip(alist, blist):  
    print(a, b)
```

Результат:

```
# a1 b1
```

```
# a2 b2
```

```
# a3 b3
```

```
alist = []
```

```
len(list(zip(alist, blist)))
```

Результат:

```
# 0
```

Для расширения списков неравной длины до самого длинного с помощью элементов `None` используйте `itertools.zip_longest` (`itertools.izip_longest` в Python 2):

```
alist = ['a1', 'a2', 'a3']
```

```
blist = ['b1']
```

```
clist = ['c1', 'c2', 'c3', 'c4']
```

```
for a,b,c in itertools.zip_longest(alist, blist, clist):  
    print(a, b, c)
```

Результат:

```
# a1 b1 c1
```

```
# a2 None c2
```

```
# a3 None c3
```

```
# None None c4
```

3. Вставка в указанный индекс значений:

```
alist = [123, 'xyz', 'zara', 'abc']
```

```
alist.insert(3, [2009])
```

```
print("Final List :", alist)
```

Результат:

```
Final List : [123, 'xyz', 'zara', 2009, 'abc']
```

20.9. Длина списка

Для получения одномерной длины списка используйте функцию `len()`.

```
len(['one', 'two'])      # возвращает 2

len(['one', [2, 3], 'four'])  # возвращает 3, не 4
```

Функция `len()` также работает с строками, словарями и другими структурами данных, похожими на списки. Обратите внимание: `len()` – это встроенная функция, а не метод объекта списка.

Также отметим, что `len()` обходится в $O(1)$, то есть потребуется одинаковое количество времени, чтобы получить результат, независимо от длины списка.

20.10. Удаление дублирующихся значений в списке

Удаление дублирующихся значений в списке может быть выполнено путем преобразования списка в `set` (множество), то есть неупорядоченную коллекцию отдельных объектов. Если необходима списочная структура данных, то `set` можно преобразовать обратно в список с помощью функции `list()`:

```
names = ["aixk", "duke", "edik", "tofp", "duke"]
list(set(names))
# Результат: ['duke', 'tofp', 'aixk', 'edik']
```

Обратите внимание, что путем преобразования списка в `set` исходный порядок теряется. Для сохранения порядка списка можно использовать `OrderedDict`:

```
import collections
>>> collections.OrderedDict.fromkeys(names).keys()
# Результат: ['aixk', 'duke', 'edik', 'tofp']
```

20.11. Сравнение списков

С помощью операторов сравнения можно лексикографически сравнивать списки и другие последовательности. Оба операнда должны быть одного типа.

```
[1, 10, 100] < [2, 10, 100]
# True, потому что 1 < 2

[1, 10, 100] < [1, 10, 100]
# False, так как списки равны

[1, 10, 100] <= [1, 10, 100]
# True, так как списки равны

[1, 10, 100] < [1, 10, 101]
# True, потому что 100 < 101

[1, 10, 100] < [0, 10, 100]
# False, так как 0 < 1
```

Если один из списков содержится в начале другого, то “побеждает” самый короткий список.

```
[1, 10] < [1, 10, 100]
# True
```

20.12. Доступ к значениям во вложенном списке

Начиная с трехмерного списка:

```
alist = [[[1,2],[3,4]], [[5,6,7],[8,9,10]], [12, 13, 14]]
```

Доступ к элементам в списке:

```
print(alist[0][0][1])
#2
#Доступ ко второму элементу первого списка в первом списке
```

```
print(alist[1][1][2])
#10
#Доступ к третьему элементу второго списка во втором списке
```

Выполнение операции поддержки:

```
alist[0][0].append(11)
print(alist[0][0][2])
#11
#Добавляет 11 в конец первого списка в первом списке
```

Использование вложенных циклов для печати списка:

```
for row in alist: #один из способов перебора вложенных списков
    for col in row:
        print(col)
#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#[12, 13, 14]
```

Обратите внимание, что эта операция может использоваться в генерации списка или даже в качестве генератора для повышения эффективности, например:

```
[col for row in alist for col in row]
#[[1, 2, 11], [3, 4], [5, 6, 7], [8, 9, 10], [12, 13, 14]]
```

Не все элементы внешних списков должны быть списками:

```
alist[1].insert(2, 15)
#Inserts 15 into the third position in the second list
```

Есть еще один способ, который иногда приходится использовать:

```
for row in range(len(alist)): # менее "питоновский" способ циклического просмотра списков
    for col in range(len(alist[row])):
        print(alist[row][col])

#[1, 2, 11]
#[3, 4]
#[5, 6, 7]
#[8, 9, 10]
#15
#[12, 13, 14]
```

Использование срезов во вложенном списке:

```
print(alist[1][1:])
#[[8, 9, 10], 15, [12, 13, 14]]
# Срезы все еще работают
```

Окончательный список:

```
print(alist)
#[[[1, 2, 11], [3, 4]], [[5, 6, 7], [8, 9, 10], 15, [12, 13, 14]]]
```

20.13. Инициализация списка с фиксированным числом элементов

Для **неизменяемых** элементов (например, None, строковые литералы и т. д.):

```
my_list = [None] * 10
my_list = ['test'] * 10
```

Для **изменяемых** элементов одна и та же конструкция приведет к тому, что все элементы списка будут ссылаться на один и тот же объект, например, для множества:

```
>>> my_list=[{1}] * 10
>>> print(my_list)
[{1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}, {1}]
>>> my_list[0].add(2)
>>> print(my_list)
[{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}]
```

Вместо этого, чтобы инициализировать список с фиксированным количеством **разных изменяемых** объектов, используйте:

```
my_list=[{1} for _ in range(10)]
```

Глава 21. Генератор списков

Генераторы списков (List comprehensions) в языке Python – это лаконичные синтаксические конструкции. С их помощью можно формировать списки из других списков, применяя функции к каждому элементу списка. В следующей главе объясняется и демонстрируется использование этих выражений.

21.1. Списковые вычисления

Генератор списка (List comprehension) создает новый список путем применения выражения к каждому элементу итерируемого списка. В самом простом виде это выглядит так:

```
[<expression> for <element> in <iterable> ]
```

Также есть необязательное условие if:

```
[<expression> for <element> in <iterable> if <condition> ]
```

Каждый <element> в <iterable> подключается к <expression>, если (необязательно) <условие> имеет значение true. Все результаты сразу возвращаются в новый список. Выражения генератора оцениваются “лениво”, а генераторы списков оценивают весь итератор сразу, занимая память, пропорциональную длине итератора.

Чтобы создать список (list) квадратов целых чисел:

```
squares = [x * x for x in (1, 2, 3, 4)]
#квадраты: [1, 4, 9, 16]
```

Выражение for устанавливает x в каждое значение по очереди из (1, 2, 3, 4). Результат выражения x * x добавляется во внутренний list. Этот внутренний список присваивается переменной squares после завершения.

Помимо ускорения генераторы списков примерно эквивалентны следующему циклу for:

```
squares = []
for x in (1, 2, 3, 4):
    squares.append(x * x)
```

```
# квадраты: [1, 4, 9, 16]
```

Выражение, применяемое к каждому элементу, может быть настолько сложным, насколько это необходимо:

```
# Получить список заглавных символов из строки
[s.upper() for s in "Hello World"]
# ['H', 'E', 'L', 'L', 'O', ' ', 'W', 'O', 'R', 'L', 'D']
```



```
# Убрать все запятые с конца строки в списке
[w.strip(',') for w in ['these', 'words,', 'mostly', 'have,commas,']]
# ['these', 'words', 'mostly', 'have,commas']

# Организовать буквы в словах в алфавитном порядке
sentence = "Beautiful is better than ugly"
[''.join(sorted(word, key = lambda x: x.lower())) for word in sentence.split()]
# ['aBefiltuu', 'is', 'beertt', 'ahnt', 'gluy']
```

Условие Else

Условие `else` можно использовать в списковых включениях, но следует быть осторожным с синтаксисом. Условие `if` или `else` следует использовать перед циклом `for`, а не после:

```
# создать список символов в слове "apple", заменяя согласные на '*'
# Ex - 'apple' -> ['a', '*', '*', '*', 'e']

[x for x in 'apple' if x in 'aeiou' else '*']
# SyntaxError: invalid syntax

# При использовании if / else используйте их перед циклом
[x if x in 'aeiou' else '*' for x in 'apple']
# ['a', '*', '*', '*', 'e']
```

Обратите внимание, что здесь используется другая языковая конструкция, условное выражение, которое само по себе не является частью синтаксиса генерации, в то время как `if` после `for...in` является частью генератора списков и используется для *фильтрации* элементов из объекта итерируемого типа.

Двойная итерация

Порядок двойной итерации `[... for x in ... for y in ...]` является или естественным, или неинтуитивным. Оптимальное правило – следовать эквивалентному циклу `for`:

```
def foo(i):
    return i, i + 0.5

for i in range(3):
    for x in foo(i):
        yield str(x)

что принимает вид:

[str(x)
 for i in range(3)
 for x in foo(i)]
```

Это может быть сжато в одну строку, как `[str(x) for i in range(3) for x in foo(i)]`.

Изменение на месте и другие побочные эффекты

Прежде чем использовать генераторы списков, нужно понять разницу между функциями, вызываемыми для их побочных эффектов (*изменяющие*, или *функции на месте*), которые обычно возвращают `None`, и функциями, которые возвращают интересное значение.

Многие функции (особенно т. н. *чистые функции*) просто принимают объект и возвращают какой-то объект. *Функция на месте* изменяет существующий объект, это называется *побочным эффектом*. Другие примеры включают операции ввода и вывода, такие как `print`.

`list.sort()` сортирует список *на месте* (это означает, что он изменяет исходный список) и возвращает значение `None`. Следовательно, это не будет работать так, как ожидается, с генераторами списков:

```
[x.sort() for x in [[2, 1], [4, 3], [0, 1]]]
# [None, None, None]
```

Вместо этого `sorted()` возвращает отсортированный `list`, а не производит сортировку на месте:

```
[sorted(x) for x in [[2, 1], [4, 3], [0, 1]]]
# [[1, 2], [3, 4], [0, 1]]
```

Допускается использование генераторов списков для побочных эффектов, таких как ввод-вывод или функции на месте. Но лучше, тем не менее, использовать цикл `for`. В Python 3 это работает так:

```
[print(x) for x in (1, 2, 3)]
```

Вместо этого используйте:

```
for x in (1, 2, 3):
    print(x)
```

В некоторых ситуациях функции с побочными эффектами подходят для генераторов списков. У `random.randrange()` есть побочный эффект изменения состояния генератора случайных чисел, но он также возвращает интересное значение. Кроме того, `next()` может вызываться на итераторе.

Следующий генератор случайных значений не является чистым, но имеет смысл, поскольку генератор случайных значений сбрасывается каждый раз, когда вычисляется выражение:

```
from random import randrange
[randrange(1, 7) for _ in range(10)]
# [2, 3, 2, 1, 1, 5, 2, 4, 3, 5]
```

Пробельные символы в генераторах списков

Сложные генераторы списков могут достигать нежелательной длины или становиться менее читаемыми. Хотя в примерах это встречается реже, можно разбить генератор списков на несколько строк, например так:

```
[
    x for x
    in 'foo'
    if x not in 'bar'
]
```

21.2. Условные генераторы списков

В генератор списка можно добавить одно или несколько `if` условий для фильтрации значений.

```
[<expression> for <element> in <iterable> if <condition>]
```

Для каждого `<element>` в `<iterable>`, если `<condition>` имеет значение `True`, добавить `<expression>` (обычно это функция `<element>`) в возвращаемый список. Например, это можно использовать для извлечения только четных чисел из последовательности целых чисел:

```
[x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]
```

Приведенный выше код эквивалентен:

```
even_numbers = []
for x in range(10):
    if x % 2 == 0:
        even_numbers.append(x)

print(even_numbers)
# [0, 2, 4, 6, 8]
```

Кроме того, условный генератор списка вида `[e for x in y if c]`, где `e` и `c` являются выражениями в терминах `x`) эквивалентно `list(filter(lambda x: c, map(lambda x: e, y)))`.

Несмотря на одинаковый результат, обратите внимание на тот факт, что первый пример работает почти в два раза быстрее, чем второй.

Обратите внимание, что это совершенно не похоже на условное выражение `... if ... else ...` (иногда называемое тернарным выражением), которые можно использовать для части `<expression>` генератора списка. Рассмотрим следующий пример:

```
[x if x % 2 == 0 else None for x in range(10)]
# Результат: [0, None, 2, None, 4, None, 6, None, 8, None]
```

Здесь условное выражение – не фильтр, а оператор, определяющий значение, которое будет использоваться для элементов списка:

`<value-if-condition-is-true> if <condition> else <value-if-condition-is-false>` (`<значение-если-условие-есть-истина> if <условие> else <значение-если-условие-есть-ложь>`)

Это становится более очевидным, если объединить его с другими операторами:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in range(10)]
# Результат: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Если вы используете Python 2.7, то `xrange` может быть лучше, чем `range`, по нескольким причинам, описанным в документации к `xrange`:

```
[2 * (x if x % 2 == 0 else -1) + 1 for x in xrange(10)]
# Результат: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Приведенный выше код эквивалентен:

```
numbers = []
for x in range(10):
    if x % 2 == 0:
        temp = x
    else:
        temp = -1
    numbers.append(2 * temp + 1)
print(numbers)
# Результат: [1, -1, 5, -1, 9, -1, 13, -1, 17, -1]
```

Можно комбинировать тернарные выражения и условия `if`. Тернарный оператор работает над отфильтрованным результатом:

```
[x if x > 2 else '*' for x in range(10) if x % 2 == 0]
# Результат: ['*', '*', 4, 6, 8]
```

Того же самого нельзя было достичь, используя только тернарный оператор:

```
[x if (x > 2 and x % 2 == 0) else '*' for x in range(10)]
# Результат: ['*', '*', '*', '*', '*', 6, '*', 8, '*']
```

21.3. Избегание повторных и ресурсоемких операций с помощью условного предложения

Рассмотрим приведенный ниже код:

```
def f(x):
    import time
    time.sleep(.1) # Имитация ресурсоемкой операции
    return x**2
```

```
[f(x) for x in range(1000) if f(x) > 10]
# [16, 25, 36, ...]
```

Это приводит к двум вызовам $f(x)$ для 1000 значений x : один вызов для генерации значения, а другой для проверки условия if . Если $f(x)$ является особенно ресурсоемкой операцией, это может существенно повлиять на производительность. Но хуже всего, если вызов $f()$ несет побочные эффекты, это может привести к неожиданным результатам.

Вместо этого можно оценивать ресурсоемкую операцию только один раз для каждого значения x , генерируя промежуточные итерируемые данные (генераторное выражение) следующим образом:

```
[v for v in (f(x) for x in range(1000)) if v > 10]
# [16, 25, 36, ...]
```

или используя встроенную эквивалентную функцию `map`:

```
[v for v in map(f, range(1000)) if v > 10]
# [16, 25, 36, ...]
```

Другой способ, который может сделать код более читабельным, – поместить частичный результат (v в предыдущем примере) в итерируемый тип (например, список или кортеж), а затем выполнить над ним итерацию. Поскольку v будет единственным элементом в этом итерируемом, в результате мы имеем ссылку на вывод нашей “медленной” функции, вычисляемой только один раз:

```
[v for x in range(1000) for v in [f(x)] if v > 10]
# [16, 25, 36, ...]
```

Однако на практике логика кода может быть более сложной, и важно сохранить его читабельность. Как правило, лучше использовать отдельную функцию-генератор вместо сложной однострочной:

```
def process_prime_numbers(iterable):
    for x in iterable:
        if is_prime(x):
            yield f(x)
```

```
[x for x in process_prime_numbers(range(1000)) if x > 10]
# [11, 13, 17, 19, ...]
```

Другой способ предотвратить многократное вычисление $f(x)$ – это использовать для $f(x)$ декоратор `@functools.lru_cache()` (Python 3.2+). Поскольку вывод f для ввода x уже был вычислен один раз, второй вызов функции исходного генератора списков будет таким же быстрым, как поиск по словарю. Этот подход использует запоминание (memoization) для повышения эффективности, что сравнимо с использованием генераторных выражений.

Допустим, вы должны “сгладить” (т. е. перевести в одномерный) список:

```
l = [[1, 2, 3], [4, 5, 6], [7], [8, 9]]
```

Одним из методов может быть:

```
reduce(lambda x, y: x+y, l)
sum(l, [])
list(itertools.chain(*l))
```

Но лучшую производительность принесет генератор списка:

```
[item for sublist in l for item in sublist]
```

Сокращения, основанные на $+$ (включая подразумеваемое использование в сумме), при наличии L -подсписков с необходимостью имеют вид $O(L^2)$ – поскольку список промежуточных результатов становится все длиннее, на каждом шаге выделяется новый объект списка промежуточных результатов, и все элементы предыдущего промежуточного результата должны быть скопированы (а также несколько новых, добавленных в конце). Поэтому для простоты допустим, что у вас есть L -подсписков с I -элементами в каждом: первые I -элементы

копируются туда и обратно $L-1$ раз, вторые I -элементы $L-2$ раза и т. д.; общее количество копий равно I , умноженному на сумму x для x от 1 до L исключенного, т. е. $I*(L**2)/2$.

Генератор списка создает только один список и копирует каждый элемент (из исходного места в результирующий список) также ровно один раз.

21.4. Генератор словаря (словарь включений)

Генератор словаря (dictionary comprehension) аналогичен генератору списка, только вместо списка создает объект словаря.

Основной пример (*Версия Python 2.x ≥ 2.7*):

```
{x: x * x for x in (1, 2, 3, 4)}
# {1: 1, 2: 4, 3: 9, 4: 16}
```

еще один способ написания:

```
dict((x, x * x) for x in (1, 2, 3, 4))
# {1: 1, 2: 4, 3: 9, 4: 16}
```

Как и в случае с генератором списка, можно использовать условный оператор внутри генератора словаря, чтобы получить только те элементы словаря, которые удовлетворяют заданному критерию.

Версия Python 2.x ≥ 2.7:

```
{name: len(name) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6}
# {'Exchange': 8, 'Overflow': 8}
```

Это можно переписать с помощью генераторного выражения:

```
dict((name, len(name)) for name in ('Stack', 'Overflow', 'Exchange') if len(name) > 6)
# {'Exchange': 8, 'Overflow': 8}
```

Начало работы со словарем и использование генератора словаря как фильтра пары “ключ-значение”

Версия Python 2.x ≥ 2.7:

```
initial_dict = {'x': 1, 'y': 2}
{key: value for key, value in initial_dict.items() if key == 'x'}
# Результат: {'x': 1}
```

Переключение ключа и значения словаря (инвертирование словаря)

Если у вас есть словарь, содержащий простые хешируемые значения (дублирование значений может привести к неожиданным результатам):

```
my_dict = {1: 'a', 2: 'b', 3: 'c'}
```

и вы хотите поменять местами ключи и значения, вы можете использовать несколько подходов в зависимости от вашего стиля программирования:

- `swapped = {v: k for k, v in my_dict.items()}`
- `swapped = dict((v, k) for k, v in my_dict.items())`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(zip(my_dict.values(), my_dict.keys()))`
- `swapped = dict(map(reversed, my_dict.items()))`

```
print(swapped)
# {a: 1, b: 2, c: 3}
```

Версия Python 2.x ≥ 2.3:

Если ваш словарь велик, импортируйте `itertools` и используйте `izip` или `imap`.

Объединение словарей

Объедините словари и при необходимости переопределите старые значения с помощью вложенного генератора словаря.

```
dict1 = {'w': 1, 'x': 1}
dict2 = {'x': 2, 'y': 2, 'z': 2}

{k: v for d in [dict1, dict2] for k, v in d.items()}
# {'w': 1, 'x': 2, 'y': 2, 'z': 2}
```

Примечание: генераторы словарей были добавлены в Python 3.0 и перенесены в версию 2.7+, в отличие от генераторов списков, которые были добавлены в 2.0. Версии до 2.7 могут использовать генераторные выражения и встроенную функцию dict() для имитации генератора словаря.

21.5. Генераторы списков с вложенными циклами

Списки могут использовать вложенные циклы for. Вы можете запрограммировать любое количество вложенных циклов for внутри генератора списка, и каждый цикл for может иметь дополнительную связанную проверку на if. При этом порядок следования for-конструкций такой же, как при написании очередности вложенных for-выражений. Общая структура списочных представлений выглядит следующим образом:

```
[expression for target1 in iterable1 [if condition1]
 for target2 in iterable2 [if condition2]...
 for targetN in iterableN [if conditionN] ]
```

Например, следующий код “уплощает” список списков с использованием нескольких выражений for:

```
data = [[1, 2], [3, 4], [5, 6]]
output = []
for each_list in data:
    for element in each_list:
        output.append(element)
print(output)

# Результат: [1, 2, 3, 4, 5, 6]
```

Можно эквивалентно записать в виде генератора списка с множественными конструкциями for:

```
data = [[1, 2], [3, 4], [5, 6]]
output = [element for each_list in data for element in each_list]
print(output)
# Результат: [1, 2, 3, 4, 5, 6]
```

Как в расширенной форме, так и в генераторе списка внешний цикл идет первым.

Использование вложенных циклов в генераторе списка и компактнее, и значительно быстрее.

```
In [1]: data = [[1,2],[3,4],[5,6]]
In [2]: def f():
...: output=[]
...: for each_list in data:
...: for element in each_list:
...: output.append(element)
...: return output
In [3]: timeit f()
1000000 loops, best of 3: 1.37 µs per loop
In [4]: timeit [inner for outer in data for inner in outer]
1000000 loops, best of 3: 632 ns per loop
```

Дополнительное время на вызов приведенной выше функции примерно 140 ns.

Строчные if вкладываются таким же образом и могут находиться в любом положении после первого for:

```
data = [[1], [2, 3], [4, 5]]
output = [element for each_list in data
          if len(each_list) == 2
          for element in each_list
          if element != 5]
print(output)
# [2, 3, 4]
```

Как бы то ни было, ради удобства чтения следует рассмотреть возможность использования традиционных циклов for. Это особенно актуально, когда глубина вложенности превышает 2 уровня и/или логика понимания слишком сложна. Генераторы списка с многократным использованием вложенных циклов могут выдавать ошибки или неожиданный результат.

21.6. Генераторные выражения

Генераторные выражения очень похожи на генераторы списков. Основное отличие состоит в том, что в них не создается сразу полный набор результатов, а создается объект-генератор, над которым можно выполнять итерации.

Например, в следующем коде можно увидеть различия:

```
# генератор списка (list comprehension)
[x**2 for x in range(10)]
# Результат: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Версия Python 2.x ≥ 2.4:

```
# генераторное выражение (generator comprehension)
(x**2 for x in xrange(10))
# Результат: <generator object <genexpr> at 0x11b4b7c80>
```

Это два совершенно разных объекта.

- В результате работы генератора списка возвращается объект типа list, а в результате работы генераторной функции – объект типа generator.
- Объекты типа generator не могут быть проиндексированы, поэтому для расстановки элементов по порядку используется функция next.

Примечание: мы используем функцию xrange, поскольку она тоже создает объект-генератор. Если бы мы использовали range, то был бы создан список. Кроме того, xrange существует только в поздней версии Python 2. В Python 3 range просто возвращает генератор. Более подробную информацию можно найти в примере “Различия между функциями range и xrange”.

Версия Python 2.x ≥ 2.4:

```
g = (x**2 for x in xrange(10))
print(g[0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object has no attribute '__getitem__'
```

```
g.next() # 0
g.next() # 1
g.next() # 4
...
g.next() # 81
g.next() # Throws StopIteration Exception
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

Версия Python 3.x ≥ 3.0:

Примечание: функцию `g.next()` следует заменить на `next(g)`, а `xrange` – на `range`, поскольку в Python 3 не существуют `Iterator.next()` и `xrange()`.

Хотя оба эти варианта можно итерировать аналогичным образом:

```
for i in [x**2 for x in range(10)]:
```

```
print(i)
```

Out:

0

1

4

81

Версия Python 2.x ≥ 2.4:

```
for i in (x**2 for x in xrange(10)):
```

```
print(i)
```

Out:

0

1

4

81

Примеры использования

Выражения генератора вычисляются “лениво”, то есть генерируют и возвращают каждое значение только при итерировании генератора. Это часто бывает полезно при итерациях по большим наборам данных, позволяя избежать необходимости создания дубликата набора данных в памяти:

```
for square in (x**2 for x in range(1000000)):
```

```
# сделать что-то
```

Другим распространенным вариантом использования является отказ от итерации по всему итерируемому объекту, если в этом нет необходимости. В данном примере при каждой итерации функции `get_objects()` из удаленного API извлекается один объект. Объектов может быть несколько тысяч, их нужно извлекать по одному, а нам всего лишь нужно знать, существует ли объект, соответствующий шаблону. Используем генераторное выражение:

```
def get_objects():
```

```
    """Gets objects from an API one by one"""
```

```
    while True:
```

```
        yield get_next_item()
```

```
def object_matches_pattern(obj):
```

```
    # выполнить потенциально сложный расчет
```

```
    return matches_pattern
```



```
def right_item_exists():
    items = (object_matched_pattern(each) for each in get_objects())
    for item in items:
        if item.is_the_right_one:

            return True
    return False
```

21.7. Генераторы наборов

Генераторы наборов подобны генераторам списков и словарей, но они создают набор, т. е. неупорядоченную коллекцию уникальных элементов.

Версия Python 2.x ≥ 2.7:

```
# Набор, содержащий каждое значение в диапазоне (5):
{x for x in range(5)}
# {0, 1, 2, 3, 4}

# Набор четных чисел от 1 до 10:
{x for x in range(1, 11) if x % 2 == 0}
# {2, 4, 6, 8, 10}

# Уникальные буквенные символы в текстовой строке:
text = "When in the Course of human events it becomes necessary for one people..."
{ch.lower() for ch in text if ch.isalpha()}
# Результат: set(['a', 'c', 'b', 'e', 'f', 'i', 'h', 'm', 'l', 'o',
#               'n', 'p', 's', 'r', 'u', 't', 'w', 'v', 'y'])
```

Имейте в виду, что наборы неупорядочены. Это значит, что порядок результатов в наборе может отличаться от того, который представлен в приведенных выше примерах.

Примечание: генераторы наборов доступны с версии Python 2.7+, в отличие от генераторов списков, которые были добавлены в Python 2.0. В Python версий от 2.2 до 2.6 функция `set()` может использоваться с генераторным выражением для получения того же результата:

```
set(x for x in range(5))
# Результат: {0, 1, 2, 3, 4}
```

21.8. Рефакторинг функций `filter` и `map` в генераторы списков

Функции `filter` или `map` часто должны быть заменены генераторами списков. Гвидо ван Россум хорошо описал это в открытом письме в 2005 году:

`filter(P, S)` почти всегда записывается понятнее в виде `[x for x in S if P(x)]`, и это имеет то огромное преимущество, что наиболее часто используются предикаты, которые являются сравнениями, например, `x=42`, и использование лямбда-выражения для этого требует гораздо больше усилий от читателя (к тому же лямбда-выражение работает медленнее, чем генератор списка). Тем более это касается функции `map(F, S)`, которая превращается в `[F(x) для x в S]`. Конечно, во многих случаях вместо этого можно использовать генераторные выражения.

Следующие строки кода считаются “не питоновскими” и будут вызывать ошибки.

```
# четные числа < 10
filter(lambda x: x % 2 == 0, range(10))
# умножение каждого числа на 2
map(lambda x: 2*x, range(10))
# сумма всех элементов в списке
functools.reduce(lambda x,y: x+y, range(10))
```

Используя то, что мы узнали из предыдущей цитаты, мы можем преобразовать эти выражения в эквивалентные им генераторы списков; также удалим лямбда-функции из каждого, чтобы сделать код более читаемым.

```
# Filter:
# P(x) = x % 2 == 0
# S = range(10)
[x for x in range(10) if x % 2 == 0]

# [0, 2, 4, 6, 8]

# Map
# F(x) = 2*x
# S = range(10)
[2*x for x in range(10)]

# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Читаемость становится еще более очевидной при работе с цепочками функций. Если для удобства чтения результаты одной функции `map` или `filter` должны быть переданы в качестве результата следующей, то в простых случаях они могут быть заменены одним списком. Кроме того, из генератора списка мы можем легко понять, каков результат нашего процесса, в отличие от большой когнитивной нагрузки при понимании процесса с использованием `map` и `filter`.

```
# Map & Filter
filtered = filter(lambda x: x % 2 == 0, range(10))
results = map(lambda x: 2*x, filtered)

# List comprehension
results = [2*x for x in range(10) if x % 2 == 0]
```

Рефакторинг – краткий справочник

- Map

```
map(F, S) == [F(x) for x in S]
```

- Filter

```
filter(P, S) == [x for x in S if P(x)]
```

где `F` и `P` являются функциями, которые соответственно преобразуют входные значения и возвращают булевы значения.

21.9. Генераторы с использованием кортежей

В предложении `for` генератора списков может быть указано более одной переменной:

```
[x + y for x, y in [(1, 2), (3, 4), (5, 6)]]
# [3, 7, 11]

[x + y for x, y in zip([1, 3, 5], [2, 4, 6])]
# [3, 7, 11]
```

Это подобно обычному использованию циклов `for`:

```
for x, y in [(1,2), (3,4), (5,6)]:
    print(x+y)
# 3
# 7
# 11
```

Однако если генератор начинается с кортежа, то он должен быть заключен в круглые скобки:

```
[x, y for x, y in [(1, 2), (3, 4), (5, 6)]]
# SyntaxError: invalid syntax

[(x, y) for x, y in [(1, 2), (3, 4), (5, 6)]]
# [(1, 2), (3, 4), (5, 6)]
```

21.10. Подсчет вхождений при использовании генераторов

Когда мы хотим подсчитать количество элементов в итерируемом объекте, удовлетворяющих некоторому условию, мы можем использовать генератор для получения идиоматического синтаксиса:

```
# Подсчет чисел в списке от 1 до 1000 (range(1000)), которые четны и содержат цифру 9:
print(sum(
    1 for x in range(1000)
    if x % 2 == 0 and
    '9' in str(x)
))
```

Результат: 95

Основная концепция может быть кратко сформулирована следующим образом.

1. Проведем итерации над элементами из range(1000).
2. Объединим все необходимые условия if.
3. Используем 1 (единицу) в качестве *выражения* для возврата 1 для каждого элемента, который соответствует условиям.
4. Суммируем все 1, чтобы определить количество элементов, которые удовлетворяют условиям.

Примечание: здесь мы не собираем все единицы в списке (обратите внимание на отсутствие квадратных скобок), но передаем их непосредственно в функцию sum. Это *выражение генератора*, похожее на генератор.

21.11. Изменение типов в списке

Количественные данные часто считываются в виде строк, которые перед обработкой необходимо преобразовать к числовым типам. Типы всех элементов списка могут быть преобразованы либо с помощью генератора списка, либо с помощью функции map().

```
# Преобразование списка строк в целые числа.
items = ["1", "2", "3", "4"]
[int(item) for item in items]
# Результат: [1, 2, 3, 4]
```

```
# Преобразование списка строк в float.
items = ["1", "2", "3", "4"]
map(float, items)
# Результат: [1.0, 2.0, 3.0, 4.0]
```

21.12. Вложенные генераторы списков

Вложенные генераторы списков, в отличие от генераторов списков с вложенными циклами, представляют собой генераторы списков внутри генераторов списков. Начальным выражением может быть любое произвольное выражение, в том числе и другой генератор списка.

```
# Генератор списка с вложенным циклом
[x + y для x в [1, 2, 3] для y в [3, 4, 5]]
#Результат: [4, 5, 6, 5, 6, 7, 6, 7, 8]
```

```
# Вложенный генератор списка
[[x + y для x в [1, 2, 3]] для y в [3, 4, 5]]
#Результат: [[4, 5, 6], [5, 6, 7], [6, 7, 8]]
```

Второй пример эквивалентен коду

```
l = []
for y in [3, 4, 5]:
```

```
temp = []
for x in [1, 2, 3]:
    temp.append(x + y)
l.append(temp)
```

Одним из примеров использования вложенного генератора является транспонирование матрицы.

```
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]

[[row[i] for row in matrix] for i in range(len(matrix))]

# [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Как и в случае с вложенными циклами `for`, нет ограничений на глубину вложения генераторов.

```
[[[i + j + k for k in 'cd'] for j in 'ab'] for i in '12']
# Результат: [[[ '1ac', '1ad'], ['1bc', '1bd']], [['2ac', '2ad'], ['2bc', '2bd']]]
```

21.13. Итерация двух и более списков одновременно внутри генератора списка

Для одновременной итерации более двух списков внутри *генератора списков* можно использовать функцию `zip()`:

```
>>> list_1 = [1, 2, 3, 4]
>>> list_2 = ['a', 'b', 'c', 'd'].
>>> list_3 = ['6', '7', '8', '9']

# Два списка
>>> [(i, j) for i, j in zip(list_1, list_2)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]

# Три списка
>>> [(i, j, k) for i, j, k in zip(list_1, list_2, list_3)]
[(1, 'a', '6'), (2, 'b', '7'), (3, 'c', '8'), (4, 'd', '9')]
# и т. д. ...
```

Глава 22. Срезы списков (выделение частей списков)

22.1. Использование третьего аргумента “шаг”

```
lst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

lst[::2]
# Результат: ['a', 'c', 'e', 'g'].

lst[::3]
# Результат: ['a', 'd', 'g'].
```

22.2. Выбор подписка из списка

```
lst = ['a', 'b', 'c', 'd', 'e']
```

```
lst[2:4]
```

```
# Результат: ['c', 'd'].
```

```
lst[2:]
```

```
# Результат: ['c', 'd', 'e'].
```

```
lst[:4]
```

```
# Результат: ['a', 'b', 'c', 'd'].
```

22.3. Реверсирование списка с помощью среза

```
a = [1, 2, 3, 4, 5]
```

```
# прохождение по списку в обратном порядке (шаг=-1)
```

```
b = a[::-1]
```

```
# встроенный метод реверсирования списка 'a'
```

```
a.reverse()
```

```
if a == b:
```

```
    print(True)
```

```
print(b)
```

```
# Результат:
```

```
# True
```

```
# [5, 4, 3, 2, 1]
```

22.4. Смещение списка с помощью среза

```
def shift_list(array, s):
```

```
    """Сдвигает элементы списка влево или вправо.
```

```
    Аргументы:
```

```
    array (массив) – список для сдвига
```

```
    s - величина сдвига списка ('+' сдвиг вправо, '-' сдвиг влево)
```

```
    Возвращает:
```

```
    shifted_array – сдвинутый список
```

```
    # вычислить фактическую величину сдвига (например, 11 → 1, если длина массива равна 5)
```

```
    s %= len(array)
```

```
    # изменить направление сдвига для большей интуитивности
```

```
    s *= -1
```

```
    # сдвиг массива со срезом списка
```

```
    shifted_array = array[s:] + array[:s]
```

```
    return shifted_array
```

```
my_array = [1, 2, 3, 4, 5]
```

```
# отрицательные числа
```

```
shift_list(my_array, -7)
```

```
>>> [3, 4, 5, 1, 2]
```

```
# отсутствие сдвига на числах, равных размеру массива
shift_list(my_array, 5)
>>> [1, 2, 3, 4, 5]
```

```
# работает для положительных чисел
shift_list(my_array, 3)
>>> [3, 4, 5, 1, 2]
```

Глава 23. Метод groupby()

Параметр **Подробности**

Итерируемое (iterable) Любой итерируемый тип данных в Python

Ключ Функция (критерий), по которой группируется итерируемое

В Python метод `itertools.groupby()` позволяет разработчикам группировать значения итерируемого класса на основе заданного свойства в другой итерируемый набор значений.

23.1. Пример

В этом примере мы видим, что происходит при использовании разных типов итерабельных данных.

```
things = [("animal", "bear"), ("animal", "duck"), ("plant", "cactus"), ("vehicle", "harley"), \
          ("vehicle", "speed boat"), ("vehicle", "school bus")]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Результаты:

```
{'animal': [('animal', 'bear'), ('animal', 'duck')],
'plant': [('plant', 'cactus')],
'vehicle': [('vehicle', 'harley'),
('vehicle', 'speed boat'),
('vehicle', 'school bus')]}
```

Приведенный ниже пример по сути такой же, как и предыдущий. Разница лишь в том, что все кортежи заменены на списки.

```
things = [["animal", "bear"], ["animal", "duck"], ["vehicle", "harley"], ["plant", "cactus"], \
          ["vehicle", "speed boat"], ["vehicle", "school bus"]]
dic = {}
f = lambda x: x[0]
for key, group in groupby(sorted(things, key=f), f):
    dic[key] = list(group)
dic
```

Результаты:

```
{'animal': [['animal', 'bear'], ['animal', 'duck']],
'plant': [['plant', 'cactus']],
'vehicle': [['vehicle', 'harley'],
['vehicle', 'speed boat'],
['vehicle', 'school bus']]}
```

23.2. Пример 2

Данный пример иллюстрирует, как выбирается ключ по умолчанию, если ничего не указано.

```
c = groupby(['goat', 'dog', 'cow', 1, 1, 2, 3, 11, 10, ('persons', 'man', 'woman')])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Результаты:

```
{1: [1, 1],
 2: [2],
 3: [3],
 ('persons', 'man', 'woman'): [('persons', 'man', 'woman')],
 'cow': ['cow'],
 'dog': ['dog'],
 10: [10],
 11: [11],
 'goat': ['goat']}
```

Обратите внимание, что кортеж в целом считается одним ключом в этом списке.

23.3. Пример 3

Обратите внимание, что в этом примере объекты 'mulato' и 'camel' не отображаются в результате. Отображается только последний элемент с указанным ключом. Последний результат для c фактически стирает два предыдущих результата. Но затем посмотрите на новую версию, в которой данные отсортированы первыми по тому же ключу.

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
c = groupby(list_things, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Результаты:

```
{'c': ['camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Отсортированная версия:

```
list_things = ['goat', 'dog', 'donkey', 'mulato', 'cow', 'cat', ('persons', 'man', 'woman'), \
               'wombat', 'mongoose', 'malloo', 'camel']
sorted_list = sorted(list_things, key = lambda x: x[0])
print(sorted_list)
print()
c = groupby(sorted_list, key=lambda x: x[0])
dic = {}
for k, v in c:
    dic[k] = list(v)
dic
```

Результаты:

```
['cow', 'cat', 'camel', 'dog', 'donkey', 'goat', 'mulato', 'mongoose', 'malloo', ('persons', 'man', 'woman'), 'wombat']
```

```
{'c': ['cow', 'cat', 'camel'],
 'd': ['dog', 'donkey'],
 'g': ['goat'],
 'm': ['mulato', 'mongoose', 'malloo'],
 'persons': [('persons', 'man', 'woman')],
 'w': ['wombat']}
```

Глава 24. Связные списки

Связный список представляет собой набор узлов, каждый из которых состоит из ссылки и значения. Узлы объединяются в последовательность с помощью своих ссылок. Связные списки могут использоваться для реализации более сложных структур данных, таких как списки, стеки, очереди и ассоциативные массивы.

24.1. Пример односвязного списка

В этом примере реализован связный список со многими из тех методов, что имеются во встроенном объекте списка (list).

```
class Node:
    def __init__(self, val):
        self.data = val
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, val):
        self.data = val

    def setNext(self, val):
        self.next = val

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        """Check if the list is empty""" # Проверка, пуст ли список
        return self.head is None

    def add(self, item):
        """Add the item to the list""" # Добавить элемент в список
        new_node = Node(item)
        new_node.setNext(self.head)
        self.head = new_node

    def size(self):
```



```

    """Return the length/size of the list""" # Возврат длины/размера списка
    count = 0
    current = self.head
    while current is not None:
        count += 1
        current = current.getNext()
    return count

def search(self, item):
    """Search for item in list. If found, return True. If not found, return False"""
    # Поиск элемента в списке. Если он найден, возвращается True. Если не найден,
    # возвращается False
    current = self.head
    found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            current = current.getNext()
    return found

def remove(self, item):
    """Remove item from list. If item is not found in list, raise ValueError"""
    # Удалить элемент из списка. Если элемент не найден в списке, возникает ошибка ValueError
    current = self.head
    previous = None
    found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            previous = current
            current = current.getNext()
    if found:
        if previous is None:
            self.head = current.getNext()
        else:
            previous.setNext(current.getNext())
    else:
        raise ValueError
        print 'Value not found.'

def insert(self, position, item):
    """
    Insert item at position specified. If position specified is
    out of bounds, raise IndexError # Вставить элемент в указанную позицию. Если указанная
    позиция выходит за границы, то возникает ошибка IndexError
    """
    if position > self.size() - 1:
        raise IndexError
        print "Index out of bounds."
    current = self.head
    previous = None
    pos = 0
    if position is 0:
        self.add(item)
    else:
        new_node = Node(item)
        while pos < position:
            pos += 1

```

```

    previous = current
    current = current.getNext()
    previous.setNext(new_node)
    new_node.setNext(current)

```

```

def index(self, item):

```

```

    """Return the index where item is found.

```

```

    If item is not found, return None. # Возвращает индекс, по которому найден элемент. Если
    элемент не найден, возвращается None

```

```

    """
    current = self.head
    pos = 0
    found = False
    while current is not None and not found:
        if current.getData() is item:
            found = True
        else:
            current = current.getNext()
            pos += 1
    if found:
        pass
    else:
        pos = None
    return pos

```

```

def pop(self, position = None):

```

```

    """If no argument is provided, return and remove the item at the head.

```

```

    If position is provided, return and remove the item at that position.

```

```

    If index is out of bounds, raise IndexError

```

```

    """
    # Если аргумент не указан, то возвращается и удаляется элемент, находящийся в головной
    части. Если указан аргумент position, то возвращается и удаляется элемент в этой позиции.
    Если индекс выходит за границы, то возникает ошибка IndexError

```

```

    if position > self.size():
        print 'Index out of bounds'
        raise IndexError

```

```

    current = self.head
    if position is None:
        ret = current.getData()
        self.head = current.getNext()
    else:
        pos = 0
        previous = None
        while pos < position:
            previous = current
            current = current.getNext()
            pos += 1
        ret = current.getData()
        previous.setNext(current.getNext())
    print ret
    return ret

```

```

def append(self, item):

```

```

    """Append item to the end of the list""" # Добавить элемент в конец списка

```

```

    current = self.head
    previous = None
    pos = 0
    length = self.size()

```

```

while pos < length:
    previous = current
    current = current.getNext()
    pos += 1
new_node = Node(item)
if previous is None:
    new_node.setNext(current)
    self.head = new_node
else:
    previous.setNext(new_node)

def printList(self):
    """Print the list""" # Вывод списка
    current = self.head
    while current is not None:
        print current.getData()
        current = current.getNext()

```

Функции использования аналогичны функциям встроенного списка.

```

ll = LinkedList()
ll.add('I')
ll.add('H')
ll.insert(1,'e')
ll.append('l')
ll.append('o')
ll.printList()

```

```

H
e
l
l
o

```

Глава 25. Узел связанного списка

25.1. Написание простого узла связанного списка на языке Python

Связный список – это либо:

- пустой список, представленный значением None, или
- узел, содержащий объект-носитель (cargo object) и ссылку на связный список.

```
#!/usr/bin/env python
```

```

class Node:
    def __init__(self, cargo=None, next=None):
        self.car = cargo
        self.cdr = next
    def __str__(self):
        return str(self.car)

def display(lst):
    if lst:
        w("%s " % lst)
        display(lst.cdr)
    else:
        w("nil\n")

```

Глава 26. Фильтр

Параметр Подробности

Функция *вызываемая функция*, определяющая или условие, или значение None, использующая затем функцию тождества для фильтрации (*только позиционно*)

Итерируемое итерируемый объект, который будет отфильтрован (*только позиционно*) (iterable)

26.1. Основное использование фильтра

Фильтр отбрасывает элементы последовательности на основе некоторых критериев:

```
names = ['Fred', 'Wilma', 'Barney']
```

```
def long_name(name):
    return len(name) > 5
```

Python 2.x 2.0:

```
filter(long_name, names)
# Результат: ['Barney']
```

```
[name for name in names if len(name) > 5] # эквивалентный генератор списка
# Результат: ['Barney']
```

```
from itertools import ifilter
ifilter(long_name, names)           # как генератор (аналогично встроенному фильтру Python 3.x)
# Результат: <itertools.ifilter at 0x4197e10>
list(ifilter(long_name, names))     # эквивалентно фильтру со списками
# Результат: ['Barney']
```

```
(name for name in names if len(name) > 5) # эквивалентное генераторное выражение
# Результат: <generator object <genexpr> at 0x0000000003FD5D38>
```

Python 2.x 2.6

```
# Помимо опций для старых версий Python 2.x существует функция future_builtin:
from future_builtins import filter
filter(long_name, names)           # идентична itertools.ifilter
# Результат: <itertools.ifilter at 0x3eb0ba8>
```

Python 3.x 3.0

```
filter(long_name, names)           # возвращает генератор
# Результат: <filter at 0x1fc6e443470>
list(filter(long_name, names))     # приведение к списку
# Результат: ['Barney']
```

```
(name for name in names if len(name) > 5) # equivalent generator expression
# Результат: <generator object <genexpr> at 0x000001C6F49BF4C0>
```

26.2. Фильтр без функции

Если параметр функции – None, тогда будет использоваться функция тождества:

```
list(filter(None, [1, 0, 2, [], 'a'])) # отбрасывает 0, [] and ''
# Результат: [1, 2, 'a']
```

Версия Python 2.x ≥ 2.0.1:

```
[i for i in [1, 0, 2, [], "a"] if i] # эквивалентный генератор списка
```

Версия Python 3.x ≥ 3.0.0:

```
(i for i in [1, 0, 2, [], "a"] if i) # эквивалентное генераторное выражение
```

26.3. Фильтр для проверки на “короткое замыкание” (вычисления по короткой схеме)

Функции `filter` (Python 3.x) и `ifilter` (Python 2.x) возвращают генератор, таким образом они могут быть очень удобны при создании тестов на “короткое замыкание” типа `or` или `and`:

Версия Python 2.x ≥ 2.0.1:

```
# не рекомендуется для реального использования, но позволяет сократить пример:
from itertools import ifilter as filter
```

Версия Python 2.x ≥ 2.6.1:

```
from future_builtins import filter
```

Чтобы найти первый элемент, который меньше 100:

```
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple))
    return name_value_tuple[1] < 100
next(filter(find_something_smaller_than, car_shop))
# Вывод: Check Toyota, 1000$
#       Check rectangular tire, 80$
# Результат: ('rectangular tire', 80)
```

Функция `next` дает следующий (в данном случае первый) элемент и, следовательно, является причиной “короткого замыкания”.

26.4. Дополняющая функция: `filterfalse`, `ifilterfalse`

В модуле `itertools` есть дополняющая функция для `filter`:

Версия Python 2.x ≥ 2.0.1:

```
# не рекомендуется для реального использования, но позволяет сохранить пример и для
Python 2.x, и для Python 3.x
from itertools import ifilterfalse as filterfalse
```

Версия Python 3.x ≥ 3.0.0:

```
from itertools import filterfalse
```

которая работает точно так же, как `filter` у генератора, но сохраняет только элементы, которые имеют ложное значение (`False`):

```
# Использование без функции (None):
list(filterfalse(None, [1, 0, 2, [], "a"])) # отбрасывает 1, 2, 'a'
# Результат: [0, [], ""]
```

```
# Использование с функцией
names = ['Fred', 'Wilma', 'Barney']
```

```
def long_name(name):
    return len(name) > 5
```

```
list(filterfalse(long_name, names))
# Результат: ['Fred', 'Wilma']

# "Короткое замыкание":
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
def find_something_smaller_than(name_value_tuple):
    print('Check {0}, {1}$'.format(*name_value_tuple))
    return name_value_tuple[1] < 100
next(filterfalse(find_something_smaller_than, car_shop))
# Вывести: Check Toyota, 1000$
# Результат: ('Toyota', 1000)

# Использование эквивалентного генератора:
car_shop = [('Toyota', 1000), ('rectangular tire', 80), ('Porsche', 5000)]
generator = (car for car in car_shop if not car[1] < 100)
next(generator)
```

Глава 27. Модуль “Heapq”

27.1. Самые большие и самые маленькие предметы в коллекции

Для нахождения наибольших элементов в коллекции в модуле `heapq` есть функция `nlargest`, которой передаются два аргумента: первый – количество элементов, которые мы хотим получить, второй – имя коллекции:

```
import heapq

numbers = [1, 4, 2, 100, 20, 50, 32, 200, 150, 8]
print(heapq.nlargest(4, numbers)) # [200, 150, 100, 50]
```

Аналогично для нахождения наименьших элементов в коллекции мы используем функцию `nsmallest`:

```
print(heapq.nsmallest(4, numbers)) # [1, 2, 4, 8]
```

Для сложных структур данных функции `nlargest` и `nsmallest` принимают дополнительный аргумент (ключевой параметр). В следующем примере показано использование свойства `age` для получения самых старых и самых молодых людей из словаря `people`:

```
people = [
    {'firstname': 'John', 'lastname': 'Doe', 'age': 30},
    {'firstname': 'Jane', 'lastname': 'Doe', 'age': 25},
    {'firstname': 'Janie', 'lastname': 'Doe', 'age': 10},
    {'firstname': 'Jane', 'lastname': 'Roe', 'age': 22},
    {'firstname': 'Johnny', 'lastname': 'Doe', 'age': 12},
    {'firstname': 'John', 'lastname': 'Roe', 'age': 45}]
```

```
oldest = heapq.nlargest(2, people, key=lambda s: s['age'])
print(oldest)
# Результат: [{'firstname': 'John', 'age': 45, 'lastname': 'Roe'}, {'firstname': 'John', 'age': 30, 'lastname': 'Doe'}]
```

```
youngest = heapq.nsmallest(2, people, key=lambda s: s['age'])
print(youngest)
# Результат: [{'firstname': 'Janie', 'age': 10, 'lastname': 'Doe'}, {'firstname': 'Johnny', 'age': 12, 'lastname': 'Doe'}]
```

27.2. Наименьший элемент в коллекции

Самым интересным свойством `heapq` является то, что его наименьший элемент всегда является первым элементом: `heapq[0]`.

```
import heapq

numbers = [10, 4, 2, 100, 20, 50, 32, 200, 150, 8]

heapq.heapify(numbers)
print(numbers)
# Результат: [2, 4, 10, 100, 8, 50, 32, 200, 150, 20]

heapq.heappop(numbers) # 2
print(numbers)
# Результат: [4, 8, 10, 100, 20, 50, 32, 200, 150]

heapq.heappop(numbers) # 4
print(numbers)
# Результат: [8, 20, 10, 100, 150, 50, 32, 200]
```

Глава 28. Кортеж (tuple)

Кортеж – это неизменяемый список значений. Кортежи являются одним из самых простых и распространенных типов коллекций в Python и могут быть созданы с помощью оператора запятой (`value = 1, 2, 3`).

28.1. Кортеж

Синтаксически кортеж представляет собой список значений, разделенных запятыми:

```
t = 'a', 'b', 'c', 'd', 'e'
```

Хотя это и необязательно, обычно принято заключать кортеж в круглые скобки:

```
t = ('a', 'b', 'c', 'd', 'e')
```

Создание пустого кортежа при помощи круглых скобок:

```
t0 = ()
type(t0)      # <type 'tuple'>
```

Чтобы создать кортеж с одним элементом, необходимо добавить заключительную запятую:

```
t1 = 'a',
type(t1)      # <type 'tuple'>
```

Обратите внимание, что одиночное значение в скобках не является кортежем:

```
t2 = ('a')
type(t2)      # <type 'str'>
```

Для создания одноэлементного кортежа необходимо использовать запятую в конце:

```
t2 = ('a',)
type(t2)      # <type 'tuple'>
```

Обратите внимание, что для одноэлементных кортежей рекомендуется (см. в руководстве PEP8 про использование закрывающих запятых) использовать круглые скобки. Кроме того, после запятой не должно быть пробелов (см. PEP8 о пробельных символах).

```
t2=('a',)      # нотация одобрена PEP8
t2='a',        # эта нотация не рекомендуется PEP8
t2=('a',)      # эта нотация не рекомендуется PEP8
```

Другим способом создания кортежа является встроенная функция `tuple`.

```
t = tuple('lupins')
print(t)      #('l', 'u', 'p', 'i', 'n', 's')
t = tuple(range(3))
print(t)      # (0, 1, 2)
```

Эти примеры основаны на материалах из книги “Think Python” Аллена В. Дауни.

28.2. Кортежи являются неизменяемыми

Одно из основных различий между списками (`list`) и кортежами (`tuple`) в Python заключается в том, что кортежи являются неизменяемыми, то есть после инициализации кортежа в него нельзя добавлять или изменять элементы. Например:

```
>>> t = (1, 4, 9)
>>> t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Аналогично кортежи не имеют методов `.append` и `.extend`, как у списков. Использование оператора `+=` возможно, но при этом изменяется привязка переменной, а не сам кортеж:

```
>>> t = (1, 2)
>>> q = t
>>> t += (3, 4)
>>> t
(1, 2, 3, 4) #output
>>> q
(1, 2) #output
```

Будьте осторожны при размещении изменяемых объектов, таких как списки, внутри кортежей. Это может привести к очень запутанным результатам при их изменении. Например:

```
>>> t = (1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3]) #output
>>> t[3] += [4, 5]
```

Это вызовет ошибку и повлечет изменение содержимого списка внутри кортежа:

```
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, 3, [1, 2, 3, 4, 5])
```

Вы можете использовать оператор `+=` для “добавления” в кортеж – это создает новый кортеж с новым элементом, который вы “добавили”, и присваивается его текущей переменной; старый кортеж не изменяется, но заменяется!

Это позволяет избежать преобразования в список и из списка, но работает медленно и является плохой практикой, особенно если вы собираетесь “добавлять” несколько раз.

28.3. Упаковка и распаковка кортежей

Кортежи в Python – это значения, разделенные запятыми. Окружающие круглые скобки для ввода кортежей необязательны, поэтому два примера

```
a = 1, 2, 3      # a – кортеж (1, 2, 3)
```

и

```
a = (1, 2, 3)    # a – кортеж (1, 2, 3)
```

эквивалентны. Присваивание `a = 1, 2, 3` также называется *упаковкой*, поскольку оно объединяет (упаковывает) значения в кортеж.

Обратите внимание, что кортеж с одним значением также является кортежем. Чтобы указать Python, что переменная является кортежем, а не одним значением, можно использовать запятую в конце:

```
a = 1    # a имеет значение 1
a = 1,   # a – это кортеж (1,)
```

Запятая нужна также, если вы используете скобки

```
a = (1,) # a – это кортеж (1,)
a = (1)  # a имеет значение 1 и не является кортежем
```

Для распаковки значений из кортежа и выполнения множественного присваивания используйте

```
# распаковка или множественное присваивание
x, y, z = (1, 2, 3)
# x == 1
# y == 2
# z == 3
```

Символ `_` может быть использован в качестве имени одноразовой (“выбрасываемой”) переменной, если нужны только некоторые элементы кортежа, выступая в роли заполнителя:

```
a = 1, 2, 3, 4
_, x, y, _ = a
# x == 2
# y == 3
```

Одноэлементные кортежи:

```
x, = 1,   # x – это значение 1
x = 1,   # x – это кортеж (1,)
```

В Python 3 целевая переменная с префиксом `*` может быть использована в качестве *универсальной переменной* (catch-all variable):

```
first, *more, last = (1, 2, 3, 4, 5)
#first == 1
#more == [2, 3, 4]
#last == 5
```

28.4. Встроенные функции кортежей

Кортежи поддерживают следующие встроенные функции:

Сравнение

Если элементы имеют одинаковый тип, Python выполняет сравнение и возвращает результат. Если элементы разного типа, то проверяется, являются ли они числами.

- Если числа, то выполняется сравнение.
- Если один из элементов является числом, то возвращается другой элемент.
- В противном случае типы сортируются в алфавитном порядке.

Если мы дошли до конца одного из списков, то более длинный список “больше”. Если оба списка одинаковы, то возвращается 0.

```
tuple1 = ('a', 'b', 'c', 'd', 'e')
tuple2 = ('1', '2', '3')
tuple3 = ('a', 'b', 'c', 'd', 'e')

cmp(tuple1, tuple2)    # 1
cmp(tuple2, tuple1)    # -1
cmp(tuple1, tuple3)    # 0
```

Длина кортежа

Функция `len` возвращает общую длину кортежа

```
len(tuple1)
# 5
```

Максимальное значение кортежа

Функция `max` возвращает элемент кортежа с максимальным значением

```
max(tuple1)    #'e'
max(tuple2)    #'3'
```

Минимальное значение кортежа

Функция `min` возвращает элемент из кортежа с минимальным значением

```
min(tuple1)    #'a'
min(tuple2)    #'1'
```

Преобразование списка в кортеж

Встроенная функция `tuple` преобразует список в кортеж

```
list = [1,2,3,4,5]
tuple(list)
>>>Out: (1, 2, 3, 4, 5)
```

Конкатенация кортежей

Используйте `+` для конкатенации двух кортежей

```
tuple1 + tuple2
>>>Out:('a', 'b', 'c', 'd', 'e', '1', '2', '3')
```

28.5. Кортежи являются поэлементно хешируемыми и сравниваемыми

```
hash( (1, 2) )           # ok
hash( ([], {"hello"}) ) # не подходит, поскольку списки и множества не хешируемы
```

Таким образом, кортеж можно поместить внутри набора (`set`) или в качестве ключа в словарь (`dict`) только тогда, когда это можно сделать с каждым из его элементов.

```
{ (1, 2) }           # ok
{ ([], {"hello"}) } # не подходит
```

28.6. Индексирование кортежей

```
x = (1, 2, 3)
x[0] # 1
x[1] # 2
x[2] # 3
x[3] # IndexError: индекс кортежа выходит за пределы диапазона
```

Индексирование с отрицательными числами будет начинаться с последнего элемента как `-1`:

```
x[-1] # 3
x[-2] # 2
x[-3] # 1
x[-4] # IndexError: индекс кортежа выходит за пределы диапазона
```

Индексирование диапазона элементов:

```
print(x[:-1])    # (1, 2)
print(x[-1:])    # (3,)
print(x[1:3])    # (2, 3)
```

28.7. Реверсирование элементов

Обратим элементы в кортеже:

```
colors = "red", "green", "blue"
rev = colors[::-1]
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Также можно использовать функцию `reversed` (дает итерируемый объект, который преобразуется в кортеж):

```
rev = tuple(reversed(colors))
# rev: ("blue", "green", "red")
colors = rev
# colors: ("blue", "green", "red")
```

Глава 29. Основы ввода и вывода данных

29.1. Использование функции вывода "print"

Версия Python 3.x ≥ 3.0:

В Python 3 функциональность вывода на экран представлена в виде функции:

```
print("This string will be displayed in the output")
# This string will be displayed in the output
```

```
print("You can print \n escape characters too.")
# You can print escape characters too.
```

Версия Python 2.x ≥ 2.3:

В Python 2 `print` изначально был оператором, как показано ниже:

```
print "This string will be displayed in the output"
# This string will be displayed in the output
print "You can print \n escape characters too."
# You can print escape characters too.
```

Примечание: использование `from __future__ import print_function` в Python 2 позволит пользователям использовать функцию `print()` так же, как и в коде Python 3. Это доступно только для версии Python 2.6 и выше.

29.2. Ввод из файла

Входные данные также могут быть считаны из файлов. Файлы можно открывать с помощью встроенной функции `open`. Использование синтаксиса `with <command> as <name>` (называемого "менеджером контекста") позволяет очень просто использовать функцию `open` и получать доступ к файлу:

```
with open('somefile.txt', 'r') as fileobj:
    # напишите здесь код, используя fileobj
```

Это гарантирует, что при выходе из блока выполнение кода автоматически закроет файл.

Файлы могут быть открыты в разных режимах. В приведенном выше примере файл открывается только для чтения. Чтобы открыть существующий файл только для чтения, используйте `r`. Если вы хотите открыть файл для двоичного чтения, используйте `rb`. Для добав-

ления данных в существующий файл используйте `a`. Используйте `w`, чтобы создать файл или перезаписать существующий файл с тем же именем. С помощью `g+` можно открыть файл как для чтения, так и для записи. Первым аргументом `open()` является имя файла, вторым – режим. Если параметр `mode` оставить пустым, то по умолчанию он будет равен `r`.

```
# создадим файл примера:
with open('shoppinglist.txt', 'w') as fileobj:
    fileobj.write('tomato\npasta\ngarlic')

with open('shoppinglist.txt', 'r') as fileobj:
    # тот метод создает список, в котором каждая строка
    # файла является элементом списка
    lines = fileobj.readlines()

print(lines)
# ['tomato\n', 'pasta\n', 'garlic']

with open('shoppinglist.txt', 'r') as fileobj:
    # здесь мы считываем все содержимое в одну строку:
    content = fileobj.read()
    # получаем список строк, как и в предыдущем примере:
    lines = content.split('\n')

print(lines)
# ['tomato', 'pasta', 'garlic']
```

Если размер файла небольшой, то можно безопасно считать все содержимое файла в память. Если же файл очень большой, то часто лучше считывать построчно или по частям, а входные данные обрабатывать в том же цикле. Для этого:

```
with open('shoppinglist.txt', 'r') as fileobj:
    # this method reads line by line:
    lines = []
    for line in fileobj:
        lines.append(line.strip())
```

При чтении файлов учитывайте характерные для операционной системы символы перевода строки. Хотя `for line in fileobj` автоматически удаляет их, всегда безопасно вызывать `strip()`, как показано выше.

Открытые файлы (`fileobj` в приведенных выше примерах) всегда указывают на определенное место в файле. Когда файл открыт впервые, дескриптор (указатель) указывает на самое начало файла, на позицию 0. Дескриптор файла может отображать текущее положение при помощи команды `tell`:

```
fileobj = open('shoppinglist.txt', 'r')
pos = fileobj.tell()
print('We are at %u.' % pos) # We are at 0.
```

После прочтения всего содержимого позиция будет указана в конце файла:

```
content = fileobj.read()
end = fileobj.tell()
print('This file was %u characters long.' % end)
# This file was 22 characters long.
fileobj.close()
```

Позиция может быть установлена в любое необходимое положение:

```
fileobj = open('shoppinglist.txt', 'r')
fileobj.seek(7)
pos = fileobj.tell()
print('We are at character #%u.' % pos)
```

Кроме того, во время данного вызова можно прочитать любую длину из содержимого файла во время данного вызова. Для этого необходимо передать аргумент для `read()`. Когда `read()` вызывается без аргументов, будет прочитано все содержимое файла до конца. Если передать аргумент, то будет прочитано указанное количество байтов или символов, в зависимости от режима (`rb` и `r` соответственно):

```
# считывает следующие 4 символа
# начиная с текущей позиции
next4 = fileobj.read(4)
# что у нас есть?
print(next4) # 'abcd'
# где мы сейчас находимся?
pos = fileobj.tell()
print('We are at %u.' % pos) # Мы находимся на позиции 11, так как были на 7, и прочитали 4
символа

fileobj.close()
```

Продemonстрируем разницу между символами и байтами:

```
with open('shoppinglist.txt', 'r') as fileobj:
    print(type(fileobj.read())) # <class 'str'>

with open('shoppinglist.txt', 'rb') as fileobj:
    print(type(fileobj.read())) # <class 'bytes'>
```

29.3. Чтение из `stdin`

Программы на Python могут читать из конвейеров Unix. Приведем простой пример чтения из `stdin`:

```
import sys

for line in sys.stdin:
    print(line)
```

Имейте в виду, что `sys.stdin` представляет собой поток. Это означает, что цикл `for` завершится только после завершения потока.

Теперь вы можете направить вывод другой программы в вашу программу на Python следующим образом:

```
$ cat myfile | python myprogram.py
```

В этом примере `cat myfile` может быть любой командой Unix, которая выводит в `stdout`. В качестве альтернативы можно использовать модуль `fileinput`:

```
import fileinput
for line in fileinput.input():
    process(line)
```

29.4. Использование функций `input()` и `raw_input()`

Версия Python 2.x ≥ 2.3:

Функция `raw_input` будет ждать, пока пользователь введет текст, а затем вернет результат в виде строки.

```
foo = raw_input("Поместите сюда сообщение, запрашивающее ввод у пользователя")
```

В приведенном выше примере `foo` будет хранить все введенные пользователем данные.

Версия Python 3.x ≥ 3.0:

Функция `input` будет ждать, пока пользователь введет текст, а затем вернет результат в виде строки.

```
foo = input("Поместите сюда сообщение, запрашивающее ввод данных у пользователя")
```

В приведенном выше примере `foo` будет хранить все введенные пользователем данные.

29.5. Функция запроса числа у пользователя

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(raw_input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

```
def input_number(msg, err_msg=None):
    while True:
        try:
            return float(input(msg))
        except ValueError:
            if err_msg is not None:
                print(err_msg)
```

Чтобы использовать эту функцию:

```
user_number = input_number("введите число:", "это не число!")
```

или, если не нужно сообщение об ошибке:

```
user_number = input_number("введите число:")
```

29.6. Печать строки без новой строки в конце

Версия Python 2.x ≥ 2.3:

В Python 2.x, чтобы продолжить строку с помощью `print`, завершите оператор `print` запятой. При этом автоматически добавляется пробел.

```
print "Hello,", print "World!" # Hello, World!
```

Версия Python 3.x ≥ 3.0:

В Python 3.x функция `print` имеет необязательный концевой параметр – то, что она выводит в конце заданной строки. По умолчанию это символ новой строки:

```
print("Hello, ", end="\n")
print("World!")
# Hello, # World!
```

Но можно передать и другие символы:

```
print("Hello, ", end="")
print("World!")
# Hello, World!
```

```
print("Hello, ", end="<br>")
print("World!")
# Hello, <br>World!
```

```
print("Hello, ", end="BREAK")
print("World!")
# Hello, BREAKWorld!
```

Если требуется больший контроль над выводом, можно использовать метод `sys.stdout.write`:

```
import sys
sys.stdout.write("Hello, ")
sys.stdout.write("World!")
# Hello, World!
```

Глава 30. Ввод/вывод файлов и папок

Параметр	Подробности
<code>filename</code> (имя файла)	путь к вашему файлу или, если файл находится в рабочем каталоге, имя файла вашего файла
<code>access_mode</code> (режим доступа)	строковое значение, определяющее способ открытия файла
<code>buffering</code> (буферизация)	целочисленное значение, используемое для необязательной буферизации строк

Когда речь идет о хранении, чтении или передаче данных, работа с файлами операционной системы одновременно необходима и легка для Python. В отличие от других языков, Python упрощает работу с файлами, требуя только команды для открытия, чтения/записи и закрытия. Далее объясняется, как Python может взаимодействовать с файлами в операционной системе.

30.1. Режимы работы с файлами

Существует несколько режимов, в которых можно открыть файл, определяемых параметром `mode`. К ним относятся:

- `'r'` – режим чтения. Устанавливается по умолчанию. Позволяет только читать файл, но не изменять его. При использовании этого режима файл должен существовать.
- `'w'` – режим записи. Создает новый файл, если тот не существует, в противном случае сотрет файл и разрешит запись в него.
- `'a'` – режим добавления. В этом режиме данные записываются в конец файла. При этом файл не стирается, и для этого режима он должен существовать.
- `'rb'` – режим чтения в двоичном формате. Он аналогичен `r`, за исключением того, что чтение принудительно осуществляется в двоичном режиме. Также выбирается по умолчанию.
- `'r+'` – режим чтения и записи одновременно. Позволяет одновременно читать и записывать в файлы без использования `r` и `w`.
- `'rb+'` – режим чтения и записи в двоичном формате. То же самое, что и `r+`, за исключением того, что данные передаются в двоичном виде.
- `'wb'` – режим записи в двоичном формате. То же, что и `w`, за исключением того, что данные передаются в двоичном виде.
- `'w+'` – режим записи и чтения. То же самое, что и `r+`, но если файл не существует, то создается новый. В противном случае файл перезаписывается.
- `'wb+'` – режим записи и чтения в двоичном режиме. То же, что и `w+`, но данные передаются в двоичном виде.
- `'ab'` – добавление в двоичном режиме. Аналогично `a`, за исключением того, что данные представлены в двоичном виде.

- 'a+' – режим добавления и чтения. Аналогичен w+, поскольку создает новый файл, если тот не существует. В противном случае указатель переходит в конец файла, если он существует.
- 'ab+' – режим добавления и чтения в двоичном формате. То же, что и a+, за исключением того, что данные передаются в двоичном виде.

```
with open(filename, 'r') as f:
    f.read()
with open(filename, 'w') as f:
    f.write(filedata)
with open(filename, 'a') as f:
    f.write("\n" + newdata)
```

	r	r+	w	w+	a	a+
Чтение	✓	✓	X	✓	X	✓
Запись	X	✓	✓	✓	✓	✓
Создает файл	X	X	✓	✓	✓	✓
Стирает файл	X	X	✓	✓	X	X
Начальное положение	Начало	Начало	Начало	Начало	Конец	Конец

В Python 3 добавлен новый режим создания эксклюзивов (exclusive creation), чтобы случайно не усечь или не перезаписать существующий файл.

- 'x' – открыть для эксклюзивного создания; если файл уже существует, будет выдана ошибка FileExistsError.
- 'xb' – открыть для записи в режиме эксклюзивного создания в двоичном формате. То же, что и x, только данные в двоичном формате.
- 'x+' – режим чтения и записи. Аналогичен w+, поскольку создаст новый file, если file не существует. В противном случае будет выдана ошибка FileExistsError.
- 'xb+' – режим записи и чтения. То же самое, что и x+, но данные в двоичном виде.

	x	x+
Чтение	X	✓
Запись	✓	✓
Создает файл	✓	✓
Стирает файл	X	X
Начальное положение	Начало	Начало

Это позволяет писать открытый код в более “питоновском” стиле:
Версия Python 3.x ≥ 3.3:

```
try:
    with open("fname", "r") as fout:
        # Работа с открытым файлом
except FileExistsError:
    # Обработка ошибок происходит здесь
```


Для Python 2:

Версия Python 2.x ≥ 2.0:

```
import os.path
if os.path.isfile(fname):
    with open(fname, "w") as fout:
        # Работа с открытым файлом
else:
    # Обработка ошибок происходит здесь
```

30.2. Построчное чтение file

Простейший способ итерации по строке файла:

```
with open('myfile.txt', 'r') as fp:
    for line in fp:
        print(line)
```

Метод `readline()` позволяет более детально управлять построчной итерацией. Пример ниже эквивалентен приведенному выше:

```
with open('myfile.txt', 'r') as fp:
    while True:
        cur_line = fp.readline()
        # Если результатом является пустая строка
        if cur_line == "":
            # Мы достигли конца файла
            break
        print(cur_line)
```

Итеративирование при помощи циклов `for` и `readline()` вместе считается плохой практикой. Чаще метод `readlines()` используется для хранения итерируемой коллекции строк файла:

```
with open("myfile.txt", "r") as fp:
    lines = fp.readlines()
for i in range(len(lines)):
    print("Line " + str(i) + ": " + line)
```

Это выведет на экран следующее:

```
Line 0: hello
Line 1: world
```

30.3. Итерация файлов (рекурсивно)

Для итерации всех файлов, включая вложенные каталоги, используйте метод `os.walk()`:

```
import os

for root, folders, files in os.walk(root_dir):
    for filename in files:
        print root, filename
```

В параметре `root_dir` можно указать `"."` для запуска из текущего каталога или любой другой стартовый путь.

Версия Python 3.x ≥ 3.5:

Если вы также хотите получить информацию о файле, вы можете использовать более эффективный метод `os.scandir()` следующим образом:

```
for entry in os.scandir(path):
    if not entry.name.startswith('.') and entry.is_file():
        print(entry.name)
```

30.4. Получение полного содержимого файла

Предпочтительным методом ввода-вывода файлов является использование ключевого слова `with`. Это гарантирует, что дескриптор файла будет закрыт после завершения чтения или записи.

```
with open('myfile.txt') as in_file:
    content = in_file.read()
```

```
print(content)
```

или, чтобы закрыть файл вручную, можно отказаться от использования `with` и просто вызвать `close`:

```
in_file = open('myfile.txt', 'r')
content = in_file.read()
print(content)
in_file.close()
```

Следует помнить, что без использования инструкции `with` вы можете случайно оставить открытым файл в случае возникновения непредвиденного исключения, как в примере ниже:

```
in_file = open('myfile.txt', 'r')
raise Exception("oops")
in_file.close() # Это никогда не будет вызвано
```

30.5. Запись в файл

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1")
    f.write("Line 2")
    f.write("Line 3")
    f.write("Line 4")
```

Если вы откроете `myfile.txt`, то вы увидите, что его содержимое имеет следующий вид:

```
Line 1Line 2Line 3Line 4
```

Python не добавляет разрывы строк автоматически, это нужно сделать вручную:

```
with open('myfile.txt', 'w') as f:
    f.write("Line 1\n")
    f.write("Line 2\n")
    f.write("Line 3\n")
    f.write("Line 4\n")
```

```
Line 1
Line 2
Line 3
Line 4
```

Не используйте метод `os.linesep` в качестве ограничителя строк при записи файлов, открытых в текстовом режиме (по умолчанию); вместо этого используйте `\n`. Если необходимо указать кодировку, достаточно добавить параметр `encoding` в функцию `open`:

```
with open('my_file.txt', 'w', encoding='utf-8') as f:
    f.write('utf-8 text')
```

Также можно использовать для записи в файл оператор `print`. Механика этого процесса отличается в Python 2 и Python 3, но концепция одна и та же: вы можете взять вывод, который был бы выведен на экран, и отправить его в файл.

Версия Python 3.x ≥ 3.0:

```
with open('fred.txt', 'w') as outfile:
    s = "I'm Not Dead Yet!"
    print(s)           # запись в stdout
    print(s, file = outfile) # запись в outfile
```

Примечание: возможно указать параметр `file` и выводить на экран, # убедившись, что файл завершается значением `None` либо напрямую, либо через переменную

```
myfile = None
print(s, file = myfile)    # запись в stdout
print(s, file = None)     # запись в stdout
```

В Python 2 вы бы сделали что-то вроде:

Версия Python 2.x ≥ 2.0:

```
outfile = open('fred.txt', 'w')
s = "I'm Not Dead Yet!"
print s           # запись в stdout
print >> outfile, s # запись в outfile
```

В отличие от функции записи функция `print` автоматически добавляет разрывы строк.

30.6. Проверка наличия файла или пути

Примените стиль кодирования EAFP и используйте метод `try` для открытия:

```
import errno

try:
    with open(path) as f:
        # Файл существует
except IOError as e:
    # Вызвать исключение, если не EWOULDBLOCK (нет такого файла или каталога)
    if e.errno != errno.ENOENT:
        raise
    # нет такого файла или каталога
```

Это также позволит избежать т. н. “состояния гонки”, если другой процесс удаляет файл между его проверкой и использованием. Это состояние гонки может произойти в следующих случаях:

- Использование модуля `os`:

```
import os
os.path.isfile('/path/to/some/file.txt')
```

Версия Python 3.x ≥ 3.4:

- Использование `pathlib`:

```
import pathlib
path = pathlib.Path('/path/to/some/file.txt')
if path.is_file():
```



Чтобы проверить, существует данный путь или нет, вы можете выполнить описанную выше процедуру EAFP или проверить путь явно:

```
import os
path = "/home/myFiles/directory1"
```

```
if os.path.exists(path):
    # Выполнить что-либо
```

30.7. Произвольный доступ к файлам с помощью mmap

Использование модуля mmap позволяет пользователю произвольно обращаться к местам в файле путем сопоставления (мэппинга) в памяти. Это альтернатива использованию обычных операций с файлами.

```
import mmap
with open('filename.ext', 'r') as fd:
    # 0: отображение всего файла
    mm = mmap.mmap(fd.fileno(), 0)

    # вывести символы с индексами с 5 по 10
    print mm[5:10]

    # вывести строку, начинающуюся с текущей позиции mm
    print mm.readline()

    # записать символ в 5-й индекс
    mm[5] = 'a'

    # вернуть позицию mm к началу файла
    mm.seek(0)

    # закрыть объект mmap
    mm.close()
```

30.8. Замена текста в файле

```
import fileinput

replacements = {'Search1': 'Replace1',
                'Search2': 'Replace2'}

for line in fileinput.input('filename.txt', inplace=True):
    for search_for in replacements:
        replace_with = replacements[search_for]
        line = line.replace(search_for, replace_with)
    print(line, end="")
```

30.9. Проверка того, что файл пуст

```
>>> import os
>>> os.stat(path_to_file).st_size == 0

или же

>>> import os
>>> os.path.getsize(path_to_file) > 0
```

Тем не менее оба способа будут генерировать исключение, если файл не существует. Чтобы избежать такой ошибки, сделайте следующее:

```
import os
def is_empty_file(fpath):
    return os.path.isfile(fpath) and os.path.getsize(fpath) > 0

который вернет логическое значение bool.
```

30.10. Чтение файла в диапазоне строк

Предположим, что вы хотите выполнить итерацию только между некоторыми конкретными строками файла. Для этого вы можете использовать модуль `itertools`.

```
import itertools
with open('myfile.txt', 'r') as f:
    for line in itertools.islice(f, 12, 30):
        # сделать здесь что-то
```

Будут прочитаны строки с 13 по 20, так как в Python индексация начинается с 0. Поэтому строка 1 индексируется как 0. Также можно прочитать некоторые дополнительные строки, используя ключевое слово `next()`.

Когда вы используете файл-объект как итеративный, не применяйте здесь инструкцию `readline()`, поскольку эти два метода не должны использоваться вместе.

30.11. Копирование дерева каталогов

```
import shutil
source='//192.168.1.2/Daily Reports'
destination='D:\\Reports\\Today'
shutil.copytree(source, destination)
```

Каталог назначения **не должен существовать** перед использованием этой функции.

30.12. Копирование содержимого файла в другой файл

```
with open(input_file, 'r') as in_file, open(output_file, 'w') as out_file:
    for line in in_file:
        out_file.write(line)
```

- Используем модуль `shutil`:

```
import shutil
shutil.copyfile(src, dst)
```

Глава 31. Модуль os.path

Этот модуль реализует несколько полезных функций для работы с именами путей. Параметры пути могут передаваться в виде строк или байтов. Приложениям рекомендуется представлять имена как строки символов (Unicode).

31.1. Объединение путей

Чтобы соединить два или более компонента пути, импортируйте в Python модуль `os`, а затем выполните следующее:

```
import os
os.path.join('a', 'b', 'c')
```

Преимущество использования `os.path` заключается в том, что он позволяет коду оставаться совместимым во всех операционных системах, поскольку в нем используется разделитель, подходящий для платформы, на которой он работает.

Например, результатом этой команды в Windows будет:

```
>>> os.path.join('a', 'b', 'c')
'a\\b\\c'
```

В Unix:

```
>>> os.path.join('a', 'b', 'c')
'a/b/c'
```

31.2. Управление компонентами пути

Чтобы отделить компонент от пути, используйте:

```
>>> p = os.path.join(os.getcwd(), 'foo.txt')
>>> p
'/Users/csaftoiu/tmp/foo.txt'
>>> os.path.dirname(p)
'/Users/csaftoiu/tmp'
>>> os.path.basename(p)
'foo.txt'
>>> os.path.split(os.getcwd())
('/Users/csaftoiu/tmp', 'foo.txt')
>>> os.path.splitext(os.path.basename(p))
('foo', '.txt')
```

31.3. Получение родительского каталога

```
os.path.abspath(os.path.join(PATH_TO_GET_THE_PARENT, os.pardir))
```

31.4. Проверка существования пути

Чтобы проверить, существует ли заданный путь, используйте:

```
path = '/home/john/temp'
os.path.exists(path)
# это возвращает false, если путь не существует или является неработающей символической
ссылкой
```

31.5. Проверка того, является ли данный путь каталогом, файлом, символической ссылкой, точкой монтирования

Для проверки того, является ли заданный путь каталогом:

```
dirname = '/home/john/python'
os.path.isdir(dirname)
```

Чтобы проверить, является ли данный путь файлом:

```
filename = dirname + 'main.py'
os.path.isfile(filename)
```

Чтобы проверить, является ли данный путь символической ссылкой:

```
symlink = dirname + 'some_sym_link'
os.path.islink(symlink)
```

Чтобы проверить, является ли данный путь точкой монтирования:

```
mount_path = '/home'
os.path.ismount(mount_path)
```

31.6. Абсолютный путь из относительного пути

Используйте функцию `os.path.abspath`:

```
>>> os.getcwd()
'/Users/csaftoiu/tmp'
>>> os.path.abspath('foo')
'/Users/csaftoiu/tmp/foo'
>>> os.path.abspath('../foo')
'/Users/csaftoiu/foo'
>>> os.path.abspath('/foo')
'/foo'
```

Глава 32. Итерируемые типы данных и итераторы

32.1. Итератор (iterator), итерируемый объект (iterable) и генератор (generator)

Итерируемый объект (iterable) – это объект, который может возвращать итератор (iterator). Точнее, любой объект, имеющий метод `iter` и возвращающий итератор, является итерируемым. Также это может быть объект, реализующий метод `__getitem__`. Этот метод может принимать индексы (начиная с нуля) и выдавать ошибку `IndexError`, если индексы становятся недействительными.

Класс `str` в Python является примером итерируемого объекта, в котором реализован метод `__getitem__`.

Итератор – это объект, который при вызове `next(*object*)` на некотором объекте выдает следующее значение в последовательности. Более того, любой объект, реализующий метод `__next__`, является итератором. После исчерпания итератора возникает сообщение `StopIteration`; повторное использование итератора в этой точке невозможно.

Итерируемые классы

Итерируемые классы определяют методы `__iter__` и `__next__`. Пример итерируемого класса:

```
class MyIterable:
```

```
    def iter(self):
```

```
        return self
```

```
    def next(self):
```

```
        # код
```

#Классический итерируемый объект (iterable) в старых версиях Python, `__getitem__` все еще поддерживается...

```
class MySequence:
```

```
    def getitem(self, index):
```

```
        if (condition):
```

```
            raise IndexError
```

```
        return (item)
```

#Можно получить простой экземпляр “итератора” с помощью `iter(MySequence())`

Попробуем инстантировать (создать экземпляр) абстрактного класса из модуля `collections`, чтобы лучше разобраться в этом. Пример:

Версия Python 2.x ≥ 2.3

```
import collections
```

```
>>> collections.Iterator()
```

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Версия Python 3.x ≥ 3.0

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Чтобы обеспечить совместимость итерируемых классов Python 3 и Python 2, выполните следующие действия:

Версия Python 2.x ≥ 2.3

```
class MyIterable(object): # или collections.Iterator, что рекомендовано....
```

```
    def __iter__(self):
```

```
        return self
```

```
def next(self): #code
    __next__ = next
```

Оба они теперь являются итераторами и по ним можно проходить циклом:

```
ex1 = MyIterableClass()
ex2 = MySequence()
```

```
for (item) in (ex1): #code
for (item) in (ex2): #code
```

Использование **генераторов** – это простой способ создания итераторов. Генератор – это итератор, по которому можно итерировать.

32.2. Извлечение значений по одному

Начните со встроенной функции `iter()` для получения **итератора** по итерируемому объекту и используйте функцию `next()` для получения элементов одного за другим, до возникновения сигнала `StopIteration`, означающего конец итерации:

```
s = {1, 2}          # или список, или генератор, или даже итератор
i = iter(s)         # получить итератор
a = next(i)         # a = 1
b = next(i)         # b = 2
c = next(i)         # вызывает StopIteration
```

32.3. Итерирование по всему итерируемому

```
s = {1, 2, 3}

# получить каждый элемент в s
for a in s:
    print a # выводит 1, затем 2, затем 3

# копирование в список
l1 = list(s) # l1 = [1, 2, 3]

# использование генератора списка
l2 = [a * 2 for a in s if a > 2] # l2 = [6]
```

32.4. Проверка только одного элемента в итерируемом

С помощью распаковки извлеките первый элемент и убедитесь, что он единственный:

```
a, = iterable

def foo():
    yield 1

a, = foo()          # a = 1

nums = [1, 2, 3]
a, = nums           # ValueError: слишком много значений для распаковки
```

32.5. Что может быть итерируемым

Итерируемым (Iterable) может быть любой объект, в котором элементы принимаются *по одному и в направлении только вперед*. Встроенные коллекции Python являются итерируемыми:

```
[1, 2, 3]          # список (list), итерация по элементам
(1, 2, 3)          # кортеж (tuple)
```



```
{1, 2, 3}      # набор (set)
{1: 2, 3: 4}   # словарь (dict), итерация по ключам
```

Генераторы возвращают итерируемые объекты:

```
def foo():      # foo еще не итерируемый...
    yield 1

res = foo()     # ...но res уже итерируемый
```

32.6. В итератор нельзя входить повторно!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a
# Каким был первый элемент в итерируемом? Теперь его не получить.
# Можно только получить новый итератор
gen()
```

Глава 33. Функции

Параметр	Подробности
arg1, ..., argN	Регулярные аргументы
*args	Неименованные позиционные аргументы
kw1, ..., kwN	Только именованные аргументы (Keyword-only arguments, аргументы только для ключевых слов)
**kwargs	Остальные именованные аргументы (аргументы ключевых слов)

Функции в Python представляют собой организованный, многократно используемый и модульный код для выполнения набора определенных действий. Функции упрощают процесс кодирования, предотвращают использование избыточной логики и делают код более понятным. В данной главе описывается объявление и использование функций в Python.

В языке Python имеется множество *встроенных функций*, таких как `print()`, `input()`, `len()`. Можно также создавать собственные функции для выполнения более специфических задач – *пользовательские функции*.

33.1. Создание и вызов простых функций

Использование инструкции `def` является наиболее распространенным способом определения функции в Python. Этот оператор является так называемым *составным выражением с одним предложением* и имеет следующий синтаксис:

```
def function_name(parameters):
    statement(s)
```

`function_name` – это *идентификатор* функции. Так как определение функции является исполняемым, его исполнение *связывает* имя функции с объектом функции, который может быть вызван позже при помощи идентификатора.

parameters – это необязательный список идентификаторов, которые при вызове функции привязываются к значениям, передаваемым в качестве аргументов. Функция может иметь произвольное количество аргументов, которые разделяются запятыми.

statement(s) – или *тело функции* – непустая последовательность операторов, выполняемых при каждом вызове функции. Тело функции не может быть пустым, так же как и любой блок с отступом.

Вот пример простого определения функции, назначение которой состоит в том, чтобы при каждом ее вызове выводить Hello:

```
def greet():  
    print("Hello")
```

Теперь вызовем определенную функцию greet():

```
greet()  
# Результат: Hello
```

Еще один пример определения функции, которая принимает один-единственный аргумент и выводит на экран переданное значение при каждом вызове:

```
def greet_two(greeting):  
    print(greeting)
```

После этого необходимо вызвать функцию greet_two() с аргументом:

```
greet_two("Howdy")  
# Howdy
```

Также вы можете задать для этого аргумента функции значение по умолчанию:

```
def greet_two(greeting="Howdy"):  
    print(greeting)
```

Теперь вы можете вызывать функцию без указания значения:

```
greet_two()  
# Howdy
```

В отличие от многих других языков вам не нужно явно объявлять тип возвращаемого значения функции. Функции Python могут возвращать значения любого типа с помощью ключевого слова return. Одна функция может возвращать любое количество разных типов!

```
def many_types(x):  
    if x < 0:  
        return "Hello!"  
    else:  
        return 0  
  
print(many_types(1))  
print(many_types(-1))  
  
# Результат:  
0  
Hello!
```

Пока это правильно обрабатывается вызывающей стороной, это совершенно правильный код Python.

Функция, которая достигает конца исполнения без оператора возврата, всегда будет возвращать значение None:

```
def do_nothing():  
    pass  
  
print(do_nothing())  
# None
```

Как упоминалось ранее, определение функции должно иметь тело функции, т. е. непустую последовательность операторов. Поэтому в данном примере в теле функции используется оператор `pass`, который является нулевой операцией – когда он выполняется, то ничего не происходит. Это полезно в качестве заполнителя, когда синтаксически требуется оператор, но код выполнять не нужно.

33.2. Определение функции с произвольным числом аргументов

Произвольное количество позиционных аргументов:

Определение функции, способной принимать произвольное число аргументов, можно осуществить, предварительно обозначив один из аргументов символом `*`.

```
def func(*args):
    # args будет представлять собой кортеж, содержащий все переданные значения
    for i in args:
        print(i)
```

```
func(1, 2, 3) # Вызов с 3 аргументами
# Результат:  1
#             2
#             3
```

```
list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values) # Вызов со списком значений, * расширяет список
# Результат:  1
#             2
#             3
```

```
func() # Вызов без аргументов
# Нет результата
```

Нельзя указать значение по умолчанию для `args`, например `func(*args=[1, 2, 3])` вызовет синтаксическую ошибку (даже не будет компилироваться).

Нельзя предоставить аргументы по имени при вызове функции, например `func(*args=[1, 2, 3])` приведет к ошибке `TypeError`.

Но если у вас уже есть аргументы в массиве (или любом другом итерируемом объекте), **можно** вызвать свою функцию следующим образом: `func(*my_stuff)`.

К этим аргументам (`*args`) можно обращаться по индексу, например `args[0]` вернет первый аргумент.

Произвольное количество именованных аргументов (аргументов ключевых слов)

Вы можете взять произвольное количество аргументов с именем, обозначая аргумент в определении **двумя** `*` перед ним:

```
def func(**kwargs):
    # kwargs будет представлять собой словарь, содержащий имена в качестве ключей
    # и значения в качестве значений
    for name, value in kwargs.items():
        print(name, value)
```

```
func(value1=1, value2=2, value3=3) # Вызов с тремя аргументами
# Результат: value1      1
#           value2      2
#           value3      3
```

```
func() # Вызов без аргументов
# Нет результата
```

```
my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)           # Вызов со словарем
# Результат:      foo 1
#                  bar 2
```

Вы не можете предоставить аргументы без имен, например `func(1, 2, 3)` вызовет `TypeError`. `kwargs` – это простой собственный (нативный) словарь языка Python. Например, `args['value1']` даст значение для аргумента `value1`. Предварительно убедитесь, что такой аргумент существует, иначе будет выдана ошибка `KeyError`.

Предупреждение

Порядок в определении имеет значение. Первыми должны быть **позиционные и именованные аргументы** (необходимые аргументы). Затем идут **произвольные** аргументы `*args` (необязательные). Затем следуют аргументы **только именованные** (необходимые). Затем **произвольное ключевое слово** `**kwargs` (необязательно).

```
# |позиционные|необязательные|---только-именованные|необязательные|
def func(arg1, arg2=10, *args, kwarg1, kwarg2=2, **kwargs):
    pass
```

- `arg1` должен быть предоставлен, в противном случае возникает `TypeError`. Он может быть задан как позиционный (`func(10)`) или именованный (`func(arg1=10)`).
- `kwarg1` также должен быть указан, но он может быть предоставлен только в качестве именованного аргумента: `func(kwarg1=10)`.
- `arg2` и `kwarg2` являются необязательными. Если значение должно быть изменено, применяются те же правила для `arg1` (либо позиционный, либо именованный), либо `kwarg1` (только именованный).
- `*args` перехватывает дополнительные позиционные параметры. Но следует учитывать, что `arg1` и `arg2` должны быть позиционными для передачи аргументов в `*args`: `func(1, 1, 1, 1)`.
- `**kwargs` перехватывает все дополнительные именованные параметры. В этом случае любой параметр, который не является `arg1`, `arg2`, `kwarg1` или `kwarg2`. Например: `func(kwarg3=10)`.
- В Python 3 вы можете использовать только `*`, чтобы указать, что все последующие аргументы должны быть указаны как именованные. Например, функция `math.isclose` в Python версии 3.5 и выше определяется с помощью `def math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)`, что означает, что первые два аргумента могут быть заданы позиционно, но необязательные третий и четвертый параметры могут быть заданы только как именованные аргументы.

Python 2.x не поддерживает только именованные параметры. Такое поведение можно эмулировать с помощью `kwargs`:

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # тело функции ...
```

Примечание о присвоении имен

Соглашение об именовании необязательных позиционных аргументов `args` и необязательных именованных аргументов `kwargs` – это просто соглашение, можно использовать любые имена, но полезно следовать этому соглашению, чтобы другие понимали, что вы делаете, или даже вы сами впоследствии, так что, пожалуйста, делайте это.

Примечание об уникальности

Любая функция может быть определена с **одним или несколькими** `*args` и **одним или несколькими** `**kwargs`, но не с более чем одним из них. При этом `*args` **должен** быть последним позиционным аргументом, а `**kwargs` должен быть последним параметром. Попытка использовать более одного из них приведет к исключению `Syntax Error`.

Примечание о вложенных функциях с необязательными аргументами

Такие функции можно вложить друг в друга, и обычно принято удалять те элементы, которые код уже обработал, **но** если вы передаете параметры, то вам необходимо передать **необязательные** позиционные аргументы с префиксом `*` и **необязательные** именованные аргументы с префиксом `**`, иначе `args` будут передаются как список или кортеж, а `kwargs` – как один словарь, например:

```
def fn(**kwargs):
    print(kwargs)
    f1(**kwargs)
```

```
def f1(**kwargs):
    print(len(kwargs))
```

```
fn(a=1, b=2)
# Результат:
# {'a': 1, 'b': 2}
# 2
```

33.3. Лямбда-функции (встроенные/анонимные)

Ключевое слово `lambda` создает встроенную функцию (inline function), содержащую одно выражение. Значение этого выражения и есть то, что возвращает функция при вызове.

Рассмотрим функцию:

```
def greeting():
    return "Hello"
```

которая при вызове:

```
print(greeting())
```

выводит:

```
Hello
```

Это можно записать в виде лямбда-функции следующим образом:

```
greet_me = lambda: "Hello"
```

См. примечание в конце этого раздела о присвоении лямбда-функций переменным. Как правило, этого делать не следует.

Это создает встроенную функцию с именем `greet_me`, которая возвращает `Hello`. Обратите внимание, что вы не пишете `return` при создании такой функции с лямбдой. Значение после двоеточия возвращается автоматически.

После присвоения переменной она может использоваться как обычная функция:

```
print(greet_me())
```

выводит:

```
Hello
```

Лямбда-функция также может принимать аргументы:

```
strip_and_upper_case = lambda s: s.strip().upper()
strip_and_upper_case(" Hello ")
```

что возвращает строку:

```
HELLO
```

Такие функции могут принимать произвольное количество аргументов и именованных аргументов, как обычные функции.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

что выводит:

```
hello ('world',) {'world': 'world'}
```

Обычно это удобно использовать для коротких функций, которые удобно определять в том месте, где они вызываются (что обычно с `sorted`, `filter` и `map`).

Например, эта строка сортирует список строк, игнорируя их регистр и пробелы в начале и в конце:

```
sorted([" foo ", "  bAR", "BaZ  "], key=lambda s: s.strip().upper())
# Результат:
# [' bAR', 'BaZ  ', 'foo ']
```

Сортировка списка с игнорированием пробелов:

```
sorted([" foo ", "  bAR", "BaZ  "], key=lambda s: s.strip())
# Результат:
# ['BaZ  ', ' bAR', 'foo ']
```

Примеры с функцией `map`:

```
sorted(map(lambda s: s.strip().upper(), [" foo ", "  bAR", "BaZ  "]))
# Результат:
# ['BAR', 'BAZ', 'FOO']
```

```
sorted(map(lambda s: s.strip(), [" foo ", "  bAR", "BaZ  "]))
# Результат:
# ['BaZ', 'bAR', 'foo']
```

Примеры с числовыми списками:

```
my_list = [3, -4, -2, 5, 1, 7]
sorted(my_list, key=lambda x: abs(x))
# Результат:
# [1, -2, 3, -4, 5, 7]
```

```
list(filter(lambda x: x>0, my_list))
# Результат:
# [3, 5, 1, 7]
```

```
list(map(lambda x: abs(x), my_list))
# Результат:
# [3, 4, 2, 5, 1, 7]
```

Можно вызвать другие функции (с аргументами или без) внутри лямбда-функции.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

Выведет:

```
hello world
```

Это полезно, потому что `lambda` может содержать только одно выражение, и с помощью вспомогательной функции можно запустить несколько операторов.

Примечание

Имейте в виду, что PEP8 (официальное руководство по стилю Python) не рекомендует назначать лямбды для переменных (как было в первых двух примерах):

Всегда используйте инструкцию `def` вместо оператора присваивания, который привязывает лямбда-выражение непосредственно к идентификатору.

Правильно:

```
def f(x): return 2*x
```

Неправильно:

```
f = lambda x: 2*x
```

Первая форма означает, что именем результирующего объекта функции является конкретное `f` вместо общего `<lambda>`. Это более полезно для отслеживания и представления строк в целом. Использование оператора присваивания исключает единственное преимущество, которое лямбда-выражение может предлагать в явном выражении `def` (т. е. то, что оно может быть встроено в большее выражение).

33.4. Создание функции с необязательными аргументами

Необязательные аргументы можно отключить, присвоив (с помощью `=`) аргументу-имени значение по умолчанию:

```
def make(action='nothing'):  
    return action
```

Вызов этой функции возможен тремя различными способами:

```
make("fun")  
# Вывод: fun
```

```
make(action="sleep")  
# Вывод: sleep
```

```
# Аргумент является необязательным, поэтому функция будет использовать значение  
# по умолчанию, если аргумент не передан.  
make()  
# Вывод: nothing
```

Предупреждение

С изменяемыми типами данных (`list`, `dict`, `set` и т. д.), если они заданы в качестве атрибута по умолчанию, следует обращаться осторожно. Любое изменение аргумента по умолчанию меняет его навсегда. См. главу “Определение функции с необязательными изменяемыми аргументами”.

33.5. Определение функции с необязательными изменяемыми аргументами

Существует проблема, связанная с использованием **необязательных аргументов с изменяемым типом по умолчанию** (описана в разделе “Определение функции с необязательными аргументами”), которая потенциально может привести к неожиданному поведению.

Пояснение

Проблема возникает потому, что аргументы по умолчанию инициализируются **один раз**, в момент *определения* функции, а **не** (как во многих других языках) при ее *вызове*. Значения по умолчанию хранятся в переменной-члене `__defaults__` объекта функции.

```
def f(a, b=42, c=[]):  
    pass
```

```
print(f.__defaults__)  
# Результат: (42, [])
```

Для **неизменяемых** типов (см. главу “Передача аргументов и изменяемость”) это не проблема, потому что нет способа изменить эту переменную; ее можно только переназначить, оставив исходное значение без изменений. Следовательно, впоследствии гарантированно сохраняется одинаковое значение по умолчанию. Однако для **изменяемого** типа исходное значение может измениться при обращении к его различным функциям. Поэтому для последующих вызовов функции не гарантируется сохранение первоначального значения по умолчанию.

```
def append(elem, to=[]):
    to.append(elem)      # Этот вызов append() изменяет переменную по умолчанию "to"
    return to

append(1)
# Результат: [1]

append(2)               # Добавляет ее к внутреннему списку
# Результат: [1, 2]

append(3, [])           # Использование нового созданного списка дает ожидаемый результат
# Результат: [3]

# Повторный вызов без аргумента снова добавит ее во внутренний список
append(4)
# Результат: [1, 2, 4]
```

Примечание. Некоторые IDE, такие как PyCharm, выдают предупреждение, если в качестве атрибута по умолчанию указан изменяемый тип.

Решение

Если вы хотите, чтобы аргумент по умолчанию всегда был тем, который вы указываете в определении функции, то **всегда** следует использовать неизменяемый тип в качестве аргумента по умолчанию.

Часто в случае, когда изменяемый тип необходим по умолчанию, выход в том, чтобы использовать значение None в качестве аргумента по умолчанию, а затем присваивать фактическое значение по умолчанию переменной аргумента, если оно равно None.

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

33.6. Передача аргументов и изменяемость

Сначала немного терминологии:

- **аргумент (фактический параметр)**: фактическая переменная, передаваемая функции;
- **параметр (формальный параметр)**: принимающая переменная, которая используется в функции.

В Python **аргументы передаются по присваиванию** (в отличие от других языков, где аргументы могут передаваться по значению/ссылке/указателю).

- Изменение параметра приводит к изменению аргумента (если тип аргумента является изменяемым).

```
def foo(x):
    x[0] = 9             # здесь x – параметр
    print(x)             # это изменяет список, помеченный как x, так и y
```



```

y = [4, 5, 6]
foo(y)          # вызов foo с y в качестве аргумента
# Вывод: [9, 5, 6] # изменен список, помеченный x
print(y)
# Вывод: [9, 5, 6] # список, помеченный y, изменен тоже
    • Переназначение параметра не приведет к переназначению аргумента.

def foo(x):      # здесь x – параметр, при вызове foo(y) мы присваиваем y значение x
    x[0] = 9      # это изменяет список, помеченный как x, так и y
    x = [1, 2, 3] # x теперь маркирует другой список (на y это не влияет)
    x[2] = 8      # это изменяет список x, а не список y

y = [4, 5, 6]    # y – аргумент, x – параметр
foo(y)           # представим, что мы написали "x = y", теперь переходим к строке 1
y
# Вывод: [9, 5, 6]

```

В Python мы **не присваиваем фактически значения переменным, а привязываем (т. е. назначаем, присоединяем) переменные (которые считаются именами) к объектам.**

- **Неизменяемые:** целые числа, строки, кортежи и т. д. Все операции создают копии.
- **Изменяемые:** списки, словари, наборы и т. д. Операции могут быть как изменяющими, так и не изменяющими.

```

x = [3, 1, 9]
y = x
x.append(5)    # изменяет список, помеченный x и y, причем и x, и y привязаны к [3, 1, 9]
x.sort()       # изменяет список, помеченный x и y (сортировка на месте)
x = x + [4]    # не изменяет список (создает копию только для x, но не для y)
z = x          # z – это x ([1, 3, 9, 4])
x += [6]       # изменяет список, помеченный x и z (используется функция extend)
x = sorted(x)  # не изменяет список (создает копию только для x)
x
# Результат: [1, 3, 4, 5, 6, 9]
y
# Результат: [1, 3, 5, 9]
z
# Результат: [1, 3, 5, 9, 4, 6]

```

33.7. Возврат значений из функций

Функции могут возвращать (return) значение, которое можно использовать напрямую:

```

def give_me_five():
    return 5

print(give_me_five())    # Вывести возвращаемое значение
# Результат: 5

или сохранить значение для последующего использования:

num = give_me_five()
print(num)               # Вывести сохраненное возвращаемое значение
# Результат: 5

```

или использовать это значение для любых операций:

```

print(give_me_five() + 10)
# Результат: 15

```

Если в функции встречается оператор `return`, то происходит немедленный выход из функции, и последующие операции не оцениваются:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')
```

```
print(give_me_another_five())
```

```
# Результат: 5
```

Вы также можете возвращать несколько значений (в виде кортежа):

```
def give_me_two_fives():
    return 5, 5          # Возвращает два значения 5
```

```
first, second = give_me_two_fives()
```

```
print(first)
```

```
# Результат: 5
```

```
print(second)
```

```
# Результат: 5
```

Функция без оператора `return` неявно возвращает значение `None`. Аналогично функция с оператором `return`, но без возвращаемого значения или переменной возвращает `None`.

33.8. Замыкание (closure)

Замыкания в Python создаются вызовами функций. Вызов `makeInc` создает привязку для `x`, на которую ссылается функция `inc`. Каждый вызов `makeInc` создает новый экземпляр этой функции, но каждый экземпляр имеет ссылку на разную `x`.

```
def makeInc(x):
    def inc(y):
        # x "присоединен" к определению inc
        return y + x
    return inc
```

```
incOne = makeInc(1)
```

```
incFive = makeInc(5)
```

```
incOne(5)
```

```
# 6
```

```
incFive(5)
```

```
# 10
```

Обратите внимание, что в то время как при обычном замыкании вложенная функция полностью наследует все переменные из окружающей ее среды, в этой конструкции вложенная функция имеет доступ только на чтение к унаследованным переменным, но не может назначать их:

```
def makeInc(x):
    def inc(y):
        # увеличение x не разрешено
        x += y
        return x
    return inc
```

```
incOne = makeInc(1)
```

```
incOne(5)
```

```
# UnboundLocalError: локальная переменная 'x' упоминается перед присваиванием
```

В Python 3 появилась инструкция `nonlocal` для реализации полного закрытия с вложенными функциями.

```
def makeInc(x):
    def inc(y):
        nonlocal x      # теперь можно присвоить значение x
        x += y
        return x
    return inc

incOne = makeInc(1)
incOne(5)

# 6
```

33.9. Принудительное использование именованных параметров

Все параметры, указанные после первой звездочки в сигнатуре функции, являются только именованными.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() отсутствует 1 обязательный именованный аргумент: 'b'
```

В Python 3 можно поместить одну звездочку в подпись функции, чтобы гарантировать, что остальные аргументы могут передаваться только с использованием именованных аргументов.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() принимает 2 позиционных аргумента, но было задано 3
f(1, 2, c=3)
# Ошибка отсутствует
```

33.10. Вложенные функции

Функции в Python являются объектами первого класса. Они могут быть определены в любой области видимости.

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Функции, захватывающие свою область видимости, могут передаваться, как и любые другие объекты.

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
```

```

add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Результат: 15
add6(10)
#Результат: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Результат: 26

```

33.11. Предел рекурсии

Существует ограничение на глубину возможной рекурсии, которое зависит от реализации Python. При достижении этого предела выдается исключение `RuntimeError`:

```

def cursing(depth):
    try:
        cursing(depth + 1)      # фактически рекурсия
    except RuntimeError as RE:
        print('I recursed {} times!'.format(depth))

cursing(0)
# Результат: I recursed 1083 times!

```

Можно изменить предел глубины рекурсии, используя метод `sys.setrecursionlimit(limit)`, и проверить этот предел на `sys.getrecursionlimit()`.

```

sys.setrecursionlimit(2000)
cursing(0)

# Результат: I recursed 1997 times!

```

Начиная с Python 3.5 исключением является `RecursionError`, производное из `RuntimeError`.

33.12. Рекурсивная лямбда-функция с использованием присваиваемой переменной

Один из методов создания рекурсивных лямбда-функций заключается в присвоении функции переменной и последующем обращении к этой переменной в самой функции. В качестве примера можно привести рекурсивное вычисление факториала числа, как показано в следующем коде:

```

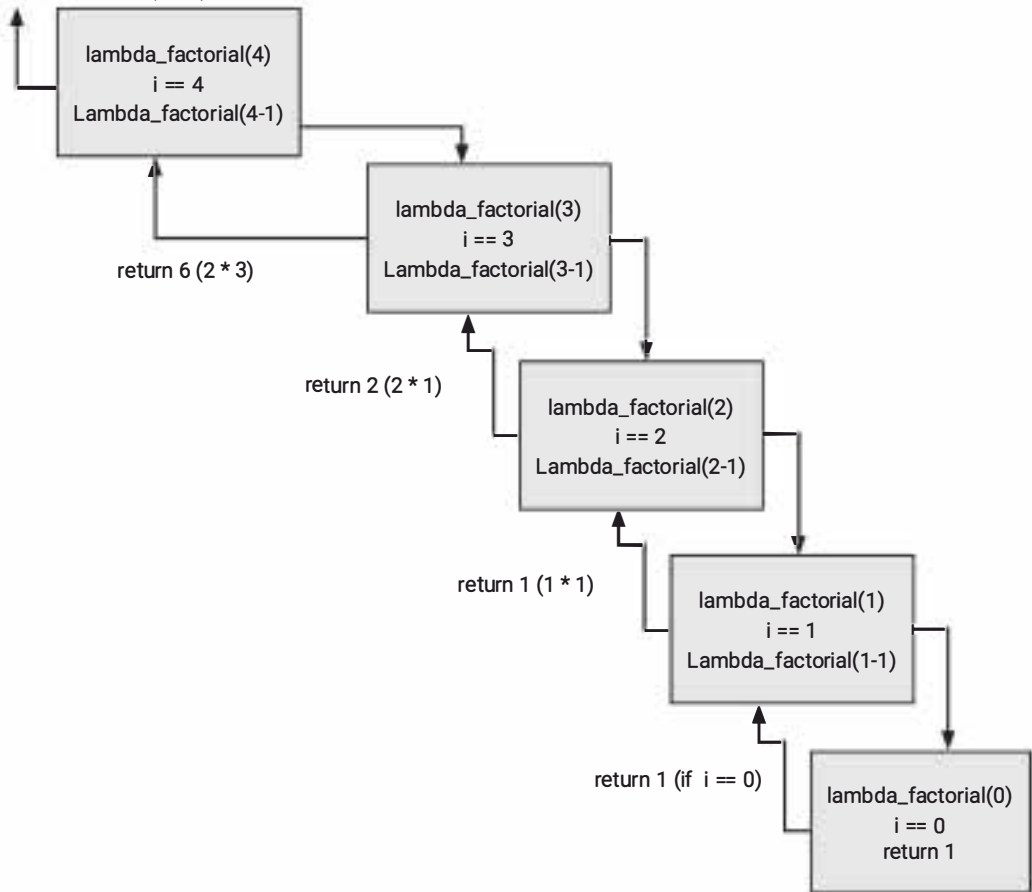
lambda_factorial = lambda i: 1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24

```

Описание кода

Лямбда-функции через присваивание переменной передается значение (4), которое она оценивает и возвращает 1, если оно равно 0, иначе возвращается текущее значение (i)* другого вычисления с помощью лямбда-функции значения -1 (i-1). Это продолжается до тех пор, пока переданное значение не уменьшится до 0 (return 1). Процесс можно визуализировать следующим образом:

Final Answer 24 (4 * 6)



33.13. Рекурсивные функции

Рекурсивная функция – это функция, которая вызывает сама себя в своем определении. Например, математическая функция факториала, определяемая как $\text{factorial}(n) = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1$, может быть запрограммирована как:

```
def factorial(n):
    #n должно быть целым числом
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

что даст следующие результаты:

```
factorial(0)
#результат 1
factorial(1)
#результат 1
factorial(2)
#результат 2
factorial(3)
#результат 6
```

как и ожидалось.

Обратите внимание, что эта функция рекурсивна, потому что второй `return factorial(n-1)`, где функция вызывает себя же в своем определении.

Некоторые рекурсивные функции могут быть реализованы с использованием лямбда-функций, к примеру функция факториала:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

Функция выводит то же, что и выше.

33.14. Определение функции с аргументами

Аргументы указываются в круглых скобках после имени функции:

```
def divide(dividend, divisor): # Имена функции и ее аргументов
    # Аргументы доступны по имени в теле функции
    print(dividend / divisor)
```

Имя функции и ее список аргументов называются *сигнатурой* функции. Каждый именованный аргумент является фактически локальной переменной функции.

При вызове функции дайте значения для аргументов, указав их по порядку:

```
divide(10, 2)
# результат: 5
```

Также их можно указать в любом порядке, используя имена из определения функции:

```
divide(divisor=2, dividend=10)
# результат: 5
```

33.15. Распаковка итерируемых объектов и словарей

Функции позволяют задавать такие типы параметров: позиционные, переменные позиционные, именованные (keyword args, kwargs). Приведем наглядное и краткое использование разных типов.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)
```

```
>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}
```

```
>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

```
>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
```

```

1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

```

```

>>> pair = (3, 4)
>>> unpacking(1, 2, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *pair)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *pair, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

```

```

>>> args_list = [3, 4]
>>> unpacking(1, 2, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, 3, 4, *args_list)
1 2 3 4 (3, 4) {}
>>> unpacking(1, 2, d=4, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, *args_list, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'

```

```

>>> arg_dict = {'c':3, 'd':4}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'d':4, 'c':3}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {}
>>> arg_dict = {'c':3, 'd':4, 'not_a_parameter': 75}
>>> unpacking(1, 2, **arg_dict)
1 2 3 4 () {'not_a_parameter': 75}

```

```

>>> unpacking(1, 2, *pair, **arg_dict)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
>>> unpacking(1, 2, 3, 4, **arg_dict)

```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'd'
```

Позиционные аргументы имеют приоритет перед любыми другими формами передачи аргументов

```
>>> unpacking(1, 2, **arg_dict, c=3)
1 2 3 4 () {'not_a_parameter': 75}
>>> unpacking(1, 2, 3, **arg_dict, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
```

33.16. Определение функции с несколькими аргументами

В функцию можно передавать любое количество аргументов, единственные жесткие правила – имя каждого аргумента должно быть уникальным, а необязательные аргументы должны находиться после обязательных:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

При вызове функции вы можете либо указать каждый именованный аргумент без имени, но порядок важен:

```
print(func(1, 'a', 100))
# Результат: 1 a 100
```

```
print(func('abc', 14))
# abc 14 10
```

или можно комбинировать передачу аргументов с именем и без него. Тогда те, у кого есть имя, должны следовать за аргументами без имени, но порядок аргументов с именем не имеет значения:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Результат: This is StackOverflow Documentation
```

Глава 34. Создание функций со списочными аргументами

34.1. Функция и вызов

Списки в качестве аргументов – это просто еще одна переменная:

```
def func(myList):
    for item in myList:
        print(item)
```

и может передаваться в самом вызове функции:

```
func([1,2,3,5,7])
1
2
3
5
7
```


или в качестве переменной:

```
aList = ['a','b','c','d']
func(aList)
a
b
c
d
```

Глава 35. Функциональное программирование в Python

Функциональное программирование позволяет разложить задачу на набор функций. В идеале функции только принимают входные данные и производят выходные данные и не имеют никакого внутреннего состояния, которое влияло бы на выход, производимый при заданном входном сигнале. Ниже рассматриваются функциональные техники, общие для многих языков, такие как `lambda`, `map`, `reduce`.

35.1. Лямбда-функция

Это анонимная встроенная функция, определенная с помощью лямбда-выражения. Параметры лямбда-функции указываются слева от двоеточия. Справа от двоеточия указывается тело функции. Результат выполнения тела функции возвращается неявно.

```
s=lambda x:x*x
s(2) =>4
```

35.2. Функция `map`

Функция `map` принимает функцию и коллекцию предметов. Она создает новую пустую коллекцию, запускает функцию для каждого элемента в исходной коллекции и вставляет каждое возвращаемое значение в новую коллекцию. Возвращает новую коллекцию. Приведем простой пример такой функции, которая принимает список имен и возвращает список длин этих имен:

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])
print(name_lengths) =>[4, 4, 3]
```

35.3. Функция `reduce`

Принимает функцию и коллекцию предметов. Возвращает значение, которое создается путем объединения элементов. Приведем пример простого использования этой функции, которая возвращает сумму всех предметов в коллекции.

```
total = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print(total) =>10
```

35.4. Функция `filter`

Принимает функцию и коллекцию. Возвращает коллекцию, состоящую из всех элементов, для которых функция вернула значение `True`.

```
arr=[1,2,3,4,5,6]
[i for i in filter(lambda x:x>4,arr)] # выводит [5,6]
```

Глава 36. Частичные функции

Параметр	Подробности
x	возводимое в степень число
y	показатель степени
raise (возведение в степень)	функция, которая будет специализируемой

Как вы, вероятно, знаете, в объектно-ориентированном программировании специализация абстрактного класса и его использование – это практика, о которой следует помнить при написании кода. Что если бы вы могли определить абстрактную функцию и специализировать эту функцию для создания различных ее версий? Это можно представить как своего рода *наследование функций*, когда привязываются конкретные параметры, чтобы сделать их подходящими для конкретного сценария использования.

36.1. Возведение в степень

Давайте предположим, что мы хотим возвести x в степень y. Вы бы написали это как:

```
def raise_power(x, y):
    return x**y
```

Но что делать, если y может принимать конечное множество значений? Давайте предположим, что y может принимать значение из диапазона [3,4,5], и вы не хотите предлагать конечному пользователю возможность использовать такую функцию, так как она требует больших вычислительных затрат. На самом деле вы должны проверить, является ли заданное y допустимым значением, и переписать свою функцию в виде:

```
def raise(x, y):
    if y in (3,4,5):
        return x**y
    raise NumberNotInRangeException("You should provide a valid exponent")
```

Выглядит беспорядочно? Воспользуемся абстрактной формой и специализируем ее для всех трех случаев: реализуем их **частично**.

```
from functors import partial
raise_to_three = partial(raise, y=3)
raise_to_four = partial(raise, y=4)
raise_to_five = partial(raise, y=5)
```

Что здесь реализовано? Мы зафиксировали параметры y и определили три различные функции. Вообще нет необходимости использовать абстрактную функцию, определенную выше, но можно использовать **частичные прикладные функции** для решения задач наподобие возведения числа в степень с фиксированным значением.

Глава 37. Декораторы

Параметр	Подробности
f	функция, которая должна быть декорирована (обернута)

Декоративные функции – это паттерны проектирования программного обеспечения. Они динамически изменяют функциональность функции, метода или класса без необходимости прямого использования подклассов или изменения исходного кода декорируемой функции. При правильном использовании декораторы могут стать мощным инструментом в процессе разработки.

37.1. Функция-декоратор

Декораторы дополняют поведение других функций или методов. В качестве **декоратора** может быть использована любая функция, принимающая в качестве параметра функцию и возвращающая дополненную функцию.

```
# Этот простейший декоратор ничего не делает с декорируемой функцией.
# Такие минимальные декораторы иногда можно использовать в качестве
# своеобразных маркеров кода.
def super_secret_function(f):
    return f
```

```
@super_secret_function
def my_function():
    print("This is my secret function.")
```

Здесь @-нотация – это “синтаксический сахар”, эквивалентный следующему:

```
my_function = super_secret_function(my_function)
```

Это важно иметь в виду, чтобы понять, как работают декораторы. Синтаксис без использования синтаксического сахара дает понять, почему функция декоратора принимает функцию в качестве аргумента и почему она должна возвращать другую функцию. Он также демонстрирует, что произойдет, если вы *не вернете* функцию:

```
def disabled(f):
    """
    Эта функция ничего не возвращает и, следовательно, удаляет декорированную функцию
    из локальной области видимости.
    """
    pass

@disabled
def my_function():
    print("Эта функция больше не может быть вызвана...")

my_function()
# TypeError: 'NoneType' object is not callable
```

Таким образом, мы обычно определяем *новую* функцию внутри декоратора и возвращаем ее. Эта новая функция сначала сделает то, что ей нужно сделать, затем вызовет исходную функцию и, наконец, обработает возвращаемое значение. Рассмотрим простую функцию декоратора, которая выводит аргументы, получаемые исходной функцией, а затем вызывает ее.

```
# Это декоратор
def print_args(func):
    def inner_func(*args, **kwargs):
        print(args)
        print(kwargs)
        return func(*args, **kwargs) # Вызов исходной функции с ее аргументами
    return inner_func
```

```
@print_args
def multiply(num_a, num_b):
    return num_a * num_b
```

```
print(multiply(3, 5))
#Результат:
# (3,5) - Это 'args', которые получает функция
# {} - Это 'kwargs', пустые, потому что мы не указали именованных аргументов
# 15 - Результат работы функции
```

37.2. Класс декоратора

Как упоминалось во введении, декоратор – это функция, которая может быть применена к другой функции, чтобы дополнить, “обернуть” ее. Синтаксический сахар эквивалентен следующей записи: `my_func = decorator(my_func)`. Но что если использовать `decorator` вместо класса? Синтаксис все равно будет работать, за исключением того, что теперь `my_func` заменяется экземпляром класса `decorator`. Если этот класс реализует “магический метод” `__call__()`, тогда все же можно будет использовать `my_func` как функцию:

```
class Decorator(object):
    """Простой класс декоратора."""

    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('До вызова функции.')
        res = self.func(*args, **kwargs)
        print('После вызова функции.')
        return res

@Decorator
def testfunc():
    print('Внутри функции.')

testfunc()
# До вызова функции.
# Внутри функции.
# После вызова функции.
```

Обратите внимание, что функция, украшенная декоратором класса, больше не будет считаться “функцией” с точки зрения проверки типов:

```
import types
isinstance(testfunc, types.FunctionType)
# False
type(testfunc)
# <class '__main__.Decorator'>
```

Методы декорирования

Для методов декорирования вам необходимо определить дополнительный метод `__get__`:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('внутри декоратора')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Возвращает метод, если он вызван на экземпляре
        return self if instance is None else MethodType(self, instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass

a = Test()

        внутри декоратора
```

Предупреждение!

Декораторы классов создают только один экземпляр для конкретной функции, поэтому при декорировании метода декоратором класса один и тот же декоратор будет использоваться всеми экземплярами данного класса:

```
from types import MethodType

class CountCallsDecorator(object):
    def __init__(self, func):
        self.func = func
        self.ncalls = 0 # Количество вызовов данного метода

    def __call__(self, *args, **kwargs):
        self.ncalls += 1 # Увеличение счетчика вызовов
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        return self if instance is None else MethodType(self, instance)

class Test(object):
    def __init__(self):
        pass

    @CountCallsDecorator
    def do_something(self):
        return 'something was done'

a = Test()
a.do_something()
a.do_something.ncalls # 1
b = Test()
b.do_something()
b.do_something.ncalls # 2
```

37.3. Декоратор с аргументами (фабрика декораторов)

Декоратор принимает только один аргумент: декорируемую функцию. Передача других аргументов не предусмотрена. Но дополнительные аргументы часто бывают нужны. Тогда хитрость заключается в том, чтобы сделать функцию, которая принимает произвольные аргументы и возвращает декоратор.

Функции декоратора

```
def decoratorfactory(message):
    def decorator(func):
        def wrapped_func(*args, **kwargs):
            print('Декоратор хочет сказать вам: {}'.format(message))
            return func(*args, **kwargs)
        return wrapped_func
    return decorator

@decoratorfactory('Hello World')
def test():
    pass

test()

    Декоратор хочет сказать вам: Hello World
```

Важная заметка:

При применении фабрик декораторов **необходимо** вызывать декоратор с парой круглых скобок:

```
@decoratorfactory # Без скобок
def test():
    pass
```

```
test()
```

```
TypeError: decorator() missing 1 required positional argument: 'func'
```

Классы декораторов

```
def decoratorfactory(*decorator_args, **decorator_kwargs):
```

```
    class Decorator(object):
        def __init__(self, func):
            self.func = func

        def __call__(self, *args, **kwargs):
            print('Внутри декоратора с аргументами {}'.format(decorator_args))
            return self.func(*args, **kwargs)
```

```
    return Decorator
```

```
@decoratorfactory(10)
def test():
    pass
```

```
test()
```

```
Внутри декоратора с аргументами (10,)
```

37.4. Приведение декоратора к виду декорируемой функции

Декораторы обычно сбрасывают метаданные функций, поскольку они не совпадают. Это может вызвать проблемы при использовании метапрограммирования для динамического доступа к метаданным функции. Метаданные также включают в себя документацию функции (docstrings) и ее имя. Функция `functools.wraps` делает декорированную функцию похожей на оригинальную путем копирования нескольких атрибутов в оборачивающую функцию.

```
from functools import wraps
```

Оба способа обертывания декоратора позволяют добиться одного и того же – скрыть, что исходная функция была декорирована. Нет причин отдавать предпочтение версии функции перед версией класса, если только вы уже не используете одну из них вместо другой.

Как функция

```
def decorator(func):
    # Копирует docstring, имя, аннотации и модуль в декоратор
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapped_func
```

```
@decorator
def test():
    pass
```

```
test.__name__
```

```
'test'
```

Как класс

```
class Decorator(object):
    def __init__(self, func):
        # Копирует имя, модуль, аннотации и docstring в экземпляр
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring of test."""
    pass

test.__doc__
    Docstring of test.
```

37.5. Использование декоратора для определения времени выполнения функции

```
import time
def timer(func):
    def inner(*args, **kwargs):
        t1 = time.time()
        f = func(*args, **kwargs)
        t2 = time.time()
        print 'Время выполнения заняло {0} секунд'.format(t2-t1)
        return f
    return inner

@timer
def example_function():
    # ВЫПОЛНИТЬ ЧТО-ТО

example_function()
```

37.6. Создание класса-синглтона (одиночки) с помощью декоратора

Синглтон (одиночка) – это паттерн, который ограничивает инстанцирование класса одним экземпляром/объектом. Используя декоратор, мы можем определить класс как синглтон, заставив класс либо вернуть существующий экземпляр класса, либо создать новый экземпляр (если он не существует).

```
def singleton(cls):
    instance = [None]
    def wrapper(*args, **kwargs):
        if instance[0] is None:
            instance[0] = cls(*args, **kwargs)
        return instance[0]

    return wrapper
```

Такой декоратор может быть добавлен в любое объявление класса и будет гарантировать создание не более одного экземпляра класса. Все последующие вызовы вернут уже существующий экземпляр класса.

```
@singleton
class SomeSingletonClass:
    x = 2
    def __init__(self):
        print("Создан!")

instance = SomeSingletonClass() # выводит: Создан!
instance = SomeSingletonClass() # ничего не выводит
print(instance.x)               # 2

instance.x = 3
print(SomeSingletonClass().x)   # 3
```

Поэтому не имеет значения, обращаетесь ли вы к экземпляру класса через локальную переменную или создаете другой “экземпляр” – вы всегда получаете один и тот же объект.

Глава 38. Классы

Python является не только популярным скриптовым языком, но и поддерживает парадигму объектно-ориентированного программирования. Классы описывают данные и предоставляют методы для манипулирования этими данными. Кроме того, классы позволяют абстрагироваться, отделяя конкретные детали реализации от абстрактных представлений данных.

Код, использующий классы, обычно легче читать, понимать и сопровождать.

38.1. Введение в классы

Класс функционирует как шаблон, определяющий основные характеристики конкретного объекта. Приведем пример:

```
class Person(object):
    """Простой класс."""           # docstring
    species = "Homo Sapiens"        # атрибут класса

    def __init__(self, name):        # специальный метод
        """Это инициализатор. Это специальный
        метод (см. ниже)."""
        self.name = name            # атрибут экземпляра

    def __str__(self):               # специальный метод
        """Этот метод запускается, когда Python
        пытается привести объект к строке.
        Возвращайте эту строку при использовании функции print() и т. д.
        """
        return self.name

    def rename(self, renamed):        # обычный метод
        """Переназначить и вывести на печать атрибут name."""
        self.name = renamed
        print("Now my name is {}".format(self.name))
```

При рассмотрении приведенного выше примера следует обратить внимание на несколько моментов.

1. Класс состоит из *атрибутов* (данных) и *методов* (функций).
 2. Атрибуты и методы определяются просто как обычные переменные и функции.
 3. Как отмечено в соответствующей docstring, метод `__init__()` называется *инициализатором*. Он эквивалентен конструктору в других объектно-ориентированных языках, и это метод, который первым запускается при создании нового объекта или нового экземпляра класса.

4. Атрибуты, которые относятся ко всему классу, определяются сначала и называются *атрибутами класса*.

5. Атрибуты, относящиеся к конкретному экземпляру класса (объекту), называются *атрибутами экземпляра*. Обычно они задаются внутри `__init__()`; это необязательно, но рекомендуется (поскольку атрибуты, определенные вне `__init__()`, подвергаются риску доступа к ним до их определения).

6. Каждый метод, включенный в определение класса, передает рассматриваемый объект в качестве первого параметра. Слово `self` используется для этого параметра (использование `self` на самом деле условно, так как слово `self` не имеет неотъемлемого значения в Python, но это одно из наиболее важных условных обозначений Python, которому необходимо следовать).

7. Тех, кто привык к объектно-ориентированному программированию на других языках, могут удивить некоторые моменты. Во-первых, у Python нет реальной концепции `private` элементов, поэтому все элементы по умолчанию имитируют поведение ключевого слова `public` в языках C++/Java. Для получения дополнительной информации см. пример “Члены частного класса”.

8. Некоторые из методов класса имеют следующий вид: `__functionname__(self, other_stuff)`. Все подобные методы называются “магическими” и являются важной частью классов в Python. Например, перегрузка операторов в Python реализуется с помощью магических методов.

Теперь давайте сделаем несколько экземпляров нашего класса `Person`!

```
>>> # Экземпляры
>>> kelly = Person("Kelly")
>>> joseph = Person("Joseph")
>>> john_doe = Person("John Doe")
```

В настоящее время у нас есть три объекта класса `Person`, это `kelly`, `joseph` и `john_doe`. Мы можем получить доступ к атрибутам класса из каждого экземпляра, используя оператор точечный “.”. Еще раз обратите внимание на разницу между атрибутами класса и экземпляра:

```
>>> # Атрибуты
>>> kelly.species
'Homo Sapiens'
>>> john_doe.species
'Homo Sapiens'
>>> joseph.species
'Homo Sapiens'
>>> kelly.name
'Kelly'
>>> joseph.name
'Joseph'
```

Мы можем выполнять методы класса, используя тот же точечный оператор “.”:

```
>>> # Методы
>>> john_doe.__str__()
'John Doe'
>>> print(john_doe)
'John Doe'
>>> john_doe.rename("John")
'Now my name is John'
```

38.2. Связанные, несвязанные и статические методы

Идея связанных и несвязанных методов в Python 3 была удалена. Когда вы объявляете в этой версии метод внутри класса, вы используете ключевое слово `def`, тем самым создавая объект функции. Это обычная функция, а окружающий класс работает как ее пространство имен. В следующем примере мы объявляем метод `f` в классе `A`, и он становится функцией `A.f`:

Версия Python 3.x ≥ 3.0

```
class A(object):
    def f(self, x):
        return 2 * x
A.f
# <function A.f at ...> (Python 3.x)
```

В Python 2 поведение было иным: функциональные объекты внутри класса были неявно заменены объектами типа `instancemethod`, которые назывались *несвязанными методами*, потому что они не были привязаны к какому-либо конкретному экземпляру класса. Можно было получить доступ к базовой функции, используя свойство `__func__`.

Python 2.x 2.3

```
A.f
# <несвязанный метод A.f> (Python 2.x)
A.f.__class__
# <type 'instancemethod'>
A.f.__func__
# <function f at ...>
```

Последнее поведение подтверждается проверкой – методы распознаются как функции в Python 3, то время как в Python 2 это различие сохраняется.

Версия Python 3.x ≥ 3.0:

```
import inspect

inspect.isfunction(A.f)
# True
inspect.ismethod(A.f)
# False
```

Версия Python 2.x ≥ 2.3:

```
import inspect

inspect.isfunction(A.f)
# False
inspect.ismethod(A.f)
# True
```

В обеих версиях Python функцию/метод `A.f` можно вызывать напрямую, при условии, что вы передаете экземпляр класса `A` в качестве первого аргумента.

```
A.f(1, 7)
# Python 2: TypeError: несвязанный метод f() должен вызывать экземпляр A
#           в качестве первого аргумента (вместо него был получен экземпляр int)
# Python 3: 14
a = A()
A.f(a, 20)
# Python 2 & 3: 40
```

Предположим теперь, что `a` является экземпляром класса `A`, что тогда `a.f`? Интуитивно это должен быть тот же метод `f` класса `A`, только он должен каким-то образом “знать”, что он был применен к объекту `a` – в Python это называется методом, *связанным* с `a`.

Тонкости заключаются в следующем: запись `a.f` вызывает магический метод `__getattr__`, который сначала проверяет, есть ли у `a` атрибут с именем `f` (такого нет), а затем проверяет класс `A`, содержит ли он метод с таким именем (содержит), и создает новый объект `m` типа `method`, который имеет ссылку на исходный `A.f` в `m.__func__` и ссылку на объект `a` в `m.__self__`. Когда этот объект вызывается как функция, он просто выполняет следующее: `m(...) => m.__func__(m.__self__, ...)`. Таким образом, этот объект называется **связанным методом**, потому что при вызове он знает, что в качестве первого аргумента нужно передать объект, к которому он был привязан. (Это работает одинаково в Python 2 и 3.)

```
a = A()
a.f
# <связанный метод A.f из <__main__.A object at ...>>
a.f(2)
# 4
```

```
# Примечание: связанный объект метода a.f создается заново *при каждом* его вызове:
a.f is a.f # False
# В качестве оптимизации производительности можно хранить связанный метод
# в __dict__ объекта, в этом случае объект метода останется фиксированным:
a.f = a.f
a.f is a.f # True
```

Наконец, Python имеет **методы класса** и **статические методы** – особые виды методов. Методы класса работают так же, как и обычные методы, за исключением того, что при вызове на объект они привязываются к *классу* объекта, а не к объекту. Таким образом, `m.__self__ = type(a)`. Когда вы вызываете такой связанный метод, он передает класс `a` в качестве первого аргумента. Статические методы еще проще: они вообще ничего не связывают и просто возвращают базовую функцию без каких-либо преобразований.

```
class D(object):
    multiplier = 2

    @classmethod
    def f(cls, x):
        return cls.multiplier * x

    @staticmethod
    def g(name):
        print("Hello, %s" % name)

D.f
# <связанный метод type.f из <class '__main__.D'>>
D.f(12)
# 24
D.g
# <function D.g at ...>
D.g("world")
# Hello, world
```

Обратите внимание, что методы класса привязаны к классу даже при обращении к экземпляру:

```
d = D()
d.multiplier = 1337
(D.multiplier, d.multiplier)
```

```
# (2, 1337)
d.f
# <связанный метод D.f из <class '__main__.D'>>
d.f(10)
# 20
```

Стоит отметить, что на самом низком уровне функции, методы, статические методы и т. д. фактически являются дескрипторами, которые вызывают специальные методы `__get__`, `__set__` и, необязательно, `__del__`.

38.3. Базовое наследование

Наследование в Python основано на тех же идеях, что и в других объектно-ориентированных языках, таких как Java, C++ и др. Новый класс может быть получен от существующего класса следующим образом.

```
class BaseClass(object):
    pass

class DerivedClass(BaseClass):
    pass
```

Здесь `BaseClass` – уже существующий (*родительский*) класс, а `DerivedClass` – новый (*дочерний*) класс, который наследует (подклассифицирует) атрибуты из `BaseClass`.

Примечание: начиная с Python 2.2 все классы неявно наследуют от класса `object`, который является базовым классом для всех встроенных типов.

Определим в приведенном ниже примере родительский класс `Rectangle` (прямоугольник), который неявно наследует от `object`:

```
class Rectangle():
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def area(self):
        return self.w * self.h

    def perimeter(self):
        return 2 * (self.w + self.h)
```

Класс `Rectangle` может использоваться как базовый класс для определения класса `Square` (квадрат), так как квадрат является частным случаем прямоугольника.

```
class Square(Rectangle):
    def __init__(self, s):
        # вызов родительского конструктора, w и h оба являются s
        super(Square, self).__init__(s, s)
        self.s = s
```

Класс `Square` автоматически наследует все атрибуты класса `Rectangle`, а также класс объекта. `super()` используется для вызова метода `__init__()` класса `Rectangle`, по сути вызывая любой переопределенный метод базового класса.

Примечание: в Python 3 `super()` не требует аргументов.

Объекты производных классов могут получать и изменять атрибуты базовых классов:

```
r.area()
# Результат: 12
r.perimeter()
# Результат: 14
```

```
s.area()
# Результат: 4
s.perimeter()
# Результат: 8
```

Встроенные функции, которые работают с наследованием

`issubclass(DerivedClass, BaseClass)`: возвращает `True`, если `DerivedClass` является подклассом `BaseClass`.

`isinstance(s, Class)`: возвращает `True`, если `s` является экземпляром `Class` или любого из производных классов `Class`:

```
# проверка подкласса
issubclass(Square, Rectangle)
# Результат: True
```

```
# создание экземпляра
r = Rectangle(3, 4)
s = Square(2)
```

```
isinstance(r, Rectangle)
# Результат: True
isinstance(r, Square)
# Результат: False
# Прямоугольник не является квадратом
```

```
isinstance(s, Rectangle)
# Результат: True
# Квадрат является прямоугольником
isinstance(s, Square)
# Результат: True
```

38.4. Манкипатчинг (Monkey Patching)

В данном случае манкипатчинг (Monkey Patching, “обезьянье исправление”) означает добавление новой переменной или метода в класс после того, как он был определен. Допустим, мы определили класс `A` как:

```
class A(object):
    def __init__(self, num):
        self.num = num

    def __add__(self, other):
        return A(self.num + other.num)
```

Но теперь мы хотим добавить в код еще одну функцию. Предположим, что эта функция такова:

```
def get_num(self):
    return self.num
```

Но как добавить это в качестве метода в `A`? Это просто, мы по существу помещаем эту функцию в `A` с помощью оператора присваивания.

```
A.get_num = get_num
```

Почему это работает? Потому что функции – это объекты, как и любые другие объекты, а методы – это функции, принадлежащие классу. Функция `get_num` должна быть доступна всем существующим (уже созданным), а также новым экземплярам класса `A`. Эти дополнения автоматически становятся доступными для всех экземпляров этого класса (или его подклассов). Например:

```
foo = A(42)

A.get_num = get_num

bar = A(6);

foo.get_num() # 42

bar.get_num() # 6
```

Обратите внимание, что в отличие от некоторых других языков этот прием не работает для определенных встроенных типов, и не считается хорошим стилем программирования.

38.5. Классы нового и старого стиля

Версия Python 2.x ≥ 2.2.0:

Классы *нового стиля* были введены в Python 2.2 для унификации классов и типов. Они наследуются от типа верхнего уровня `object`. Класс нового стиля – это *тип, определяемый пользователем*, и он очень похож на встроенные типы.

```
# класс нового стиля
class New(object):
    pass

# экземпляр нового стиля
new = New()

new.__class__
# <class '__main__.New'>
type(new)
# <class '__main__.New'>
issubclass(New, object)
# True
```

Классы *старого стиля* не наследуются от `object`. Экземпляры старого стиля всегда реализуются со встроенным типом `instance`.

```
# класс старого стиля
class Old:
    pass

# экземпляр старого стиля
old = Old()

old.__class__
# <class '__main__.Old at ...'>
type(old)
# <type 'instance'>
issubclass(Old, object)
# False
```

Версия Python 3.x ≥ 3.0.0:

В Python 3 классы старого стиля были удалены. Классы нового стиля в Python 3 неявно наследуют от `object`, поэтому больше не нужно указывать `MyClass(object)`.

```
class MyClass:
    pass

my_inst = MyClass()
```

```

type(my_inst)
# <class '__main__.MyClass'>
my_inst.__class__
# <class '__main__.MyClass'>
issubclass(MyClass, object)
# True

```

38.6. Методы класса: альтернативные инициализаторы

Методы классов представляют собой альтернативные способы создания экземпляров классов. Для иллюстрации рассмотрим пример. Предположим, что у нас есть относительно простой класс `Person`:

```

class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

    def greet(self):
        print("Здравствуйте, меня зовут" + self.full_name + ".")

```

Возможно, было бы удобно создать экземпляры этого класса, указав полное имя вместо имени и фамилии отдельно. Один из способов сделать это – сделать `last_name` необязательным параметром и считать, что если он не задан, то мы передали полное имя:

```

class Person(object):

    def __init__(self, first_name, age, last_name=None):
        if last_name is None:
            self.first_name, self.last_name = first_name.split(" ", 2)
        else:
            self.first_name = first_name
            self.last_name = last_name

        self.full_name = self.first_name + " " + self.last_name
        self.age = age

    def greet(self):
        print("Здравствуйте, меня зовут" + self.full_name + ".")

```

Однако с этим кодом есть две основные проблемы.

1. Параметры `first_name` и `last_name` теперь вводят в заблуждение, так как вы можете ввести полное имя для `first_name`. Кроме того, если случаев и/или параметров, обладающих такой гибкостью, больше, то ветвление `if/elif/else` может быстро надоесть.

2. Не так важно, но все же стоит обратить внимание: что если `last_name` имеет значение `None`, но `first_name` не разбивается на две или более части пробелами? Получаем еще один уровень проверки ввода и/или обработки исключений ...

Введем методы класса. Вместо того чтобы иметь один инициализатор создадим отдельный инициализатор, называемый `from_full_name`, и декорируем его (встроенным) `classmethod` декоратором.

```

class Person(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
        self.full_name = first_name + " " + last_name

```

```
@classmethod
def from_full_name(cls, name, age):
    if " " not in name:
        raise ValueError
    first_name, last_name = name.split(" ", 2)
    return cls(first_name, last_name, age)

def greet(self):
    print("Здравствуйте, меня зовут" + self.full_name + ".")
```

Обратите внимание на `cls` вместо `self` в качестве первого аргумента `from_full_name`. Методы класса применяются к классу в целом, а не к экземпляру данного класса (что обычно обозначается `self`). Итак, если `cls` – это наш класс `Person`, тогда возвращаемое значение из метода класса `from_full_name` – это `Person(first_name, last_name, age)`, которое использует `__init__` класса `Person` для создания экземпляра класса `Person`. В частности, если мы должны создать подкласс `Employee` класса `Person`, то `from_full_name` будет работать и в классе `Employee`.

Чтобы показать, что все работает как надо, создадим экземпляры `Person` более чем одним способом без ветвления в `__init__`:

```
In [2]: bob = Person("Bob", "Bobberson", 42)

In [3]: alice = Person.from_full_name("Alice Henderson", 31)

In [4]: bob.greet()
Здравствуйте, меня зовут Bob Bobberson.

In [5]: alice.greet()
Здравствуйте, меня зовут Alice Henderson.
```

38.7. Множественное наследование

Python использует алгоритм линеаризации C3 (C3 linearization algorithm) для определения порядка разрешения атрибутов класса, включая методы. Этот порядок известен как порядок разрешения методов (Method Resolution Order, MRO).

Приведем простой пример:

```
class Foo(object):
    foo = 'attr foo of Foo'

class Bar(object):
    foo = 'attr foo of Bar' # мы этого не увидим
    bar = 'attr bar of Bar'

class FooBar(Foo, Bar):
    foobar = 'attr foobar of FooBar'
```

Теперь, если мы создаем `FooBar`, если мы найдем атрибут `foo`, мы увидим, что атрибут `Foo` найден первым:

```
fb = FooBar()
а также
>>> fb.foo
'attr foo of Foo'
```

Вот MRO для `FooBar`:

```
>>> FooBar.mro()
[<class '__main__.FooBar'>, <class '__main__.Foo'>, <class '__main__.Bar'>, <type 'object'>]
```


Проще говоря, алгоритм MRO Python является следующим:

1. Прежде всего глубина (т. е. сначала FooBar, затем Foo), кроме случая если
2. общий родитель (object) блокируется дочерним элементом (Bar) и
3. круговые отношения не допускаются.

То есть, например, Bar не может наследовать FooBar, а FooBar наследуется от Bar.

Еще один мощный инструмент в наследовании – функция super. Она может извлекать из родительских классов их возможности.

```
class Foo(object):
    def foo_method(self):
        print "foo Method"

class Bar(object):
    def bar_method(self):
        print "bar Method"

class FooBar(Foo, Bar):
    def foo_method(self):
        super(FooBar, self).foo_method()
```

Множественное наследование с методом init класса: когда каждый класс имеет собственный метод init, при попытке выполнения множественного наследования вызывается только метод init того класса, который наследуется первым. В примере ниже вызывается только метод init класса Foo, а метод init класса Bar не вызывается.

```
class Foo(object):
    def __init__(self):
        print "foo init"

class Bar(object):
    def __init__(self):
        print "bar init"

class FooBar(Foo, Bar):
    def __init__(self):
        print "foobar init"
        super(FooBar, self).__init__()
```

```
a = FooBar()
```

Результат:

```
foobar init
foo init
```

Но это не значит, что класс Bar не наследуется. Экземпляр конечного класса FooBar также является экземпляром класса Bar и класса Foo.

```
print isinstance(a, FooBar)
print isinstance(a, Foo)
print isinstance(a, Bar)
```

Выход:

```
True
True
True
```

38.8. Свойства

Классы Python поддерживают **свойства**, которые выглядят как обычные переменные объектов, но с возможностью прикрепления пользовательского поведения и документации.

```
class MyClass(object):

    def __init__(self):
        self._my_string = ""

    @property
    def string(self):
        """Очень важная строка."""
        return self._my_string

    @string.setter
    def string(self, new_value):
        assert isinstance(new_value, str), \
            "Дайте мне строку, а не %r!" % type(new_value)
        self._my_string = new_value

    @string.deleter
    def x(self):
        self._my_string = None
```

Кажется, что объект класса `MyClass` имеет свойство `.string`, однако его поведение теперь жестко контролируется:

```
mc = MyClass()
mc.string = "String!"
print(mc.string)
del mc.string
```

Помимо полезного синтаксиса, описанного выше, синтаксис свойств позволяет проверять или добавлять другие атрибуты к этим атрибутам. Это может быть особенно полезно для общедоступных API, где пользователю должен быть предоставлен определенный уровень помощи.

Другим распространенным способом использования свойств является предоставление классу возможности производить “виртуальные атрибуты” – атрибуты, которые фактически не хранятся, а вычисляются только при запросе.

```
class Character(object):
    def __init__(name, max_hp):
        self._name = name
        self._hp = max_hp
        self._max_hp = max_hp

    # Сделать hp только для чтения, не предоставляя метода set
    @property
    def hp(self):
        return self._hp

    # Сделать имя только для чтения, не предоставляя метода set
    @property
    def name(self):
        return self._name

    def take_damage(self, damage):
        self._hp -= damage
        self._hp = 0 if self._hp < 0 else self._hp

    @property
    def is_alive(self):
        return self._hp != 0
```

```

@property
def is_wounded(self):
    return self.hp < self.max_hp if self.hp > 0 else False

@property
def is_dead(self):
    return not self.is_alive

bilbo = Character('Bilbo Baggins', 100)
bilbo.hp
# результат : 100
bilbo.hp = 200
# результат : AttributeError: невозможно установить атрибут
# атрибут hp только для чтения.

bilbo.is_alive
# результат : True
bilbo.is_wounded
# результат : False
bilbo.is_dead
# результат : False

bilbo.take_damage( 50 )

bilbo.hp
# результат : 50

bilbo.is_alive
# результат : True
bilbo.is_wounded
# результат : True
bilbo.is_dead
# результат : False

bilbo.take_damage( 50 )
bilbo.hp
# результат : 0

bilbo.is_alive
# результат : False
bilbo.is_wounded
# результат : False
bilbo.is_dead
# результат : True

```

38.9. Значения по умолчанию для переменных экземпляра

Если переменная содержит значение неизменяемого типа (например строчного), то можно присвоить значение по умолчанию следующим образом:

```

class Rectangle(object):
    def __init__(self, width, height, color='blue'):
        self.width = width
        self.height = height
        self.color = color

    def area(self):
        return self.width * self.height

```

```
# Создать несколько экземпляров класса
default_rectangle = Rectangle(2, 3)
print(default_rectangle.color) # blue

red_rectangle = Rectangle(2, 3, 'red')
print(red_rectangle.color) # red
```

Нужно быть осторожным при инициализации изменяемых объектов, таких как списки в конструкторе. Рассмотрим следующий пример:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=[0,0], color='blue'):
        self.width = width
        self.height = height
        self.pos = pos
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [4, 0] Положение r2 также изменилось
```

Такое поведение вызвано тем, что в Python параметры по умолчанию связываются при выполнении функции, а не при ее объявлении. Чтобы получить переменную экземпляра по умолчанию, не разделяемую между экземплярами, необходимо использовать конструкцию, подобную этой:

```
class Rectangle2D(object):
    def __init__(self, width, height, pos=None, color='blue'):
        self.width = width
        self.height = height
        self.pos = pos or [0, 0] # значение по умолчанию - [0, 0]
        self.color = color

r1 = Rectangle2D(5,3)
r2 = Rectangle2D(7,8)
r1.pos[0] = 4
r1.pos # [4, 0]
r2.pos # [0, 0] позиция r2 не изменилась
```

38.10. Переменные класса и экземпляра

Переменные экземпляра уникальны для каждого экземпляра, в то время как переменные класса являются общими для всех экземпляров.

```
class C:
    x = 2          # переменная класса

    def __init__(self, y):
        self.y = y # переменная экземпляра

C.x
# 2
C.y
# AttributeError: объект типа 'C' не имеет атрибута 'y'

c1 = C(3)
c1.x
# 2
c1.y
# 3
```

```
c2 = C(4)
c2.x
# 2
c2.y
# 4
```

Доступ к переменным класса можно получить в экземплярах этого класса, но присвоение атрибуту класса создает переменную экземпляра, которая “затеняет” переменную класса.

```
c2.x = 4
c2.x
# 4
C.x
# 2
```

Обратите внимание, что *изменение* переменных класса из экземпляров может привести к неожиданным последствиям.

```
class D:
    x = []
    def __init__(self, item):
        self.x.append(item) # обратите внимание, что это не присваивание!

d1 = D(1)
d2 = D(2)

d1.x
# [1, 2]
d2.x
# [1, 2]
D.x
# [1, 2]
```

38.11. Композиция классов

Композиция классов позволяет устанавливать явные отношения между объектами. В данном примере люди живут в городах, которые принадлежат странам. Композиция позволяет людям получить доступ к количеству всех людей, живущих в их стране:

```
class Country(object):
    def __init__(self):
        self.cities=[]

    def addCity(self,city):
        self.cities.append(city)

class City(object):
    def __init__(self, numPeople):
        self.people = []
        self.numPeople = numPeople

    def addPerson(self, person):
        self.people.append(person)

    def join_country(self,country):
        self.country = country
        country.addCity(self)

    for i in range(self.numPeople):
        person(i).join_city(self)
```

```
class Person(object):
    def __init__(self, ID):
        self.ID=ID

    def join_city(self, city):
        self.city = city
        city.addPerson(self)

    def people_in_my_country(self):
        x= sum([len(c.people) for c in self.city.country.cities])
        return x

US=Country()
NYC=City(10).join_country(US)
SF=City(5).join_country(US)

print(US.cities[0].people[0].people_in_my_country())

# 15
```

38.12. Перечисление всех членов класса

Функция `dir()` может быть использована для получения списка членов класса:

```
dir(Class)
```

Например:

```
>>> dir(list)
['_add_', '_class_', '_contains_', '_delattr_', '_delitem_', '_dir_', '_doc_', '_eq_', '_format_',
'_ge_', '_getattribute_', '_getitem_', '_gt_', '_hash_', '_iadd_', '_imul_', '_init_', '_iter_', '_le_',
'_len_', '_lt_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_reversed_',
'_rmul_', '_setattr_', '_setitem_', '_sizeof_', '_str_', '_subclasshook_', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Обычно требуется искать только “магические” члены класса. Т. е. нужно перечисление членов, имена которых не начинаются с `_`:

```
>>> [m for m in dir(list) if not m.startswith('_')]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Предостережения:

Классы могут определять метод `__dir__()`. Если этот метод существует, вызов `dir()` вызовет `__dir__()`, в противном случае Python попытается создать список членов класса. Это означает, что функция `dir` может произвести неожиданные результаты. Две важные цитаты из официальной документации Python :

Если объект не предоставляет `dir()`, функция пытается собрать информацию из атрибута `dict` объекта, если он определен, и из его типа объекта. Полученный список не обязательно будет полным и может быть неточным, если у объекта есть пользовательский `getattr()`.

Примечание. Поскольку `dir()` используется в основном как удобное средство для использования в интерактивном режиме, он пытается предоставить интересный набор имен, а не строго или последовательно детерминированный их набор, и его поведение может меняться в разных выпусках. Например, атрибуты метаклассов не попадают в список результатов, если аргументом является класс.

38.13. Класс-синглтон

Синглтон (“одиночка”, Singleton) – это паттерн, который ограничивает инстанцирование класса одним экземпляром/объектом.

```
class Singleton:
    def __new__(cls):
        try:
```

```

        it = cls.__it__
    except AttributeError:
        it = cls.__it__ = object.__new__(cls)
    return it

def __repr__(self):
    return '<{}>'.format(self.__class__.__name__.upper())

def __eq__(self, other):
    return other is self

```

Другой способ – декорировать ваш класс. Создадим класс-синглтон:

```
class Singleton:
```

Непотокбезопасный вспомогательный класс, облегчающий реализацию синглтонов. Он должен использоваться в качестве декоратора – не метакласса – для класса, который должен быть синглтоном.

Декорированный класс может определить одну функцию `init`, которая принимает только аргумент `self`. В остальном для декорированного класса не существует никаких ограничений.

Чтобы получить экземпляр синглтона, используйте метод `Instance`. Попытка использовать `__call__` приведет к возникновению ошибки `TypeError`.

Ограничения: От декорированного класса нельзя наследовать.

```
***
```

```

def __init__(self, decorated):
    self._decorated = decorated

```

```
def Instance(self):
```

```
***
```

Возвращает экземпляр синглтона. При первом вызове создается новый экземпляр декорированного класса и вызывается его метод `__init__`. При всех последующих вызовах возвращается уже созданный экземпляр.

```
***
```

```

try:
    return self._instance
except AttributeError:
    self._instance = self._decorated()
    return self._instance

```

```

def __call__(self):
    raise TypeError('Доступ к синглтонам должен осуществляться через `Instance()`.')

```

```

def __instancecheck__(self, inst):
    return isinstance(inst, self._decorated)

```

Теперь можно воспользоваться методом Instance:

```

@Singleton
class Single:
    def __init__(self):
        self.name=None
        self.val=0
    def getName(self):
        print(self.name)

```

```
x=Single.Instance()
y=Single.Instance()
x.name='I\'m single'
x.getName() # выводит I'm single
y.getName() # выводит I'm single
```

38.14. Дескрипторы и точечный поиск

Дескрипторы – это объекты, которые являются (обычно) атрибутами классов и имеют любой из специальных методов `__get__`, `__set__`, или `__delete__`. **Дескрипторы данных** имеют или метод `__set__`, или `__delete__`. Они могут управлять точечным поиском экземпляра и используются для реализации функций, `staticmethod`, `classmethod` и `property`. Точечный поиск (например, экземпляр `foo` класса `Foo` ищет атрибут `bar` – т. е. `foo.bar`) использует следующий алгоритм:

1. `bar` ищется в классе `Foo`. Если он там есть и является *дескриптором данных*, то используется дескриптор данных. Именно так `property` может управлять доступом к данным в экземпляре, и экземпляры не могут переопределить его. Если дескриптор данных отсутствует, то:

2. `bar` ищется в `__dict__` экземпляра. Поэтому мы можем переопределять или блокировать методы, вызываемые из экземпляра с помощью точечного поиска. Если `bar` существует в экземпляре, то он используется. Если нет, то:

3. ищем `bar` в классе `Foo`. Если это *дескриптор*, то используется протокол дескриптора. Именно так реализуются функции (в данном контексте – несвязанные методы), `classmethod` и `staticmethod`. В противном случае просто возвращается объект либо возникает ошибка `AttributeError`.

Глава 39. Метаклассы

Метаклассы позволяют глубоко изменять поведение классов Python (в плане их определения, инстанцирования, доступа и т. д.), заменяя метакласс `type`, который новые классы используют по умолчанию.

39.1. Базовые метаклассы

Когда `type` вызывается с тремя аргументами, он ведет себя как (мета)класс, которым он является, и создает новый экземпляр, т. е. порождает новый класс/тип.

```
Dummy = type('OtherDummy', (), dict(x=1))
Dummy.__class__          # <type 'type'>
Dummy().__class__.__class__ # <type 'type'>
```

Можно подклассифицировать `type` для создания собственного метакласса.

```
class mytype(type):
    def __init__(cls, name, bases, dict):
        # вызов базового инициализатора
        type.__init__(cls, name, bases, dict)

        # выполнить пользовательскую инициализацию...
        cls.__custom_attribute__ = 2
```

Теперь у нас есть новый пользовательский метакласс `mytype`, который можно использовать для создания классов таким же образом, как и `type`.

```
MyDummy = mytype('MyDummy', (), dict(x=2))
MyDummy.__class__          # <class '__main__.mytype'>
MyDummy().__class__.__class__ # <class '__main__.mytype'>
MyDummy.__custom_attribute__ # 2
```


При создании нового класса с помощью ключевого слова `class` метакласс по умолчанию выбирается на основе базовых классов.

```
>>> class Foo(object):
...     pass

>>> type(Foo)
type
```

В приведенном выше примере единственным базовым классом является `object`, поэтому наш метакласс будет с типом `object`, который является `type`. Можно переопределить значение по умолчанию, однако это зависит от того, используем ли мы Python 2 или Python 3:

Версия Python 2.x ≤ 2.7:

Для определения метакласса можно использовать специальный атрибут `__metaclass__`.

```
class MyDummy(object):
    __metaclass__ = mytype
type(MyDummy) # <class '__main__.mytype'>
```

Версия Python 3.x ≥ 3.0:

Специальный именованный аргумент `metaclass` определяет метакласс.

```
class MyDummy(metaclass=mytype):
    pass
type(MyDummy) # <class '__main__.mytype'>
```

Любые аргументы ключевых слов (кроме `metaclass`) в объявлении класса будут переданы в метакласс. Таким образом, `class MyDummy(metaclass=mytype, x=2)` передаст `x=2` в качестве именованного аргумента в конструктор `mytype`.

39.2. Синглтоны, использующие метаклассы

Синглтон – это паттерн, который ограничивает инстанцирование класса одним экземпляром/объектом.

```
class SingletonType(type):
    def __call__(cls, *args, **kwargs):
        try:
            return cls.__instance
        except AttributeError:
            cls.__instance = super(SingletonType, cls).__call__(*args, **kwargs)
            return cls.__instance
```

Версия Python 2.x ≤ 2.7

```
class MySingleton(object):
    __metaclass__ = SingletonType
```

Версия Python 3.x ≥ 3.0

```
class MySingleton(metaclass=SingletonType):
    pass
MySingleton() is MySingleton() # True, происходит только одно инстанцирование
```

39.3. Использование метакласса

Синтаксис метаклассов

Версия Python 2.x ≤ 2.7:

```
class MyClass(object):
    __metaclass__ = SomeMetaclass
```

Версия Python 3.x ≥ 3.0:

```
class MyClass(metaclass=SomeMetaclass):
    pass
```

Совместимость Python 2 и 3 с библиотекой six

```
import six

class MyClass(six.with_metaclass(SomeMetaclass)):
    pass
```

39.4. Введение в метаклассы

Что такое метакласс?

В языке Python все является объектами: целые числа, строки, списки, даже функции и классы. И каждый объект является экземпляром класса. Чтобы проверить класс объекта *x*, можно вызвать `type(x)`:

```
>>> type(5)
<type 'int'>
>>> type(str)
<type 'type'>
>>> type([1, 2, 3])
<type 'list'>

>>> class C(object):
    pass

>>> type(C)
<type 'type'>
```

Большинство классов в Python являются экземплярами `type`. Сам `type` также является классом. Такие классы, экземпляры которых также являются классами, называются метаклассами.

Самый простой метакласс

Итак, мы уже знаем один метакласс в Python: `type`. Можем ли мы создать еще один?

```
class SimplestMetaclass(type):
    pass

class MyClass(object):
    __metaclass__ = SimplestMetaclass
```

Это не добавляет каких-либо функциональных возможностей, но это новый метакласс, `MyClass` теперь является экземпляром `SimplestMetaclass`:

```
>>> type(MyClass)
<class '__main__.SimplestMetaclass'>
```

Метакласс, который что-то делает

Метакласс, который что-то делает, обычно переопределяет у `type` метод `__new__`, чтобы изменить некоторые свойства создаваемого класса, прежде чем вызывать исходный `__new__`, который создает класс:

```
class AnotherMetaclass(type):
    def __new__(cls, name, parents, dct):
        # cls – этот класс
        # name – имя создаваемого класса
        # parents – список родительских классов класса
```

```
# dct – список атрибутов класса (методов, статических переменных)
# здесь все атрибуты могут быть изменены до создания класса, например:

dct['x'] = 8 # теперь класс будет иметь статическую переменную x = 8

# возвращаемое значение – новый класс super позаботится об этом
return super(AnotherMetaclass, cls).__new__(cls, name, parents, dct)
```

39.5. Пользовательская функциональность в метаклассах

Функциональность в метаклассах может быть изменена таким образом, чтобы при сборке класса на стандартный вывод выводилась строка или возникало исключение. Этот метакласс выводит имя собираемого класса.

```
class VerboseMetaclass(type):
```

```
    def __new__(cls, class_name, class_parents, class_dict):
        print("Создание класса", class_name)
        new_class = super().__new__(cls, class_name, class_parents, class_dict)
        return new_class
```

Вы можете использовать метакласс следующим образом:

```
class Spam(metaclass=VerboseMetaclass):
    def eggs(self):
        print("[вставьте здесь пример строки]")
s = Spam()
s.eggs()
```

Стандартным выходом будет:

```
Создание класса Spam
[вставьте здесь пример строки]
```

39.6. Метакласс по умолчанию

В Python все является объектами, и у всех объектов есть класс:

```
>>> type(1)
int
```

Литерал 1 является экземпляром int. Объявим класс:

```
>>> class Foo(object):
...     pass
```

Теперь давайте создадим экземпляр:

```
>>> bar = Foo()
```

Какой класс у bar?

```
>>> type(bar)
Foo
```

Отлично, bar – это экземпляр Foo. Но каков класс самого Foo?

```
>>> type(Foo)
type
```

Хорошо, сам Foo является экземпляром type. Как насчет type?

```
>>> type(type)
type
```

Итак, что такое метакласс? Пока представим, что это просто причудливое имя для класса класса. Выводы:

- В Python всё является объектом, поэтому всё имеет класс.
- Класс класса называется метаклассом.
- Метакласс по умолчанию – это `type`, и на сегодняшний день это самый распространенный метакласс.

Но почему вы должны знать о метаклассах? Python сам по себе является довольно “взламываемым” языком, и понятие метакласса важно, если вы занимаетесь продвинутым программированием, например метапрограммированием, или хотите контролировать инициализацию своих классов.

Глава 40. Форматирование строк

При хранении и преобразовании данных для восприятия человеком форматирование строк может оказаться очень важным. Python предлагает широкий спектр методов форматирования строк, которые описаны в этой главе.

40.1. Основы форматирования строк

```
foo = 1
bar = 'bar'
baz = 3.14
```

Вы можете использовать `str.format` для форматирования вывода. Пары скобок заменяются аргументами в том порядке, в котором передаются аргументы:

```
print('{} , {} and {}'.format(foo, bar, baz))
# Вывод: "1, bar and 3.14"
```

Индексы также могут быть указаны внутри скобок. Номера соответствуют индексам аргументов, переданных функции `str.format` (на основе 0).

```
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Вывод: "1, bar, 3.14, and bar"
print('{0}, {1}, {2}, and {3}'.format(foo, bar, baz))
# Вывод: ошибка выхода индекса за пределы диапазона
```

Можно также использовать именованные аргументы:

```
print("X value is: {x_val}. Y value is: {y_val}.".format(x_val=2, y_val=3))
# Вывод: "X value is: 2. Y value is: 3."
```

На атрибуты объектов можно ссылаться при передаче в `str.format`:

```
class AssignValue(object):
    def __init__(self, value):
        self.value = value
my_value = AssignValue(6)
print('Мое значение: {0.value}'.format(my_value)) # "0" необязателен
# Вывод: "Мое значение: 6"
```

Также можно использовать словарные ключи:

```
my_dict = {'key': 6, 'other_key': 7}
print("Мой дпругой ключ: {0[other_key]}".format(my_dict)) # "0" is optional
# Вывод: "Мой дпругой ключ: 7"
```

То же самое относится к индексам списка и кортежей:

```
my_list = ['zero', 'one', 'two']
print("2-й элемент: {0[2]}".format(my_list)) # ""0" необязателен
# Вывод: "2-й элемент: two"
```

Примечание. В дополнение к `str.format` Python предоставляет и оператор `%`, также известный как оператор форматирования строк или интерполяции (см. PEP 3101) для форматирования строк. `str.format` является преемником `%`, и он обеспечивает большую гибкость, например, упрощая выполнение нескольких замен.

Помимо индексов аргументов вы также можете включить внутри фигурных скобок *спецификацию формата*. Это выражение, которое подчиняется специальным правилам и должно предваряться двоеточием (:). Примером спецификации формата является директива выравнивания: `~^20` (^ обозначает выравнивание по центру, общая ширина 20, заполнение символом ~):

```
{:~^20}'.format('centered')
# Вывод: '~~~~~centered~~~~~'
```

Функция `format` позволяет то, что невозможно с `%`, например повторение аргументов:

```
t = (12, 45, 22222, 103, 6)
print '{0} {2} {1} {2} {3} {2} {4} {2}'.format(*t)
# Вывод: 12 22222 45 22222 103 22222 6 22222
```

Поскольку `format` является функцией, его можно использовать в качестве аргумента в других функциях:

```
number_list = [12,45,78]
print map('число равно {}'.format, number_list)
# Вывод: ['число равно 12', 'число равно 45', 'число равно 78']
```

```
from datetime import datetime, timedelta
```

```
once_upon_a_time = datetime(2010, 7, 1, 12, 0, 0)
delta = timedelta(days=13, hours=8, minutes=20)
```

```
gen = (once_upon_a_time + x * delta for x in xrange(5))
```

```
print '\n'.join(map('{:%Y-%m-%d %H:%M:%S}'.format, gen))
#Вывод: 2010-07-01 12:00:00
#       2010-07-14 20:20:00
#       2010-07-28 04:40:00
#       2010-08-10 13:00:00
#       2010-08-23 21:20:00
```

40.2. Выравнивание и заполнение

Версия Python 2.x ≥ 2.6

Метод `format()` может быть использован для изменения выравнивания строки. Для этого необходимо использовать выражение формата вида `:[fill_char][align_operator][width]`, где `align_operator` может принимать значения:

- < заставляет поле выравниваться влево в пределах width.
- > заставляет поле выравниваться вправо в пределах width.
- ^ заставляет поле выравниваться по центру в пределах width.
- = заставляет помещать заполнение после знака (только для числовых типов).

`fill_char` (если не указано, по умолчанию – пробел) – это символ, используемый для заполнения.

```
{:~<9s}, World'.format('Hello')
# 'Hello~~~~, World'

{:~>9s}, World'.format('Hello')
# '~~~~Hello, World'

{:~^9s}'.format('Hello')
# '~^Hello~^'

{:0=6d}'.format(-123)
# '-00123'
```

Примечание: вы можете добиться тех же результатов, используя строковые функции `ljust()`, `rjust()`, `center()`, `zfill()`, однако эти функции устарели с версии 2.5.

40.3. Форматные литералы (f-строка)

В PEP 498 (Python 3.6 и выше) были введены строки буквенного формата, позволяющие добавлять `f` к началу строкового литерала, чтобы эффективно применить к нему `.format` со всеми переменными в текущей области видимости.

```
>>> foo = 'bar'
>>> f'Foo is {foo}'
'Foo is bar'
```

Это также работает с более продвинутыми строками формата, включая выравнивание и точечную нотацию.

```
>>> f'{foo:^7s}'
' bar '
```

Примечание: `f''` не обозначает конкретный тип типа `b''` для `bytes` или `u''` для `unicode` в Python 2. Формирование применяется немедленно, в результате чего получается обычная строка.

Строки формата также могут быть *вложенными*:

```
>>> price = 478.23
>>> f'f${price:0.2f}:*>20s)'
'*****$478.23'
```

Выражения в `f`-строке оцениваются в порядке слева направо. Это можно обнаружить только в том случае, если выражения имеют побочные эффекты:

```
>>> def fn(l, incr):
...     result = l[0]
...     l[0] += incr
...     return result
...
>>> lst = [0]
>>> f'{fn(lst,2)} {fn(lst,3)}'
'0 2'
>>> f'{fn(lst,2)} {fn(lst,3)}'
'5 7'
>>> lst
[10]
```

40.4. Форматирование чисел с плавающей точкой

```
>>> '{0:.0f}'.format(42.12345)
'42'

>>> '{0:1f}'.format(42.12345)
'42.1'
```

```
>>> '{0:.3f}'.format(42.12345)
'42.123'
```

```
>>> '{0:.5f}'.format(42.12345)
'42.12345'
```

```
>>> '{0:.7f}'.format(42.12345)
'42.1234500'
```

То же самое относится и к другим способам ссылки:

```
>>> '{:.3f}'.format(42.12345)
'42.123'
>>> '{answer:.3f}'.format(answer=42.12345)
'42.123'
```

Числа с плавающей точкой также могут быть отформатированы в научной нотации или в процентах:

```
>>> '{0:.3e}'.format(42.12345)
'4.212e+01'
```

```
>>> '{0:.0%}'.format(42.12345)
'4212%'
```

Также можно комбинировать обозначения {0} и {name}. Это особенно полезно, если вы хотите округлить все переменные до заданного числа десятичных знаков:

```
>>> s = 'Hello'
>>> a, b, c = 1.12345, 2.34567, 34.5678
>>> digits = 2

>>> '{0}! {1:.{n}f}, {2:.{n}f}, {3:.{n}f}'.format(s, a, b, c, n=digits)
'Hello! 1.12, 2.35, 34.57'
```

40.5. Именованные заполнители

Строки формата могут содержать именованные заполнители, которые интерполируются с помощью именованных аргументов для format.

Использование словаря (Python 2.x)

```
>>> data = {'first': 'Hodor', 'last': 'Hodor!'}
>>> '{first} {last}'.format(**data)
'Hodor Hodor!'
```

Использование словаря (Python 3.2+)

```
>>> '{first} {last}'.format_map(data)
'Hodor Hodor!'
```

Метод str.format_map позволяет использовать словари без их предварительной распаковки. Также класс data (который может быть пользовательским) используется вместо только что заполненного dict.

Без словаря

```
>>> '{first} {last}'.format(first='Hodor', last='Hodor!')
'Hodor Hodor!'
```

40.6. Форматирование строк с использованием типа datetime

Любой класс может создать свой собственный синтаксис форматирования строк с помощью метода `__format__`. В стандартной библиотеке Python для этого удобно использовать тип `datetime`, в котором можно использовать коды форматирования, подобные `strftime`, непосредственно в `str.format`:

```
>>> from datetime import datetime
>>> 'Северная Америка: {dt:%m/%d/%Y}. ISO: {dt:%Y-%m-%d}'.format(dt=datetime.now())
'Северная Америка: 07/21/2016. ISO: 2016-07-21.'
```

Полный список списков форматов даты и времени можно найти в официальной документации.

40.7. Форматирование числовых значений

Метод `.format()` может интерпретировать число в различных форматах, например:

```
>>> '{:c}'.format(65)      # символ Юникода
'A'
>>> '{:d}'.format(0x0a)    # основание 10
'10'
>>> '{:n}'.format(0x0a)    # основание 10 с использованием текущего языкового стандарта
                             # для разделителей
'10'
```

Форматирование целых чисел по разным основаниям (шестнадцатеричное, восьмеричное, двоичное)

```
>>> '{:x}'.format(10) # основание 16, строчные буквы – шестнадцатеричная система
'a'
>>> '{:X}'.format(10) # основание 16, верхний регистр – шестнадцатеричная система
'A'
>>> '{:o}'.format(10) # основание 8 – восьмеричное
'12'
>>> '{:b}'.format(10) # основание 2 – двоичное
'1010'
>>> '{:0:#b}, {0:#o}, {0:#x}'.format(42) # С префиксом
'0b10101010, 0o52, 0x2a'
>>> '8 bit: {0:08b}; Three bytes: {0:06x}'.format(42) # С нулевым заполнением
'8 bit: 00101010; Three bytes: 00002a'
```

Используйте форматирование для преобразования кортежа RGB со значениями с плавающей точкой в шестнадцатеричную строку координат цвета:

```
>>> r, g, b = (1.0, 0.4, 0.0)
>>> '#{0:02X}{0:02X}{0:02X}'.format(int(255 * r), int(255 * g), int(255 * b))
'FF6600'
```

Преобразовывать можно только целые числа:

```
>>> '{:x}'.format(42.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'x' for object of type 'float'
```

40.8. Вложенное форматирование

Некоторые форматы могут принимать дополнительные параметры, например, ширину формируемой строки или выравнивание:

```
>>> '{:.*>10}'.format('foo')
'.....foo'
```


Они также могут предоставляться как параметры для `format` путем вложения большего количества `{}` внутри `{}`:

```
>>> '{:>{}}'.format('foo', 10)
'.....foo'
'{:0{}}'.format('foo', '*', '^', 15)
'*****foo*****'
```

В последнем примере строка форматирования `{:0{}}` изменена на `{:*^15}` (т. е. “выровнять по центру и заполнить * на общую длину 15”), перед применением ее к реальной строке `'foo'`, которая должна быть отформатирована таким образом. Это может быть полезно в случаях, когда параметры неизвестны заранее, например для выравнивания табличных данных:

```
>>> data = ["a", "bbbbbbb", "ccc"]
>>> m = max(map(len, data))
>>> for d in data:
...     print('{:>{}}'.format(d, m))
      a
 bbbbbb
  ccc
```

40.9. Форматирование с помощью функций `Getitem` и `Getattr`

Любая структура данных, поддерживающая функцию `__getitem__`, может форматировать свою вложенную структуру:

```
person = {'first': 'Arthur', 'last': 'Dent'}
'{p[first]} {p[last]}'.format(p=person)
# 'Arthur Dent'
```

Доступ к объектным атрибутам можно получить с помощью `getattr()`:

```
class Person(object):
    first = 'Zaphod'
    last = 'Beeblebrox'

'{p.first} {p.last}'.format(p=Person())
# 'Zaphod Beeblebrox'
```

40.10. Комбинированное заполнение и усечение строк

Допустим, вы хотите вывести переменные в столбец из 3 символов. Обратите внимание: удвоение `{}` и `}` экранирует их.

```
s = ""

pad
{{:3}}           :{a:3}:

truncate
{{:.3}}          :(e:.3):

combined
{{:>3.3}}         :(a:>3.3):
{{:3.3}}          :(a:3.3):
{{:3.3}}          :(c:3.3):
{{:3.3}}          :(e:3.3):

print (s.format(a="1"*1, c="3"*3, e="5"*5))
```

Результат:

```
pad
{:3}           :1
```

```
truncate
{:.3}          :555:

combined
{:>3.3}         1:
{.3.3}          :1
{.3.3}          :333:
{.3.3}          :555:
```

40.11. Пользовательское форматирование класса

Примечание: все приведенное ниже относится как к методу `str.format`, так и к функции `format`. В приведенном далее тексте они взаимозаменяемы.

Для каждого значения, передаваемого в функцию `format`, Python ищет метод `__format__` для этого аргумента. Поэтому ваш собственный класс может иметь свой собственный метод `__format__` для определения того, как функция `format` будет отображать и форматировать ваш класс и его атрибуты. Это отличается от метода `__str__`, так как в методе `__format__` можно учитывать язык форматирования, включая выравнивание, ширину полей и т. д., и даже (при желании) реализовать свои собственные спецификации формата и расширения языка форматирования.

```
object.__format__(self, format_spec)
```

Например:

Пример на Python 2, но может быть легко применен к Python 3

```
class Example(object):
    def __init__(self,a,b,c):
        self.a, self.b, self.c = a,b,c

    def __format__(self, format_spec):
        """ Реализовать специальную семантику для спецификатора формата 's' """
        # Отклонить все, что не является s
        if format_spec[-1] != 's':
            raise ValueError('{} спецификатор формата не понят для этого объекта', format_spec[-1])

        # Выходные данные в данном примере будут иметь вид (<a>,<b>,<c>)
        raw = "(" + ",".join([str(self.a), str(self.b), str(self.c)]) + ")"
        # Соблюдать язык форматирования, используя встроенный формат строк
        # Поскольку мы знаем, что исходный format_spec заканчивается на 's',
        # мы можем воспользоваться методом str.format со строковым аргументом,
        # который мы создали выше
        return "{r:{f}}".format( r=raw, f=format_spec )

inst = Example(1,2,3)
print "{0:>20s}".format( inst )
# вывод :              (1,2,3)
# Обратите внимание, как соблюдены выравнивание по правому краю и ширина поля 20.
```

Замечания:

Если у вашего пользовательского класса нет настраиваемого метода `__format__` и экземпляр класса передается в функцию `format`, **Python 2** всегда будет использовать возвращаемое значение метода `__str__` или метода `__repr__`, чтобы определить, что печатать (и если они не существуют, то по умолчанию будет использоваться `repr`), и для форматирования потребуется использовать спецификатор формата `s`. В **Python 3**, чтобы передать ваш пользовательский класс в функцию `format`, вам нужно будет определить метод `__format__` в вашем пользовательском классе.

Глава 41. Строковые методы

41.1. Изменение регистра букв в строке

Тип `string` в Python предоставляет множество функций, которые работают с регистром букв в строке. К ним относятся:

- `str.casefold`
- `str.upper`
- `str.lower`
- `str.capitalize`
- `str.title`
- `str.swapcase`

С `unicode`-строками (по умолчанию в Python 3) эти операции не являются отображаемыми 1: 1 или обратимыми. Большинство из этих операций предназначены только для показа, а не для нормализации.

Версия Python 3.x ≥ 3.3

```
str.casefold()
```

Метод `str.casefold` создает строку в нижнем регистре (из строчных букв), пригодную для сравнения без учета регистра. Действует более “агрессивно”, чем `str.lower`, и может изменять строки, которые уже находятся в нижнем регистре, или увеличивать длину строк, не предназначен для отображения.

```
"ΧΒΣ".casefold()
# 'xssσ'
```

```
"ΧΒΣ".lower()
# 'xβς'
```

Преобразования, происходящие при действиях с регистрами, определяются консорциумом Unicode в файле `CaseFolding.txt` на их веб-сайте.

```
str.upper()
```

Метод `str.upper` берет каждый символ в строке и преобразует его в свой эквивалент верхнего регистра (заглавных букв), например:

```
"This is a 'string'".upper()
# "THIS IS A 'STRING'."
```

```
str.lower()
```

Метод `str.lower` делает обратное; он принимает каждый символ в строке и преобразует его в его нижний регистр:

```
"This IS a 'string'".lower()
# "this is a 'string'."
```

```
str.capitalize()
```

Метод `str.capitalize` возвращает “капитализированную” версию строки, т. е. делает первый символ верхним регистром, а остальные ниже (как в предложениях):

```
"this IS A 'String'".capitalize() # Первый символ заглавными буквами, остальные – строчными
# "This is a 'string'."
```

```
str.title()
```

Метод `str.title` возвращает “заголовочную” версию строки, то есть каждая буква в начале слова будет в верхнем регистре, а все остальные – в нижнем регистре:

```
"this is a 'String'".title()
# "This Is A 'String'"
```

```
str.swapcase()
```

Метод `str.swapcase` возвращает новый строковый объект, в котором все символы нижнего регистра меняются местами с символами верхнего регистра, а все символы верхнего регистра – с символами нижнего:

```
"this iS A STRiNG".swapcase() # Поменяем регистр каждого символа
# "THIS is a strIng"
```

Использование методов класса `str`

Стоит отметить, что эти методы могут вызываться либо на строковых объектах (как показано выше), либо как метод класса класса `str` (с явным вызовом – `str.upper` и т. д.).

```
str.upper("This is a 'string'")
# "THIS IS A 'STRING'"
```

Это наиболее полезно при применении одного из этих методов ко многим строкам сразу, к примеру, в функции `map`.

```
map(str.upper, ["These", "are", "some", "strings"])
# ["THESE", "ARE", "SOME", "STRINGS"]
```

41.2. str.translate: замена символов в строке

Python поддерживает метод `translate` для типа данных `str`, который позволяет указать так называемую таблицу замены (`translation table`), которая используется для замены символов, а также символы, которые должны быть удалены в процессе работы.

```
str.translate(table[, deletechars])
```

Параметр	Описание
<code>table</code>	таблица поиска, которая определяет соответствие одного символа другому.
<code>deletechars</code>	список символов, которые должны быть удалены из строки.

Метод `maketrans` (`str.maketrans` в Python 3 и `string.maketrans` в Python 2) позволяет сгенерировать таблицу замены.

```
>>> translation_table = str.maketrans("aeiou", "12345")
>>> my_string = "This is a string!"
>>> translated = my_string.translate(translation_table)
'Th3s 3s 1 str3ng!'
```

Метод `translate` возвращает строку, которая является копией исходной строки с произведенными заменами. Если требуется удалить только символы, можно установить аргумент `table` в `None`.

```
>>> 'this syntax is very useful'.translate(None, 'aeiou')
'ths syntx s vry sfl'
```

41.3. str.format и f-strings: форматирование значений в строку

Язык Python предоставляет возможность интерполяции и форматирования строк с помощью функции `str.format`, появившейся в версии 2.6, и с помощью нового типа форматированных строковых литералов `f-strings`, появившихся в версии 3.6.

Даны следующие переменные:

```
i = 10
f = 1.5
s = "foo"
l = ['a', 1, 2]
d = {'a': 1, 2: 'foo'}
```

Следующие утверждения эквивалентны:

```
"10 1.5 foo ['a', 1, 2] {'a': 1, 2: 'foo'}"
>>> "{} {} {} {}".format(i, f, s, l, d)
>>> str.format("{} {} {} {}", i, f, s, l, d)
>>> "{0} {1} {2} {3} {4}".format(i, f, s, l, d)
>>> "{0:d} {1:0.1f} {2} {3!r} {4!r}".format(i, f, s, l, d)
>>> "{i:d} {f:0.1f} {s} {!r} {d!r}".format(i=i, f=f, s=s, l=l, d=d)
>>> f"{i} {f} {s} {l} {d}"
>>> f"{i:d} {f:0.1f} {s} {!r} {d!r}"
```

Для справки: Python также поддерживает квалификаторы в стиле C для форматирования строк. Приведенные ниже примеры эквивалентны приведенным выше, но версии с использованием `str.format` являются предпочтительными из-за преимуществ гибкости, согласованности обозначений и расширяемости:

```
"%d %0.1f %s %r %r" % (i, f, s, l, d)
"% (i)d % (f)0.1f % (s)s % (l)r % (d)r" % dict(i=i, f=f, s=s, l=l, d=d)
```

Фигурные скобки, используемые для интерполяции в `str.format`, также могут быть пронумерованы для уменьшения дублирования при форматировании строк. Например, следующие варианты являются эквивалентными:

```
"I am from Australia. I love cupcakes from Australia!"
>>> "I am from {}. I love cupcakes from {}!".format("Australia", "Australia")
>>> "I am from {0}. I love cupcakes from {0}!".format("Australia")
```

Хотя официальная документация на Python, как обычно, достаточно основательна, можно обратиться к сайту pyformat.info, где есть большой набор примеров с подробными объяснениями.

Символы `{}` и `}` могут быть экранированы следующим образом:

```
"{'a': 5, 'b': 6}"
>>> "{{{'a': 5, 'b': 6}}".format("a", 5, "b", 6)

>>> f"{{{ 'a': 5, 'b': 6}}}"
```

См. главу “Форматирование строк” для получения дополнительной информации. Функция `str.format()` была предложена в PEP 3101, а `f-strings` – в PEP 498.

41.4. Полезные константы модуля `string`

Модуль `string` языка Python предоставляет константы для операций, связанных со строками. Чтобы воспользоваться ими, импортируйте модуль `string`:

```
>>> import string
```

```
string.ascii_letters :
```

Конкатенация `ascii_lowercase` и `ascii_uppercase`:

```
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
string.ascii_lowercase:
```

Содержит все символы ASCII нижнего регистра:

```
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

```
string.ascii_uppercase:
```

Содержит все символы ASCII верхнего регистра:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

`string.digits:`

Содержит все десятичные знаковые символы:

```
>>> string.digits
'0123456789'
```

`string.hexdigits:`

Содержит все шестнадцатеричные символы цифр:

```
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

`string.octaldigits:`

Содержит все восьмеричные символы цифр:

```
>>> string.octaldigits
'01234567'
```

`string.punctuation:`

Содержит все символы, которые считаются символами пунктуации в локализации C:

```
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~'
```

`string.whitespace:`

Содержит все символы ASCII, считающиеся пробелами:

```
>>> string.whitespace
'\t\n\r\x0b\x0c'
```

В скриптовом режиме `print(string.whitespace)` будет печатать фактические символы, используйте `str` для получения строки, возвращенной выше.

`string.printable:`

Содержит все символы, которые считаются пригодными для печати; это комбинация `string.digits`, `string.ascii_letters`, `string.punctuation` и `string.whitespace`.

```
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ! "$%&'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

41.5. Удаление ненужных ведущих и завершающих символов из строки

Для удаления из строки ведущих и завершающих символов предусмотрены три метода: `str.strip`, `str.rstrip` и `str.lstrip`. Все три метода имеют одинаковую сигнатуру и все три возвращают новый объект `string` с удаленными ненужными символами.

`str.strip([chars])`

Метод `str.strip` работает с заданной строкой и удаляет (обрезает) все ведущие и завершающие символы, содержащиеся в аргументе `chars`; если `chars` не предоставлены или имеют значение `None`, все пробельные символы удаляются по умолчанию. Например:

```
>>> " строка с ведущими и завершающими пробельными символами ".strip()
'строка с ведущими и завершающими пробельными символами'
```

Если заданы `chars` в качестве аргумента, то все содержащиеся символы удаляются из строки, которая возвращается. Например:

```
>>> ">>> а Python prompt".strip('> ') # удаляет символ '>' и символ пробела
'а Python prompt'
```

`str.rstrip([chars])` и `str.lstrip([chars])`

Эти методы имеют сходную семантику и аргументы с `str.strip()`, различие заключается в направлении, с которого они начинают действовать. Метод `str.rstrip()` начинается с конца строки, `str.lstrip()` – с начала строки.

Используем `str.rstrip()`:

```
>>> "  spacious string  ".rstrip()
'  spacious string'
```

Используем `str.lstrip()`:

```
>>> "  spacious string  ".lstrip()
'spacious string'
```

41.6. Реверсирование строки

Строка может быть перевернута с помощью встроенной функции `reversed()`, которая принимает строку и возвращает итератор в обратном порядке.

```
>>> reversed('hello')
<reversed object at 0x0000000000000000>
>>> [char for char in reversed('hello')]
['o', 'l', 'l', 'e', 'h']
```

Функция `reversed()` может быть “завернута” в вызов `.join()` для создания строки из итератора.

```
>>> ".join(reversed('hello'))
'olleh'
```

Хотя использование `reversed()` может быть более понятным для начинающих пользователей Python, использование расширенного среза с шагом -1 выполняется быстрее и является более кратким. Реализуем его в виде функции:

```
>>> def reversed_string(main_string):
...     return main_string[::-1]
...
>>> reversed_string('hello')
'olleh'
```

41.7. Разбиение строки по разделителю на список строк

`str.split(sep=None, maxsplit=-1)`

Метод `str.split` принимает строку и возвращает список подстрок исходной строки. Поведение зависит от того, предоставлен или исключен аргумент `sep`. Если `sep` не предоставлен или равен `None`, то разбиение происходит везде, где есть пробелы. Однако ведущие и завершающие пробелы игнорируются, а несколько последовательных символов пробела обрабатываются как один символ пробела:

```
>>> "This is a sentence.".split()
['This', 'is', 'a', 'sentence.']

>>> " This is a sentence. ".split()
['This', 'is', 'a', 'sentence.']

>>> "      ".split()
[]
```

Параметр `sep` может использоваться для определения строки-разделителя. Исходная строка разделяется в том месте, где встречается разделитель, а сам разделитель отбрасывается. Несколько последовательных разделителей не рассматриваются так же, как одно вхождение, а приводят к созданию пустых строк.

```
>>> "This is a sentence.".split(' ')
['This', 'is', 'a', 'sentence.']
```

```
>>> "Earth,Stars,Sun,Moon".split(',')
['Earth', 'Stars', 'Sun', 'Moon']
```

```
>>> " This is a sentence. ".split(' ')
['', 'This', 'is', '', '', 'a', 'sentence.', '', '']
```

```
>>> "This is a sentence.".split('e')
['This is a s', 'nt', 'nc', '.']
```

```
>>> "This is a sentence.".split('en')
['This is a s', 't', 'ce.']
```

По умолчанию используется разделение на *каждое* вхождение разделителя, однако параметр `maxsplit` ограничивает количество разделителей. Значение по умолчанию -1 означает отсутствие ограничений:

```
>>> "This is a sentence.".split('e', maxsplit=0)
['This is a sentence.']
```

```
>>> "This is a sentence.".split('e', maxsplit=1)
['This is a s', 'ntence.']
```

```
>>> "This is a sentence.".split('e', maxsplit=2)
['This is a s', 'nt', 'nce.']
```

```
>>> "This is a sentence.".split('e', maxsplit=-1)
['This is a s', 'nt', 'nc', '.']
```

str.rsplit(sep=None, maxsplit=-1)

Метод `str.rsplit` ("right split", "разделение справа") отличается от `str.split` ("left split", "разделение слева") при указании `maxsplit`. Разбиение начинается с конца строки, а не с начала:

```
>>> "This is a sentence.".rsplit('e', maxsplit=1)
['This is a sentenc', '.']
```

```
>>> "This is a sentence.".rsplit('e', maxsplit=2)
['This is a sent', 'nc', '.']
```

Примечание. Python указывает максимальное количество выполняемых *разбиений*, в то время как большинство других языков программирования определяют максимальное количество созданных *подстрок*. Это может создать путаницу при переносе или сравнении кода.

41.8. Замена всех вхождений одной подстроки на другую подстроку

Тип `str` в Python также имеет метод замены вхождений одной подстроки на другую подстроку в данной строке. Для более сложных случаев можно использовать метод `re.sub`.

`str.replace(old, new[, count])` :

Метод `str.replace` принимает два аргумента `old` и `new`, содержащих `old` подстроку, которая должна быть заменена `new` подстрокой. Необязательный аргумент `count` указывает количество замен, которые должны быть произведены. Например, чтобы заменить 'foo' на 'spam' в следующей строке, мы можем вызвать `str.replace` с `old = 'foo'` и `new = 'spam'`:

```
>>> "Make sure to foo your sentence.".replace('foo', 'spam')
'Make sure to spam your sentence.'
```

Если заданная строка содержит несколько примеров, которые соответствуют аргументу `old`, то **все** вхождения заменяются значением, указанным в `new`:


```
>>> "It can foo multiple examples of foo if you want.".replace('foo', 'spam')
"It can spam multiple examples of spam if you want."
```

если, конечно, мы не зададим значение для `count`. В этом случае вхождения `count` будут заменены:

```
>>> """It can foo multiple examples of foo if you want, \
... or you can limit the foo with the third argument.""".replace('foo', 'spam', 1)
'It can spam multiple examples of foo if you want, or you can limit the foo with the third argument.'
```

41.9. Проверка содержимого строк

Тип `str` в Python также имеет ряд методов, которые могут быть использованы для оценки содержимого строки. Это `str.isalpha`, `str.isdigit`, `str.isalnum`, `str.isspace`. Проверка регистра может быть выполнена с помощью методов `str.isupper`, `str.islower` и `str.istitle`.

`str.isalpha`

Этот метод не принимает никаких аргументов и возвращает `True`, если все символы в данной строке являются буквенными (алфавитными), например:

```
>>> "Hello World".isalpha()      # содержит пробел
False
>>> "Hello2World".isalpha()      # содержит число
False
>>> "HelloWorld!".isalpha()      # содержит знаки
False
>>> "HelloWorld".isalpha()
True
```

Как крайний случай пустая строка при использовании `"".isalpha()` оценивается как `False`.

`str.isupper`, `str.islower`, `str.istitle`

Эти методы проверяют регистр в заданной строке.

`str.isupper` – метод, возвращающий `True`, если все символы в заданной строке имеют верхний регистр, и `False` противном случае.

```
>>> "HeLLo WORLD".isupper()
False
>>> "HELLO WORLD".isupper()
True
>>> "".isupper()
False
```

И наоборот, `str.islower` – это метод, который возвращает `True`, если все символы в данной строке являются строчными, и `False` в противном случае.

```
>>> "Hello world".islower()
False
>>> "hello world".islower()
True
>>> "".islower()
False
```

`str.istitle` возвращает значение `True`, если заданная строка “капитализирована”, то есть каждое слово начинается с символа верхнего регистра, за которым следуют строчные буквы.

```
>>> "hello world".istitle()
False
>>> "Hello world".istitle()
False
>>> "Hello World".istitle()
True
>>> "".istitle()
False
```

str.isdecimal , str.isdigit , str.isnumeric

`str.isdecimal` возвращает, является ли данная строка последовательностью десятичных цифр, пригодной для представления десятичного числа.

`str.isdigit` включает цифры, не имеющие формы, пригодной для представления десятичного числа, например надстрочные цифры.

`str.isnumeric` включает любые числовые значения, даже если они не являются цифрами, например значения вне диапазона 0–9.

	isdecimal	isdigit	isnumeric
12345	True	True	True
?2??5	True	True	True
?2 ³ ????	False	True	True
??	False	False	True
Five	False	False	False

Строки байтов (`bytes` в Python 3, `str` в Python 2) поддерживают только функцию `isdigit`, которая проверяет только наличие основных ASCII-цифр. Как и в случае с `str.isalpha`, пустая строка оценивается как `False`.

str.isalnum

Это комбинация из `str.isalpha` и `str.isnumeric`, в частности, принимает значение `True`, если все символы в данной строке являются буквенно-цифровыми, то есть они состоят из буквенных (алфавитных) или числовых символов:

```
>>> "Hello2World".isalnum()
True
>>> "HelloWorld".isalnum()
True
>>> "2016".isalnum()
True
>>> "Hello World".isalnum() # содержит пробел
False
```

str.isspace

Оценивается как `True`, если строка содержит только пробельные символы.

```
>>> "\t\r\n".isspace()
True
>>> " ".isspace()
True
```

Иногда строка выглядит “пустой”, но мы не знаем, связано ли это с тем, что она содержит пробельные символы или вообще ничего не содержит.

```
>>> "".isspace()
False
```

Для этого случая нам необходима дополнительная проверка:

```
>>> my_str = ""
>>> my_str.isspace()
False
>>> my_str.isspace() or not my_str
True
```

Но самый короткий способ проверить, является ли строка пустой или содержит пробельные символы – использовать функцию `strip` (без аргументов она удаляет все ведущие и завершающие пробельные символы):

```
>>> not my_str.strip()
True
```

41.10. Проверка содержания подстроки в строке

В Python интуитивно понятна проверка того, содержит ли строка заданную подстроку. Для этого достаточно воспользоваться оператором `in`:

```
>>> "foo" in "foo.baz.bar"
True
```

Примечание: при проверке пустой строки всегда будет получен результат `True`:

```
>>> "" in "test"
True
```

41.11. Объединение списка строк в одну строку

Строка может быть использована в качестве разделителя для объединения списка строк в одну строку с помощью метода `join()`. Например, можно создать строку, в которой каждый элемент списка отделяется пробелом.

```
>>> ".join(["once", "upon", "a", "time"])
"once upon a time"
```

В следующем примере элементы строки разделяются тремя дефисами.

```
>>> "-".join(["once", "upon", "a", "time"])
"once-upon-a-time"
```

41.12. Подсчет количества вхождений подстроки в строке

Для подсчета количества вхождений подстроки в другую строку существует один метод – `str.count`.

`str.count(sub[, start[, end]])`

Метод `str.count` возвращает целочисленное `int` значение, указывающее на количество непересекающихся вхождений подстроки `sub` в другую строку. Необязательные аргументы `start` и `end` указывают начальные и конечные позиции, между которыми будет производиться поиск. По умолчанию `start = 0` и `end = len(str)`, то есть поиск будет производиться во всей строке:

```
>>> s = "She sells seashells by the seashore."
>>> s.count("sh")
2
>>> s.count("se")
3
>>> s.count("sea")
2
>>> s.count("seashells")
1
```

Задавая разные значения для `start` и `end`, мы можем получить более локализованный поиск и подсчет, например, если `start` равно 13, то вызов:

```
>>> s.count("sea", start)
1
```

эквивалентен:

```
>>> t = s[start:]
>>> t.count("sea")
1
```

41.13. Сравнение строк без учета регистра

Сравнение строк без учета регистра кажется чем-то тривиальным, но это не так. В этом разделе рассматриваются только строки в Unicode (в Python 3 – по умолчанию). Отметим, что Python 2 может иметь незначительные недостатки по сравнению с Python 3 – работа с Unicode в последнем гораздо более полная.

Прежде всего следует отметить, что преобразования для снятия регистра в Unicode не являются тривиальными. Существует текст, для которого

```
text.lower() != text.upper().lower(), such as "ß":
>>> "ß".lower()
'ß'
>>> "ß".upper().lower()
'ss'
```

Но, допустим, вы хотели сравнить "BUSSE" и "Buße" без учета регистра. Или также хотите сравнить "BUSSE" и "BUßE" равными – с новой формой заглавного символа. Рекомендваемым способом является использование `casefold`:

Версия Python 3.x ≥ 3.3:

```
>>> help(str.casefold)
```

Help on method_descriptor:

```
casefold(...)
    S.casefold() -> str
```

Return a version of S suitable for caseless comparisons.

(Возвращает версию S, пригодную для сравнений без учета регистра)

Если `casefold` недоступен, выполнение `.upper().lower()` помогает (но только в некоторой степени).

Теперь рассмотрим работу с ударениями. Вы, вероятно, думаете, что "ê" == "ê", но это не так:

```
>>> "ê" == "ê"
False
```

Это потому, что на самом деле в первом случае мы имеем дело с буквой с циркумфлексом, а во втором – с буквой с комбинированным циркумфлексным ударением:

```
>>> import unicodedata

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E WITH CIRCUMFLEX']

>>> [unicodedata.name(char) for char in "ê"]
['LATIN SMALL LETTER E', 'COMBINING CIRCUMFLEX ACCENT']
```

Самый простой способ решения этой проблемы – `unicodedata.normalize`. Скорее всего, вы хотите использовать нормализацию NFKD, но проверяйте документацию. Затем выполните:

```
>>> unicodedata.normalize("NFKD", "ê") == unicodedata.normalize("NFKD", "ê")
True
```

В довершение это же с применением функций:

```
import unicodedata

def normalize_caseless(text):
    return unicodedata.normalize("NFKD", text.casefold())

def caseless_equal(left, right):
    return normalize_caseless(left) == normalize_caseless(right)
```

41.14. Выравнивание строк

Python предоставляет функции для выравнивания строк, позволяющие использовать заполнение текстом, что значительно упрощает выравнивание. Ниже приведен пример использования функций `str.ljust` и `str.rjust`:

```
interstates_lengths = {
    5: (1381, 2222),
    19: (63, 102),
    40: (2555, 4112),
    93: (189, 305),
}
for road, length in interstates_lengths.items():
    miles, kms = length
    print('{ } -> { } mi. ({} km.)'.format(str(road).rjust(4), str(miles).ljust(4), str(kms).ljust(4)))
40 -> 2555 mi. (4112 km.)
19 -> 63   mi. (102 km.)
5  -> 1381 mi. (2222 km.)
93 -> 189 mi. (305 km.)
```

Функции `ljust` и `rjust` очень похожи. Обе имеют параметр `width` и необязательный параметр `fillchar`. Любая строка, созданная этими функциями, имеет длину не менее параметра `width`, который был передан в функцию. Если длина строки превышает ширину `width`, то она не усекается. Аргумент `fillchar`, в качестве которого по умолчанию используется символ пробела ' ', должен быть одним символом, а не многосимвольной строкой.

Функция `ljust` заполняет конец вызываемой строки символом `fillchar` до тех пор, пока длина строки не станет равной ширине. Функция `rjust` аналогичным образом заполняет начало строки. Таким образом, `l` и `r` в названиях этих функций означают сторону, с которой исходная строка, а не символ-заполнитель, располагается в выходной строке.

41.15. Проверка начальных и конечных символов строки

Для проверки начала и конца строки в Python можно использовать методы `str.startswith()` и `str.endswith()`.

`str.startswith(prefix[, start[, end]])`

Как следует из названия, `str.startswith` используется для проверки того, начинается ли заданная строка с символов, заданных в `prefix`.

```
>>> s = "This is a test string"
>>> s.startswith("T")
True
>>> s.startswith("Thi")
True
>>> s.startswith("thi")
False
```

Необязательные аргументы `start` и `end` задают начальную и конечную точки, с которых будет начинаться и заканчиваться тестирование. В следующем примере при указании значения `start`, равного 2, поиск строки будет производиться с позиции 2:

```
>>> s.startswith("is", 2)
True
```

Это дает `True`, так как `s[2] == 'i'` и `s[3] == 's'`. Вы также можете проверить, начинается ли кортеж (tuple) с любой из множества строк:

```
>>> s.startswith(('This', 'That'))
True
>>> s.startswith(('ab', 'bc'))
False
```

str.endswith(prefix[, start[, end]])

Метод `str.endswith` полностью аналогичен `str.startswith`, с той лишь разницей, что ищет не начальные, а конечные символы. Например, чтобы проверить, заканчивается ли строка символом полной остановки, т. е. точкой, можно написать:

```
>>> s = "this ends in a full stop."
>>> s.endswith('.')
True
>>> s.endswith('!')
False
```

Как и в случае с `startswith`, в качестве завершающей последовательности может использоваться более одного символа:

```
>>> s.endswith('stop.')
True
>>> s.endswith('Stop.')
False
```

Вы также можете проверить, заканчивается ли кортеж какой-либо из множества строк.

```
>>> s.endswith(('.', 'something'))
True
>>> s.endswith(('ab', 'bc'))
False
```

41.16. Преобразование между строковыми или байтовыми данными и символами Unicode

Содержимое файлов и сетевых сообщений может представлять собой закодированные символы. Для правильного отображения их часто необходимо преобразовать в Unicode. В Python 2 может потребоваться преобразование строковых данных в символы Unicode. По умолчанию ("", "" и т. д.) – это ASCII-строка, причем любые значения, выходящие за пределы ASCII-диапазона, отображаются как экранированные значения. Строки Unicode имеют вид `u` (или `u"`, и т. д.).

Версия Python 2.x ≥ 2.3:

Вы получаете "© abc" в кодировке UTF-8 из файла, сети или другого источника данных

```
s = '\xc2\xa9 abc'      # s – это массив байтов, а не строка символов
                        # Не знает, что оригинал был UTF-8,
                        # форма строковых литералов по умолчанию в Python 2
s[0]                    # '\xc2' – бессмысленный байт (без контекста, например, кодировки)
type(s)                 # str – несмотря на то, что он не является полезным,
                        # без знания кодировки
```

```
u = s.decode('utf-8')    # u'\xa9 abc'
                        # Теперь у нас есть строка Unicode, которая может быть
                        # прочитана как UTF-8 и выведена на печать правильно
                        # В Python 2 строковые литералы Unicode нуждаются в ведущем символе u
                        # str.decode преобразует строку, которая может содержать экранированные байты,
                        # в Unicode-строку
```

```
u[0]                    # u'\xa9' - символ Unicode 'Знак копирайта' (U+00A9) '©'
type(u)                 # Unicode
```

```
u.encode('utf-8')       # '\xc2\xa9 abc'
                        # unicode.encode выдает строку с экранированными байтами для
                        # не-ASCII символов
```

В Python 3 может возникнуть необходимость преобразования массивов байтов (называемых “байтовыми литералами”) в строки символов Unicode. По умолчанию используется строка Unicode, и литералы байтовых строк теперь должны вводиться как `b''`, `b'''` и т. д. Байтовый литерал вернет `True` на `isinstance(some_val, byte)`, предполагая, что `some_val` – это строка, которая может быть закодирована в байтовом виде.

Версия Python 3.x ≥ 3.0:

```
# Вы получаете “© abc” в кодировке UTF-8 из файла или сети
s = b'\xc2\xa9 abc'      # s – массив байтов, а не символов
                           # В Python 3 строковый литерал по умолчанию имеет формат Unicode;
                           # для литералов байтовых массивов требуется ведущий символ b
s[0]                     # b'\xc2' – бессмысленный байт (без контекста, например, кодировки)
type(s)                  # bytes – теперь, когда массивы байтов стали явными, Python может
                           # показать это.

u = s.decode('utf-8')    # '© abc' на терминале Unicode
                           # bytes.decode преобразует массив байтов в строку,
                           # (которая в Python 3 будет иметь вид Unicode)

u[0]                     # '\u00a9' – символ Unicode ‘Знак копирайта’ (U+00A9) ‘©’
type(u)                  # str
                           # Строковым литералом по умолчанию в Python 3
                           # является UTF-8 Unicode
u.encode('utf-8')        # b'\xc2\xa9 abc'
                           # str.encode создает массив байтов, отображающий байты
                           # диапазона ASCII как неэкранированные символы
```

Глава 42. Использование циклов внутри функций

В Python функция будет возвращена, как только выполнение перейдет к оператору возврата.

42.1. Оператор возврата внутри цикла в функции

В данном примере функция вернется, как только значение переменной станет равным 1.

```
def func(params):
    for value in params:
        print('Got value {}'.format(value))

        if value == 1:
            # Возвращает из функции, как только значение становится равным 1
            print(">>> Got 1")
            return

    print("Still looping")

    return "Couldn't find 1"

func([5, 3, 1, 2, 8, 9])
```

результат:

```
Got value 5
Still looping
Got value 3
Still looping
Got value 1
>>> Got 1
```

Глава 43. Импорт модулей

43.1. Импорт модуля

Используйте оператор `import`:

```
>>> import random
>>> print(random.randint(1, 10))
4
```

`import module` импортирует модуль и затем позволяет ссылаться на его объекты – например, значения, функции и классы – с помощью синтаксиса `module.name`. В приведенном выше примере импортируется модуль `random`, который содержит функцию `randint`. Поэтому, импортировав `random`, можно вызвать функцию `randint` с помощью `random.randint`. Вы можете импортировать модуль и присвоить ему другое имя:

```
>>> import random as rn
>>> print(rn.randint(1, 10))
4
```

Если ваш файл на Python `main.py` находится в той же папке, что и `custom.py`, вы можете импортировать его следующим образом:

```
import custom
```

Также возможен импорт функции из модуля:

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Для импорта специальных функций глубже в модуль точечный оператор может использоваться **только** в левой части от ключевого слова `import`:

```
from urllib.request import urlopen
```

В Python есть два способа вызова функции с верхнего уровня. Один из них – `import`, а другой – `from`. Первый способ следует использовать, когда есть вероятность столкновения имен. Допустим, у нас есть файлы `hello.py` и `world.py`, в которых есть одна и та же функция с именем `function`. Тогда оператор `import` будет работать хорошо.

```
from hello import function
from world import function
function() #будет вызвана функция из world. А не функция hello
```

В общем случае `import` предоставляет пространство имен.

```
import hello
import world
hello.function() # будет вызвана исключительно функция из hello
world.function() # будет вызвана исключительно функция из world
```


Но если вы уверены, что во всем вашем проекте не может быть такого, чтобы одно и то же имя функции вы использовали неоднократно, используйте оператор `from`. В одной строке может быть выполнено несколько операций импорта:

```
>>> # Множественные модули
>>> import time, sockets, random
>>> # Множественные функции
>>> from math import sin, cos, tan
>>> # Множественные константы
>>> from math import pi, e
```

```
>>> print(pi)
3.141592653589793
>>> print(cos(45))
0.5253219888177297
>>> print(time.time())
1482807222.7240417
```

Приведенные выше ключевые слова и синтаксис могут также использоваться в комбинациях:

```
>>> from urllib.request import urlopen as geturl, pathname2url as path2url, getproxies
>>> from math import factorial as fact, gamma, atan as arctan
>>> import random.randint, time, sys
>>> print(time.time())
1482807222.7240417
>>> print(arctan(60))
1.554131203080956
>>> filepath = "/dogs/jumping poodle (december).png"
>>> print(path2url(filepath))
/dogs/jumping%20poodle%20%28december%29.png
```

43.2. Специальная переменная `__all__`

Модули могут иметь специальную переменную `__all__` для ограничения того, какие переменные импортируются при использовании `from mymodule import *`.

Дан следующий модуль:

```
# mymodule.py

__all__ = ['imported_by_star']

imported_by_star = 42
not_imported_by_star = 21
```

При использовании функции `from mymodule import *` импортируется только `imported_by_star`:

```
>>> from mymodule import *
>>> imported_by_star
42
>>> not_imported_by_star
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'not_imported_by_star' is not defined
```

Однако `not_imported_by_star` может быть импортирован явным образом:

```
>>> from mymodule import not_imported_by_star
>>> not_imported_by_star
```

43.3. Импорт модулей из произвольного места файловой системы

Если вы хотите импортировать модуль, который еще не существует ни как встроенный модуль в стандартной библиотеке Python (Python Standard Library), ни как побочный пакет, вы можете добавить в `sys.path` путь к каталогу, в котором находится ваш модуль. Это может быть полезно, если установлено несколько сред Python.

```
import sys
sys.path.append("/path/to/directory/containing/your/module")
import mymodule
```

Важно, чтобы в этом случае добавлялся путь к *каталогу*, в котором находится `mymodule`, а не путь к самому модулю.

43.4. Импорт всех имен из модуля

```
from module_name import *
```

Например:

```
from math import *
sqrt(2)          # вместо math.sqrt(2)
ceil(2.7)        # вместо math.ceil(2.7)
```

При этом в глобальное пространство имен будут импортированы все имена, заданные в модуле `math`, за исключением имен, начинающихся с символа подчеркивания (который указывает на то, что автор считает это имя предназначенным только для внутреннего использования).

Предупреждение: Если функция с таким же именем уже была определена или импортирована, то она будет **перезаписана**. Практически всегда **рекомендуется** импортировать только конкретные имена `from math import sqrt, ceil`:

```
def sqrt(num):
    print("Я не знаю, чему равен квадратный корень из {}".format(num))
sqrt(4)
# Вывод: Я не знаю, чему равен квадратный корень из 4.
from math import *
sqrt(4)
# Вывод: 2.0
```

Импорт со звездочками разрешен только на уровне модуля. Попытки выполнить их в определениях классов или функций приводят к ошибке `SyntaxError`.

```
def f():
    from math import *
class A:
    from math import *
```

`SyntaxError: import * only allowed at module level`

43.5. Программный импорт

Версия Python 2.x ≥ 2.7:

Чтобы импортировать модуль через вызов функции, используйте модуль `importlib` (входит в состав Python начиная с версии 2.7):

```
import importlib
random = importlib.import_module("random")
```

Функция `importlib.import_module()` также будет импортировать подмодуль пакета напрямую:

```
collections_abc = importlib.import_module("collections.abc")
```

Для старых версий Python используйте модуль `imp`.

Версия Python 2.x ≤ 2.7:

Для выполнения программного импорта используйте функции `imp.find_module` и `imp.load_module`.

```
import imp, sys
def import_module(name):
    fp, pathname, description = imp.find_module(name)
    try:
        return imp.load_module(name, fp, pathname, description)
    finally:
        if fp:
            fp.close()
```

Не используйте `__import__()` для программного импорта модулей! Существуют тонкие детали, связанные с `sys.modules`, аргументом `fromlist` и т. д., которые легко упустить из виду и которые `importlib.import_module()` решает за вас.

43.6. Правила PEP8 для импорта

Некоторые рекомендации по стилю PEP8 для импорта.

1. Импорт должен быть на отдельных строках:

```
from math import sqrt, ceil      # Не рекомендуется
from math import sqrt           # Рекомендуется
from math import ceil
```

2. Располагайте операторы импорта в верхней части модуля в следующем порядке:

- Импорт стандартных библиотек
- Импорт от сторонних производителей
- Конкретный импорт локальных приложений/библиотек

3. Следует избегать импорта с подстановочными знаками, поскольку это приводит к путанице в именах в текущем пространстве имен. Если вы выполняете `from module import *`, то может быть неясно, происходит ли конкретное имя в коде из `module` или нет. Это вдвойне верно, если у вас есть несколько утверждений типа `from module import *`.

4. Избегайте использования относительного импорта, вместо него используйте явный импорт.

43.7. Импорт конкретных имен из модуля

Вместо импорта всего модуля можно импортировать только определенные, конкретные имена:

```
from random import randint      # Синтаксис "from MODULENAME import NAME1[, NAME2[, ...]]"
print(randint(1, 10))           # Результат: 5
```

`from random` необходим, поскольку интерпретатор Python должен знать, из какого ресурса ему следует импортировать функцию или класс, а `import randint` указывает на саму функцию или класс. Еще один пример (аналогичный приведенному выше):

```
from math import pi
print(pi)                       # Результат: 3.14159265359
```

Следующий пример приведет к ошибке, поскольку мы не импортировали модуль:

```
random.randrange(1, 10) # работает, только если "import random" запускался ранее
```

Выведет:

```
NameError: name 'random' is not defined
```

Интерпретатор Python не понимает, что вы имеете в виду под словом `random`. Его необходимо объявить, добавив, например `import random`:

```
import random
random.randrange(1, 10)
```

43.8. Импорт submodule

```
from module.submodule import function
```

Это импортирует `function` из `module.submodule`.

43.9. Повторный импорт модуля

При работе с интерактивным интерпретатором может потребоваться перезагрузка модуля. Это может быть полезно, если вы редактируете модуль и хотите импортировать самую новую версию, или если вы провели манкипатчинг какого-то элемента существующего модуля и хотите вернуть изменения. Обратите внимание, что для возврата **нельзя** просто импортировать модуль заново:

```
import math
math.pi = 3
print(math.pi) # 3
import math
print(math.pi) # 3
```

Это происходит потому, что интерпретатор регистрирует каждый импортируемый модуль. И когда вы пытаетесь повторно импортировать модуль, интерпретатор видит его в реестре и ничего не делает. Поэтому сложный способ повторного импорта – использовать `import` после удаления соответствующего элемента из реестра:

```
print(math.pi) # 3
import sys
if 'math' in sys.modules: # Есть ли модуль ``math`` в реестре?
    del sys.modules['math'] # Если да, то удалите его.
import math
print(math.pi) # 3.141592653589793
```

Но есть и более простой и понятный путь.

Python 2

Используем функцию `reload`:

Python 2.x версии ≥ 2.3:

```
import math
math.pi = 3
print(math.pi) # 3
reload(math)
print(math.pi) # 3.141592653589793
```

Python 3

Функция `reload` перемещена в `importlib`:

Python 3.x версии ≥ 3.0:

```
import math
math.pi = 3
print(math.pi) # 3
from importlib import reload
reload(math)
print(math.pi) # 3.141592653589793
```

43.10. Функция `__import__()`

Функция `__import__()` может быть использована для импорта модулей в случае, когда их имена известны только во время выполнения программы:

```
if user_input == "os":
    os = __import__("os")
# эквивалентно import os
```

Эта функция также может быть использована для указания пути к модулю:

```
mod = __import__(r"C:/path/to/file/anywhere/on/computer/module.py")
```

Глава 44. Разница между модулем и пакетом

44.1. Модули

Модуль – это отдельный элемент языка Python, который может быть импортирован. Использование модуля выглядит следующим образом:

```
module.py
```

```
def hi():
    print("Hello world!")
```

```
my_script.py
```

```
import module
module.hi()
```

в интерпретаторе:

```
>>> from module import hi
>>> hi()
# Hello world!
```

44.2. Пакеты

Пакет состоит из нескольких файлов (или модулей) Python и даже может включать библиотеки, написанные на языках C или C++. Вместо того чтобы быть одним файлом, он представляет собой целую структуру папок, которая может выглядеть следующим образом:

Папка package

- `__init__.py`
- `dog.py`
- `hi.py`

```
__init__.py
```

```
from package.dog import woof
from package.hi import hi
```

```
dog.py
```

```
def woof():
    print("WOOF!!!")
```

```
hi.py
```

```
def hi():
    print("Hello world!")
```

Все пакеты Python должны содержать файл `__init__.py`. Когда вы импортируете пакет в свой скрипт (сценарий) при помощи `import package`, будет запущен сценарий `__init__.py`, который предоставит вам доступ ко всем функциям пакета. В приведенном случае он позволяет использовать функции `package.hi` и `package.woof`.

Глава 45. Математический модуль math

45.1. Округление: round, floor, ceil, trunc

Помимо встроенной функции `round` математический модуль предоставляет функции `floor`, `ceil` и `trunc`.

```
x = 1.55
y = -1.55
```

```
# округлить до ближайшего целого числа
round(x)      # 2
round(y)      # -2
```

```
# второй аргумент указывает, до скольких знаков после запятой следует округлять
# (по умолчанию 0)
round(x, 1)   # 1.6
round(y, 1)   # -1.6
```

```
# math – это модуль, поэтому сначала импортируйте его, а затем используйте
import math
```

```
# получить наибольшее целое число, меньшее x
math.floor(x) # 1
math.floor(y) # -2
```

```
# получить наименьшее целое число, большее x
math.ceil(x)  # 2
math.ceil(y)  # -1
```

```
# отбросить дробную часть x
math.trunc(x) # 1, эквивалентно math.floor для положительных чисел
math.trunc(y) # -1, эквивалентно math.ceil для отрицательных чисел
```

Версия Python 2.x ≤ 2.7:

`floor`, `ceil`, `trunc` и `round` всегда возвращают число типа `float`.

```
round(1.3) # 1.0
```

`round` всегда округляет половинные значения “в сторону от нуля”.

```
round(0.5) # 1.0
round(1.5) # 2.0
```

Версия Python 3.x ≥ 3.0:

`floor`, `ceil`, `trunc` всегда возвращают целочисленное значение, а `round` возвращает целочисленное значение, если вызывается с одним аргументом.

```
round(1.3) # 1
round(1.33, 1) # 1.3
```

round округляет в сторону ближайшего четного числа. Это исправляет смещение в сторону больших чисел при выполнении большого количества вычислений.

```
round(0.5) # 0
round(1.5) # 2
```

Внимание!

Как и в любом другом представлении с плавающей точкой, некоторые дроби *не могут быть представлены точно*. Это может привести к неожиданному поведению при округлении.

```
round(2.675, 2) # 2.67, не 2.68!
```

Предупреждение о функциях floor, trunc и целочисленном делении отрицательных чисел

В языке Python (а также в C++ и Java) отрицательные числа округляются от нуля. Пример:

```
>>> math.floor(-1.7)
-2.0
>>> -5 // 2
-3
```

45.2. Тригонометрия

Вычисление длины гипотенузы

```
math.hypot(2, 4) # Просто сокращение для SquareRoot(2**2 + 4**2)
# Результат: 4.47213595499958
```

Преобразование градусов в радианы и обратно

Все функции модуля math предполагают использование радиан, поэтому необходимо преобразовать градусы в радианы:

```
math.radians(45) # Преобразование 45 градусов в радианы
# Результат: 0.7853981633974483
```

Все результаты обратных тригонометрических функций возвращают результат в радианах, поэтому может потребоваться обратное преобразование в градусы:

```
math.degrees(math.asin(1)) # Преобразование результата в градусы
# Результат: 90.0
```

Синус, косинус, тангенс и обратные функции

```
# Синус и арксинус:
math.sin(math.pi / 2)
# Результат: 1.0
math.sin(math.radians(90)) # Синус 90 градусов
# Результат: 1.0
```

```
math.asin(1)
# Результат: 1.5707963267948966 # "= pi / 2"
math.asin(1) / math.pi
# Результат: 0.5
```

```
# Косинус и арккосинус:
math.cos(math.pi / 2)
# Результат: 6.123233995736766e-17
# Почти ноль, но не совсем, потому что "pi" – число типа float с ограниченной точностью!
```

```
math.acos(1)
# Результат: 0.0

# Тангенс и арктангенс:
math.tan(math.pi/2)
# Результат: 1.633123935319537e+16
# Очень большое, но не бесконечное число, потому что "pi" – число типа float
# с ограниченной точностью
```

Версия Python 3.x ≥ 3.5:

```
math.atan(math.inf)
# Результат: 1.5707963267948966 # Это просто "pi / 2"

math.atan(float("inf"))
# Результат: 1.5707963267948966 # Это просто "pi / 2"
```

Помимо `math.atan` существует также двухаргументная функция `math.atan2`, которая вычисляет правильный квадрант и позволяет избежать ловушек, связанных с делением на ноль:

```
math.atan2(1, 2) # Эквивалентно "math.atan(1/2)"
# Результат: 0.4636476090008061 # ≈ 26.57 градусов, 1-й квадрант

math.atan2(-1, -2) # Не равно "math.atan(-1/-2)" == "math.atan(1/2)"
# Результат: -2.677945044588987 # ≈ -153.43 градусов (или 206.57 градусов), 3-й квадрант

math.atan2(1, 0) # math.atan(1/0) вызовет ошибку ZeroDivisionError
# Результат: 1.5707963267948966 # Это просто "pi / 2"
```

Гиперболические синус, косинус и тангенс

```
# Функция гиперболического синуса
math.sinh(math.pi) # = 11.548739357257746
math.asinh(1) # = 0.8813735870195429

# Функция гиперболического косинуса
math.cosh(math.pi) # = 11.591953275521519
math.acosh(1) # = 0.0

# Функция гиперболического тангенса
math.tanh(math.pi) # = 0.99627207622075
math.atanh(0.5) # = 0.5493061443340549
```

45.3. Функция pow для быстрого возведения в степень

Используем модуль `timeit` из командной строки:

```
> python -m timeit 'for x in xrange(50000): b = x**3'
10 loops, best of 3: 51.2 msec per loop
> python -m timeit 'from math import pow' 'for x in xrange(50000): b = pow(x,3)'
100 loops, best of 3: 9.15 msec per loop
```

Встроенный оператор `**` часто бывает полезен, но если производительность важна, то используйте `math.pow`. При этом следует учитывать, что `pow` возвращает данные типа `float`, даже если аргументы являются целыми числами:

```
> from math import pow
> pow(5,5)
3125.0
```


45.4. Бесконечность и NaN (“не число”)

Во всех версиях Python мы можем представить бесконечность и NaN (“не число”, “not a number”) следующим образом:

```
pos_inf = float('inf')      # положительная бесконечность
neg_inf = float('-inf')     # отрицательная бесконечность
not_a_num = float('nan')    # NaN (“не число”)
```

В Python 3.5 и выше мы также можем использовать задекларированные константы `math.inf` и `math.nan`:

Версия Python 3.x ≥ 3.5

```
pos_inf = math.inf
neg_inf = -math.inf
not_a_num = math.nan
```

Строковые представления отображаются в виде `inf` и `-inf` и `nan`:

```
pos_inf, neg_inf, not_a_num
# Результат: (inf, -inf, nan)
```

С помощью метода `isinf` мы можем провести проверку на положительную или отрицательную бесконечность:

```
math.isinf(pos_inf)
# Результат: True
```

```
math.isinf(neg_inf)
# Результат: True
```

Мы можем провести проверку на положительную или отрицательную бесконечность путем прямого сравнения:

```
pos_inf == float('inf') # или == math.inf в Python 3.5+
# Результат: True
```

```
neg_inf == float('-inf') # или == -math.inf в Python 3.5+
# Результат: True
```

```
neg_inf == pos_inf
# Результат: False
```

Python 3.2 и выше также позволяет проверять конечность:

Версия Python 3.x ≥ 3.2

```
math.isfinite(pos_inf)
# Результат: False
```

```
math.isfinite(0.0)
# Результат: True
```

Операторы сравнения работают, как и ожидалось, для положительной и отрицательной бесконечности:

```
import sys
sys.float_info.max
# Результат: 1.7976931348623157e+308 (это зависит от системы)
```

```
pos_inf > sys.float_info.max
# Результат: True
```

```
neg_inf < -sys.float_info.max
# Результат: True
```

Но если арифметическое выражение выдает значение, превышающее максимум, который может быть представлен в виде числа типа float, то оно становится бесконечностью:

```
pos_inf == sys.float_info.max * 1.0000001
# Результат: True
```

```
neg_inf == -sys.float_info.max * 1.0000001
# Результат: True
```

Однако деление на ноль не дает результата, равного бесконечности (или отрицательной бесконечности), а вызывает исключение `ZeroDivisionError`.

```
try:
    x = 1.0 / 0.0
    print(x)
except ZeroDivisionError:
    print("Деление на ноль")

# Результат: Деление на ноль
```

Арифметические операции над бесконечностью дают в результате бесконечность, или иногда NaN:

```
-5.0 * pos_inf == neg_inf
# Результат: True
```

```
-5.0 * neg_inf == pos_inf
# Результат: True
```

```
pos_inf * neg_inf == neg_inf
# Результат: True
```

```
0.0 * pos_inf
# Результат: nan
```

```
0.0 * neg_inf
# Результат: nan
```

```
pos_inf / pos_inf
# Результат: nan
```

NaN никогда не равно ничему, даже самому себе. Мы можем проверить это с помощью метода `isnan`:

```
not_a_num == not_a_num
# Результат: False
```

```
math.isnan(not_a_num)
Результат: True
```

NaN всегда сравнивается как "не равно", но никогда не меньше или больше:

```
not_a_num != 5.0 # или любое случайное значение
# Результат: True
```

```
not_a_num > 5.0 or not_a_num < 5.0 or not_a_num == 5.0
# Результат: False
```

Арифметические операции над NaN всегда дают NaN. Это касается и умножения на -1: “отрицательного NaN” не существует.

```
5.0 * not_a_num
# Результат: nan

float('-nan')
# Результат: nan
```

Python 3.x Version ≥ 3.5

```
-math.nan
# Результат: nan
```

Существует одно тонкое различие между старыми float-версиями NaN и бесконечности и константами математической библиотеки math в Python 3.5+:

Версия Python 3.x ≥ 3.5 :

```
math.inf is math.inf, math.nan is math.nan
# Результат: (True, True)

float('inf') is float('inf'), float('nan') is float('nan')
# Результат: (False, False)
```

45.5. Логарифмы

`math.log(x)` вычисляет натуральный (по основанию e) логарифм от x .

```
math.log(math.e)      # 1.0
math.log(1)           # 0.0
math.log(100)         # 4.605170185988092
```

`math.log` может терять точность при вычислении чисел, близких к 1, из-за ограничений чисел типа float. Для точного вычисления логарифмов, близких к 1, следует использовать `math.log1p`, который вычисляет натуральный логарифм от 1 плюс аргумент:

```
math.log(1 + 1e-20) # 0.0
math.log1p(1e-20) # 1e-20
```

`math.log10` может быть использован для вычисления логарифмов по основанию 10:

```
math.log10(10) # 1.0
```

Версия Python 2.x $\geq 2.3.0$

При использовании двух аргументов `math.log(x, base)` выдает логарифм x по заданному основанию (т.е. $\log(x) / \log(\text{base})$).

```
math.log(100, 10) # 2.0
math.log(27, 3) # 3.0
math.log(1, 10) # 0.0
```

45.6. Константы

Модули math включают в себя две часто используемые математические константы.

- `math.pi` – математическая константа числа “пи”
- `math.e` – математическая константа e (основание натурального логарифма)

```
>>> from math import pi, e
>>> pi
3.141592653589793
```

```
>>> e
2.718281828459045
>>>
```

В Python 3.5 и выше имеются константы для бесконечности и NaN (“не числа”). Более старый синтаксис передачи строки в `float()` по-прежнему работает.

Версия Python 3.x ≥ 3.5:

```
math.inf == float('inf')
# Результат: True

-math.inf == float('-inf')
# Результат: True

# NaN никогда не сравнивается ни с чем, даже с самим собой
math.nan == float('nan')
# Результат: False
```

45.7. Мнимые числа

Мнимые числа в Python обозначаются символом “j” или “J”, стоящим после заданного числа.

```
1j      # Эквивалентно квадратному корню из -1.
1j * 1j # = (-1+0j)
```

45.8. Копирование знаков

В Python 2.6 и выше функция `math.copysign(x, y)` возвращает `x` со знаком `y`. Возвращаемое значение всегда имеет тип `float`.

Версия Python 2.x ≥ 2.6

```
math.copysign(-2, 3)    # 2.0
math.copysign(3, -3)    # -3.0
math.copysign(4, 14.2)  # 4.0
math.copysign(1, -0.0)  # -1.0, на платформе, поддерживающей “знаковый ноль”
```

45.9. Комплексные числа и модуль `cmath`

Модуль `cmath` аналогичен модулю `math`, но определяет функции, подходящие для комплексной плоскости. Прежде всего комплексные числа – это числовой тип, который является частью самого языка Python, а не предоставляется библиотечным классом. Поэтому нам не нужно выполнять `import cmath` для обычных арифметических выражений. Обратите внимание, что мы используем `j` (или `J`), а не `i`.

```
z = 1 + 3j
```

Следует использовать `1j`, поскольку `j` будет именем переменной, а не числовым литералом.

```
1j * 1j
Результат: (-1+0j)
```

```
1j ** 1j
# Результат: (0.20787957635076193+0j) # “i to the i” == math.e ** -(math.pi/2)
```

Мы имеем действительную (`real`) и мнимую (`imag`) части, а также комплексную сопряженную (`conjugate`):

```
# действительная часть и мнимая часть имеют тип float
z.real, z.imag
# Результат: (1.0, 3.0)
```

```
z.conjugate()
# Результат: (1-3j) # z.conjugate() == z.real - z.imag * 1j
```

Встроенные функции `abs` и `complex` также являются частью самого языка и не требуют импорта:

```
abs(1 + 1j)
# Результат: 1.4142135623730951 # квадратный корень из 2
```

```
complex(1)
# Результат: (1+0j)
```

```
complex(imag=1)
# Результат: (1j)
```

```
complex(1, 1)
# Результат: (1+1j)
```

Функция `complex` может принимать строку, но в ней не может быть пробелов:

```
complex("1+1j")
# Результат: (1+1j)
```

```
complex("1 + 1j")
# Exception: ValueError: complex() arg is a malformed string (complex() arg – неправильно сформированная строка)
```

Но для большинства функций модуль все же нужен, например для `sqrt`:

```
import cmath
cmath.sqrt(-1)
# Результат: 1j
```

Естественно, поведение `sqrt` различно для комплексных и вещественных чисел. Для не-комплексных при применении модуля `math` квадратный корень из отрицательного числа вызывает исключение:

```
import math
math.sqrt(-1)
# Exception: ValueError: math domain error (ошибка математической области (домена))
```

Предусмотрены функции для преобразования в полярные координаты и обратно:

```
cmath.polar(1 + 1j)
# Результат: (1.4142135623730951, 0.7853981633974483) # == (sqrt(1 + 1), atan2(1, 1))
```

```
abs(1 + 1j), cmath.phase(1 + 1j)
# Результат: (1.4142135623730951, 0.7853981633974483) # аналогично предыдущему
```

```
cmath.rect(math.sqrt(2), math.atan(1))
# Результат: (1.0000000000000002+1.0000000000000002j)
```

Математическая область комплексного анализа выходит за рамки данного примера, но многие функции в комплексной плоскости имеют “срез ветви”, обычно вдоль действительной или мнимой оси. Большинство современных платформ поддерживают “подписанный ноль”, как указано в IEEE 754, что обеспечивает непрерывность этих функций по обе стороны от среза ветви. Следующий пример взят из документации по языку Python:

```
cmath.phase(complex(-1.0, 0.0))
# Результат: 3.141592653589793
```

```
cmath.phase(complex(-1.0, -0.0))
# Результат: -3.141592653589793
```

Модуль cmath также предоставляет множество функций, имеющих прямые аналоги из модуля math. Кроме sqrt существуют комплексные версии exp, log, log10, тригонометрических функций и их инверсий (sin, cos, tan, asin, acos, atan), а также гиперболических функций и их инверсий (sinh, cosh, tanh, asinh, acosh, atanh). Заметим, однако, что комплексный аналог math.atan2, двухаргументной формы арктангенса, отсутствует.

```
cmath.log(1+1j)
# Результат: (0.34657359027997264+0.7853981633974483j)
```

```
cmath.exp(1j * cmath.pi)
# Результат: (-1+1.2246467991473532e-16j) # e для i pi == -1, в пределах ошибки округления
```

Константы pi и e предоставляются Python. Обратите внимание, что они имеют тип данных float, а не complex.

```
type(cmath.pi)
# Результат: <class 'float'>
```

Модуль cmath также предоставляет комплексные версии isinf и (для Python 3.2+) isfinite (см. главу “Бесконечность и NaN (“не число”)”). Комплексное число считается бесконечным, если либо его действительная часть, либо мнимая часть являются бесконечными.

```
cmath.isinf(complex(float('inf'), 0.0))
# Результат: True
```

Аналогично модуль cmath предоставляет комплексную версию isnan. (См. главу “Бесконечность и NaN (“не число”)”). Комплексное число считается “не числом”, если либо его действительная часть, либо мнимая часть “не число”.

```
cmath.isnan(0.0, float('nan'))
# Результат: True
```

Обратите внимание, что в cmath аналог для констант math.inf и math.nan отсутствует (начиная с Python 3.5 и выше):

Версия Python 3.x ≥ 3.5:

```
cmath.isinf(complex(0.0, math.inf))
# Результат: True
```

```
cmath.isnan(complex(math.nan, 0.0))
# Результат: True
```

```
cmath.inf
# Exception: AttributeError: module 'cmath' has no attribute 'inf' (модуль 'cmath' не имеет атрибута 'inf')
```

В Python 3.5 и выше в модулях cmath и math есть метод isclose.

Версия Python 3.x ≥ 3.5:

```
z = cmath.rect(*cmath.polar(1+1j))
```

```
z
# Результат: (1.0000000000000002+1.0000000000000002j)
```

```
cmath.isclose(z, 1+1j)
# True
```

Глава 46. Комплексная математика

46.1. Расширенная комплексная арифметика

Модуль `cmath` включает дополнительные функции для работы с комплексными числами.

```
import cmath
```

Данный модуль позволяет вычислять фазу комплексного числа в радианах:

```
z = 2+3j          # Комплексное число
cmath.phase(z)    # 0.982793723247329
```

Он позволяет преобразовывать комплексные числа между декартовым (прямоугольным или тригонометрическим) и полярным представлениями:

```
cmath.polar(z)      # (3.605551275463989, 0.982793723247329)
cmath.rect(2, cmath.pi/2) # (0+2j)
```

Модуль содержит версию для работы с комплексными числами для:

- экспоненциальных и логарифмических функции (как обычно, \log – натуральный логарифм, \log_{10} – десятичный логарифм):

```
cmath.exp(z)        # (-7.315110094901103+1.0427436562359045j)
cmath.log(z)         # (1.2824746787307684+0.982793723247329j)
cmath.log10(-100)    # (2+1.3643763538418412j)
```

- квадратных корней:

```
cmath.sqrt(z)        # (1.6741492280355401+0.8959774761298381j)
```

- тригонометрических функций и обратных к ним:

```
cmath.sin(z)          # (9.15449914691143-4.168906959966565j)
cmath.cos(z)           # (-4.189625690968807-9.109227893755337j)
cmath.tan(z)           # (-0.003764025641504249+1.00323862735361j)
cmath.asin(z)          # (0.5706527843210994+1.9833870299165355j)
cmath.acos(z)          # (1.0001435424737972-1.9833870299165355j)
cmath.atan(z)          # (1.4099210495965755+0.22907268296853878j)
cmath.sin(z)**2 + cmath.cos(z)**2 # (1+0j)
```

- гиперболических функций и обратных к ним:

```
cmath.sinh(z)         # (-3.59056458998578+0.5309210862485197j)
cmath.cosh(z)          # (-3.7245455049153224+0.5118225699873846j)
cmath.tanh(z)          # (0.965385879022133-0.009884375038322495j)
cmath.asinh(z)         # (0.5706527843210994+1.9833870299165355j)
cmath.acosh(z)         # (1.9833870299165355+1.0001435424737972j)
cmath.atanh(z)         # (0.14694666622552977+1.3389725222944935j)
cmath.cosh(z)**2 - cmath.sin(z)**2 # (1+0j)
cmath.cosh((0+1j)*z) - cmath.cos(z) # 0j
```

46.2. Основы комплексной арифметики

В Python имеется встроенная поддержка комплексной арифметики. Мнимая единица обозначается `j`:

```
z = 2+3j          # Комплексное число
w = 1-7j          # Еще одно комплексное число
```

Комплексные числа можно складывать, вычитать, умножать, делить и возводить в степень:

```
z + w              # (3-4j)
z - w              # (1+10j)
```

```
z * w      # (23-11j)
z / w      # (-0.38+0.34j)
z**3       # (-46+9j)
```

Python также может выделять действительную и мнимую части комплексных чисел, вычислять их абсолютное значение и сопряженность:

```
z.real      # 2.0
z.imag      # 3.0
abs(z)      # 3.605551275463989
z.conjugate() # (2-3j)
```

Глава 47. Модуль collections

Встроенный пакет collections предоставляет несколько специализированных, гибких типов коллекций, которые отличаются высокой производительностью и являются альтернативой общим типам коллекций dict, list, tuple и set. Модуль также определяет абстрактные базовые классы, описывающие различные виды функциональности коллекций (например, MutableSet и ItemsView).

47.1. collections.Counter

Counter – это подкласс, позволяющий легко подсчитывать объекты. В нем имеются служебные методы для работы с частотой встречаемости объектов, которые вы подсчитываете.

```
import collections
counts = collections.Counter([1,2,3])
```

Приведенный выше код создает объект counts, который содержит частоты всех элементов, передаваемых в конструктор. В данном примере значение Counter({1: 1, 2: 1, 3: 1}).

Примеры конструкторов

Счетчик букв

```
>>> collections.Counter('Happy Birthday')
Counter({'a': 2, 'p': 2, 'y': 2, 'i': 1, 'r': 1, 'B': 1, ' ': 1, 'H': 1, 'd': 1, 'h': 1, 't': 1})
```

Счетчик слов

```
>>> collections.Counter('I am Sam Sam I am That Sam-I-am That Sam-I-am! I do not like that Sam-I-am'.
split())
Counter({'I': 3, 'Sam': 2, 'Sam-I-am': 2, 'That': 2, 'am': 2, 'do': 1, 'Sam-I-am!': 1, 'that': 1, 'not': 1, 'like': 1})
```

Рецепты

```
>>> c = collections.Counter({'a': 4, 'b': 2, 'c': -2, 'd': 0})
```

Получение количества отдельных элементов:

```
>>> c['a']
4
```

Установка количества отдельных элементов:

```
>>> c['c'] = -3
>>> c
Counter({'a': 4, 'b': 2, 'd': 0, 'c': -3})
```

Получение общего количества элементов в счетчике (4 + 2 + 0 – 3):

```
>>> sum(c.itervalues()) # отрицательные числа учитываются!
3
```


Получить элементы (сохраняются только те, которые имеют положительный счетчик):

```
>>> list(c.elements())
['a', 'a', 'a', 'a', 'b', 'b']
```

Удаление ключей с 0 или отрицательным значением:

```
>>> c = collections.Counter()
Counter({'a': 4, 'b': 2})
```

Удалить все:

```
>>> c.clear()
>>> c
Counter()
```

Добавить/удалить отдельные элементы:

```
>>> c.update({'a': 3, 'b': 3})
>>> c.update({'a': 2, 'c': 2})          # добавляет к существующим, устанавливает, если их нет
>>> c
Counter({'a': 5, 'b': 3, 'c': 2})
>>> c.subtract({'a': 3, 'b': 3, 'c': 3})  # вычитает (допускаются отрицательные значения)
>>> c
Counter({'a': 2, 'b': 0, 'c': -1})
```

47.2. collections.OrderedDict

Порядок ключей в словарях Python произвольный: он не зависит от порядка их добавления. Например:

```
>>> d = {'foo': 5, 'bar': 6}
>>> print(d)
{'foo': 5, 'bar': 6}
>>> d['baz'] = 7
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6}
>>> d['foobar'] = 8
>>> print(a)
{'baz': 7, 'foo': 5, 'bar': 6, 'foobar': 8}
```

(Произвольное упорядочивание, подразумеваемое выше, означает, что при использовании приведенного кода можно получить разные результаты по сравнению с тем, что показано здесь).

Порядок появления ключей соответствует порядку их итерации, например в цикле for. Класс collections.OrderedDict предоставляет объекты словарей, сохраняющие порядок следования ключей. Упорядоченные словари (OrderedDicts) могут быть созданы, как показано ниже, с помощью серии упорядоченных элементов (здесь – список кортежей пар “ключ-значение”):

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('foo', 5), ('bar', 6)])
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6)])
>>> d['baz'] = 7
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7)])
>>> d['foobar'] = 8
>>> print(d)
OrderedDict([('foo', 5), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Или мы можем создать пустой `OrderedDict` и затем добавить в него элементы:

```
>>> o = OrderedDict()
>>> o['key1'] = "value1"
>>> o['key2'] = "value2"
>>> print(o)
OrderedDict([('key1', 'value1'), ('key2', 'value2')])
```

Итерация в `OrderedDict` позволяет получить доступ к ключам в том порядке, в котором они были добавлены. Что произойдет, если мы присвоим новое значение существующему ключу?

```
>>> d['foo'] = 4
>>> print(d)
OrderedDict([('foo', 4), ('bar', 6), ('baz', 7), ('foobar', 8)])
```

Ключ сохраняет свое первоначальное место в `OrderedDict`.

47.3. collections.defaultdict

`collections.defaultdict(default_factory)` возвращает подкласс `dict`, имеющий значение по умолчанию для отсутствующих ключей. Аргументом должна быть функция, возвращающая значение по умолчанию при вызове без аргументов. Если ничего не передано, то по умолчанию возвращается значение `None`.

```
>>> state_capitals = collections.defaultdict(str)
>>> state_capitals
defaultdict(<class 'str'>, {})
```

Этот код возвращает ссылку на `defaultdict`, который создаст строковый объект с помощью своего метода `default_factory`.

Типичным применением `defaultdict` является использование в качестве `default_factory` одного из встроенных типов, таких как `str`, `int`, `list` или `dict`, так как они возвращают пустые типы при вызове без аргументов:

```
>>> str()
""
>>> int()
0
>>> list()
[]
```

Вызов `defaultdict` с несуществующим ключом не приводит к ошибке, как это было бы в обычном словаре.

```
>>> state_capitals['Alaska']
""
>>> state_capitals
defaultdict(<class 'str'>, {'Alaska': ""})
```

Другой пример, с типом данных `int`:

```
>>> fruit_counts = defaultdict(int)
>>> fruit_counts['apple'] += 2           # Ошибки возникать не должны
>>> fruit_counts
defaultdict(int, {'apple': 2})
>>> fruit_counts['banana']             # Ошибки возникать не должны
0
>>> fruit_counts                        # Создается новый ключ
defaultdict(int, {'apple': 2, 'banana': 0})
```

Обычные словарные методы работают со словарем по умолчанию:

```
>>> state_capitals['Alabama'] = 'Montgomery'
>>> state_capitals
defaultdict(<class 'str'>, {'Alabama': 'Montgomery', 'Alaska': ""})
```

Использование `list` в качестве `default_factory` приведет к созданию списка для каждого нового ключа.

```
>>> s = [('NC', 'Raleigh'), ('VA', 'Richmond'), ('WA', 'Seattle'), ('NC', 'Asheville')]
>>> dd = collections.defaultdict(list)
>>> for k, v in s:
...     dd[k].append(v)
>>> dd
defaultdict(<class 'list'>,
{'VA': ['Richmond'],
'NC': ['Raleigh', 'Asheville'],
'WA': ['Seattle']})
```

47.4. collections.namedtuple

Создайте новый тип `Person` с помощью `namedtuple` следующим образом:

```
Person = namedtuple('Person', ['age', 'height', 'name'])
```

Второй аргумент представляет собой список атрибутов, которыми будет обладать кортеж. Эти атрибуты также можно перечислить в виде строки, разделенной пробелом или запятой:

```
Person = namedtuple('Person', 'age, height, name')
```

или

```
Person = namedtuple('Person', 'age height name')
```

После определения именованный кортеж может быть инстанцирован путем вызова объекта с необходимыми параметрами, например:

```
dave = Person(30, 178, 'Dave')
```

Также могут использоваться именованные аргументы:

```
jack = Person(age=30, height=178, name='Jack S.')
```

Теперь можно получить доступ к атрибутам именованного кортежа:

```
print(jack.age) # 30
print(jack.name) # 'Jack S.'
```

Первым аргументом конструктора именованного кортежа (в нашем примере `'Person'`) является имя типа – `typename`. Обычно для конструктора и имени типа используется одно и то же слово, но они могут быть разными:

```
Human = namedtuple('Person', 'age, height, name')
dave = Human(30, 178, 'Dave')
print(dave) # yields: Person(age=30, height=178, name='Dave')
```

47.5. collections.deque

Возвращает новый объект `deque`, инициализированный слева направо (с помощью функции `append()`) с данными из итерируемого объекта. Если итерируемый объект не указан, то новый `deque` будет пустым.

`Deque` – это обобщение стеков и очередей (название является сокращением от “double-ended queue”, “двусторонняя очередь”). Поддерживаются потокобезопасные, экономящие память добавления и удаления с любой стороны `deque` с примерно одинаковой производительностью $O(1)$ в любом направлении. Хотя списочные объекты поддерживают аналогичные операции, они оптимизированы для быстрых операций с фиксированной длиной и несут $O(n)$ затрат на перемещение памяти для операций `pop()` и `insert(0, v)`, которые изменяют как размер, так и положение базового представления данных.

Начиная с версии Python 2.4, если параметр `maxlen` не задан или равен `None`, то `deque` может расти до произвольной длины. В противном случае `deque` ограничивается заданной

максимальной длиной. После заполнения deque ограниченной длины при добавлении новых элементов соответствующее количество элементов удаляется с противоположного конца. Дескрипторы ограниченной длины обеспечивают функциональность, аналогичную хвостовому фильтру (tail filter) в Unix. Они также полезны для отслеживания транзакций и других массивов данных, где интерес представляет только самая последняя активность.

```
>>> from collections import deque
>>> d = deque('ghi')          # создать новый deque с тремя элементами
>>> for elem in d:            # итерация по элементам deque
... print elem.upper()
G
H
I

>>> d.append('j')             # добавить новую запись в правую часть
>>> d.appendleft('f')         # добавить новую запись в левую часть
>>> d                          # показать представление deque
deque(['f', 'g', 'h', 'i', 'j'])

>>> d.pop()                   # вернуть и удалить крайний правый элемент
j
>>> d.popleft()               # вернуть и удалить крайний левый элемент
f
>>> list(d)                   # перечислить содержимое deque
['g', 'h', 'i']
>>> d[0]                       # заглянуть (peek) в крайний левый элемент
'g'
>>> d[-1]                     # заглянуть (peek) в крайний правый элемент
'i'

>>> list(reversed(d))         # перечислить содержимое deque в обратном порядке
['i', 'h', 'g']
>>> 'h' in d                  # выполнить поиск в deque
True
>>> d.extend('jkl')           # добавить несколько элементов сразу
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> d.rotate(1)               # правый поворот
>>> d
deque(['l', 'g', 'h', 'i', 'j', 'k'])
>>> d.rotate(-1)              # левый поворот
>>> d
deque(['g', 'h', 'i', 'j', 'k', 'l'])

>>> deque(reversed(d))        # создать новый deque в обратном порядке
deque(['l', 'k', 'j', 'i', 'h', 'g'])
>>> d.clear()                  # очистить deque
>>> d.pop()                    # невозможно выполнить операцию выталкивания (pop) из пустого deque
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <toplevel>
    d.pop()
IndexError: pop from an empty deque

>>> d.extendleft('abc')       # extendleft() меняет порядок ввода
>>> d
deque(['c', 'b', 'a'])
```

47.6. collections.ChainMap

Инструмент управления поиском в словарях ChainMap появился в версии Python 3.3. Он возвращает новый объект ChainMap при заданном количестве map. Этот объект объединяет несколько словарей или других отображений для создания единого, обновляемого представления.

ChainMaps полезны для управления вложенными контекстами и оверлеями. Примером в мире Python может служить реализация класса Context в шаблонизаторе Django. Он полезен для быстрого связывания нескольких отображений, чтобы результат можно было рассматривать как единое целое. Зачастую это гораздо быстрее, чем создавать новый словарь и выполнять несколько вызовов update().

В любом случае, когда имеется цепочка значений поиска, можно использовать ChainMap. Примером может служить наличие как значений, задаваемых пользователем, так и словаря значений по умолчанию. Другой пример – карты параметров POST и GET, используемые в веб-приложениях, например Django или Flask. С помощью ChainMap можно получить комбинированное представление двух разных словарей.

Список параметров maps упорядочен от первого поиска до последнего поиска. При поиске последовательно перебираются все нижележащие отображения, пока не будет найден ключ. В отличие от этого записи, обновления и удаления выполняются только на первом отображении.

```
import collections
```

```
# определим два словаря, в которых хотя бы некоторые ключи пересекаются
```

```
dict1 = {'apple': 1, 'banana': 2}
```

```
dict2 = {'coconut': 1, 'date': 1, 'apple': 3}
```

```
# создадим две ChainMap с разным упорядочиванием словарей
```

```
combined_dict = collections.ChainMap(dict1, dict2)
```

```
reverse_ordered_dict = collections.ChainMap(dict2, dict1)
```

Обратите внимание на влияние порядка нахождения первого значения на последующий поиск:

```
for k, v in combined_dict.items():  
    print(k, v)
```

```
date 1  
apple 1  
banana 2  
coconut 1
```

```
for k, v in reverse_ordered_dict.items():  
    print(k, v)
```

```
date 1  
apple 3  
banana 2  
coconut 1
```

Глава 48. Модуль operator

48.1. Функция Itemgetter

Группировка пар “ключ-значение” словаря по значению с помощью itemgetter:

```
from itertools import groupby
from operator import itemgetter
adict = {'a': 1, 'b': 5, 'c': 1}
dict((i, dict(v)) for i, v in groupby(adict.items(), itemgetter(1)))
# Результат: {1: {'a': 1, 'c': 1}, 5: {'b': 5}}
```

что эквивалентно (но быстрее) использования лямбда-функции такого типа:

```
dict((i, dict(v)) for i, v in groupby(adict.items(), lambda x: x[1]))
```

Сортировка списка кортежей по второму элементу, сначала первый элемент как вторичный:

```
alist_of_tuples = [(5,2), (1,3), (2,2)]
sorted(alist_of_tuples, key=itemgetter(1,0))
# Результат: [(2, 2), (5, 2), (1, 3)]
```

48.2. operator как альтернатива инфиксному оператору

Для каждого инфиксного оператора, например “+”, существует оператор-функция (для “+” это operator.add):

```
1 + 1
# Результат: 2
from operator import add
add(1, 1)
# Результат: 2
```

Несмотря на то, что в основной документации указано, что для арифметических операторов допускается только числовой ввод, возможно нижеследующее:

```
from operator import mul
mul('a', 10)
# Результат: 'aaaaaaaaaa'
mul([3], 3)
# Результат: [3, 3, 3]
```

48.3. Methodcaller

Вместо нижеприведенной лямбда-функции, которая вызывает метод явно:

```
alist = ['wolf', 'sheep', 'duck']
list(filter(lambda x: x.startswith('d'), alist)) # Сохранять только элементы, начинающиеся с 'd'
# Результат: ['duck']
```

можно использовать оператор-функцию, которая делает то же самое:

```
from operator import methodcaller
list(filter(methodcaller('startswith', 'd'), alist)) # Делает то же самое, но быстрее.
# Результат: ['duck']
```

Глава 49. Модуль JSON

49.1. Хранение данных в файле

Следующий фрагмент кода кодирует данные, хранящиеся в `d`, в формат JSON и сохраняет их в файл (замените параметр `filename` на реальное имя файла).

```
import json

d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}

with open(filename, 'w') as f:
    json.dump(d, f)
```

49.2. Получение данных из файла

Следующий фрагмент кода открывает закодированный в формате JSON файл (замените параметр `filename` на реальное имя файла) и возвращает объект, который хранится в этом файле.

```
import json

with open(filename, 'r') as f:
    d = json.load(f)
```

49.3. Форматирование вывода JSON

Допустим, у нас есть следующие данные:

```
>>> data = {"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Просто “сброс” в JSON-формате не дает ничего особенного:

```
>>> print(json.dumps(data))
{"cats": [{"name": "Tubbs", "color": "white"}, {"name": "Pepper", "color": "black"}]}
```

Установка отступов для получения более красивого результата

Если мы хотим вывести данные в более эстетичном виде, мы можем задать размер отступа:

```
>>> print(json.dumps(data, indent=2))
{
  "cats": [
    {
      "name": "Tubbs",
      "color": "white"
    },
    {
      "name": "Pepper",
      "color": "black"
    }
  ]
}
```

Сортировка ключей по алфавиту для получения последовательного вывода

По умолчанию порядок ключей в выводе не определен. Мы можем расположить их в алфавитном порядке, чтобы всегда получать один и тот же результат:

```
>>> print(json.dumps(data, sort_keys=True))
{'cats': [{'color': 'white', 'name': 'Tubbs'}, {'color': 'black', 'name': 'Pepper'}]}
```

Избавление от пробельных символов для получения компактного вывода

Можно избавиться от лишних пробелов, что делается путем установки строк-разделителей, отличных от стандартных ' и ': :

```
>>> print(json.dumps(data, separators=(',', ':')))
{"cats":[{"name":"Tubbs","color":"white"}, {"name":"Pepper","color":"black"}]}
```

49.4. "load" и "loads", "dump" и "dumps"

Модуль json содержит функции для чтения строк в Unicode и записи в них, а также для чтения из файлов и записи в них. Они отличаются друг от друга наличием в названии функции конечной буквы "-s". В этих примерах мы используем строковый объект ввода-вывода, но те же функции применимы для любого файлового объекта.

Здесь мы используем строковые функции:

```
import json
```

```
data = {"foo": "bar", "baz": []}
json_string = json.dumps(data)
# u'{"foo": "bar", "baz": []}'
json.loads(json_string)
# {"foo": "bar", "baz": []}
```

А здесь мы используем функции для файлов:

```
import json
```

```
from io import StringIO
```

```
json_file = StringIO()
data = {"foo": "bar", "baz": []}
json.dump(data, json_file)
json_file.seek(0) # Возврат к началу файла перед чтением
json_file_content = json_file.read()
# u'{"foo": "bar", "baz": []}'
json_file.seek(0) # Возврат к началу файла перед чтением
json.load(json_file)
# {"foo": "bar", "baz": []}
```

Как видно, основное отличие заключается в том, что при дампе JSON-данных необходимо передавать в функцию дескриптор файла, а не перехватывать возвращаемое значение. Также стоит отметить, что во избежание повреждения данных перед чтением или записью необходимо обращаться к началу файла. При открытии файла курсор помещается в позицию 0, поэтому приведенное ниже описание также будет работать:

```
import json
```

```
json_file_path = './data.json'
data = {"foo": "bar", "baz": []}
with open(json_file_path, 'w') as json_file:
    json.dump(data, json_file)
```

```
with open(json_file_path) as json_file:
    json_file_content = json_file.read()
    # u'{"foo": "bar", "baz": []}'
```

```
with open(json_file_path) as json_file:
    json.load(json_file)
    # {"foo": "bar", "baz": []}
```


Наличие обоих способов работы с JSON-данными позволяет удобно работать с форматами, основанными на JSON, такими как PySpark's json-per-line:

```
# loading from a file
data = [json.loads(line) for line in open(file_path).splitlines()]

# dumping to a file
with open(file_path, 'w') as json_file:
    for item in data:
        json.dump(item, json_file)
        json_file.write('\n')
```

49.5. Вызов “json.tool” из командной строки для эстетичного вывода JSON

Имеется некоторый JSON-файл типа “foo.json”:

```
{“foo”: {“bar”: {“baz”: 1}}}
```

Мы можем вызвать модуль непосредственно из командной строки (передав в качестве аргумента имя файла), чтобы вывести его в эстетичном виде:

```
$ python -m json.tool foo.json
{
  "foo": {
    "bar": {
      "baz": 1
```

Модуль также будет принимать входные данные из STDOUT, поэтому (в командном языке Bash) мы можем сделать то же самое:

```
$ cat foo.json | python -m json.tool
```

49.6. JSON-кодирование пользовательских объектов

Если мы попробуем сделать следующее:

```
import json
from datetime import datetime
data = {'datetime': datetime(2016, 9, 26, 4, 44, 0)}
print(json.dumps(data))
```

мы получаем ошибку `TypeError: datetime.datetime(2016, 9, 26, 4, 44) is not JSON serializable`. Чтобы правильно сериализовать объект даты-времени (`datetime`), нам необходимо написать собственный код для его преобразования:

```
class DatetimeJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        try:
            return obj.isoformat()
        except AttributeError:
            # obj не имеет метода isoformat; пусть встроенный кодировщик JSON справится с этой задачей
            return super(DatetimeJSONEncoder, self).default(obj)
```

и затем использовать этот класс кодировщика вместо `json.dumps`:

```
encoder = DatetimeJSONEncoder()
print(encoder.encode(data))
# prints {"datetime": "2016-09-26T04:44:00"}
```

49.7. Создание JSON из словаря Python

```
import json
d = {
    'foo': 'bar',
    'alice': 1,
    'wonderland': [1, 2, 3]
}
json.dumps(d)
```

Этот фрагмент кода вернет следующий результат:

```
'{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
```

49.8. Создание словаря Python из JSON

```
import json
s = '{"wonderland": [1, 2, 3], "foo": "bar", "alice": 1}'
json.loads(s)
```

Этот фрагмент кода вернет следующий результат:

```
{u'alice': 1, u'foo': u'bar', u'wonderland': [1, 2, 3]}
```

Глава 50. Модуль Sqlite3

50.1. Sqlite3 не требует отдельного серверного процесса

Модуль `sqlite3` был написан Герхардом Херингом. Для использования модуля необходимо сначала создать объект `Connection`, представляющий базу данных. Здесь данные будут храниться в файле `example.db`:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

Можно также указать специальное имя `:memory:` для создания базы данных в оперативной памяти. После создания объекта `Connection` можно создать объект `Cursor` и вызвать его метод `execute()` для выполнения SQL-команд:

```
c = conn.cursor()
```

```
# Создать таблицу
c.execute("CREATE TABLE stocks
(date text, trans text, symbol text, qty real, price real)")
```

```
# Вставить строку данных
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
```

```
# Сохранить (зафиксировать) изменения
conn.commit()
```

```
# Мы также можем закрыть соединение, если закончили с ним работать.
# Убедитесь, что все изменения были зафиксированы, иначе они будут потеряны.
conn.close()
```

50.2. Получение значений из базы данных и обработка ошибок

Рассмотрим получение значений из базы данных `SQLite3`. Выведем значения строк, возвращаемых запросом `select`:

```
import sqlite3
conn = sqlite3.connect('example.db')
```

```
c = conn.cursor()
c.execute("SELECT * from table_name where id=cust_id")
for row in c:
    print row # получится список
```

Для получения одного совпадения используется метод `fetchone()`:

```
print c.fetchone()
```

Для получения нескольких строк используйте метод `fetchall()`:

```
a=c.fetchall() # это аналогично методу list(cursor), использовавшемуся ранее
for row in a:
    print row
```

Обработка ошибок может быть выполнена с помощью встроенной функции `sqlite3.Error`:

```
try:
    #SQL код
except sqlite3.Error as e:
    print "An error occurred:", e.args[0]
```

Глава 51. Модуль os

Параметр Подробности

path	Путь к файлу. Разделитель пути может быть определен с помощью <code>os.path.sep</code> .
mode	Необходимое разрешение, в восьмеричном исчислении (например 0700)

Этот модуль обеспечивает переносимый способ использования функциональности, зависящей от операционной системы.

51.1. makedirs – рекурсивное создание каталогов

Дана локальная директория со следующим содержимым:

```
├─ dir1
│   ├── subdir1
│   └── subdir2
```

Мы хотим создать те же подкаталоги `subdir1`, `subdir2` в новом каталоге `dir2`, который еще не существует.

```
import os
```

```
os.makedirs("./dir2/subdir1")
os.makedirs("./dir2/subdir2")
```

В результате выполнения этой операции:

```
├─ dir1
│   ├── subdir1
│   └── subdir2
└─ dir2
    ├── subdir1
    └── subdir2
```

Каталог `dir2` создается в первый раз только тогда, когда он нужен для создания `subdir1`. Если бы вместо этого мы использовали `os.mkdir`, то возникло бы исключение, поскольку `dir2` еще не существовало бы.

```
os.mkdir("./dir2/subdir1")
OSError: [Errno 2] No such file or directory: './dir2/subdir1'
```

os.mkdir “не понравится”, если целевой каталог уже существует. Если мы запустим его снова:

```
OSError: [Errno 17] File exists: './dir2/subdir1'
```

Однако это можно легко исправить, перехватив исключение и проверив, что каталог был создан.

```
try:
    os.makedirs("./dir2/subdir1")
except OSError:
    if not os.path.isdir("./dir2/subdir1"):
        raise
```

```
try:
    os.makedirs("./dir2/subdir2")
except OSError:
    if not os.path.isdir("./dir2/subdir2"):
        raise
```

51.2. Создание каталога

```
os.mkdir('newdir')
```

Если необходимо указать права доступа, можно использовать необязательный аргумент mode:

```
os.mkdir('newdir', mode=0700)
```

51.3. Получение текущего каталога

Используйте функцию os.getcwd():

```
print(os.getcwd())
```

51.4. Определение операционной системы

Модуль os предоставляет интерфейс для определения типа операционной системы, на которой в данный момент работает код.

```
os.name
```

В Python 3 это может вернуть одно из следующих значений:

- posix
- nt
- ce
- java

51.5. Удаление каталога

Удаление каталога по адресу path:

```
os.rmdir(path)
```

Не следует использовать os.remove() для удаления каталога. Эта функция предназначена для файлов, и ее использование для каталогов приведет к ошибке OSError.

51.6. Следование по симлинку (POSIX)

Иногда требуется определить цель симлинка. Для этого используется os.readlink:

```
print(os.readlink(path_to_symlink))
```

51.7. Изменение разрешений файла

```
os.chmod(path, mode)
```

где mode – желаемое разрешение, в восьмеричном исчислении.

Глава 52. Модуль locale

52.1. Форматирование валюты в долларах США с использованием модуля locale

```
import locale
```

```
locale.setlocale(locale.LC_ALL, "")
```

```
Результат[2]: 'English_United States.1252'
```

```
locale.currency(762559748.49)
```

```
Результат[3]: '$762559748.49'
```

```
locale.currency(762559748.49, grouping=True)
```

```
Результат[4]: '$762,559,748.49'
```

Глава 53. Модуль Itertools

53.1. Метод комбинаций в модуле Itertools

`itertools.combinations` вернет генератор последовательности k-комбинаций списка.

Другими словами, он возвращает генератор кортежей всех возможных k-образных комбинаций входного списка.

Например, если у вас есть список:

```
a = [1,2,3,4,5]
```

```
b = list(itertools.combinations(a, 2))
```

```
print b
```

Результат:

```
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

Вышеприведенный результат представляет собой генератор, преобразованный в список кортежей всех возможных *парных комбинаций* входного списка a.

Вы также можете найти все тройные комбинации:

```
a = [1,2,3,4,5]
```

```
b = list(itertools.combinations(a, 3))
```

```
print b
```

Результат:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),  
(1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),  
(2, 4, 5), (3, 4, 5)]
```

53.2. itertools.dropwhile

Функция `itertools.dropwhile` позволяет брать элементы из последовательности после того, как условие впервые принимает значение `False`.

```
def is_even(x):
    return x % 2 == 0

lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

Это дает результаты `[13, 14, 22, 23, 44]`.

Заметим, что первым числом, нарушающим предикат (т. е. функцию, возвращающую булево значение) `is_even`, является 13. Все элементы до этого отбрасываются.

Результат, выдаваемый `dropwhile`, аналогичен результату, полученному из приведенного ниже кода.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

Конкатенация результатов, полученных с помощью `takewhile` и `dropwhile`, дает исходный итерируемый объект.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

53.3. Использование `zip_longest` для двух итераторов до тех пор, пока они оба не будут исчерпаны

Подобно встроенной функции `zip()`, функция `itertools.zip_longest` продолжит итерировать до конца более короткого из двух итерируемых объектов.

```
from itertools import zip_longest
a = [i for i in range(5)] # Длина равна 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Длина равна 7
for i in zip_longest(a, b):
    x, y = i # Обратите внимание, что zip_longest возвращает значения в виде кортежа
    print(x, y)
```

Необязательный аргумент `fillvalue` (по умолчанию это `''`) может быть передан следующим образом:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i # Обратите внимание, что zip_longest возвращает значения в виде кортежа
    print(x, y)
```

В Python 2.6 и 2.7 эта функция называется `itertools.izip_longest`.

53.4. Получение среза генератора

Функция `itertools.islice` позволяет получить срез генератора:

```
results = fetch_paged_results() # возвращает генератор
limit = 20 # Необходимо получить только первые 20 результатов
for data in itertools.islice(results, limit):
    print(data)
```

Обычно нельзя получить срез генератора:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[3]:
    print(part)
```

Результатом будет:

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[3]:
TypeError: 'generator' object is not subscriptable
```

Следующий код работает:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Обратите внимание: как и в обычном срезе, здесь также можно использовать аргументы `start`, `stop` и `step`:

```
itertools.islice(iterable, 1, 30, 3)
```

53.5. Группировка элементов из итерируемого объекта с помощью функции

Начнем с итерируемого переменного, которое необходимо сгруппировать:

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Сформируем сгруппированный генератор, группируя по второму элементу в каждом кортеже:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))
```

```
testGroupBy(lst)
```

```
# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Группируются только группы последовательных элементов. Перед вызовом `GroupBy` может потребоваться сортировка по одному и тому же ключу. Например (последний элемент изменен):

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)
```

```
# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

Группа, возвращаемая GroupBy, является итератором, который будет недействителен до следующей итерации. Например, следующий пример не будет работать, если вы хотите, чтобы группы были отсортированы по ключу. Группа 5 будет пустой, так как при извлечении группы 2 группа 5 станет недействительной.

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
    print(key, list(group))
```

```
# 2 [('c', 2, 6)]
# 5 []
```

Чтобы правильно выполнить сортировку, перед сортировкой создайте список из итератора.

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))
```

```
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

53.6. itertools.takewhile

Функция `itertools.takewhile` позволяет брать элементы из последовательности до тех пор, пока условие впервые не примет значение `False`.

```
def is_even(x):
    return x % 2 == 0

lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

В результате получается `[0, 2, 4, 12, 18]`.

Заметим, что первым числом, нарушающим предикат (т. е. функцию, возвращающую булево значение) `is_even`, является 13. Как только `takewhile` встречает значение, дающее `False` для данного предиката, он выходит. Результат, выдаваемый `takewhile`, похож на результат, полученный из приведенного ниже кода.

```
def takewhile(predicate, iterable):
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Примечание: конкатенация результатов, полученных с помощью `takewhile` и `dropwhile`, дает исходный итерируемый объект.

```
result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))
```

53.7. itertools.permutations

Функция `itertools.permutations` возвращает генератор с последовательными перестановками элементов итерируемого множества длиной `r`.

```
a = [1, 2, 3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```



```
list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Если список `a` имеет дублирующиеся элементы, то и результирующие перестановки будут иметь дублирующиеся элементы, для получения уникальных перестановок можно использовать `set`:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

53.8. itertools.repeat

Повторим что-либо `n` раз:

```
>>> import itertools
>>> for i in itertools.repeat('еще-и-еще', 3):
...     print(i)
еще-и-еще
еще-и-еще
еще-и-еще
```

53.9. Получение накопленной суммы чисел в итерируемом объекте

Версия Python 3.x ≥ 3.2:

`accumulate` выдает кумулятивную сумму (или произведение) чисел.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]

>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

53.10. Циклический переход по элементам итератора

`cycle` – это бесконечный итератор.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Поэтому во избежание образования замкнутого цикла при его использовании необходимо указывать границы. Пример:

```
>>> # Итерация по каждому элементу цикла для фиксированного диапазона
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

53.11. itertools.product

Эта функция позволяет выполнять итерацию по декартову произведению списка итерируемых переменных. Например:

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

ЭКВИВАЛЕНТНО

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Как и все функции Python, принимающие переменное число аргументов, мы можем передать список в `itertools.product` для распаковки с помощью оператора `*`. Таким образом,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

дает те же результаты, что и оба предыдущих примера:

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
    print i
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

53.12. itertools.count

Введение:

Эта простая функция генерирует бесконечный ряд чисел. Например:

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Обратите внимание, что мы должны прерваться, иначе он будет выводить вечно! Результат:

```
0 1 2 3 4 5 6 7 8 9 10
```

Arguments:

Функция `count()` принимает два аргумента: `start` and `step`:

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Результат:

```
10 14 18 22
```

53.13. Объединение нескольких итераторов в цепочку

С помощью `itertools.chain` можно создать один генератор, который будет последовательно выдавать значения из нескольких генераторов.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
''.join(chain(a, b))
```

Результатом будет:

```
'1 2 3 4 x y z'
```

В качестве альтернативного конструктора можно использовать метод класса `chain.from_iterable`, который принимает в качестве единственного параметра итерируемый объект из итерируемых объектов. Для получения того же результата, что и выше, используйте:

```
''.join(chain.from_iterable([a,b]))
```

Хотя `chain` может принимать произвольное число аргументов, `chain.from_iterable` – единственный способ выстроить цепочку из *бесконечного* числа итерируемых объектов.

Глава 54. Модуль Asyncio

54.1. Синтаксис корутин (сопрограмм) и делегирования

До выхода версии Python 3.5 модуль `asyncio` использовал генераторы для имитации асинхронных вызовов и поэтому имел синтаксис, отличающийся от синтаксиса текущего релиза Python 3.5.

Версия Python 3.x ≥ 3.5:

В Python 3.5 появились ключевые слова `async` и `await`. Обратите внимание на отсутствие круглых скобок вокруг вызова `await func()`.

```
import asyncio

async def main():
    print(await func())

async def func():
    # Выполнить трудоемкие действия...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Версия Python 3.x ≥ 3.3 < 3.5:

До версии Python 3.5 для определения корутин использовался декоратор `@asyncio.coroutine`. Для делегирования генератора использовался выход из выражения. Обратите внимание на круглые скобки вокруг `yield from func()`.

```
import asyncio

@asyncio.coroutine
def main():
    print((yield from func()))
```

```
@asyncio.coroutine
def func():
    # Выполнить трудоемкие действия...
    return "Hello, world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())
```

Версия Python 3.x ≥ 3.5:

Приведем пример, показывающий, как можно асинхронно выполнять две функции:

```
import asyncio

async def cor1():
    print("cor1 start")
    for i in range(10):
        await asyncio.sleep(1.5)
    print("cor1", i)

async def cor2():
    print("cor2 start")
    for i in range(15):
        await asyncio.sleep(1)
    print("cor2", i)

loop = asyncio.get_event_loop()
cors = asyncio.wait([cor1(), cor2()])
loop.run_until_complete(cors)
```

54.2. Асинхронные исполнители (Executors)

Примечание: используется синтаксис `async/await` языка Python 3.5+

`asyncio` поддерживает использование объектов `Executor`, находящихся в `concurrent.futures`, для асинхронного планирования задач. В циклах событий имеется функция `run_in_executor()`, которая принимает объект `Executor`, вызываемую переменную (`Callable`) и параметры вызываемой переменной.

Планирование задачи для исполнителя:

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

def func(a, b):
    # Выполнить трудоемкие действия...
    return a + b

async def main(loop):
    executor = ThreadPoolExecutor()
    result = await loop.run_in_executor(executor, func, "Hello", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main(loop))
```

Каждый цикл событий также имеет слот `Executor` “по умолчанию”, который может быть назначен исполнителю. Для назначения исполнителя и планирования задач из цикла используется метод `set_default_executor()`.

```
import asyncio
from concurrent.futures import ThreadPoolExecutor
```

```
def func(a, b):
    # Выполнить трудоемкие действия...
    return a + b

async def main(loop):
    # Примечание: использование `None` в качестве первого параметра обозначает Executor
    # 'по умолчанию'.
    result = await loop.run_in_executor(None, func, "Hello", " world!")
    print(result)

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.set_default_executor(ThreadPoolExecutor())
    loop.run_until_complete(main(loop))
```

В `concurrent.futures` существует два основных типа исполнителей: `ThreadPoolExecutor` и `ProcessPoolExecutor`. `ThreadPoolExecutor` содержит пул потоков, который может быть либо задан вручную в конструкторе, либо по умолчанию равен количеству ядер на машине, умноженному на 5. `ThreadPoolExecutor` использует пул потоков для выполнения назначенных ему задач и, как правило, лучше справляется с операциями, связанными с процессором, а не с операциями ввода-вывода. Это отличает его от `ProcessPoolExecutor`, который порождает новый процесс для каждой назначенной ему задачи. `ProcessPoolExecutor` может принимать только задачи и параметры, которые относятся к категории `picklable` (т. е. преобразуются в поток байтов и обратно). Наиболее распространенные `non-picklable` задачи – это методы объектов. Если необходимо запланировать метод объекта в качестве задачи в исполнителе, необходимо использовать `ThreadPoolExecutor`.

54.3. Использование UVLoop

`uvloop` – это реализация `asyncio.AbstractEventLoop`, основанная на библиотеке `libuv` (используется в `nodejs`). Она совместима с 99% функций модуля `asyncio` и работает гораздо быстрее, чем традиционный `asyncio.EventLoop`. В настоящее время `uvloop` недоступен для Windows, установите его с помощью `pip install uvloop`.

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop(uvloop.new_event_loop())
    # Выполняйте свои процессы здесь...
```

Можно также изменить фабрику циклов событий, установив в `uvloop` значение `EventLoopPolicy`.

```
import asyncio
import uvloop

if __name__ == "__main__":
    asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
    loop = asyncio.new_event_loop()
```

54.4. ПрIMITив синхронизации: Event

Концепция

Используйте `Event` для синхронизации планирования нескольких корутин (сопрограмм). Проще говоря, событие (`Event`) – это как стартовый выстрел в беге: оно позволяет бегунам выйти на старт.

Пример

```
# event trigger function (функция запуска события)
def trigger(event):
```

```

print('EVENT SET')
event.set() # пробуждение ожидающих корутин

# потребители событий
async def consumer_a(event):
    consumer_name = 'Consumer A'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

async def consumer_b(event):
    consumer_name = 'Consumer B'
    print('{} waiting'.format(consumer_name))
    await event.wait()
    print('{} triggered'.format(consumer_name))

# событие
event = asyncio.Event()

# обернуть корутины в одно будущее
main_future = asyncio.wait([consumer_a(event),
                           consumer_b(event)])

# цикл событий
event_loop = asyncio.get_event_loop()
event_loop.call_later(0.1, functools.partial(trigger, event)) # срабатывание события через 0,1 с

# завершить main_future
done, pending = event_loop.run_until_complete(main_future)

Результат:
Consumer B waiting
Consumer A waiting
EVENT SET
Consumer B triggered
Consumer A triggered

```

54.5. Простой Websocket

Создадим простой echo websocket с использованием модуля `asyncio`. Для соединения с сервером и отправки/получения сообщений определим корутины. Обмен сообщениями в websocket происходит в главной корутине `main`, которая запускается циклом событий.

```

import asyncio
import aiohttp

session = aiohttp.ClientSession() # работает с менеджером контекста
class EchoWebsocket:

    async def connect(self):
        self.websocket = await session.ws_connect("wss://echo.websocket.org")

    async def send(self, message):
        self.websocket.send_str(message)

    async def receive(self):
        result = (await self.websocket.receive())
        return result.data

async def main():
    echo = EchoWebsocket()

```

```

await echo.connect()
await echo.send("Hello World!")
print(await echo.receive())      # "Hello World!"

if __name__ == '__main__':
    # главный цикл
    loop = asyncio.get_event_loop()
    loop.run_until_complete(main())

```

54.6. Распространенные заблуждения, связанные с модулем asyncio

Вероятно, *самое* распространенное заблуждение об asyncio заключается в том, что он будто бы позволяет выполнять любые задачи параллельно, обходя блокировку GIL (global interpreter lock) и, следовательно, параллельно (в отдельных потоках) позволяет выполнять блокирующие задания. Это **не так**!

Модуль asyncio и библиотеки, созданные для совместной работы с asyncio, построены на основе корутин. Обратите внимание на asyncio.sleep в примерах выше. Это пример не-блокирующей корутины, которая ждет “в фоновом режиме” и передает управление обратно вызывающей функции (при вызове с помощью await). time.sleep – пример блокирующей функции. Выполнение программы просто остановится и вернется только после завершения time.sleep. Реальным примером может служить библиотека requests, состоящая только из блокирующих функций. При вызове любой из ее функций в рамках asyncio параллельность отсутствует. aiohttp, напротив, создавалась с учетом требований asyncio. Ее корутины будут выполняться параллельно.

- Если у вас есть длительные задачи, привязанные к процессору, которые вы хотели бы выполнять параллельно, то asyncio **не для вас**. Для этого вам нужны потоки (threads) или многопроцессорная обработка (multiprocessing).
- Если у вас есть задания, связанные с вводом-выводом (IO), вы *можете* запускать их параллельно с помощью asyncio.

Глава 55. Модуль Random

55.1. Создание случайного пароля пользователя

Для создания случайного пароля пользователя мы можем использовать символы, представленные в модуле string. В частности, punctuation – для знаков препинания, ascii_letters – для букв и digits – для цифр:

```
from string import punctuation, ascii_letters, digits
```

Затем мы можем объединить все эти символы в имя с названием symbols:

```
symbols = ascii_letters + digits + punctuation
```

Удалив любой из них, можно получить пул символов с меньшим количеством элементов. После этого мы можем использовать random.SystemRandom для генерации пароля. Для пароля длиной в 10 символов:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password) # '^@g;J?]M6e'
```

Отметим, что другие корутины (подпрограммы), которые сразу же становятся доступными в модуле random, такие как random.choice, random.randint и т. д., **не подходят для криптографических целей**. “За кулисами” этих процедур скрывается использование генератора псевдослучайных чисел Mersenne Twister PRNG, который не удовлетворяет требованиям

CSPRNG (Криптографически стойкого генератора псевдослучайных чисел). Поэтому, в частности, не следует использовать ни одну из них для генерации паролей, которые вы планируете использовать. Всегда используйте экземпляр `SystemRandom`, как показано выше.

Версия Python 3.x ≥ 3.6

Начиная с Python 3.6 доступен модуль `secrets`, который использует криптографически безопасную функциональность. Цитируя официальную документацию, для создания “десяти-символьного буквенно-цифрового пароля, содержащего как минимум один строчный символ, как минимум один прописной символ и не менее трех цифр”, можно сделать следующее:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = "".join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
        and any(c.isupper() for c in password)
        and sum(c.isdigit() for c in password) >= 3):
        break
```

55.2. Создание криптографически защищенных случайных чисел

По умолчанию модуль `random` в Python использует для генерации случайных чисел генератор псевдослучайных чисел Mersenne Twister PRNG, который, хотя и подходит для таких областей, как моделирование, не удовлетворяет требованиям безопасности в более сложных средах. Для создания криптографически защищенного псевдослучайного числа можно использовать `SystemRandom`, который, используя `os.urandom`, способен выступать в роли криптографически защищенного генератора псевдослучайных чисел, CPRNG.

Самый простой способ его использования заключается в инициализации класса `SystemRandom`. Предоставляемые методы аналогичны методам, экспортируемым модулем `random`.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

Для создания случайной последовательности из 10 чисел типа `int` в диапазоне `[0, 20]` можно просто вызвать `randrange()`:

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

Для создания случайного целого числа в заданном диапазоне можно использовать `randint`:

```
print(secure_rand_gen.randint(0, 20))
# 5
```

Подходы для всех остальных методов аналогичны. Интерфейс точно такой же, меняется только генератор чисел, лежащий в основе. Для получения криптографически защищенных случайных байтов можно также напрямую использовать метод `os.urandom`.

55.3. Случайности и последовательности: перемешивание, выбор и выборка

```
import random
```

```
shuffle()
```

Вы можете использовать `random.shuffle()` для перемешивания/рандомизации элементов в изменяемой и индексируемой последовательности. Например, списка:

```
laughs = ["Hi", "Ho", "He"]
random.shuffle(laughs)    # Перемешивает на месте! Не делайте: laughs = random.shuffle(laughs)
print(laughs)
# Вывод: ["He", "Hi", "Ho"] # Результат может отличаться!
```


choice()

Берет случайный элемент из произвольной **последовательности**:

```
print(random.choice(laughs))
```

Вывод: He

Результат может отличаться!

sample()

Как и в случае с choice, он берет случайные элементы из произвольной **последовательности**, но можно указать их количество:

```
        # |—последовательность—|—number—|
print(random.sample(laughs, 1)) # Взять один элемент
# Вывод: ['Ho']                 # Результат может отличаться!
```

он не будет брать один и тот же элемент дважды:

```
print(random.sample(laughs, 3)) # Возьмем 3 случайных элемента из последовательности
# Вывод: ['Ho', 'He', 'Hi']    # Результат может отличаться!
print(random.sample(laughs, 4)) # Возьмем 4 случайных элемента из последовательности
```

ValueError: Sample larger than population (выборка больше количества элементов)

55.4. Создание чисел типов int и float: randint, randrange, random и uniform

```
import random
```

randint()

Возвращает случайное целое число в диапазоне от x до y (включительно):

```
random.randint(x, y)
```

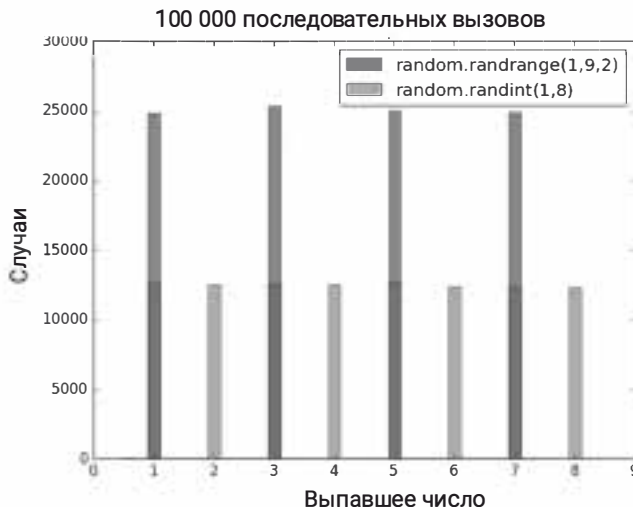
Например, получим случайное число от 1 до 8:

```
random.randint(1, 8) # Результат: 8
```

randrange()

У random.randrange тот же синтаксис, что и у range, но, в отличие от random.randint, последнее значение не включается:

```
random.randrange(100) # Случайное целое число от 0 до 99
random.randrange(20, 50) # Случайное целое число от 20 до 49
random.randrange(10, 20, 3) # Случайное целое число от 10 до 19 с шагом 3 (10, 13, 16 и 19)
```



random

Возвращает случайное число с плавающей точкой, в диапазоне от 0 до 1:

```
random.random() # Результат: 0.66486093215306317
```

uniform

Возвращает случайное число с плавающей точкой, между x и y (включительно):

```
random.uniform(1, 8) # Результат: 3.726062641730108
```

55.5. Воспроизводимые случайные числа: методы Seed и State

При задании конкретного значения seed (случайного начального значения) создается серия случайных чисел:

```
random.seed(5)                # Создать фиксированное состояние
print(random.randrange(0, 10)) # Получить случайное целое число от 0 до 9
# Результат: 9
print(random.randrange(0, 10))
# Результат: 4
```

Сброс Seed снова создаст ту же самую “случайную” последовательность:

```
random.seed(5) # Сбросить модуль random в то же фиксированное состояние
print(random.randrange(0, 10))
# Результат: 9
print(random.randrange(0, 10))
# Результат: 4
```

Поскольку seed зафиксировано, результаты всегда равны 9 и 4. Если наличие конкретных чисел не требуется, а требуется только, чтобы значения были одинаковыми, то можно также просто использовать `getstate` и `setstate` для возврата в предыдущее состояние:

```
save_state = random.getstate() # Получить текущее состояние
print(random.randrange(0, 10))
# Результат: 5
print(random.randrange(0, 10))
# Результат: 8

random.setstate(save_state)    # Сброс в текущее состояние
print(random.randrange(0, 10))
# Результат: 5
print(random.randrange(0, 10))
# Результат: 8
```

Чтобы создать псевдослучайную последовательность, используйте seed со значением `None`:

```
random.seed(None)

или вызовите метод seed без аргументов:
random.seed()
```

55.6. Случайное двоичное решение

```
import random
```

```
probability = 0.3
```

```
if random.random() < probability:
    print("Решение с вероятностью 0.3")
else:
    print("Решение с вероятностью 0.7")
```

Глава 56. Модуль Functools

56.1. Функция partial

Частичная функция `partial` создает частичное применение функции из другой функции. Она используется для *привязки значений* к некоторым аргументам функции (или именованных аргументов) и создания *вызываемой функции* без уже заданных аргументов.

```
>>> from functools import partial
>>> unhex = partial(int, base=16)
>>> unhex.__doc__ = 'Convert base16 string to int'
>>> unhex('callable')
3390155550
```

`partial()`, как следует из названия, позволяет частично оценить функцию. Рассмотрим следующий пример:

```
In [2]: from functools import partial

In [3]: def f(a, b, c, x):
...:     return 1000*a + 100*b + 10*c + x

In [4]: g = partial(f, 1, 1, 1)

In [5]: print g(2)
1112
```

При создании `g` функция `f`, принимающая четыре аргумента (`a`, `b`, `c`, `x`), также частично оценивается для первых трех аргументов – `a`, `b`, `c`. Оценка `f` завершается при вызове `g`, `g(2)`, который передает `f` четвертый аргумент. Один из способов представления `partial` – это сдвиговый регистр, вставляющий в некоторую функцию по одному аргументу за раз. `partial` удобен в тех случаях, когда данные поступают в виде потока и мы не можем передать более одного аргумента.

56.2. cmp_to_key

В Python изменились методы сортировки, которые принимают функцию ключа. Эти функции принимают значение и возвращают ключ, который используется для сортировки массивов.

Старые функции сравнения принимали два значения и возвращали `-1`, `0` или `+1`, если первый аргумент был меньше, равен или больше второго аргумента соответственно. Это несовместимо с новой функцией `key`. Вот тут-то и приходит на помощь `functools.cmp_to_key`:

```
>>> import functools
>>> import locale
>>> sorted(['A', 'S', 'F', 'D'], key=functools.cmp_to_key(locale.strcoll))
['A', 'D', 'F', 'S']
```

56.3. @lru_cache

Декоратор `@lru_cache` можно использовать для обертывания дорогой, требующей больших вычислений функции кэшем Least Recently Used (с наименьшим количеством последних использованных данных). Это позволяет мемоизировать вызовы функций, так что последующие вызовы с теми же параметрами могут возвращаться мгновенно, а не вычисляться заново.

```
@lru_cache(maxsize=None) # Безграничный кэш
def fibonacci(n):
    if n < 2:
```

```

    return n
    return fibonacci(n-1) + fibonacci(n-2)

```

```
>>> fibonacci(15)
```

В приведенном примере значение `fibonacci(3)` вычисляется только один раз, тогда как если бы у `fibonacci` не было LRU-кэша, `fibonacci(3)` вычислялось бы до 230 раз. Следовательно, `@lru_cache` особенно хорошо подходит для рекурсивных функций или динамического программирования, когда ресурсоемкая функция может быть вызвана несколько раз с одними и теми же параметрами.

`@lru_cache` имеет два аргумента:

- `maxsize`: Количество сохраняемых вызовов. Когда количество уникальных вызовов превысит `maxsize`, LRU-кэш удалит “наименее недавно использованные” вызовы.
- `typed` (добавлено в версии 3.3): это флаг для определения принадлежности эквивалентных аргументов разных типов к разным записям кэша (т. е. считаются ли 3 и 3.0 разными аргументами)

Мы также можем видеть статистику кэша:

```

>>> fib.cache_info()
CacheInfo(hits=13, misses=16, maxsize=None, currsize=16)

```

Примечание: поскольку `@lru_cache` использует словари для кэширования результатов, все параметры для функции должны быть хешируемыми, чтобы кэш работал.

56.4. @total_ordering

Когда мы хотим создать упорядоченный класс, обычно нам необходимо определить методы `__eq__()`, `__lt__()`, `__le__()`, `__gt__()` и `__ge__()`.

Декоратор `total_ordering`, примененный к классу, позволяет определить `__eq__()` и только один из `__lt__()`, `__le__()`, `__gt__()` и `__ge__()`, при этом допускаются все операции упорядочивания класса.

```

@total_ordering
class Employee:

```

```

    def __eq__(self, other):
        return ((self.surname, self.name) == (other.surname, other.name))

```

```

    def __lt__(self, other):
        return ((self.surname, self.name) < (other.surname, other.name))

```

Декоратор использует сочетание предоставленных методов и алгебраических операций для получения других методов сравнения. Например, если мы определим `__lt__()` и `__eq__()` и хотим получить `__gt__()`, можно просто проверить `not __lt__()` and `not __eq__()`.

Примечание: функция `total_ordering` доступна только начиная с версии Python 2.7.

56.5. reduce

В Python 3.x функция `reduce`, о которой уже упоминалось, была удалена из встроенных модулей и теперь должна импортироваться из `functools`.

```

from functools import reduce
def factorial(n):
    return reduce(lambda a, b: (a*b), range(1, n+1))

```

Глава 57. Модуль dis

57.1. Что такое байт-код Python?

Python является гибридным интерпретатором. При выполнении программы он сначала ассемблирует ее в *байт-код*, который затем может быть запущен в интерпретаторе Python (также называемом *виртуальной машиной Python*). Модуль `dis` из стандартной библиотеки позволяет сделать байт-код Python человекочитаемым путем дизассемблирования классов, методов, функций и объектов кода.

```
>>> def hello():
...     print "Hello, World"
...
>>> dis.dis(hello)
2      0 LOAD_CONST      1 ('Hello, World')
      3 PRINT_ITEM
      4 PRINT_NEWLINE
      5 LOAD_CONST      0 (None)
      8 RETURN_VALUE
```

Интерпретатор Python основан на стеке и использует систему *first-in last-out* (“первым пришел – последним ушел”).

Каждый код операции (опкод) в языке ассемблера Python (байт-код) забирает из стека определенное количество элементов и возвращает в стек определенное количество элементов. Если в стеке не хватает элементов для выполнения опкода, то интерпретатор Python завершает работу, возможно, без сообщения об ошибке.

57.2. Константы в модуле dis

```
EXTENDED_ARG = 145 # Все опкоды, превышающие это значение, имеют 2 операнда
HAVE_ARGUMENT = 90 # Все опкоды, превышающие это значение, имеют хотя бы 1 операнд
```

```
cmp_op = ('<', '<=', '==', '!=', '>', '>=', 'in', 'not in', 'is', 'is ...
# Список идентификаторов компараторов. Индексы используются в качестве
# операндов в некоторых операционных кодах
```

```
# Все опкоды в этих списках имеют соответствующие типы в качестве операндов
hascompare = [107]
hasconst = [100]
hasfree = [135, 136, 137]
hasjabs = [111, 112, 113, 114, 115, 119]
hasjrel = [93, 110, 120, 121, 122, 143]
haslocal = [124, 125, 126]
hasname = [90, 91, 95, 96, 97, 98, 101, 106, 108, 109, 116]
```

```
# Карта соответствия опкодов идентификаторам
opmap = {'BINARY_ADD': 23, 'BINARY_AND': 64, 'BINARY_DIVIDE': 21, 'BIN...
# Карта идентификаторов к опкодам
opname = ['STOP_CODE', 'POP_TOP', 'ROT_TWO', 'ROT_THREE', 'DUP_TOP', '...
```

57.3. Демонтаж модулей

Для дизассемблирования модуля Python необходимо сначала превратить его в файл `.рус` (скомпилированный Python). Для этого выполните команду:

```
python -m compileall <file>.py
```

Затем в интерпретаторе выполните:

```
import dis
import marshal
with open("<file>.пyc", "rb") as code_f:
    code_f.read(8) # Магическое число и время изменения
    code = marshal.load(code_f) # Возвращает объект кода, который может быть дизассемблирован
    dis.dis(code) # Вывод дизассемблирования
```

Это скомпилирует модуль Python и выведет инструкции байт-кода с dis. Модуль никогда не импортируется, поэтому его безопасно использовать с непроверенным кодом.

Глава 58. Модуль base64

Параметр	Описание
base64.b64encode(s, altchars=None)	
s	Байтоподобный объект
altchars	Байтоподобный объект длиной более 2 символов для замены символов "+" и "=" при создании алфавита Base64. Дополнительные символы игнорируются.
base64.b64decode(s, altchars=None, validate=False)	
s	Байтоподобный объект
altchars	Байтоподобный объект длиной более 2 символов для замены символов "+" и "=" при создании алфавита Base64. Дополнительные символы игнорируются.
validate	Если validate равно True, то символы, не входящие в обычный или альтернативный алфавит Base64, не отбрасываются перед проверкой заполнения.
base64.standard_b64encode(s)	
s	Байтоподобный объект
base64.standard_b64decode(s)	
s	Байтоподобный объект
base64.urlsafe_b64encode(s)	
s	Байтоподобный объект
base64.urlsafe_b64decode(s)	
s	Байтоподобный объект
b32encode(s)	
s	Байтоподобный объект
b32decode(s)	
s	Байтоподобный объект
base64.b16encode(s)	
s	Байтоподобный объект
base64.b16decode(s)	
s	Байтоподобный объект
base64.a85encode(b, foldspaces=False, wrapcol=0, pad=False, adobe=False)	
b	Байтоподобный объект

Параметр	Описание
foldspaces	Если значение foldspaces равно True, то вместо 4 последовательных пробелов будет использоваться символ 'у'.
wrapcol	Количество символов перед новой строкой (0 означает отсутствие новой строки)
pad	Если значение pad равно True, то перед кодированием байты будут дозаполнены до значений, кратных 4
adobe	Если значение adobe равно True, то кодированная последовательность будет обрамлена символами '<-' и '->', используемыми в продуктах Adobe
base64.a85decode(b, foldspaces=False, adobe=False, ignorechars=b'\t\n\r\v')	
b	Байтоподобный объект
foldspaces	Если значение foldspaces равно True, то вместо 4 последовательных пробелов будет использоваться символ 'у'.
adobe	Если значение adobe равно True, то кодированная последовательность будет обрамлена символами '<-' и '->', используемыми в продуктах Adobe
ignorechars	Байтоподобный объект, содержащий символы, которые необходимо игнорировать в кодировке
base64.b85encode(b, pad=False)	
b	Байтоподобный объект
pad	Если значение pad равно True, то перед кодированием байты будут дозаполнены до значений, кратных 4
base64.b85decode(b)	
b	Байтоподобный объект

Кодировка Base 64 представляет собой общую схему кодирования двоичных данных в строковый формат ASCII с использованием разновидности кодирования radix 64. Модуль base64 является частью стандартной библиотеки, то есть устанавливается вместе с Python. Понимание байтов и строк очень важно для данной темы. В этой главе объясняется, как использовать различные возможности и основания систем счисления модуля base64.

58.1. Кодирование и декодирование Base64

Чтобы включить модуль base64 в свой скрипт, необходимо сначала его импортировать:

```
import base64
```

Для работы функций кодирования и декодирования base64 требуется байтоподобный объект. Чтобы перевести нашу строку в байты, мы должны закодировать ее с помощью встроенной в Python функции. Чаще всего используется кодировка UTF-8, однако полный список этих стандартных кодировок (включая языки с разными символами) можно найти в официальной документации по Python. Ниже приведен пример кодирования строки в байты:

```
s = "Hello World!"
b = s.encode("UTF-8")
```

Вывод последней строки будет таким:

```
b'Hello World!'
```

Префикс b используется для обозначения того, что значение является байтовым объектом. Для Base64-кодирования этих байтов мы используем функцию base64.b64encode():

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
print(e)
```

Этот код выведет результат:

```
b'SGVsbG8gV29ybGQh'
```

который по-прежнему является байтовым объектом. Чтобы получить из этих байтов строку, мы можем использовать метод Python `decode()`, применив кодировку UTF-8:

```
import base64
s = "Hello World!"
b = s.encode("UTF-8")
e = base64.b64encode(b)
s1 = e.decode("UTF-8")
print(s1)
```

На выходе получаем:

```
SGVsbG8gV29ybGQh
```

Если бы мы хотели закодировать строку, а затем декодировать, то могли бы использовать метод `base64.b64decode()`:

```
import base64
# Создание строки
s = "Hello World!"
# Кодирование строки в байты
b = s.encode("UTF-8")
# Base64-кодирование байтов
e = base64.b64encode(b)
# Декодирование байтов Base64 в строку
s1 = e.decode("UTF-8")
# Вывод строки в кодировке Base64
print("Base64 Encoded:", s1)
# Кодирование строки в кодировке Base64 в байты
b1 = s1.encode("UTF-8")
# Декодирование байтов в формате Base64
d = base64.b64decode(b1)
# Декодирование байтов в строку
s2 = d.decode("UTF-8")
print(s2)
```

Как и следовало ожидать, результатом будет исходная строка:

```
Base64 Encoded: SGVsbG8gV29ybGQh
Hello World!
```

58.2. Кодирование и декодирование Base32

Модуль `base64` также включает в себя функции кодирования и декодирования для Base32. Эти функции очень похожи на функции Base64:

```
import base64
# Создание строки
s = "Hello World!"
# Кодирование строки в байты
b = s.encode("UTF-8")
# Base32-кодирование байтов
e = base64.b32encode(b)
# Декодирование байтов Base32 в строку
s1 = e.decode("UTF-8")
# Вывод строки в кодировке Base32
```



```
print("Base32 Encoded:", s1)
# Кодирование строки в кодировке Base32 в байты
b1 = s1.encode("UTF-8")
# Декодирование байтов Base32
d = base64.b32decode(b1)
# Декодирование байтов в строку
s2 = d.decode("UTF-8")
print(s2)
```

В результате будет получено:

```
Base32 Encoded: JBSWY3DPEBLW64TMMQQQ====
Hello World!
```

58.3. Кодирование и декодирование Base16

Модуль base64 также включает функции кодирования и декодирования для Base16, чаще всего называемой **шестнадцатеричной** системой. Эти функции очень похожи на функции Base64 и Base32:

```
import base64
# Создание строки
s = "Hello World!"
# Кодирование строки в байты
b = s.encode("UTF-8")
# Base16-кодирование байтов
e = base64.b16encode(b)
# Декодирование байтов Base16 в строку
s1 = e.decode("UTF-8")
# Вывод строки в кодировке Base16
print("Base16 Encoded:", s1)
# Кодирование строки в кодировке Base16 в байты
b1 = s1.encode("UTF-8")
# Декодирование байтов Base16
d = base64.b16decode(b1)
# Декодирование байтов в строку
s2 = d.decode("UTF-8")
print(s2)
```

В результате будет получено:

```
Base16 Encoded: 48656C6C6F20576F726C6421
Hello World!
```

58.4. Кодирование и декодирование ASCII85

Компания Adobe создала собственную кодировку **ASCII85**, которая похожа на Base85, но имеет отличия. Эта кодировка часто используется в файлах Adobe PDF. Эти функции появились в Python версии 3.4. В остальном функции base64.a85encode() и base64.a85decode() аналогичны предыдущим:

```
import base64
# Создание строки
s = "Hello World!"
# Кодирование строки в байты
b = s.encode("UTF-8")
# ASCII85 кодирование байтов
e = base64.a85encode(b)
# Декодирование байтов ASCII85 в строку
s1 = e.decode("UTF-8")
# Вывод строки в кодировке ASCII85
print("ASCII85 Encoded:", s1)
# Кодирование строки в кодировке ASCII85 в байты
```

```
b1 = s1.encode("UTF-8")
# Декодирование байтов ASCII85
d = base64.a85decode(b1)
# Декодирование байтов в строку
s2 = d.decode("UTF-8")
print(s2)
```

В результате будет получено:

```
ASCII85 Encoded: 87cURDji,"Ebo80
Hello World!
```

58.5. Кодирование и декодирование Base85

Как и функции Base64, Base32 и Base16, функции кодирования и декодирования Base85 являются base64.b85encode() и base64.b85decode():

```
import base64
# Создание строки
s = "Hello World!"
# Кодирование строки в байты
b = s.encode("UTF-8")
# Base85 Кодирование байтов
e = base64.b85encode(b)
# Декодирование байтов Base85 в строку
s1 = e.decode("UTF-8")
# Печать строки в кодировке Base85
print("Base85 Encoded:", s1)
# Кодирование строки в кодировке Base85 в байты
b1 = s1.encode("UTF-8")
# Декодирование байтов Base85
d = base64.b85decode(b1)
# Декодирование байтов в строку
s2 = d.decode("UTF-8")
print(s2)
```

В результате будет получено:

```
Base85 Encoded: NM&qnZy;B1a%^NF
Hello World!
```

Глава 59. Модуль Queue

Модуль Queue реализует очереди с несколькими производителями и потребителями. Он особенно полезен при многопоточном программировании, когда необходимо обеспечить безопасный обмен информацией между несколькими потоками. Модуль Queue предоставляет три типа очередей: 1. Queue 2. LifoQueue 3. PriorityQueue. Исключение, которая может быть: 1) полной (очередь переполнена) или 2) пустой (очередь опустошена).

59.1. Простой пример

```
from Queue import Queue

question_queue = Queue()

for x in range(1,10):
    temp_dict = {'key', x}
    question_queue.put(temp_dict)
```

```
while(not question_queue.empty()):
    item = question_queue.get()
    print(str(item))
```

Результат:

```
('key', 1)
('key', 2)
('key', 3)
('key', 4)
('key', 5)
('key', 6)
('key', 7)
('key', 8)
('key', 9)
```

Глава 60. Модуль Deque

Параметр	Подробности
iterable	Создает deque с начальными элементами, скопированными из другого итерируемого объекта
maxlen	Ограничивает размер deque, вытесняя старые элементы по мере добавления новых

60.1. Базовое использование deque

Основные методы, которые полезно использовать в этом классе – `popleft` и `appendleft`.

```
from collections import deque
```

```
d = deque([1, 2, 3])
p = d.popleft()    # p = 1, d = deque([2, 3])
d.appendleft(5)    # d = deque([5, 2, 3])
```

60.2. Доступные методы в deque

Создание пустого объекта deque:

```
dl = deque() # deque([]) создание пустого deque
```

Создание deque с некоторыми элементами:

```
dl = deque([1, 2, 3, 4]) # deque([1, 2, 3, 4])
```

Добавление элемента в deque:

```
dl.append(5) # deque([1, 2, 3, 4, 5])
```

Добавление элемента в левую часть deque:

```
dl.appendleft(0) # deque([0, 1, 2, 3, 4, 5])
```

Добавление списка элементов в deque:

```
dl.extend([6, 7]) # deque([0, 1, 2, 3, 4, 5, 6, 7])
```

Добавление списка элементов с левой стороны:

```
dl.extendleft([-2, -1]) # deque([-1, -2, 0, 1, 2, 3, 4, 5, 6, 7])
```

Использование `.pop()` приведет к удалению элемента из правой части:

```
dl.pop() # 7 => deque([-1, -2, 0, 1, 2, 3, 4, 5, 6])
```

Использование элемента `.popleft()` для удаления элемента с левой стороны:

```
dl.popleft() # -1 deque([-2, 0, 1, 2, 3, 4, 5, 6])
```

Удалить элемент по его значению:

```
dl.remove(1) # deque([-2, 0, 2, 3, 4, 5, 6])
```

Обратный порядок элементов в deque:

```
dl.reverse() # deque([6, 5, 4, 3, 2, 0, -2])
```

60.3. Ограничение размера deque

Используйте параметр `maxlen` при создании deque для ограничения размера:

```
from collections import deque
d = deque(maxlen=3)      # вмещает только 3 элемента
d.append(1)              # deque([1])
d.append(2)              # deque([1, 2])
d.append(3)              # deque([1, 2, 3])
d.append(4)              # deque([2, 3, 4]) (1 удалена, так как maxlen равен 3)
```

60.4. Поиск по ширине (Breadth First Search)

Deque – единственная структура данных Python, поддерживающая быстрыми операции очереди. (Заметим, что `queue.Queue` обычно не подходит, поскольку он предназначен для взаимодействия между потоками.) Основным вариантом использования `Queue` является поиск по ширине.

```
from collections import deque
```

```
def bfs(graph, root):
    distances = {}
    distances[root] = 0
    q = deque([root])
    while q:
        # Самым старым (но еще не посещенным) узлом будет крайний левый узел.
        current = q.popleft()
        for neighbor in graph[current]:
            if neighbor not in distances:
                distances[neighbor] = distances[current] + 1
                # Когда видим новый узел, то добавляем его в правую часть очереди.
                q.append(neighbor)
    return distances
```

Допустим, у нас есть простой направленный граф:

```
graph = {1:[2,3], 2:[4], 3:[4,5], 4:[3,5], 5:[]}
```

Теперь мы можем находить расстояния от некоторого начального положения:

```
>>> bfs(graph, 1)
{1: 0, 2: 1, 3: 1, 4: 2, 5: 2}

>>> bfs(graph, 3)
{3: 0, 4: 1, 5: 1}
```

Глава 61. Модуль Webbrowser

Параметр	Подробности
<code>webbrowser.open()</code>	
<code>url</code>	URL-адрес для открытия в веб-браузере
<code>new</code>	0 открывает URL в существующей вкладке, 1 открывает в новом окне, 2 открывает в новой вкладке
<code>autoraise</code>	если установлено значение True, то окно будет перемещаться поверх других окон
<code>webbrowser.open_new()</code>	
<code>url</code>	URL-адрес для открытия в веб-браузере
<code>webbrowser.open_new_tab()</code>	
<code>url</code>	URL-адрес для открытия в веб-браузере
<code>webbrowser.get()</code>	
<code>using</code>	браузер для использования
<code>webbrowser.register()</code>	
<code>url</code>	URL-адрес для открытия в веб-браузере
<code>constructor</code>	путь к исполняемому браузеру
<code>instance</code>	Экземпляр веб-браузера, возвращаемый из метода <code>webbrowser.get()</code>

Согласно стандартной документации Python, модуль `webbrowser` предоставляет высокоуровневый интерфейс, позволяющий отображать пользователям Web-документы. В данной теме рассказывается и демонстрируется правильное использование модуля `webbrowser`.

61.1. Открытие URL-адреса браузером по умолчанию

Чтобы просто открыть URL, используйте метод `webbrowser.open()`:

```
import webbrowser
webbrowser.open("http://stackoverflow.com")
```

Если в данный момент открыто окно браузера, то метод откроет новую вкладку по указанному URL. Если окно не открыто, метод откроет браузер по умолчанию операционной системы и перейдет на URL, указанный в параметре. Метод `open` поддерживает следующие параметры:

- `url` – URL-адрес для открытия в веб-браузере (строка) [обязательный]
- `new` – при значении параметра 0 открывается в существующей вкладке, 1 – открывается в новом окне, 2 – открывается в новой вкладке (целое число) [по умолчанию 0]
- `autoraise` – если установлено значение True, то окно будет перемещаться поверх окон других приложений (булево значение) [по умолчанию False]

Обратите внимание, что аргументы `new` и `autoraise` редко работают, так как большинство современных браузеров отказываются от этих команд. `Webbrowser` также может попытаться открыть URL в новом окне с помощью метода `open_new`:

```
import webbrowser
webbrowser.open_new("http://stackoverflow.com")
```

Этот метод обычно игнорируется современными браузерами, и URL обычно открывается в новой вкладке. Открытие новой вкладки может быть опробовано модулем с помощью метода `open_new_tab`:

```
import webbrowser
webbrowser.open_new_tab("http://stackoverflow.com")
```

61.2. Открытие URL-адреса с помощью различных браузеров

Модуль `webbrowser` также поддерживает различные браузеры с помощью методов `register()` и `get()`. Метод `get` используется для создания контроллера браузера по заданному пути к исполняемому файлу, а метод `register` – для прикрепления этих исполняемых файлов к заданным типам браузеров для последующего использования, обычно при использовании нескольких типов браузеров.

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
ff.open("http://stackoverflow.com/")
```

Registering a browser type:

```
import webbrowser
ff_path = webbrowser.get("C:/Program Files/Mozilla Firefox/firefox.exe")
ff = webbrowser.get(ff_path)
webbrowser.register('firefox', None, ff)
# Теперь для ссылки на использование Firefox в будущем можно использовать следующее
webbrowser.get('firefox').open("https://stackoverflow.com/")
```

Глава 62. tkinter

Tkinter – самая популярная в Python библиотека графического интерфейса пользователя (GUI). В этой главе рассказывается о правильном использовании этой библиотеки и ее возможностях.

62.1. Менеджеры геометрии

В Tkinter существует три механизма управления геометрией: `place`, `pack` и `grid`. Менеджер `place` использует абсолютные пиксельные координаты. Менеджер `pack` размещает виджеты в одну из 4 сторон. Новые виджеты размещаются рядом с существующими. Менеджер `grid` размещает виджеты в сетке, похожей на динамически изменяемую электронную таблицу.

place

Наиболее распространенными именованными аргументами для `widget.place` являются следующие:

- `x`, абсолютная x-координата виджета
- `y`, абсолютная y-координата виджета
- `height`, абсолютная высота виджета
- `width`, абсолютная ширина виджета

Пример кода с использованием `place`:

```
class PlaceExample(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
```

```

self.grid()
top_text=Label(master,text="Это находится сверху в начале координат")
#top_text.pack()
top_text.place(x=0,y=0,height=50,width=200)
bottom_right_text=Label(master,text="Это находится в позиции 200,400")
#top_text.pack()
bottom_right_text.place(x=200,y=400,height=50,width=200)
# Spawn Window
if __name__=="__main__":
    root=Tk()
    place_frame=PlaceExample(root)
    place_frame.mainloop()

```

Pack

widget.pack может принимать следующие именованные аргументы:

- expand, заполнять ли пространство, оставленное родительским объектом
- fill, заполнять ли все пространство (NONE (по умолчанию), X, Y, или BOTH)
- side, относительно которой будет производиться упаковка (TOP (по умолчанию), BOTTOM, LEFT, или RIGHT)

Grid

Наиболее часто используемые именованные аргументы widget.grid следующие:

- row, строка виджета (по умолчанию наименьшая незанятая)
- rowspan, количество строк, которые охватывает виджет (по умолчанию 1)
- column, столбец виджета (по умолчанию 0)
- columnspan, количество колонок, которые охватывает виджет (по умолчанию 1)
- sticky, куда поместить виджет, если ячейка сетки больше него (комбинация из N,NE,E,SE,S,SW,W,NW)

Строки и столбцы имеют нулевой индекс. Строки увеличиваются вниз, а столбцы – вправо. Пример кода с использованием grid:

```
from tkinter import *
```

```

class GridExample(Frame):
    def __init__(self,master):
        Frame.__init__(self,master)
        self.grid()
        top_text=Label(self,text="Этот текст появляется слева вверху")
        top_text.grid() # Положение по умолчанию 0, 0
        bottom_text=Label(self,text="Этот текст появляется слева внизу")
        bottom_text.grid() # Положение по умолчанию 1, 0
        right_text=Label(self,text="Этот текст появляется справа и охватывает обе строки",
            wraplength=100)

        # Положение 0,1
        # Rowspan означает, что фактическое положение [0-1],1
        right_text.grid(row=0,column=1,rowspan=2)

# Spawn Window
if __name__=="__main__":
    root=Tk()
    grid_frame=GridExample(root)
    grid_frame.mainloop()

```

Никогда не используйте pack и grid в одном фрейме! Это приведет приложение к взаимоблокировке!

62.2. Минимальное приложение tkinter

tkinter – это набор инструментов для создания графического интерфейса, который представляет собой обертку вокруг библиотеки графического интерфейса Tk/Tcl GUI и входит в состав Python. Следующий код создает новое окно с помощью tkinter и помещает некоторый текст в тело окна.

```
import tkinter as tk

# Окно графического интерфейса является подклассом базового объекта tkinter Frame
class HelloWorldFrame(tk.Frame):
    def __init__(self, master):
        # Вызов конструктора суперкласса
        tk.Frame.__init__(self, master)
        # Помещение фрейма в главное окно
        self.grid()
        # Создать текстовое поле с текстом "Hello World"
        hello = tk.Label(self, text="Hello World! This label can hold strings!")
        # Поместить текстовое поле во фрейм
        hello.grid(row=0, column=0)

# Spawn window
if __name__ == "__main__":
    # Создать объект главного окна
    root = tk.Tk()
    # Установить заголовок окна
    root.title("Hello World!")
    # Инстанцировать объект HelloWorldFrame
    hello_frame = HelloWorldFrame(root)
    # Запустить графический интерфейс
    hello_frame.mainloop()
```

Глава 63. Модуль pyautogui

pyautogui – это модуль, предназначенный для управления мышью и клавиатурой. В основном этот модуль используется для автоматизации задач щелчка мыши и нажатия клавиш клавиатуры. Для мыши координаты экрана (0,0) начинаются с левого верхнего угла. Если вы не можете управлять мышью, то быстро переместите курсор мыши в левый верхний угол, это перехватит управление мышью и клавиатурой у Python и вернет его вам.

63.1. Функции мыши

Вот некоторые из полезных функций мыши для управления ею.

```
size() #указывает размер экрана
position() # возврат текущего положения мыши
moveTo(200,0,duration=1.5) #переместить курсор в позицию (200,0) с задержкой в 1,5 секунды
moveRel() #переместить курсор относительно текущей позиции
click(337,46) #щелкнет мышью на упомянутой позиции
dragRel() #перетаски мышь относительно позиции
pyautogui.displayMousePosition() # выдает текущее положение мыши, но должно выполняться в терминале
```


63.2. Функции клавиатуры

Вот некоторые из полезных функций клавиатуры для автоматизации нажатия клавиш.

```
typewrite("") #это выведет строку на экране, на котором сфокусировано текущее окно
typewrite(['a','b','left','left','X','Y'])
pyautogui.KEYBOARD_KEYS #получить список всех клавиш_клавиатуры
pyautogui.hotkey('ctrl','o') #для комбинации клавиш для ввода
```

63.3. Распознавание скриншотов и изображений

Эти функции помогут вам сделать снимок экрана, а также сопоставить изображение с частью экрана.

```
.screenshot('c:\\path') #получить скриншот
.locateOnScreen('c:\\path') #поиск этого изображения на экране и получение координат
locateCenterOnScreen('c:\\path') #получить координаты для изображения на экране
```

Глава 64. Индексация и нарезка

Параметр	Подробности
obj	Объект, из которого вы хотите извлечь “подобъект”
start	Индекс obj, с которого должен начинаться подобъект (следует помнить, что Python имеет нулевую индексацию, то есть первый элемент obj имеет индекс 0). Если это значение опущено, то по умолчанию оно равно 0.
stop	(неполный) индекс obj, на котором должен заканчиваться подобъект. Если это значение опущено, то по умолчанию оно равно len(obj).
step	Позволяет выбрать только каждый элемент с шагом step. Если опущено, по умолчанию принимается значение 1.

64.1. Базовая нарезка

Для любого итерируемого объекта (например строки, списка и т. д.) Python позволяет нарезать и вернуть подстроку или подсписок его данных. Формат для нарезки:

```
iterable_name[start:stop:step]
```

где

- start – первый индекс фрагмента. По умолчанию равен 0 (индекс первого элемента)
- stop – последний индекс фрагмента. По умолчанию равен len(iterable)
- step – размер шага (лучше пояснен на примерах ниже)

Примеры:

```
a = "abcdef"
a # "abcdef"
# То же, что и a[:] или a[:, поскольку для всех трех индексов используются
# значения по умолчанию
a[-1] # "f"
a[:] # "abcdef"
a[:,] # "abcdef"
```

```
a[3:] # "def" (от индекса 3 до конца (по умолчанию равен размеру итерируемого объекта))
a[4] # "abcd" (от начала (по умолчанию 0) до позиции 4 (исключая))
a[2:4] # "cd" (от позиции 2 до позиции 4 (исключая))
```

Кроме того, может быть использован любой из вышеперечисленных способов с заданным размером шага:

```
a[::2] # "ace" (каждый 2-й элемент)
a[1:4:2] # "bd" (от индекса 1 до индекса 4 (исключая), каждый 2-й элемент)
```

Индексы могут быть отрицательными, в этом случае они вычисляются с конца последовательности:

```
a[-1] # "abcde" (от индекса 0 (по умолчанию) до 2-го последнего элемента
# (последний элемент -1))
a[-2] # "abcd" (от индекса 0 (по умолчанию) до 3-го последнего элемента
# (последний элемент -2))
a[-1:] # "e" (от последнего элемента до конца (по умолчанию len()))
```

Размер шага может быть и отрицательным, в этом случае нарезка будет выполнять итерации по списку в обратном порядке:

```
a[3:1:-1] # "dc" (от индекса 2 до None (по умолчанию), в обратном порядке)
```

Эта конструкция полезна для инвертирования итерируемого объекта

```
a[::-1] # "fedcba" (от последнего элемента (по умолчанию len()-1) к первому,
# в обратном порядке(-1))
```

Обратите внимание, что для отрицательных шагов `end_index` по умолчанию равен `None`.

```
a[5:None:-1] # "fedcba" (это эквивалентно a[::-1])
a[5:0:-1] # "fedcb" (от последнего элемента (индекс 5) до второго элемента (индекс 1))
```

64.2. Реверсирование объекта

С помощью срезов можно очень просто перевернуть строку, список или кортеж (или, по сути, любой объект коллекции, реализующий срез с параметром `step`). Здесь приведен пример инвертирования строки, хотя это в равной степени относится и к другим типам, перечисленным выше:

```
s = 'reverse me!'
s[::-1] # '!em esrever'
```

Кратко рассмотрим синтаксис. `[::-1]` означает, что срез должен быть от начала до конца строки (так как `start` и `end` опущены), а шаг `-1` означает, что он должен двигаться по строке в обратном направлении.

64.3. Назначение среза

Python позволяет назначать новые срезы для замены старых срезов списка за одну операцию. Это означает, что если у вас есть список, то вы можете заменить несколько его членов в одном назначении:

```
lst = [1, 2, 3]
lst[1:3] = [4, 5]
print(lst) # Результат: [1, 4, 5]
```

Назначение не должно совпадать по размеру, поэтому, если вы хотите заменить старый срез новым, отличающимся по размеру, вы можете это сделать:

```
lst = [1, 2, 3, 4, 5]
lst[1:4] = [6]
print(lst) # Результат: [1, 6, 5]
```

Также можно использовать известный синтаксис нарезки для выполнения таких действий, как замена всего списка:

```
lst = [1, 2, 3]
lst[:] = [4, 5, 6]
print(lst) # Результат: [4, 5, 6]
```

или только двух последних элементов списка:

```
lst = [1, 2, 3]
lst[-2:] = [4, 5, 6]
print(lst) # Результат: [1, 4, 5, 6]
```

64.4. Создание неглубокой копии массива

Быстрый способ создания копии массива (в отличие от присвоения переменной другой ссылки на исходный массив) заключается в следующем:

```
arr[:]
```

Рассмотрим синтаксис. `[:]` означает, что значения `start`, `end` и `slice` опущены. По умолчанию они равны 0, `len(arr)` и 1 соответственно, то есть запрашиваемый нами подмассив будет содержать все элементы `arr` с начала и до самого конца. На практике это выглядит так:

```
arr = ['a', 'b', 'c']
copy = arr[:]
arr.append('d')
print(arr) # ['a', 'b', 'c', 'd']
print(copy) # ['a', 'b', 'c']
```

Как видно, `arr.append('d')` добавила `d` в `arr`, но копия (`copy`) осталась неизменной! Заметим, что при этом выполняется *неглубокое* копирование, идентичное функции `arr.copy()`.

64.5. Индексация пользовательских классов:

`__getitem__`, `__setitem__` и `__delitem__`

```
class MultiIndexingList:
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return repr(self.value)

    def __getitem__(self, item):
        if isinstance(item, (int, slice)):
            return self.__class__(self.value[item])
        return [self.value[i] for i in item]

    def __setitem__(self, item, value):
        if isinstance(item, int):
            self.value[item] = value
        elif isinstance(item, slice):
            raise ValueError('Cannot interpret slice with multiindexing')
        else:
            for i in item:
                if isinstance(i, slice):
                    raise ValueError('Cannot interpret slice with multiindexing')
                self.value[i] = value
```

```
def __delitem__(self, item):
    if isinstance(item, int):
        del self.value[item]
    elif isinstance(item, slice):
        del self.value[item]
    else:
        if any(isinstance(elem, slice) for elem in item):
            raise ValueError('Cannot interpret slice with multiindexing')
        item = sorted(item, reverse=True)
        for elem in item:
            del self.value[elem]
```

Это позволяет осуществлять нарезку и индексацию для доступа к элементам:

```
a = MultIndexingList([1,2,3,4,5,6,7,8])
a
# Результат: [1, 2, 3, 4, 5, 6, 7, 8]
a[1,5,2,6,1]
# Результат: [2, 6, 3, 7, 2]
a[4, 1, 5, 2, ::2]
# Результат: [5, 2, [6, 7, 8], 3, [1, 3, 5, 7]]
#          4|1-|---50:--|2-|---:2--- <- указывает, какой элемент пришел из какого индекса
```

При установке и удалении элементов допускается только индексация целых чисел, разделенных *запятыми* (без нарезки):

```
a[4] = 1000
a
# Результат: [1, 2, 3, 4, 1000, 6, 7, 8]
a[2,6,1] = 100
a
# Результат: [1, 100, 100, 4, 1000, 6, 100, 8]
del a[5] a
# Результат: [1, 100, 100, 4, 1000, 100, 8]
del a[4,2,5] a
# Результат: [1, 100, 4, 8]
```

64.6. Базовая индексация

Списки Python являются 0-ориентированными, т. е. доступ к первому элементу списка осуществляется по индексу 0:

```
arr = ['a', 'b', 'c', 'd'].
print(arr[0])
>> 'a'
```

По индексу 1 можно получить доступ ко второму элементу списка, по индексу 2 – к третьему и т. д.:

```
print(arr[1])
>> 'b'
print(arr[2])
>> 'c'
```

Для доступа к элементам из конца списка можно также использовать отрицательные индексы. Например, индекс -1 позволяет получить последний элемент списка, а индекс -2 – предпоследний элемент списка:

```
print(arr[-1])
>> 'd'
print(arr[-2])
>> 'c'
```

Если попытаться обратиться к индексу, которого нет в списке, то будет выдана ошибка `IndexError`:

```
print arr[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>.
IndexError: list index out of range (индекс списка выходит за пределы диапазона)
```

Глава 65. Построение графиков с помощью Matplotlib

Matplotlib (<https://matplotlib.org/>) – это библиотека для двумерного черчения, основанная на NumPy.

65.1. Графики с общей осью x, но разными осями y: использование `twinx()`

В данном примере мы построим на одном графике синусоиду и гиперболическую синусоиду с общей осью x и разными осями y. Для этого используется команда `twinx()`.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.sinh(x)

# отделить объект figure и объект axes
# от объекта plotting
fig, ax1 = plt.subplots()

# Дублирование осей с другой осью y
# и той же осью x
ax2 = ax1.twinx() # ax2 и ax1 будут иметь общую ось x и разные оси y

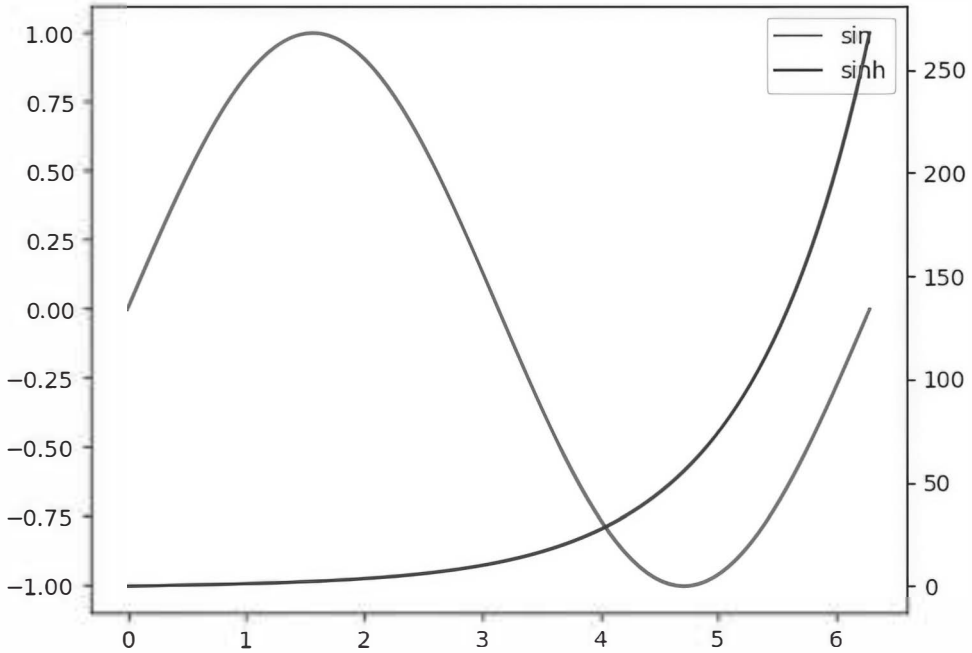
# построить кривые на осях 1 и 2
curve1, = ax1.plot(x, y, label="sin", color='r') curve2, = ax2.plot(x, z, label="sinh", color='b')

# Составить список кривых для доступа к параметрам в кривых
кривые = [кривая1, кривая2]

# добавить легенду по осям 1 или 2 объекта.
# обычно достаточно одной команды
# ax1.legend() # не будет отображаться легенда ax2
# ax2.legend() # не будет отображаться легенда ax1
ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # также допустимо

# Глобальные свойства фигуры
plt.title("График синусоиды и гиперболической синусоиды")
plt.show()
```

График синусоиды и гиперболической синусоиды



65.2. Графики с общей осью y, но разными осями x: использование функции `twiny()`

В данном примере с помощью метода `twiny()` демонстрируется график с кривыми, имеющими общую ось y, но разные оси x. Кроме того, на график добавлены некоторые дополнительные функции, такие как заголовок, легенда, метки, сетки, отметки на осях и цвета.

```
import numpy as np
import matplotlib.pyplot as plt

y = np.linspace(0, 2.0*np.pi, 101)
x1 = np.sin(y)
x2 = np.sinh(y)

# значения для создания отметок на осях x и y
ynumbers = np.linspace(0, 7, 15)
xnumbers1 = np.linspace(-1, 1, 11)
xnumbers2 = np.linspace(0, 300, 7)

# отделить объект figure и объект axes
# от объекта plotting
fig, ax1 = plt.subplots()

# Дублирование осей с другой осью x
# и той же осью y
ax2 = ax1.twinx() # ax2 и ax1 будут иметь общую ось y и разные оси x

# построить кривые на осях 1 и 2
curve1, = ax1.plot(x1, y, label="sin", color='r')
curve2, = ax2.plot(x2, y, label="sinh", color='b')
```

```
# Составить список кривых для доступа к параметрам в кривых
curves = [curve1, curve2]

# добавить легенду по осям 1 или 2 объекта
# обычно достаточно одной команды
# ax1.legend() # не будет отображаться легенда ax2
# ax2.legend() # не будет отображаться легенда ax1
# ax1.legend(curves, [curve.get_label() for curve in curves])
# ax2.legend(curves, [curve.get_label() for curve in curves]) # также допустимо

# метки оси x через оси
ax1.set_xlabel("Величина", color=curve1.get_color())
ax2.set_xlabel("Величина", color=curve2.get_color())

# метка оси y по осям
ax1.set_ylabel("Угол/Значение", color=curve1.get_color())
# ax2.set_ylabel("Величина", color=curve2.get_color()) # не работает
# в ax2 нет управления свойствами по оси y

# метки на оси y – сделать их также цветными
ax1.tick_params(axis='y', colors=curve1.get_color())
# ax2.tick_params(axis='y', colors=curve2.get_color()) # не работает
# в ax2 нет управления свойствами по оси y

# метки на оси x через оси
ax1.tick_params(axis='x', colors=curve1.get_color())
ax2.tick_params(axis='x', colors=curve2.get_color())

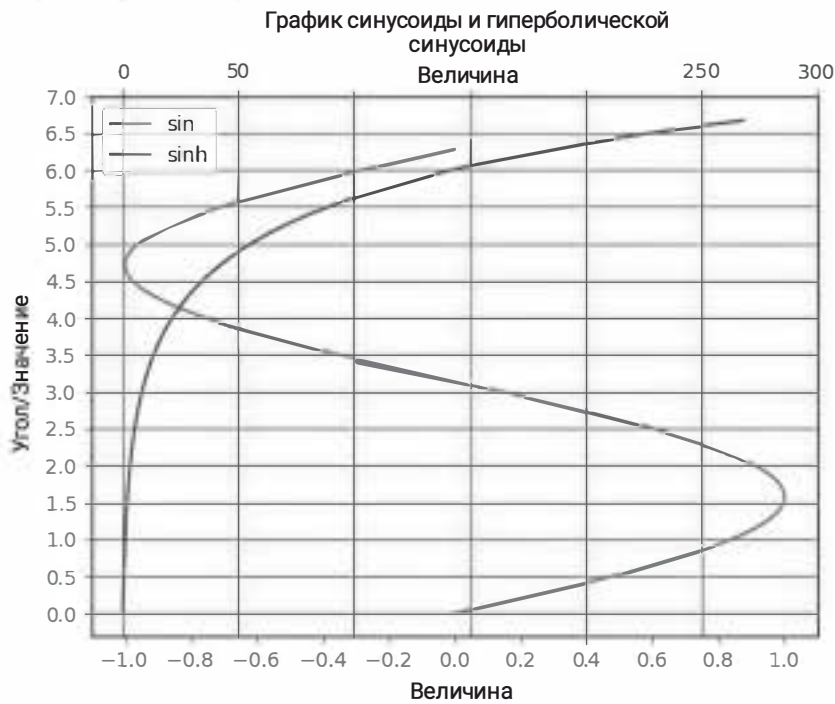
# установить метки по оси x
ax1.set_xticks(xnumbers1)
ax2.set_xticks(xnumbers2)

# установить метки по оси y
ax1.set_yticks(ynumbers)
# ax2.set_yticks(ynumbers) # также работает

# Сетки по осям 1 # используйте это, если ось 1 используется для
# определения свойств общей оси x
# ax1.grid(color=curve1.get_color())

# Для создания сеток с использованием осей 2
ax1.grid(color=curve2.get_color())
ax2.grid(color=curve2.get_color())
ax1.xaxis.grid(False)

# Глобальные свойства фигуры
plt.title("График синусоиды и гиперболической синусоиды")
plt.show()
```



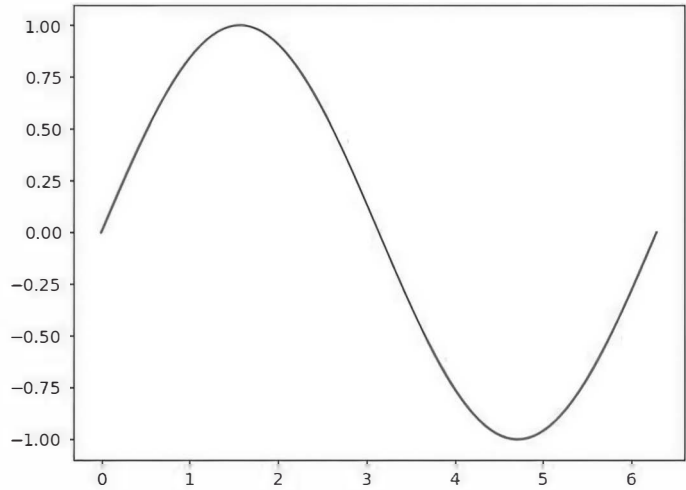
65.3. Простой график в Matplotlib

Данный пример иллюстрирует создание простой синусоиды с помощью Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt

# угол, изменяющийся в пределах от 0 до 2*pi
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x) # функция синуса

plt.plot(x, y)
plt.show()
```



65.4. Добавление дополнительных функций к простому графику: названия осей, заголовков, метки осей, сетка и легенда

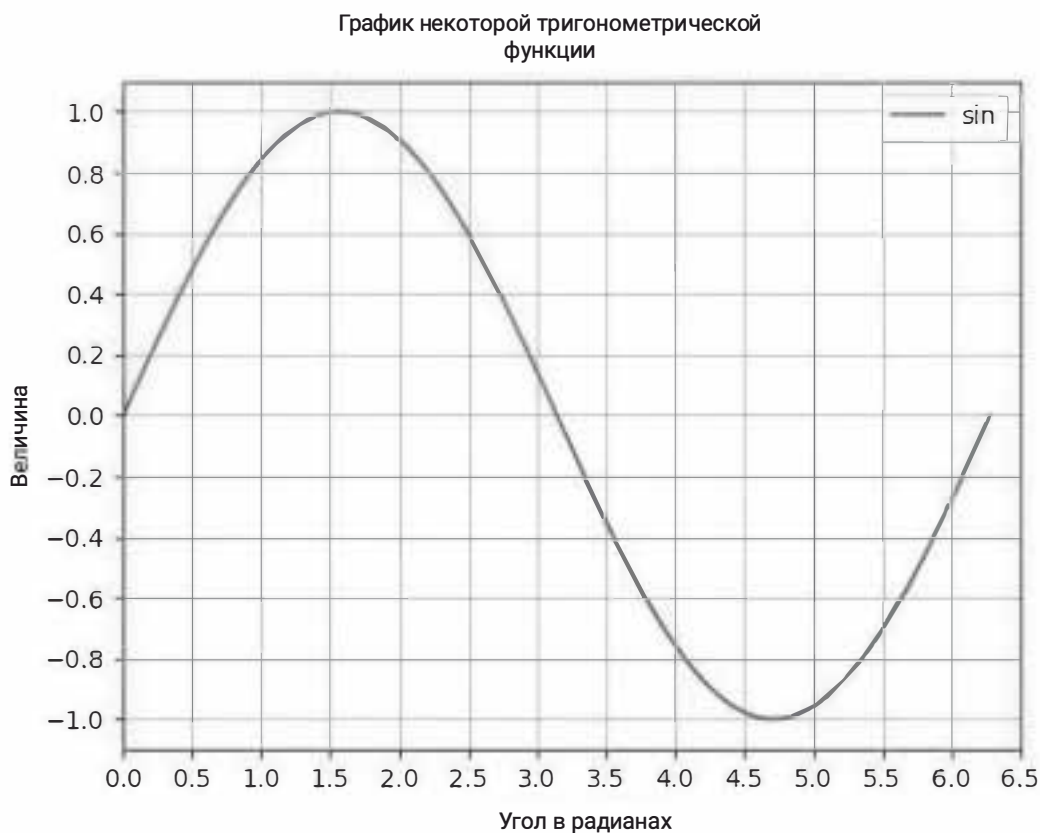
В этом примере мы берем график синусоиды и добавляем к нему заголовок, названия осей, метки осей, сетку и легенду.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)

# значения для создания меток по осям x и y
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)

plt.plot(x, y, color='r', label='sin') # r - красный цвет
plt.xlabel("Угол в радианах")
plt.ylabel("Величина")
plt.title("График некоторой тригонометрической функции")
plt.xticks(xnumbers)
plt.yticks(ynumbers) plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



65.5. Создание нескольких кривых на одном графике путем наложения, аналогично MATLAB

В данном примере синусоида и косинусоида построены на одном графике путем наложения.

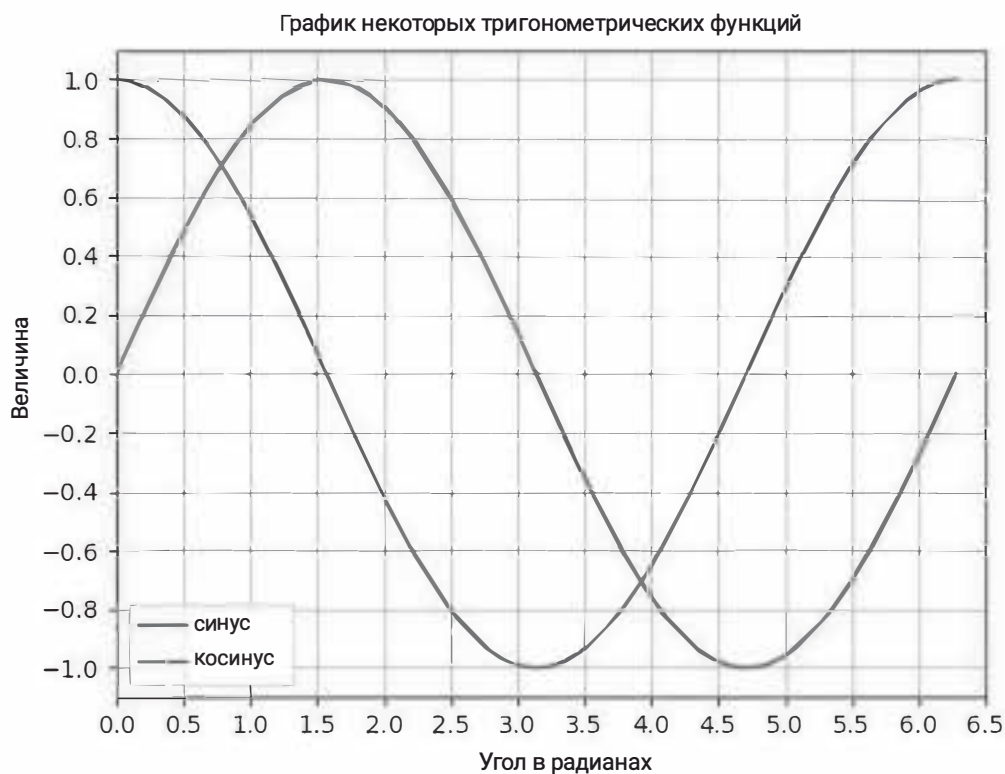
Хорошо подходит для графиков с близкими значениями x , y
Использование одной команды `plot` и `legend`

```
import numpy as np  
import matplotlib.pyplot as plt
```

```
x = np.linspace(0, 2.0*np.pi, 101)  
y = np.sin(x)  
z = np.cos(x)
```

```
# значения для создания меток по осям  $x$  и  $y$   
xnumbers = np.linspace(0, 7, 15)  
ynumbers = np.linspace(-1, 1, 11)
```

```
plt.plot(x, y, 'r', x, z, 'g') # r, g – красный, зеленый цвет  
plt.xlabel("Угол в радианах")  
plt.ylabel("Величина")  
plt.title("График некоторых тригонометрических функций")  
plt.xticks(xnumbers)  
plt.yticks(ynumbers)  
plt.legend(['синус', 'косинус'])  
plt.grid()  
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]  
plt.show()
```



65.6. Создание нескольких кривых на одном графике с помощью наложения с использованием отдельных команд plot

В этом примере синусоида и косинусоида строятся на одном и том же графике с помощью отдельных команд plot. Это более “питонично”.

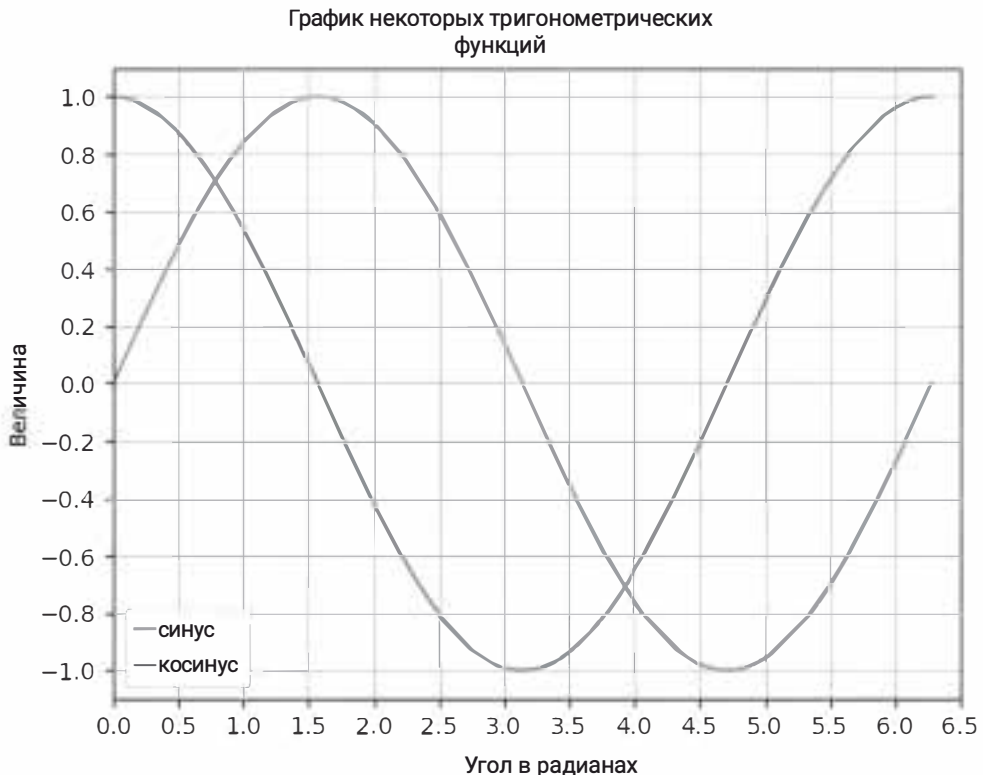
```
# Хорошо подходит для графиков с близкими значениями x, y
# Использование нескольких команд построения графиков plot
# Лучше и предпочтительнее, чем предыдущий пример
```

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = np.linspace(0, 2.0*np.pi, 101)
y = np.sin(x)
z = np.cos(x)
```

```
# значения для создания меток по осям x и y
xnumbers = np.linspace(0, 7, 15)
ynumbers = np.linspace(-1, 1, 11)
```

```
plt.plot(x, y, color='r', label='sin') # r – красный цвет
plt.plot(x, z, color='g', label='cos') # g – зеленый цвет
plt.xlabel("Угол в радианах")
plt.ylabel("Величина")
plt.title("График некоторых тригонометрических функций")
plt.xticks(xnumbers)
plt.yticks(ynumbers)
plt.legend()
plt.grid()
plt.axis([0, 6.5, -1.1, 1.1]) # [xstart, xend, ystart, yend]
plt.show()
```



Глава 66. graph-tool

Инструменты Python можно использовать для создания графов.

66.1. PyDotPlus

PyDotPlus – это улучшенная версия старого проекта pydot, обеспечивающая Python-интерфейс для языка Dot в Graphviz.

Установка

Для получения последней стабильной версии:

```
pip install pydotplus
```

Для версии разработки:

```
pip install https://github.com/carlos-jenkins/pydotplus/archive/master.zip
```

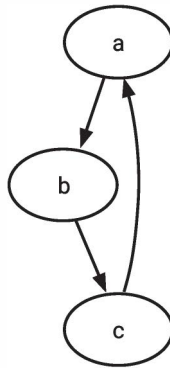
Загрузите граф, определенный в файле DOT.

- Предполагается, что файл имеет формат DOT. Он будет загружен, разобран (парсирован), затем будет возвращен класс Dot, представляющий граф. Например, простой demo.dot:

```
digraph demo1{ a -> b -> c; c -> a; }
```

```
import pydotplus
graph_a = pydotplus.graph_from_dot_file('demo.dot')
graph_a.write_svg('test.svg') # создать граф в формате svg
```

Вы получите файл в формате *.svg (Scalable Vector Graphics) следующего вида:



66.2. PyGraphviz

PyGraphviz можно получить из Python Package Index по адресу <http://pypi.python.org/pypi/pygraphviz> или установить его с помощью следующей команды:

```
pip install pygraphviz
```

Будет сделана попытка найти и установить подходящую версию, соответствующую вашей операционной системе и версии Python.

Установить версию для разработки (на github.com) можно с помощью:

```
pip install git://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

Загрузите граф, определенный в файле DOT.

- Предполагается, что файл имеет формат DOT. Он будет загружен, разобран (парсирован), затем будет возвращен класс Dot, представляющий граф. Например, простой demo.dot:

```
digraph demo1{ a -> b -> c; c -> a; }
```

```
import pygraphviz as pgv
G = pgv.AGraph("demo.dot")
G.draw('test', format='svg', prog='dot')
```

Вы получите файл в формате *.svg (Scalable Vector Graphics), такой же, как и в предыдущей главе.

Глава 67. Генераторы

Генераторы – это “ленивые” итераторы, создаваемые функциями-генераторами (с использованием yield) или выражениями-генераторами (с использованием (an_expression for x in an_iterator)).

67.1. Введение

Выражения-генераторы аналогичны генераторам (comprehensions) списков, словарей и множеств, но заключены в круглые скобки. Скобки не обязательно должны присутствовать, если они используются в качестве единственного аргумента вызова функции.

```
expression = (x**2 for x in range(10))
```

Получено 10 первых совершенных квадратов, включая 0 (в котором $x = 0$).

Функции-генераторы похожи на обычные функции, за исключением того, что в их теле имеются один или несколько операторов yield. Такие функции не могут возвращать (return) никаких значений (однако пустые возвраты допускаются, если необходимо остановить генератор раньше времени).

```
def function():
    for x in range(10):
        yield x**2
```

Эта генераторная функция эквивалентна предыдущему генераторному выражению, она выводит то же самое.

Примечание: все выражения-генераторы имеют свои *эквивалентные* функции, но не наоборот.

Генераторное выражение может быть использовано без скобок, если в противном случае обе скобки повторялись бы:

```
sum(i for i in range(10) if i % 2 == 0) #Результат: 20
any(x = 0 for x in foo)                #Результат: True или False в зависимости от foo
type(a > b for a in foo if a % 2 == 1) #Результат: <class 'generator'>
```

Вместо:

```
sum((i for i in range(10) if i % 2 == 0))
any((x = 0 for x in foo))
type((a > b for a in foo if a % 2 == 1))
```

Но не:

```
fooFunction(i for i in range(10) if i % 2 == 0,foo,bar)
return x = 0 for x in foo
barFunction(baz, a > b for a in foo if a % 2 == 1)
```

При вызове функции-генератора создается **объект-генератор**, над которым в дальнейшем можно выполнять итерации. В отличие от других типов итераторов, объекты-генераторы могут быть обойдены только один раз.

```
g1 = function()
print(g1) # Out: <generator object function at 0x1012e1888>
```

Обратите внимание, что тело генератора выполняется **не сразу**: при вызове функции `function()` в приведенном примере она сразу возвращает объект генератора, не выполняя даже первого оператора `print`. Это позволяет генераторам занимать меньше памяти, чем функции, возвращающие список, соответственно можно создавать генераторы, выдающие бесконечно длинные последовательности. По этой причине генераторы часто используются в науке о данных и других контекстах, связанных с большими объемами данных. Еще одним преимуществом является то, что другой код может сразу использовать значения, выданные генератором, не дожидаясь получения полной последовательности.

Однако если необходимо использовать значения, полученные генератором, более одного раза, и если их генерация обходится дороже хранения, то, возможно, лучше хранить полученные значения в виде списка, чем заново генерировать последовательность. Более подробная информация приведена ниже в подтеме “Сброс генератора”.

Обычно объект-генератор используется в цикле или в любой функции, требующей итерируемого объекта:

```
for x in g1:
    print("Получено", x)

# Output:
# Получено 0
# Получено 1
# Получено 4
# Получено 9
# Получено 16
# Получено 25
# Получено 36
# Получено 49
# Получено 64
# Получено 81
arr1 = list(g1)
# arr1 = [], так как в цикле выше уже были использованы все значения
g2 = function()
arr2 = list(g2) # arr2 = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Поскольку объекты-генераторы являются итераторами, их можно перебирать вручную с помощью функции `next()`. При этом при каждом последующем вызове будут возвращаться поочередно все полученные значения.

Кстати, каждый раз, когда вы вызываете `next()` на генераторе, Python выполняет операторы в теле функции генератора, пока не дойдет до следующего оператора `yield`. В этот момент он возвращает аргумент команды `yield` и запоминает точку, в которой это произошло. Повторный вызов `next()` возобновит выполнение с этой точки и будет продолжаться до следующего оператора `yield`.

Если Python доходит до конца генераторной функции, не встречая больше `yield`, возникает исключение `StopIteration` (это нормально, все итераторы ведут себя таким образом).

```
g3 = function()
a = next(g3) # a становится 0
b = next(g3) # b становится 1
c = next(g3) # c становится 2

j = next(g3) # Возникает StopIteration, j остается неопределенной
```

Отметим, что в Python 2 объекты генераторов использовали метод `.next()`, который можно было использовать для ручного итерационного перебора получаемых значений. В Python 3 этот метод был заменен стандартным для всех итераторов `__next__()`.

Сброс генератора

Помните, что перебирать объекты, созданные генератором, можно *только один раз*. Если вы уже перебирали объекты в скрипте (сценарии), то любая дальнейшая попытка сделать это приведет к результату None.

Если необходимо использовать объекты, созданные генератором, более одного раза, можно либо определить генераторную функцию повторно и использовать ее во второй раз, или при первом использовании можно сохранить вывод функции-генератора в списке. Повторное создание функции-генератора будет хорошим вариантом, если вы имеете дело с большими объемами данных, и хранение списка всех элементов данных займет много места на диске. И наоборот, если первоначальная генерация элементов требует больших затрат, лучше хранить сгенерированные элементы в виде списка, чтобы можно было использовать их повторно.

67.2. Бесконечные последовательности

Генераторы можно использовать для представления бесконечных последовательностей:

```
def integers_starting_from(n):
    while True:
        yield n
        n += 1
```

```
natural_numbers = integers_starting_from(1)
```

Бесконечная последовательность чисел, как приведенная выше, также может быть сгенерирована с помощью `itertools.count`. Приведенный выше код можно переписать следующим образом:

```
natural_numbers = itertools.count(1)
```

Для создания новых генераторов можно использовать генераторные концепции (generator comprehensions) на бесконечных генераторах:

```
multiples_of_two = (x * 2 for x in natural_numbers)
multiples_of_three = (x for x in natural_numbers if x % 3 == 0)
```

Следует помнить, что передача бесконечного генератора в любую функцию, которая попытается использовать генератор целиком, приведет к плачевным последствиям:

```
list(multiples_of_two) # никогда не завершится или вызовет ошибку, специфичную для ОС
```

Вместо этого используйте генераторы списков или наборов (list/set comprehensions), указывая `range` (или `xrange` для Python < 3.0):

```
first_five_multiples_of_three = [next(multiples_of_three) for _ in range(5)]
# [3, 6, 9, 12, 15]
```

или используйте `itertools.islice()` для разбиения итератора на подмножества:

```
from itertools import islice
multiples_of_four = (x * 4 for x in integers_starting_from(1))
first_five_multiples_of_four = list(islice(multiples_of_four, 5))
# [4, 8, 12, 16, 20]
```

Обратите внимание, что исходный генератор тоже обновляется, как и все остальные генераторы, исходящие из того же “корня”:

```
next(natural_numbers) # дает 16
next(multiples_of_two) # дает 34
next(multiples_of_four) # дает 24
```

Итерация бесконечной последовательности также может быть выполнена с помощью цикла `for`. Обязательно включите в него условный оператор `break`, чтобы цикл в конце концов завершился:

```
for idx, number in enumerate(multiples_of_two):
    print(number)
```

```
if idx == 9:
    break # остановиться после выполнения первых 10 умножений на два
```

Классический пример – числа Фибоначчи

```
import itertools
```

```
def fibonacci():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a + b
```

```
first_ten_fibs = list(itertools.islice(fibonacci(), 10))
# [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
def nth_fib(n):
    return next(itertools.islice(fibonacci(), n - 1, n))
```

```
ninety_ninth_fib = nth_fib(99) # 354224848179261915075
```

67.3. Отправка объектов генератору

Помимо получения значений от генератора можно *отправить* объект генератору с помощью метода `send()`.

```
def accumulator():
    total = 0
    value = None
    while True:
        # получить отправленное значение
        value = yield total
        if value is None: break
        # агрегирование значений
        total += value
```

```
generator = accumulator()
```

```
# продвигаться до первого "yield"
next(generator) # 0
```

```
# с этого момента генератор агрегирует значения
generator.send(1) # 1
generator.send(10) # 11
generator.send(100) # 111
# ...
```

```
# Вызов next(generator) эквивалентен вызову generator.send(None)
next(generator) # StopIteration
```

Здесь происходит следующее:

- Когда вы сначала вызываете `next(generator)`, программа выполняется до первого оператора `yield` и возвращает значение `total` в этой точке, которое равно 0. Выполнение генератора приостанавливается в этой точке.
- Когда вы вызываете `generator.send(x)`, интерпретатор принимает аргумент `x` и делает его возвращаемым значением последнего оператора `yield`, которому присваивается `value`. Затем генератор выполняется как обычно, до тех пор, пока он не выдаст следующее значение.
- Когда вы наконец вызываете `next(generator)`, программа рассматривает это так, как будто вы отправляете `None` в генератор. В этом примере используется `None` как специальное значение, чтобы указать генератору остановиться.

67.4. Предоставление всех значений из другого итерируемого объекта

Версия Python 3.x ≥ 3.3

Используйте `yield from`, если вы хотите получить все значения из другого итерируемого объекта:

```
def foob(x):
    yield from range(x * 2)
    yield from range(2)
```

```
list(foob(5)) # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1]
```

Это работает и с генераторами.

```
def fibto(n):
    a, b = 1, 1
    while True:
        if a >= n: break
        yield a
        a, b = b, a + b
```

```
def usefib():
    yield from fibto(10)
    yield from fibto(20)
```

```
list(usefib()) # [1, 1, 2, 3, 5, 8, 1, 1, 2, 3, 5, 8, 13]
```

67.5. Итерация

Объект генератора поддерживает *протокол итератора*. То есть он предоставляет метод `next()` (`__next__()` в Python 3.x, который используется для пошагового выполнения, а его метод `__iter__` возвращает сам себя. Это означает, что генератор может быть использован в любой языковой конструкции, поддерживающей общие итерируемые объекты.

частичная реализация `xrange()` в Python 2.x

```
def xrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

заикливание

```
for i in xrange(10):
    print(i) # выводит значения 0, 1, ..., 9
```

распаковка

```
a, b, c = xrange(3) # 0, 1, 2
```

построение списка

```
l = list(xrange(10)) # [0, 1, ..., 9]
```

67.6. Функция `next()`

Встроенная функция `next()` представляет собой удобную обертку, которая может быть использована для получения значения из любого итератора (в том числе и генератора итераторов) и для предоставления значения по умолчанию в случае, если итератор исчерпан.

```
def nums():
    yield 1
```

```

yield 2
yield 3
generator = nums()

next(generator, None) # 1
next(generator, None) # 2
next(generator, None) # 3
next(generator, None) # None
next(generator, None) # None
# ...

```

Синтаксис следующий: `next(iterator[, default])`. Если итератор заканчивается и было передано значение по умолчанию, то оно возвращается. Если значение по умолчанию не было передано, то выдается сообщение `StopIteration`.

67.7. Корутины (сопрограммы)

Генераторы могут использоваться для реализации корутин (сопрограмм):

```

# создание и продвижение генератора до первого yield
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        next(cr)
        return cr
    return start

# пример корутины
@coroutine
def adder(sum = 0):
    while True:
        x = yield sum
        sum += x

# пример использования
s = adder()
s.send(1) # 1
s.send(2) # 3

```

Корутины широко используются для реализации машин состояний, так как в первую очередь они удобны для создания однометодных процедур, требующих для своей работы учета состояния. Они действуют в существующем состоянии и возвращают значение, полученное по завершении операции.

67.8. Рефакторинг кода построения списков

Предположим, у вас есть сложный код, который создает и возвращает список, начиная с пустого списка и многократно добавляя к нему:

```

def create():
    result = []
    # программная логика здесь...
    result.append(value) # возможно в нескольких местах
    # еще логика...
    return result # возможно в нескольких местах

values = create()

```

Если замена внутренней логики на генератор списка нецелесообразна, можно превратить всю функцию в генератор на месте, а затем собрать результаты:

```

def create_gen():
    # программная логика...

```

```
yield value
# еще программная логика
return # не требуется, если в конце функции, конечно
```

```
values = list(create_gen())
```

Если логика рекурсивна, используйте `yield from`, чтобы включить все значения из рекурсивного вызова в “уплощенный” результат:

```
def preorder_traversal(node):
    yield node.value
    for child in node.children:
        yield from preorder_traversal(child)
```

67.9. Использование оператора `yield` с рекурсией: рекурсивное перечисление всех файлов в каталоге

Сначала импортируйте библиотеки, которые работают с файлами:

```
from os import listdir
from os.path import isfile, join, exists
```

Вспомогательная функция для чтения только файлов из каталога:

```
def get_files(path):
    for file in listdir(path):
        full_path = join(path, file)
        if isfile(full_path):
            if exists(full_path):
                yield full_path
```

Другая вспомогательная функция для получения только подкаталогов:

```
def get_directories(path):
    for directory in listdir(path):
        full_path = join(path, directory)
        if not isfile(full_path):
            if exists(full_path):
                yield full_path
```

Теперь используйте эти функции для рекурсивного получения всех файлов в каталоге и во всех его подкаталогах (с использованием генераторов):

```
def get_files_recursive(directory):
    for file in get_files(directory):
        yield file
    for subdirectory in get_directories(directory):
        for file in get_files_recursive(subdirectory): # здесь рекурсивный вызов
            yield file
```

Эту функцию можно упростить, используя `yield from`:

```
def get_files_recursive(directory):
    yield from get_files(directory)
    for subdirectory in get_directories(directory):
        yield from get_files_recursive(subdirectory)
```

67.10. Генераторные выражения

Можно создавать генераторы итераторов, используя следующий синтаксис:

```
generator = (i * 2 for i in range(3))
next(generator) # 0
next(generator) # 2
next(generator) # 4
next(generator) # вызывает StopIteration
```

Если функции не обязательно передавать список, можно сэкономить на символах (и улучшить читаемость), поместив выражение-генератор внутрь вызова функции. Скобки из вызова функции неявно превращают выражение в генераторное выражение.

```
sum(i ** 2 for i in range(4)) # 0^2 + 1^2 + 2^2 + 3^2 = 0 + 1 + 4 + 9 = 14
```

Кроме того, вы сэкономите память, поскольку вместо загрузки всего списка, по которому выполняется итерация ([0, 1, 2, 3] в этом примере), генератор позволяет Python использовать значения по мере необходимости.

67.11. Использование генератора для получения чисел Фибоначчи

Практический пример использования генератора состоит в том, чтобы перебирать значения бесконечного ряда. Вот пример нахождения первых десяти членов последовательности Фибоначчи:

```
def fib(a=0, b=1):
    """Генератор, выдающий числа Фибоначчи, `a` и `b` – начальные значения"""
    while True:
        yield a
        a, b = b, a + b

f = fib()
print(', '.join(str(next(f)) for _ in range(10)))

0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

67.12. Поиск

Функция `next` полезна даже без итерации. Передача выражения генератора для `next` – это быстрый способ поиска первого вхождения элемента, удовлетворяющего некоторому предикату. Процедурный код наподобие

```
def find_and_transform(sequence, predicate, func):
    for element in sequence:
        if predicate(element):
            return func(element)
    raise ValueError

item = find_and_transform(my_sequence, my_predicate, my_func)
```

можно заменить на:

```
item = next(my_func(x) for x in my_sequence if my_predicate(x))
# StopIteration будет вызвано, если нет совпадений; это исключение может быть
# перехвачено и преобразовано, если это необходимо.
```

Для этой цели желательно создать псевдоним (alias), например `first = next`, или функцию-оболочку для преобразования исключения:

```
def first(generator):
    try:
        return next(generator)
    except StopIteration:
        raise ValueError
```

67.13. Параллельный итерационный перебор генераторов

Для параллельного перебора нескольких генераторов используйте встроенную функцию `zip`:

```
for x, y in zip(a,b):
    print(x,y)
```

Результаты:

```
1 x
2 y
3 z
```

В Python 2 вместо этого следует использовать `itertools.zip`. Здесь мы также видим, что все функции `zip` дают кортежи. Обратите внимание, что `zip` прекратит итерацию, как только в одном из итерируемых объектов закончится количество элементов. Если вы хотите выполнять итерацию столько, сколько длится самый длинный итерируемый объект, используйте `itertools.zip_longest()`.

Глава 68. Функция reduce

Параметр	Подробности
<code>function</code>	функция, используемая для уменьшения итерируемого объекта (должна принимать два аргумента) <i>(только позиционно)</i>
<code>iterable</code>	итерируемый объект, который будет уменьшен <i>(только позиционно)</i>
<code>initializer</code>	начальное значение уменьшения <i>(необязательно, только позиционно)</i>

68.1. Обзор

```
# импорт не требуется
from functools import reduce # ... но его можно загрузить из модуля functools

from functools import reduce # обязательно
reduce уменьшает итерируемый объект путем многократного применения функции
к очередному элементу iterable и суммарному результату на данный момент.
```

```
def add(s1, s2):
    return s1 + s2
```

```
asequence = [1, 2, 3]
```

```
reduce(add, asequence) # эквивалентно: add(add(1,2),3)
# Результат: 6
```

В этом примере мы создали собственную функцию `add`. Однако в Python имеется стандартная эквивалентная функция в модуле `operator`:

```
import operator
reduce(operator.add, asequence)
# Результат: 6
```

В `reduce` также может быть передано начальное значение:

```
reduce(add, asequence, 10)
# Результат: 16
```

68.2. Использование reduce

```
def multiply(s1, s2):
    print('{arg1} * {arg2} = {res}'.format(arg1=s1,
                                          arg2=s2,
                                          res=s1*s2))

    return s1 * s2

asequence = [1, 2, 3]
```

При наличии инициализатора (`initializer`) функция запускается путем применения его к инициализатору и первому итерируемому элементу:

```
cumprod = reduce(multiply, asequence, 5)
# Результат: 5 * 1 = 5
#           5 * 2 = 10
#           10 * 3 = 30
print(cumprod)
# Результат: 30
```

Без параметра `initializer` использование `reduce` начинается с применения функции к первым двум элементам списка:

```
cumprod = reduce(multiply, asequence)
# Результат: 1 * 2 = 2
#           2 * 3 = 6
print(cumprod)
# Результат: 6
```

68.3. Накопительный продукт

```
import operator
reduce(operator.mul, [10, 5, -3])
# Результат: -150
```

68.4. Вариант без “короткого замыкания” функций `any/all`

Функция `reduce` не прервет итерацию до того, как итерируемый объект (`iterable`) будет полностью пройден, поэтому его можно использовать для создания функции `any()` или `all()` без “короткого замыкания”:

```
import operator
# “all” без короткого замыкания
reduce(operator.and_, [False, True, True, True]) # = False

# “any” без короткого замыкания
reduce(operator.or_, [True, False, False, False]) # = True
```

Глава 69. Функция `map`

Параметр	Подробности
<code>function</code>	функция, используемая для отображения (должна принимать столько параметров, сколько итерируемых объектов) (<i>только позиционно</i>)
<code>iterable</code>	функция, которая применяется к каждому элементу итерируемого объекта (<i>только позиционно</i>)
<code>*additional_iterables</code>	как <code>iterable</code> , но применяется сколько угодно раз (<i>необязательно, только позиционно</i>)

69.1. Базовое использование `map`, `itertools.imap` и `future_builtins.map`

Функция `map` является самой простой среди встроенных в Python, используемых для функционального программирования. `map()` применяет указанную функцию к каждому элементу в итерируемом объекте:

```
names = ['Fred', 'Wilma', 'Barney']
```

Версия Python 3.x ≥ 3.0:

```
map(len, names) # map в Python 3.x – это класс; его экземпляры итерируемы
# Результат: <map object at 0x00000198B32E2CF8>
```

Совместимая с Python 3 map включена в модуль future_builtins:

Версия Python 2.x ≥ 2.6:

```
from future_builtins import map    # содержит map(), совместимую с Python 3.x
map(len, names)                   # см. ниже
# Результат: <itertools.imap instance at 0x3eb0a20>
```

В качестве альтернативы в Python 2 можно использовать imap из itertools для получения генератора.

Версия Python 2.x ≥ 2.3

```
map(len, names)                  # map() возвращает список
# Результат: [4, 5, 6]
```

```
from itertools import imap
imap(len, names)                 # itertools.imap() возвращает генератор
# Результат: <itertools.imap at 0x405ea20>
```

Результат может быть явно преобразован в list (список), чтобы убрать различия между Python 2 и 3:

```
list(map(len, names))
# Результат: [4, 5, 6]
```

map() можно заменить эквивалентным *генератором списка* или *выражением генератора*:

```
[len(item) for item in names] # эквивалентно Python 2.x map()
# Результат: [4, 5, 6]
```

```
(len(item) for item in names) # эквивалентно Python 3.x map()
# Результат: <generator object <genexpr> at 0x00000195888D5FC0>
```

69.2. Сопоставление каждого значения в итерируемом объекте

К примеру, можно принять абсолютное значение каждого элемента:

```
list(map(abs, (1, -1, 2, -2, 3, -3))) # вызов `list` в Python 2.x не нужен
# Результат: [1, 1, 2, 2, 3, 3]
```

Анонимная функция также поддерживает отображение (mapping) списка:

```
map(lambda x:x*2, [1, 2, 3, 4, 5])
# Результат: [2, 4, 6, 8, 10]
```

или преобразование десятичных значений в проценты:

```
def to_percent(num):
    return num * 100
```

```
list(map(to_percent, [0.95, 0.75, 1.01, 0.1]))
# Результат: [95.0, 75.0, 101.0, 10.0]
```

или конвертации валют (с учетом обменного курса):

```
from functools import partial
from operator import mul
```

```
rate = 0.9 # фиктивный обменный курс, 1 доллар = 0,9 евро
dollars = {'under_my_bed': 1000,
           'jeans': 45,
           'bank': 5000}
```

```
sum(map(partial(mul, rate), dollars.values()))
# Результат: 5440.5
```

`functools.partial` – удобный способ исправить параметры функций, чтобы их можно было использовать с `map` вместо использования `lambda` или создания настраиваемых функций.

69.3. Отображение (mapping) значений различных итерируемых объектов

Например, вычислим среднее значение каждого *i* элемента из множества итерируемых объектов:

```
def average(*args):
    return float(sum(args)) / len(args) # приведение к float – обязательно только для Python 2.x
```

```
measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117, 91, 102]
measurement3 = [104, 102, 95, 101]
```

```
list(map(average, measurement1, measurement2, measurement3))
# Результат: [102.0, 110.0, 95.0, 100.0]
```

Существуют разные требования, если более чем один итерируемый объект передается в `map` в зависимости от версии Python:

- Функция должна принимать столько параметров, сколько есть итерируемых объектов:

```
def median_of_three(a, b, c):
    return sorted((a, b, c))[1]
```

```
list(map(median_of_three, measurement1, measurement2))
```

`TypeError: median_of_three () missing 1 required positional argument: 'c' (в median_of_three () отсутствует 1 требуемый позиционный аргумент: 'c')`

```
list(map(median_of_three, measurement1, measurement2, measurement3, measurement3))
```

`TypeError: median_of_three() takes 3 positional arguments but 4 were given (median_of_three () принимает 3 позиционных аргумента, но даны 4)`

Версия Python 2.x ≥ 2.0.1

- `map`: отображение повторяется до тех пор, пока один итерируемый объект еще не полностью исчерпан, но принимает значение `None` из полностью исчерпанных итераций:

```
import operator
```

```
measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]
```

```
# Вычислить разность между элементами
list(map(operator.sub, measurement1, measurement2))
```

`TypeError: unsupported operand type(s) for -: 'int' and 'NoneType' (неподдерживаемый тип операндов для -: 'int' и 'NoneType')`

- `itertools.imap` и `future_builtins.map`: отображение останавливается, как только один из итерируемых объектов останавливается:

```
import operator
from itertools import imap

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Вычислить разность между элементами
list(imap(operator.sub, measurement1, measurement2))
# Результат: [-2, -6]
list(imap(operator.sub, measurement2, measurement1))
# Результат: [2, 6]
```

Версия Python 3.x ≥ 3.0.0

- отображение останавливается, как только один из итерируемых объектов останавливается:

```
import operator

measurement1 = [100, 111, 99, 97]
measurement2 = [102, 117]

# Вычислить разность между элементами
list(map(operator.sub, measurement1, measurement2))
# Результат: [-2, -6]
list(map(operator.sub, measurement2, measurement1))
# Результат: [2, 6]
```

69.4. Транспонирование с помощью map: использование “None” в качестве аргумента функции (только для Python 2.x)

```
from itertools import imap
from future_builtins import map as fmap # Разное имя, чтобы подчеркнуть различия
```

```
image = [[1, 2, 3],
         [4, 5, 6],
         [7, 8, 9]]

list(map(None, *image))
# Результат: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(fmap(None, *image))
# Результат: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
list(imap(None, *image))
# Результат: [(1, 4, 7), (2, 5, 8), (3, 6, 9)]

image2 = [[1, 2, 3],
          [4, 5],
          [7, 8, 9]]

list(map(None, *image2))
# Результат: [(1, 4, 7), (2, 5, 8), (3, None, 9)] # Заполнить отсутствующие значения значением None
list(fmap(None, *image2))
# Результат: [(1, 4, 7), (2, 5, 8)] # игнорировать столбцы с отсутствующими значениями
list(imap(None, *image2))
# Результат: [(1, 4, 7), (2, 5, 8)]
```

Версия Python 3.x ≥ 3.0.0:

```
list(map(None, *image))
```

TypeError: 'NoneType' object is not callable (объект "NoneType" не может быть вызван)

Но есть обходное решение для получения аналогичных результатов:

```
def conv_to_list(*args):
    return list(args)
```

```
list(map(conv_to_list, *image))
```

Результат: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]

69.5. Последовательное и параллельное отображение

map () – это встроенная функция, что означает, что она доступна повсюду, без необходимости использовать оператор import. Она доступна везде, как print (). В примере 5 необходимо использовать оператор импорта (import pprint), прежде чем появится возможность использовать функцию pretty print. Таким образом, pprint не является встроенной функцией.

Последовательное отображение

В этом случае каждый аргумент итерируемого объекта передается в качестве аргумента функции отображения в порядке возрастания. Эта ситуация возникает, когда у нас есть только один итерируемый объект для отображения, а для функции отображения требуется один аргумент.

Пример 1.

```
insects = ['fly', 'ant', 'beetle', 'cankerworm']
f = lambda x: x + ' is an insect'
print(list(map(f, insects))) # функция, определяемая f, выполняется над каждым элементом
                             # итерируемого объекта insects
```

Результат:

```
['fly is an insect', 'ant is an insect', 'beetle is an insect', 'cankerworm is an insect']
```

Пример 2.

```
print(list(map(len, insects))) # функция len выполняется для каждого элемента списка insects
```

Результат:

```
[3, 3, 6, 10]
```

Параллельное отображение

В этом случае каждый аргумент функции сопоставления выталкивается из всех итерируемых объектов (по одному от каждого итерируемого) параллельно. Таким образом, количество предоставленных итерируемых объектов должно соответствовать количеству аргументов, требуемых функцией.

```
carnivores = ['lion', 'tiger', 'leopard', 'arctic fox']
herbivores = ['african buffalo', 'moose', 'okapi', 'parakeet']
omnivores = ['chicken', 'dove', 'mouse', 'pig']
```

```
def animals(w, x, y, z):
    return '{0}, {1}, {2}, and {3} ARE ALL ANIMALS'.format(w.title(), x, y, z)
```

Пример 3.

Слишком много аргументов

здесь можно заметить, что map пытается передать в len по одному элементу из каждого из

```
# четырех итерируемых объектов. Это приводит к тому, что len "жалуется", будто
# его "кормят" слишком большим количеством аргументов
print(list(map(len, insects, carnivores, herbivores, omnivores)))
```

Результат:

TypeError: len() takes exactly one argument (4 given) (len() принимает ровно один аргумент (дано 4))

Пример 4.

```
# Слишком мало аргументов
# здесь следует заметить, что map должен выполнять animal на отдельных элементах insects
# поочередно.
# Но animals "жалуются", когда
# приходит только один аргумент, тогда как ожидается четыре.
print(list(map(animals, insects)))
```

результат:

TypeError: animals() missing 3 required positional arguments: 'x', 'y', and 'z' (animals() не хватает 3 требуемых позиционных аргументов: 'x', 'y', and 'z')

Пример 5.

```
# здесь map снабжает w, x, y, z одним значением из списка
import pprint
pprint.pprint(list(map(animals, insects, carnivores, herbivores, omnivores)))
```

результат:

```
['Fly, lion, african buffalo, and chicken ARE ALL ANIMALS',
'Ant, tiger, moose, and dove ARE ALL ANIMALS',
'Beetle, leopard, okapi, and mouse ARE ALL ANIMALS',
'Cankerworm, arctic fox, parakeet, and pig ARE ALL ANIMALS']
```

Глава 70. Возведение в степень

70.1. Возведение в степень с использованием встроенных модулей: ** и pow()

Для возведения в степень можно использовать встроенную функцию pow или оператор **:

```
2 ** 3 # 8
pow(2, 3) # 8
```

Для большинства (в Python 2.x – для всех) арифметических операций тип результата будет соответствовать типу более широкого операнда. Это не так для **; следующие случаи являются исключениями из этого правила:

- Основание: int, показатель степени: int < 0:

```
2 ** -3
# Out: 0.125 (result is a float)
```

- Это справедливо и для Python 3.x.
- До версии Python 2.2.0 это приводило к ошибке ValueError.
- Основание: int < 0 или float < 0, показатель степени float != int

```
(-2) ** (0.5) # also (-2.) ** (0.5)
# Результат: (8.659560562354934e-17+1.4142135623730951j) (результат типа complex)
```

- До версии Python 3.0.0 это приводило к ошибке `ValueError`.

Модуль `operator` содержит две функции, эквивалентные `**`-оператору:

```
import operator
operator.pow(4, 2)      # 16
operator.__pow__(4, 3)  # 64
```

или можно напрямую вызвать метод `__pow__`:

```
val1, val2 = 4, 2
val1.__pow__(val2) # 16
val2.__rpow__(val1) # 16
# in-place операция возведения в степень не поддерживается такими неизменяемыми
# классами, как int, float, complex:
# val1.__ipow__(val2)
```

70.2. Квадратный корень: `math.sqrt()` и `cmath.sqrt`

Модуль `math` содержит функцию `math.sqrt()`, которая позволяет вычислить квадратный корень из любого числа (которое может быть преобразовано в тип `float`), причем результат всегда будет иметь тип `float`:

```
import math

math.sqrt(9)          # 3.0
math.sqrt(11.11)      # 3.3331666624997918
math.sqrt(Decimal('6.25')) # 2.5
```

Функция `math.sqrt()` выдает ошибку `ValueError`, если результат будет комплексным числом (`complex`):

```
math.sqrt(-10)
```

`ValueError: math domain error` (ошибка математической области)

Функция `math.sqrt(x)` работает быстрее, чем `math.pow(x, 0.5)` или `x ** 0.5`, но точность результатов одинакова. Модуль `cmath` очень похож на модуль `math`, за исключением того, что он может вычислять комплексные числа и все его результаты имеют вид `a + bi`. Он также может использовать функцию `.sqrt()`:

```
import cmath

cmath.sqrt(4) # 2+0j
cmath.sqrt(-4) # 2j
```

Здесь `j` – это квадратный корень из `-1`. Все числа можно представить в виде `a + bi`, или, как в данном случае, `a + bj`. `a` – это действительная часть числа, как `2` в `2+0j`. Поскольку у него нет мнимой части, `b` равно `0`. `b` представляет собой часть мнимой части числа, например `2` в `2j`. Поскольку в нем нет действительной части, `2j` можно записать как `0 + 2j`.

70.3. Возведение в степень по модулю: функция `pow()` с тремя аргументами

При задании функции `pow()` с тремя аргументами `pow(a, b, c)` выполняется возведение в степень по модулю `ab mod c`:

```
pow(3, 4, 17) # 13
```

эквивалентное неоптимизированное выражение:
`3 ** 4 % 17` # 13

```
# шаги:
3 ** 4      # 81
81 % 17     # 13
```

Для встроенных типов возведение в степень по модулю возможно только в том случае, если:

- Первый аргумент имеет тип int
- Второй аргумент – это $\text{int} \geq 0$
- Третий аргумент – это $\text{int} \neq 0$

Эти ограничения также присутствуют в Python 3.x.

Например, можно использовать 3-аргументную форму pow для определения функции обратного по модулю:

```
def modular_inverse(x, p):
    """Найдите такое a, что  $a \cdot x \equiv 1 \pmod{p}$ , предполагая, что p – простое."""
    return pow(x, p-2, p)

[modular_inverse(x, 13) for x in range(1,13)]
# Результат: [1, 7, 9, 10, 8, 11, 2, 5, 3, 4, 6, 12]
```

70.4. Вычисление больших целых корней

Несмотря на то, что Python изначально поддерживает большие целые числа, извлечение n-го корня из очень больших чисел в Python может быть неудачным.

```
x = 2 ** 100
cube = x ** 3
root = cube ** (1.0 / 3)
```

OverflowError: long int too large to convert to float (long int слишком велик для преобразования во float)

При работе с такими большими целыми числами вам нужно будет использовать пользовательскую функцию для вычисления n-го корня числа.

```
def nth_root(x, n):
    # Начнем с определения некоторых разумных границ для n-го корня.
    upper_bound = 1
    while upper_bound ** n <= x:
        upper_bound *= 2
    lower_bound = upper_bound // 2
    # Продолжим поиск лучшего результата до тех пор, пока границы имеют смысл.
    while lower_bound < upper_bound:
        mid = (lower_bound + upper_bound) // 2
        mid_nth = mid ** n
        if lower_bound < mid and mid_nth < x:
            lower_bound = mid
        elif upper_bound > mid and mid_nth > x:
            upper_bound = mid
        else:
            # Найден идеальный n-й корень.
            return mid
    return mid + 1

x = 2 ** 100
cube = x ** 3
root = nth_root(cube, 3)
x == root
# True
```

70.5. Возведение в степень с использованием модуля math: math.pow()

Модуль math содержит еще одну функцию – math.pow(). Отличие от встроенной функции pow() или оператора ** заключается в том, что результат всегда является числом с плавающей точкой, float:

```
import math
math.pow(2, 2) # 4.0
math.pow(-2, 2) # 4.0
```

Это исключает вычисления с комплексными числами в качестве входных данных

```
math.pow(2, 2+0j)
```

TypeError: can't convert complex to float (не удастся преобразовать комплексные числа в числа с плавающей точкой)

а также вычислений, которые привели бы к получению комплексных чисел:

```
math.pow(-2, 0.5)
```

ValueError: math domain error (ошибка математической области)

70.6. Экспоненциальная функция: math.exp() и cmath.exp()

Оба модуля – math и cmath – содержат число Эйлера e, и использование его со встроенной функцией pow() или оператором ** работает как функция math.exp():

```
import math

math.e ** 2 # 7.3890560989306495
math.exp(2) # 7.38905609893065

import cmath
cmath.e ** 2 # 7.3890560989306495
cmath.exp(2) # (7.38905609893065+0j)
```

Однако результат получается разным, и использование экспоненциальной функции напрямую более надежно, чем встроенное возведение в степень с основанием math.e:

```
print(math.e ** 10) # 22026.465794806703
print(math.exp(10)) # 22026.465794806718
print(cmath.exp(10).real) # 22026.465794806718
# различие появляется здесь ^
```

70.7. Экспоненциальная функция -1: math.expm1()

Модуль math содержит функцию expm1(), которая позволяет вычислить выражение $\text{math.e}^x - 1$ для очень малых значений x с большей точностью, чем позволяют math.exp(x) или cmath.exp(x):

```
import math

print(math.e ** 1e-3 - 1) # 0.0010005001667083846
print(math.exp(1e-3) - 1) # 0.0010005001667083846
print(math.expm1(1e-3)) # 0.0010005001667083417
# ^
```

Для очень маленького значения x разница становится больше:

```
print(math.e ** 1e-15 - 1) # 1.1102230246251565e-15
print(math.exp(1e-15) - 1) # 1.1102230246251565e-15
print(math.expm1(1e-15)) # 1.0000000000000007e-15
# ^
```

Это улучшение значимо для научных вычислений. Например, закон Планка содержит экспоненциальную функцию минус 1:

```
def planks_law(lambda_, T):
    from scipy.constants import h, k, c # Если scipy не установлен, жестко закодируйте их
    return 2 * h * c ** 2 / (lambda_ ** 5 * math.expm1(h * c / (lambda_ * k * T)))

def planks_law_naive(lambda_, T):
    from scipy.constants import h, k, c # Если scipy не установлен, жестко закодируйте их
    return 2 * h * c ** 2 / (lambda_ ** 5 * (math.e ** (h * c / (lambda_ * k * T)) - 1))

planks_law(100, 5000)          # 4.139080074896474e-19
planks_law_naive(100, 5000)    # 4.139080073488451e-19
#                               ^_____

planks_law(1000, 5000)         # 4.139080128493406e-23
planks_law_naive(1000, 5000)   # 4.139080233183142e-23
#                               ^_____
```

70.8. Магические методы и возведение в степень: builtin, math и cmath

Предположим, у вас есть класс, хранящий исключительно целочисленные значения:

```
class Integer(object):
    def __init__(self, value):
        self.value = int(value) # Приведение к целому числу

    def __repr__(self):
        return '{cls}({val})'.format(cls=self.__class__.__name__,
                                      val=self.value)

    def __pow__(self, other, modulo=None):
        if modulo is None:
            print('Using __pow__')
            return self.__class__(self.value ** other)
        else:
            print('Using __pow__ with modulo')
            return self.__class__(pow(self.value, other, modulo))

    def __float__(self):
        print('Using __float__')
        return float(self.value)

    def __complex__(self):
        print('Using __complex__')
        return complex(self.value, 0)
```

Использование встроенной функции pow или **-оператора всегда вызывает __pow__ :

```
Integer(2) ** 2          # Integer(4)
# Выводит: Using __pow__
Integer(2) ** 2.5        # Integer(5)
# Выводит: Using __pow__
pow(Integer(2), 0.5)     # Integer(1)
# Выводит: Using __pow__
operator.pow(Integer(2), 3) # Integer(8)
# Выводит: Using __pow__
operator.__pow__(Integer(3), 3) # Integer(27)
# Выводит: Using __pow__
```

Второй аргумент метода `__pow__()` может быть предоставлен только с помощью встроенной функции `pow()` или путем прямого вызова метода:

```
pow(Integer(2), 3, 4)          # Integer(0)
# Выводит: Using __pow__ with modulo
Integer(2).__pow__(3, 4)      # Integer(0)
# Выводит: Using __pow__ with modulo
```

Хотя функции `math` всегда преобразуют его в тип `float` и используют числа с плавающей точкой:

```
import math

math.pow(Integer(2), 0.5) # 1.4142135623730951
# Prints: Using __float__
```

Функции `cmath` пытаются преобразовать его в `complex`, но могут также возвращаться к `float`, если нет явного преобразования в `complex`:

```
import cmath

cmath.exp(Integer(2)) # (7.38905609893065+0j)
# Prints: Using __complex__

del Integer.__complex__ # Удален метод __complex__ - экземпляры не могут быть приведены к complex

cmath.exp(Integer(2)) # (7.38905609893065+0j)
# Prints: Using __float__
```

Модули `math` и `cmath` не будут работать, если отсутствует метод `__float__()`:

```
del Integer.__float__      # Удален метод __complex__

math.sqrt(Integer(2))      # также cmath.exp(Integer(2))
```

`TypeError: a float is required`

70.9. Корень n-й степени с дробным показателем степени

Хотя функция `math.sqrt` предназначена для извлечения квадратного корня, для выполнения операций с корнем n-й степени, например кубическим, часто удобно использовать оператор возведения в степень (`**`) с дробными экспонентами.

Обратным действием для возведения в степень является возведение в степень с обратным показателем. Так, если можно возвести число в куб, используя показатель степени 3, то можно извлечь кубический корень из числа, возведя его в степень $1/3$.

```
>>> x = 3
>>> y = x ** 3
>>> y
27
>>> z = y ** (1.0 / 3)
>>> z
3.0
>>> z == x
True
```


Глава 71. Поиск

71.1. Поиск элемента

Все встроенные типы коллекций в Python реализуют способ проверки принадлежности элементов с помощью функции `in`.

Список (List)

```
alist = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 in alist      # True
10 in alist     # False
```

Кортеж (Tuple)

```
atuple = ('0', '1', '2', '3', '4')
4 in atuple      # False
'4' in atuple    # True
```

Строка (String)

```
astring = 'i am a string'
'a' in astring   # True
'am' in astring  # True
'I' in astring   # False
```

Набор (Set)

```
aset = {(10, 10), (20, 20), (30, 30)}
(10, 10) in aset # True
10 in aset       # False
```

Словарь (Dict)

Случай с `dict` немного особенный: обычно `in` проверяет только ключи. Если вы хотите искать в значениях, то необходимо указать это. То же самое, если вы хотите искать пары “ключ-значение”.

```
adict = {0: 'a', 1: 'b', 2: 'c', 3: 'd'}
1 in adict      # True – неявный поиск в ключах
'a' in adict     # False
2 in adict.keys() # True – явный поиск в ключах
'a' in adict.values() # True – явный поиск в значениях
(0, 'a') in adict.items() # True – явный поиск пар “ключ/значение”
```

71.2. Поиск в пользовательских классах: `__contains__` и `__iter__`

Для того чтобы разрешить использование `in` в пользовательских классах, класс должен предоставить магический метод `__contains__` или, если это невозможно, метод `__iter__`.

Предположим, имеется класс, содержащий список списков:

```
class ListList:
    def __init__(self, value):
        self.value = value
        # Создать набор всех значений для быстрого доступа
        self.setofvalues = set(item for sublist in self.value for item in sublist)

    def __iter__(self):
        print('Using __iter__.')
        # Генератор по всем элементам подсписка
        return (item for sublist in self.value for item in sublist)

    def __contains__(self, value):
        print('Using __contains__.')
```

```
# Просто поиск, есть ли значение в наборе
return value in self.setofvalues
```

```
# Даже без использования набора можно использовать метод iter для проверки на содержание:
# return any(item == value for item in iter(self))
```

Использование проверки на принадлежность возможно при использовании in:

```
a = ListList([[1,1,1],[0,1,1],[1,5,1]])
10 in a # False
# Prints: Using __contains__
5 in a # True
# Prints: Using __contains__
```

даже после удаления метода __contains__:

```
del ListList.__contains__
5 in a # True
# Prints: Using __iter__
```

Примечание: заикливание in (как for i in a) всегда будет использовать __iter__, даже если класс реализует метод __contains__.

71.3. Получение индекса для строк: str.index(), str.rindex() и str.find(), str.rfind()

Строковый тип данных также имеет метод index, но с более расширенными возможностями и дополнительной функцией str.find. Для обоих вышеперечисленных существует дополнительный *обратный* метод.

```
astring = 'Hello on StackOverflow'
astring.index('o') # 4
astring.rindex('o') # 20
```

```
astring.find('o') # 4
astring.rfind('o') # 20
```

Разница между index/rindex и find/rfind заключается в том, что именно происходит, если подстрока не найдена в строке:

```
astring.index('q') # ValueError: substring not found (подстрока не найдена)
astring.find('q') # -1
```

Все эти методы допускают указание начального и конечного индекса:

```
astring.index('o', 5) # 6
astring.index('o', 6) # 6 - начало включено
astring.index('o', 5, 7) # 6
astring.index('o', 5, 6) # - конец не включен
```

ValueError: substring not found (подстрока не найдена)

```
astring.rindex('o', 20) # 20
astring.rindex('o', 19) # 20 - по-прежнему слева направо
astring.rindex('o', 4, 7) # 6
```

71.4. Получение индексов в списках и кортежах: list.index(), tuple.index()

Списки и кортежи поддерживают метод index для получения позиции элемента:

```
alist = [10, 16, 26, 5, 2, 19, 105, 26]
# поиск 16 в списке
alist.index(16) # 1
```

```
alist[1]          # 16
alist.index(15)

ValueError: 15 is not in list (15 не в списке)
```

Но возвращается только позиция первого найденного элемента:

```
atuple = (10, 16, 26, 5, 2, 19, 105, 26)
atuple.index(26) # 2
atuple[2]        # 26
atuple[7]        # 26 - тоже 26!
```

71.5. Поиск ключа или ключей для значения в словаре

В словарях (dict) нет встроенного метода поиска значения или ключа, поскольку словари неупорядочены. Можно создать функцию, которая получает ключ (или ключи) для заданного значения:

```
def getKeysForValue(dictionary, value):
    foundkeys = []
    for keys in dictionary:
        if dictionary[keys] == value:
            foundkeys.append(keys)
    return foundkeys
```

Это также можно записать в виде эквивалентного генератора списка:

```
def getKeysForValueComp(dictionary, value):
    return [key for key in dictionary if dictionary[key] == value]
```

Если вам нужен только один найденный ключ:

```
def getOneKeyForValue(dictionary, value):
    return next(key for key in dictionary if dictionary[key] == value)
```

Первые две функции возвращают список (list) всех ключей, имеющих указанное значение:

```
adict = {'a': 10, 'b': 20, 'c': 10}
getKeysForValue(adict, 10)      # ['c', 'a'] - порядок случайный, с тем же успехом ['a', 'c']
getKeysForValueComp(adict, 10)  # ['c', 'a']
getKeysForValueComp(adict, 20)  # ['b']
getKeysForValueComp(adict, 25)  # []
```

Следующая функция возвращает только один ключ:

```
getOneKeyForValue(adict, 10)     # 'c' - в зависимости от обстоятельств это может быть и 'a'
getOneKeyForValue(adict, 20)     # 'b'
```

и вызвать StopIteration Exception, если значения нет в словаре:

```
getOneKeyForValue(adict, 25)

StopIteration
```

71.6. Получение индекса в отсортированных последовательностях: bisect.bisect_left()

Отсортированные последовательности позволяют использовать более быстрые алгоритмы поиска, такие как bisect.bisect_left():

```
import bisect

def index_sorted(sorted_seq, value):
    """Найти крайнее левое значение, точно равное x, или вызвать ошибку ValueError"""
    i = bisect.bisect_left(sorted_seq, value)
```

```

if i != len(sorted_seq) and sorted_seq[i] == value:
    return i
raise ValueError

```

```

alist = [i for i in range(1, 100000, 3)] # Отсортированный список от 1 до 100000 с шагом 3
index_sorted(alist, 97285) # 32428
index_sorted(alist, 4) # 1
index_sorted(alist, 97286)

ValueError

```

Для очень больших отсортированных последовательностей увеличение скорости может быть довольно значительным. В случае первого поиска примерно в 500 раз быстрее:

```

%timeit index_sorted(alist, 97285)
# 100000 циклов, лучший из 3: 3 мкс на цикл
%timeit alist.index(97285)
# 1000 циклов, лучший из 3: 1.58 мс на цикл

```

Хотя это немного медленнее, если элемент является одним из первых:

```

%timeit index_sorted(alist, 4)
# 100000 циклов, лучший из 3: 2.98 мкс на цикл
%timeit alist.index(4)
# 1000000 loops, best of 3: 580 нс на цикл

```

71.7. Поиск во вложенных последовательностях

Поиск во вложенных последовательностях, таких как список кортежей, требует подхода, подобного поиску ключей для значений в словаре (dict), но нуждается в специализированных функциях. Индекс крайней последовательности, если значение было найдено в последовательности:

```

def outer_index(nested_sequence, value):
    return next(index for index, inner in enumerate(nested_sequence)
                for item in inner
                if item == value)

```

```

alist_of_tuples = [(4, 5, 6), (3, 1, 'a'), (7, 0, 4.3)]
outer_index(alist_of_tuples, 'a') # 1
outer_index(alist_of_tuples, 4.3) # 2

```

или индекс внешней и внутренней последовательности:

```

def outer_inner_index(nested_sequence, value):
    return next((oindex, iindex) for oindex, inner in enumerate(nested_sequence)
                for iindex, item in enumerate(inner)
                if item == value)

```

```

outer_inner_index(alist_of_tuples, 'a') # (1, 2)
alist_of_tuples[1][2] # 'a'

```

```

outer_inner_index(alist_of_tuples, 7) # (2, 0)
alist_of_tuples[2][0] # 7

```

В общем случае (*не всегда*) использование `next` и **генераторного выражения** с условиями для нахождения первого вхождения искомого значения является наиболее эффективным подходом.

Глава 72. Сортировка, минимум и максимум

72.1. Упорядоченность в пользовательских классах

`min`, `max` и `sorted` требуют, чтобы объекты были упорядочены. Чтобы класс был упорядоченным, в нем должны быть определены все 6 методов `__lt__`, `__gt__`, `__ge__`, `__le__`, `__ne__` и `__eq__`:

```
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{!r} - Test less than {!r}'.format(self, other))
        return self.value < other.value

    def __le__(self, other):
        print('{!r} - Test less than or equal to {!r}'.format(self, other))
        return self.value <= other.value

    def __gt__(self, other):
        print('{!r} - Test greater than {!r}'.format(self, other))
        return self.value > other.value

    def __ge__(self, other):
        print('{!r} - Test greater than or equal to {!r}'.format(self, other))
        return self.value >= other.value

    def __eq__(self, other):
        print('{!r} - Test equal to {!r}'.format(self, other))
        return self.value == other.value

    def __ne__(self, other):
        print('{!r} - Test not equal to {!r}'.format(self, other))
        return self.value != other.value
```

Хотя реализация всех этих методов будет казаться ненужной, опускание некоторых из них сделает ваш код подверженным ошибкам. Примеры:

```
alist = [IntegerContainer(5), IntegerContainer(3),
         IntegerContainer(10), IntegerContainer(7)]

res = max(alist)
# Вывод: IntegerContainer(3) - Test greater than IntegerContainer(5)
#        IntegerContainer(10) - Test greater than IntegerContainer(5)
#        IntegerContainer(7) - Test greater than IntegerContainer(10)
print(res)
# Вывод: IntegerContainer(10)

res = min(alist)
# Вывод: IntegerContainer(3) - Test less than IntegerContainer(5)
#        IntegerContainer(10) - Test less than IntegerContainer(3)
#        IntegerContainer(7) - Test less than IntegerContainer(3)
print(res)
# Вывод: IntegerContainer(3)
```

```
res = sorted(alist)
# Вывод:      IntegerContainer(3) - Test less than IntegerContainer(5)
#             IntegerContainer(10) - Test less than IntegerContainer(3)
#             IntegerContainer(10) - Test less than IntegerContainer(5)
#             IntegerContainer(7) - Test less than IntegerContainer(5)
#             IntegerContainer(7) - Test less than IntegerContainer(10)
print(res)
# Вывод:      [IntegerContainer(3), IntegerContainer(5), IntegerContainer(7), IntegerContainer(10)]
```

sorted с помощью reverse=True также использует __lt__:

```
res = sorted(alist, reverse=True)
# Вывод:      IntegerContainer(10) - Test less than IntegerContainer(7)
#             IntegerContainer(3) - Test less than IntegerContainer(10)
#             IntegerContainer(3) - Test less than IntegerContainer(10)
#             IntegerContainer(3) - Test less than IntegerContainer(7)
#             IntegerContainer(5) - Test less than IntegerContainer(7)
#             IntegerContainer(5) - Test less than IntegerContainer(3)
print(res)
# Вывод:      [IntegerContainer(10), IntegerContainer(7), IntegerContainer(5), IntegerContainer(3)]
```

Но sorted может использовать __gt__ вместо этого, если значение по умолчанию не реализовано:

```
del IntegerContainer.__lt__ # The IntegerContainer больше не реализует "less than"
```

```
res = min(alist)
# Вывод:      IntegerContainer(5) - Test greater than IntegerContainer(3)
#             IntegerContainer(3) - Test greater than IntegerContainer(10)
#             IntegerContainer(3) - Test greater than IntegerContainer(7)
print(res)
# Вывод:      IntegerContainer(3)
```

Методы сортировки вызовут TypeError, если не реализованы ни __lt__, ни __gt__:

```
del IntegerContainer.__gt__ # The IntegerContainer больше не реализует "greater then"
```

```
res = min(alist)
TypeError: unorderable types: IntegerContainer () < IntegerContainer ()
```

Декоратор functools.total_ordering можно использовать для упрощения работы с этими методами сравнения. Если вы декорируете свой класс при помощи total_ordering, вам нужно реализовать __eq__, __ne__ и только один из методов __lt__, __le__, __ge__ или __gt__, а декоратор сам заполнит остальные:

```
import functools

@functools.total_ordering
class IntegerContainer(object):
    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return "{}({})".format(self.__class__.__name__, self.value)

    def __lt__(self, other):
        print('{} - Test less than {}'.format(self, other))
        return self.value < other.value

    def __eq__(self, other):
        print('{} - Test equal to {}'.format(self, other))
        return self.value == other.value
```

```
def __ne__(self, other):
    print('{!r} - Test not equal to {!r}'.format(self, other))
    return self.value != other.value
```

```
IntegerContainer(5) > IntegerContainer(6)
# Вывод: IntegerContainer(5) - Test less than IntegerContainer(6)
# Возвращает: False
```

```
IntegerContainer(6) > IntegerContainer(5)
# Вывод: IntegerContainer(6) - Test less than IntegerContainer(5)
# Вывод: IntegerContainer(6) - Test equal to IntegerContainer(5)
# Возвращает: True
```

Обратите внимание, что теперь `>` (*больше чем, greater than*) вызывает метод *меньше, чем* (*less than*), а в некоторых случаях даже метод `__eq__`. Это также означает, что если скорость имеет большое значение, то вы должны реализовать каждый метод сравнения самостоятельно.

72.2. Особый случай: словари

Получение минимального или максимального значения или использование функции `sorted` зависит от итераций над объектом. В случае словаря итерация происходит только по ключам:

```
adict = {'a': 3, 'b': 5, 'c': 1}
min(adict)
# Результат: 'a'
max(adict)
# Результат: 'c'
sorted(adict)
# Результат: ['a', 'b', 'c']
```

Чтобы сохранить структуру словаря, необходимо выполнить итерацию по `.items()`:

```
min(adict.items())
# Результат: ('a', 3)
max(adict.items())
# Результат: ('c', 1)
sorted(adict.items())
# Результат: [('a', 3), ('b', 5), ('c', 1)]
```

Для `sorted` можно создать упорядоченный `OrderedDict`, чтобы сохранить сортировку и структуру, подобную структуре словаря:

```
from collections import OrderedDict
OrderedDict(sorted(adict.items()))
# Результат: OrderedDict([('a', 3), ('b', 5), ('c', 1)])
res = OrderedDict(sorted(adict.items()))
res['a']
# Результат: 3
```

По значению

Следует использовать ключевой аргумент:

```
min(adict.items(), key=lambda x: x[1])
# Результат: ('c', 1)
max(adict.items(), key=operator.itemgetter(1))
# Результат: ('b', 5)
sorted(adict.items(), key=operator.itemgetter(1), reverse=True)
# Результат: [('b', 5), ('a', 3), ('c', 1)]
```

72.3. Использование ключевого аргумента

Нахождение минимума/максимума последовательности последовательностей возможно следующим образом:

```
list_of_tuples = [(0, 10), (1, 15), (2, 8)]
min(list_of_tuples)
# Результат: (0, 10)
```

но если вы хотите сортировать по конкретному элементу в каждой последовательности, используйте ключевой аргумент:

```
min(list_of_tuples, key=lambda x: x[0]) # Сортировка по первому элементу
# Результат: (0, 10)
```

```
min(list_of_tuples, key=lambda x: x[1]) # Сортировка по второму элементу
# Результат: (2, 8)
```

```
sorted(list_of_tuples, key=lambda x: x[0]) # Сортировка по первому элементу (по возрастанию)
# Результат: [(0, 10), (1, 15), (2, 8)]
```

```
sorted(list_of_tuples, key=lambda x: x[1]) # Сортировка по второму элементу
# Результат: [(2, 8), (0, 10), (1, 15)]
```

```
import operator
# Модуль operator содержит эффективные альтернативы использованию лямбда-функции
max(list_of_tuples, key=operator.itemgetter(0)) # Сортировка по первому элементу
# Результат: (2, 8)
```

```
max(list_of_tuples, key=operator.itemgetter(1)) # Сортировка по второму элементу
# Результат: (1, 15)
```

```
sorted(list_of_tuples, key=operator.itemgetter(0), reverse=True) # Реверсивный (убывающий)
# Результат: [(2, 8), (1, 15), (0, 10)]
```

```
sorted(list_of_tuples, key=operator.itemgetter(1), reverse=True) # Reversed (убывающий)
# Результат: [(1, 15), (0, 10), (2, 8)]
```

72.4. Аргумент по умолчанию для max, min

В функции max или min нельзя передать пустую последовательность:

```
min([])
```

ValueError: min() arg is an empty sequence (min() arg – пустая последовательность)

Однако в Python 3 можно передать ключевой аргумент default со значением, которое будет возвращено, если последовательность пуста, вместо того чтобы вызывать исключение:

```
max([], default=42)
# Результат: 42
max([], default=0)
# Результат: 0
```

72.5. Получение отсортированной последовательности

Использование одной последовательности:

```
sorted((7, 2, 1, 5)) # кортеж
# Результат: [1, 2, 5, 7]
```

```
sorted(['c', 'A', 'b']) # список
# Результат: ['A', 'b', 'c']
```



```
sorted({11, 8, 1})           # набор
# Результат: [1, 8, 11]

sorted({'11': 5, '3': 2, '10': 15}) # словарь
# Результат: ['10', '11', '3']      # итерация только по ключам

sorted('bdca')               # строка
# Результат: ['a','b','c','d'].
```

Результатом всегда является новый список, исходные данные остаются неизменными.

72.6. Извлечение N наибольших или N наименьших элементов из итерируемого объекта

Чтобы найти некоторое количество (больше одного) наибольших или наименьших значений итерируемого объекта, можно воспользоваться функциями `nlargest` и `nsmallest` из модуля `heapq`:

```
import heapq

# получить 5 самых больших элементов из диапазона

heapq.nlargest(5, range(10))
# Результат: [9, 8, 7, 6, 5]

heapq.nsmallest(5, range(10))
# Результат: [0, 1, 2, 3, 4]
```

Это гораздо удобнее, чем сортировать весь итерируемый объект, а затем делать срезы с конца или начала. Внутри этих функций используется структура данных “приоритетной очереди двоичной кучи” (`binary heap priority queue`), что очень удобно для данного случая. Подобно `min`, `max` и `sorted`, эти функции принимают необязательный ключевой аргумент, который должен быть функцией, которая, получая элемент, возвращает его ключ сортировки.

Приведем код, который извлекает 1000 самых длинных строк из файла:

```
import heapq
with open(filename) as f:
    longest_lines = heapq.nlargest(1000, f, key=len)
```

Здесь мы открываем файл и передаем дескриптор файла `f` в функцию `nlargest`. Итерация по файлу позволяет получить каждую строку файла в виде отдельной строки `string`; затем `nlargest` передает каждый элемент (или строку) в функцию `len` для определения ключа сортировки.

Функция `len`, получая строку `string`, возвращает длину строки в символах. Требуется хранение только списка из 1000 наибольших на данный момент строчек, что можно противопоставить коду

```
longest_lines = sorted(f, key=len)[1000:]
    который должен будет держать в памяти весь файл.
```

72.7. Получение минимального или максимального из нескольких значений

```
min(7,2,1,5)
# Результат: 1

max(7,2,1,5)
# Результат: 7
```

72.8. Минимум и максимум последовательности

Получение минимума последовательности (итерируемого объекта) эквивалентно доступу к первому элементу отсортированной (sorted) последовательности:

```
min([2, 7, 5])
# Результат: 2
sorted([2, 7, 5])[0]
# Результат: 2
```

С максимумом немного сложнее, так как sorted сохраняет порядок, а max возвращает первое встретившееся значение. В случае отсутствия дубликатов максимальное значение совпадает с последним элементом отсортированного возврата:

```
max([2, 7, 5])
# Результат: 7
sorted([2, 7, 5])[-1]
# Результат: 7
```

Но не в том случае, если имеется несколько элементов, которые оцениваются как имеющие максимальное значение:

```
class MyClass(object):
    def __init__(self, value, name):
        self.value = value
        self.name = name

    def __lt__(self, other):
        return self.value < other.value

    def __repr__(self):
        return str(self.name)

sorted([MyClass(4, 'первый'), MyClass(1, 'второй'), MyClass(4, 'третий')])
# Результат: [второй, первый, третий]
max([MyClass(4, 'первый'), MyClass(1, 'второй'), MyClass(4, 'третий')])
# Результат: первый
```

Допускается использование любого итерируемого объекта, содержащего элементы, поддерживающие операции < или >.

Глава 73. Подсчет

73.1. Подсчет всех вхождений всех элементов в итерируемом объекте: collections.Counter

```
from collections import Counter

c = Counter(["a", "b", "c", "d", "a", "b", "a", "c", "d"])
c
# Результат: Counter({'a': 3, 'b': 2, 'c': 2, 'd': 2})
c["a"]
# Результат: 3
c[7] # нет в списке (7 встретилось 0 раз!)
# Результат: 0
```

Можно использовать collections.Counter для любого итерируемого объекта и подсчет каждого вхождения для каждого элемента.

Примечание: исключение составляет случай, когда задан словарь dict или другой класс, похожий на collections.Mapping, тогда они не будут подсчитаны, а будет создан счетчик (Counter) с этими значениями:

```
Counter({"e": 2})  
# Результат: Counter({"e": 2})
```

```
Counter({"e": "e"}) # предупреждение Счетчик не проверяет, что значения имеют тип int  
# Результат: Counter({"e": "e"})
```

73.2. Получение наиболее часто встречающегося значения: collections.Counter.most_common()

Подсчет ключей отображения (Mapping) при помощи collections.Counter невозможен, но мы можем подсчитать значения:

```
from collections import Counter  
adict = {'a': 5, 'b': 3, 'c': 5, 'd': 2, 'e': 2, 'q': 5}  
Counter(adict.values())  
# Результат: Counter({2: 2, 3: 1, 5: 3})
```

Наиболее распространенные элементы можно получить, используя метод most_common:

```
# Сортировка от наиболее часто встречающегося к наименее часто встречающемуся значению:  
Counter(adict.values()).most_common()  
# Результат: [(5, 3), (2, 2), (3, 1)]
```

```
# Получение наиболее часто встречающегося значения  
Counter(adict.values()).most_common(1)  
# Результат: [(5, 3)]
```

```
# Получение двух наиболее частых значений  
Counter(adict.values()).most_common(2)  
# Результат: [(5, 3), (2, 2)].
```

73.3. Подсчет количества вхождений элемента в последовательность: list.count() и tuple.count()

```
alist = [1, 2, 3, 4, 1, 2, 1, 3, 4]  
alist.count(1)  
# Результат: 3  
  
atuple = ('bear', 'weasel', 'bear', 'frog')  
atuple.count('bear')  
# Результат: 2  
atuple.count('fox')  
# Результат: 0
```

73.4. Подсчет вхождений подстроки в строку: str.count()

```
astring = 'thisisashorttext'  
astring.count('t')  
# Результат: 4
```

Это работает для подстрок длиной более одного символа:

```
astring.count('th')  
# Результат: 1  
astring.count('is')  
# Результат: 2  
astring.count('text')  
# Результат: 1
```

что было бы невозможно при использовании `collections.Counter`, который подсчитывает только одиночные символы:

```
from collections import Counter
Counter('astring')
# Результат: Counter({'a': 1, 'e': 1, 'h': 2, 'i': 2, 'o': 1, 'r': 1, 's': 3, 't': 4, 'x': 1})
```

73.5. Подсчет вхождений в numpy-массиве

Для подсчета вхождений значения в numpy-массиве можно использовать:

```
>>> import numpy as np
>>> a=np.array([0,3,4,3,5,4,7])
>>> print np.sum(a==3)
2
```

Логическое выражение создает массив, в котором все вхождения запрашиваемых значений равны 1, а все остальные равны нулю. Таким образом, суммирование этих значений дает количество вхождений. Это работает для массивов любой формы и типа.

Для подсчета вхождений всех уникальных значений в numpy-массиве можно использовать методы `unique` и `bincount`. `Unique` автоматически “уплощает” многомерные массивы, а `bincount` работает только с одномерными массивами, содержащими только положительные целые числа.

```
>>> unique,counts=np.unique(a,return_counts=True)
>>> print unique,counts # counts[i] равен вхождениям unique[i] в a
[0 3 4 5 7] [1 2 2 1 1]
>>> bin_count=np.bincount(a)
>>> print bin_count # bin_count[i] равен вхождениям i в a
[1 0 0 2 2 1 0 1]
```

Если ваши данные представляют собой numpy-массивы, то, как правило, гораздо быстрее использовать эти методы, чем преобразовывать данные в общие методы.

Глава 74. Функция print

74.1. Основы вывода на экран

В Python 3 и выше `print` является функцией, а не ключевым словом.

```
print('hello world!')
# Результат: hello world!
```

```
foo = 1 bar = 'bar' baz = 3.14
print(foo) # Результат: 1
print(bar) # Результат: bar
print(baz) # Результат: 3.14
```

Также можно передать на печать несколько параметров:

```
print(foo, bar, baz)
# Результат: 1 bar 3.14
```

Другой способ вывести множественные параметры – это использование символа `+`

```
print(str(foo) + " " + bar + " " + str(baz))
# Результат: 1 bar 3.14
```

Однако при использовании + для печати нескольких параметров следует обратить внимание на то, что тип параметров должен быть одинаковым. Попытка вывести приведенный выше пример без приведения к типу string приведет к ошибке, поскольку к строке "bar" Python будет пытаться добавить цифру 1, а затем прибавить число 3.14.

```
# Неправильно:
# type: int str float
print(foo + bar + baz)
# приведет к ошибке
```

Это связано с тем, что в первую очередь будет оцениваться содержимое:

```
print(4 + 5)
# Результат: 9
print("4" + "5")
# Результат: 45
print([4] + [5])
# Результат: [4, 5]
```

Использование знака + может быть очень полезным для пользователя при чтении вывода переменных. В приведенном ниже примере вывод очень легко читается:

```
import random
#указание python включить функцию для создания случайных чисел
randnum = random.randint(0, 12)
#создать случайное число от 0 до 12 и присвоить его переменной
print("Случайно сгенерированное число было - " + str(randnum))
```

С помощью параметра end можно запретить функции печати автоматически печатать новую строку:

```
print("у этого нет новой строки в конце... ", end="")
print("видите?")
# Результат: здесь нет новой строки в конце... видите?
```

Если требуется запись в файл, это можно передать в качестве параметра file:

```
with open('my_file.txt', 'w+') as my_file:
    print("это идет в файл!", file=my_file)
```

74.2. Параметры функции print

При помощи этой функции можно не только печатать текст. У print также есть несколько параметров, которые помогут вам.

Аргумент sep: поместить строку между аргументами. К примеру, вам нужно вывести список слов, разделенных запятой или какой-либо другой строкой:

```
>>> print('apples','bananas', 'cherries', sep=', ')
apple, bananas, cherries
>>> print('apple','banana', 'cherries', sep=', ')
apple, banana, cherries
>>>
```

Аргумент end: использование в конце чего-то кроме новой строки. Без этого аргумента все функции print() пишут строку, а затем переходят к началу следующей строки. Вы можете изменить его так, чтобы он ничего не делал (использовать пустую строку ""), или сделать двойной интервал между абзацами, используя две новые строки.

```
>>> print("<a", end=""); print(" class='jldn' if 1 else \"\", end=""); print(">")
<a class='jldn'>
>>> print("paragraph1", end="\n\n"); print("paragraph2")
paragraph1
```

```
paragraph2
>>>
```

Аргумент `file`: отправить вывод куда-либо, кроме `sys.stdout`. Теперь вы можете отправлять текст либо в `stdout`, либо в файл, либо в `StringIO`.

```
>>> def sendit(out, *values, sep=' ', end='\n'):
...     print(*values, sep=sep, end=end, file=out)

>>> sendit(sys.stdout, 'apples', 'bananas', 'cherries', sep='\t')
apples bananas cherries
>>> with open("delete-me.txt", "w+") as f:
...     sendit(f, 'apples', 'bananas', 'cherries', sep=' ', end='\n')

>>> with open("delete-me.txt", "rt") as f:
...     print(f.read())

apples bananas cherries
>>>
```

Существует и четвертый параметр – `flush`, который принудительно очищает поток.

Глава 75. Регулярные выражения (Regex)

В Python регулярные выражения доступны через модуль `re`.

Регулярные выражения представляют собой комбинации символов, которые интерпретируются как правила сопоставления подстрок. Например, выражение `'amount\D+\d+'` будет соответствовать любой строке, состоящей из слова `amount` плюс целое число, разделенное одной или несколькими не-цифрами, например: `amount=100`, `amount is 3`, `amount is equal to: 33` и т. д.

75.1. Сопоставление начала строки

Первым аргументом `re.match()` является регулярное выражение, вторым – строка, которую нужно сопоставить:

```
import re
pattern = r"123"
string = "123zzb"

re.match(pattern, string)
# Результат: <_sre.SRE_Match object; span=(0, 3), match='123'>

match = re.match(pattern, string)
match.group()
# Результат: '123'
```

Вы можете заметить, что переменная `pattern` представляет собой строку с префиксом `r`, что указывает на то, что строка является *необработанным строковым литералом*.

Необработанный строковый литерал имеет несколько иной синтаксис, чем строковый литерал, а именно: обратная косая черта (обратный слеш) `\` в необработанном строковом литерале означает "просто обратную косую черту", и нет необходимости удваивать обратные косые черты для экранирования новых строк (`\n`), табуляции (`\t`) и т. д. В обычных строковых литералах каждый обратный слеш должен быть удвоен.

Следовательно, `r"\n"` – это строка из двух символов: `\` и `n`. В шаблонах регулярных выражений также используются обратные слешы, например, `\d` обозначает любой цифровой сим-

вол. Мы можем избежать двойного экранирования наших строк ("\\d"), используя необработанные строки (r"\\d"). Например:

```
string = "\\t123zzb"      # здесь обратная косая черта экранирована, поэтому нет табуляции,
                          # только \' и \'
pattern = "\\t123"        # будет соответствовать \\t (с экранированием обратного следа),
                          # за которым следует 123
re.match(pattern, string).group() # нет совпадения
re.match(pattern, "\\t123zzb").group() # совпадает с '\\t123'

pattern = r"\\t123"
re.match(pattern, string).group() # совпадает с '\\t123'
```

Поиск выполняется только с начала строки. Если необходимо выполнить поиск в любом месте, используйте `re.search`:

```
match = re.match(r"(123)", "a123zzb")

match is None
# Результат: True

match = re.search(r"(123)", "a123zzb")

match.group()
# Результат: '123'
```

75.2. Поиск

```
pattern = r"(your base)"
sentence = "All your base are belong to us."

match = re.search(pattern, sentence)
match.group(1)
# Результат: 'your base'

match = re.search(r"(belong.*)", sentence)
match.group(1)
# Результат: 'belong to us.'
```

В отличие от `re.match` поиск осуществляется в любом месте строки. Также можно использовать `re.findall`, а также искать в начале строки (используйте символ `^`),

```
match = re.search(r"^123", "123zzb")
match.group(0)
# Результат: '123'

match = re.search(r"^123", "a123zzb")
match is None
# Результат: True

    в конце строки (используйте $),

match = re.search(r"123$", "zzb123")
match.group(0)
# Результат: '123'

match = re.search(r"123$", "123zzb")
match is None
# Результат: True
```

```
    или и там, и там (используйте и ^, и $):
match = re.search(r"^123$", "123")
match.group(0)
# Результат: '123'
```

75.3. Предварительно скомпилированные шаблоны (precompiled patterns)

```
import re
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.search("Ответ - 41!")
matches.group(1)
# Результат: 41

matches = precompiled_pattern.search("Или это было 42?")
matches.group(1)
# Результат: 42
```

Компиляция шаблона позволяет повторно использовать его в дальнейшем в программе. Однако следует учитывать, что Python кэширует недавно использованные шаблоны выражений (docs, ответ SO), поэтому, как сообщает официальная документация, *“программам, использующим за раз только несколько регулярных выражений, не нужно беспокоиться о компиляции регулярных выражений”*.

```
import re
precompiled_pattern = re.compile(r"(\d+)")
matches = precompiled_pattern.match("Ответ - 41!")
print(matches.group(1))
# Результат: Ответ - 41

matches = precompiled_pattern.match("Или это было 42?")
print(matches.group(1))
# Результат: Или это было 42
```

Его можно использовать вместе с `re.match()`.

75.4. Флаги

Для некоторых особых случаев необходимо изменить поведение регулярного выражения, что делается с помощью флагов (flags). Флаги могут быть установлены двумя способами: использованием ключевого слова `flags` или непосредственно в выражении.

Ключевое слово `flags`

Ниже приведен пример для `re.search`, но он работает для большинства функций модуля `re`.

```
m = re.search("b", "ABC")
m is None
# Результат: True

m = re.search("b", "ABC", flags=re.IGNORECASE)
m.group()
# Результат: 'B'

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE)
m is None
# Результат: True

m = re.search("a.b", "A\nBC", flags=re.IGNORECASE|re.DOTALL)
m.group()
# Результат: 'A\nB'
```


Часто используемые флаги

Флаг	Краткое описание
re.IGNORECASE, re.I	Заставляет шаблон игнорировать регистр
re.DOTALL, re.S	Заставляет . соответствовать всему, включая новые строки
re.MULTILINE, re.M	Заставляет ^ соответствовать началу строки, а \$ – концу строки
re.DEBUG	Включает отладочную информацию

Полный список всех доступных флагов можно найти в официальной документации.

Флаги непосредственно в выражении

Из официальной документации:

(?iLmsux) (Одна или несколько букв из набора 'i', 'L', 'm', 's', 'u', 'x').

Эта группа соответствует пустой строке, буквы задают соответствующие флаги: re.I (игнорирование регистра), re.L (зависит от локализации), re.M (многострочный), re.S (точка соответствует всем), re.U (зависит от Unicode) и re.X (verbose) для всего регулярного выражения. Это удобно, если вы хотите включить флаги в регулярное выражение, а не передавать аргумент flag функции re.compile().

Обратите внимание, что флаг (?x) изменяет способ разбора выражения. Он должен использоваться первым в строке выражения или после одного или нескольких пробельных символов. Если перед флагом имеются символы, не являющиеся пробелами, то результат будет неопределенным.

75.5. Замена

Замены в строках могут осуществляться с использованием re.sub.

```
re.sub(r"t[0-9][0-9]", "foo", "my name t13 is t44 what t99 ever t44")
# Результат: 'my name foo is foo what foo ever foo'
```

Использование групповых ссылок

Замены с небольшим числом групп могут быть выполнены следующим образом:

```
re.sub(r"t([0-9])([0-9])", r"t2\1", "t13 t19 t81 t25")
# Результат: 't31 t91 t18 t52'
```

Однако если вы сделаете идентификатор группы (group ID), например '10', то это не сработает: \10 будет прочитано как "ID номер 1, за которым следует 0". Поэтому необходимо быть более конкретным и использовать обозначение \g<i>:

```
re.sub(r"t([0-9])([0-9])", r"t\g<2>\g<1>", "t13 t19 t81 t25")
# Результат: 't31 t91 t18 t52'
```

Использование функции замены

```
items = ["zero", "one", "two"]
re.sub(r"a\{([0-3])\}", lambda match: items[int(match.group(1))], "Items: a[0], a[1], something, a[2]")
# Результат: 'Items: zero, one, something, two'
```

75.6. Поиск всех непересекающихся совпадений

```
re.findall(r"[0-9]{2,3}", "some 1 text 12 is 945 here 4445588899")
# Результат: ['12', '945', '444', '558', '889']
```

Обратите внимание, что символ r перед "[0-9]{2,3}" указывает Python интерпретировать строку как есть, как "сырую" строку. Можно также использовать re.finditer(), которая работает так же, как re.findall(), но возвращает итератор с объектами SRE_Match вместо списка строк:

```

results = re.finditer(r"([0-9]{2,3})", "some 1 text 12 is 945 here 4445588899")
print(results)
# Результат: <callable-iterator object at 0x105245890>
for result in results:
    print(result.group(0))
''' Результат:
12
945
444
558
889
'''

```

75.7. Проверка на разрешенные символы

Если необходимо проверить, что строка содержит только определенный набор символов, в данном случае a-z, A-Z и 0-9, то это можно сделать следующим образом:

```

import re

def is_allowed(string):
    characterRegex = re.compile(r'^a-zA-Z0-9.'])
    string = characterRegex.search(string)
    return not bool(string)

print (is_allowed("abyzABYZ0099"))
# Результат: 'True'

print (is_allowed("#*@#$$%^"))
# Результат: 'False'

```

Можно изменить строку выражения с `[^a-zA-Z0-9.]` на `[^a-z0-9.]`, чтобы, например, запретить использование заглавных букв.

75.8. Разбиение строки с помощью регулярных выражений

Для разбиения строки можно также использовать регулярные выражения. Например:

```

import re
data = re.split(r'\s+', 'James 94 Samantha 417 Scarlett 74')
print( data )
# Результат: ['James', '94', 'Samantha', '417', 'Scarlett', '74']

```

75.9. Группировка

Группировка осуществляется с помощью круглых скобок. Вызов функции `group()` возвращает строку, сформированную из совпадающих подгрупп, заключенных в круглые скобки.

```

match.group() # Группа без аргумента возвращает все найденное совпадение
# Результат: '123'
match.group(0) # Указание 0 дает тот же результат, что и указание отсутствия аргумента
# Результат: '123'

```

Для получения конкретной подгруппы в `group()` также могут быть переданы аргументы. Из документации:

Если аргумент один, то результатом будет одна строка; если аргументов несколько, то результатом будет кортеж с одним элементом на аргумент.

С другой стороны, вызов функции `groups()` возвращает список кортежей, содержащих подгруппы.

```
sentence = "Это телефонный номер 672-123-456-9910"  
pattern = r"*(phone)*?([\d-]+)"
```

```
match = re.match(pattern, sentence)
```

```
match.groups() # Полное совпадение в виде списка кортежей подгрупп в скобках  
# Результат: ('phone', '672-123-456-9910')
```

```
m.group() # Полное совпадение в виде строки  
# Результат: 'Это телефонный номер 672-123-456-9910'
```

```
m.group(0) # Полное совпадение в виде строки  
# Результат: 'Это телефонный номер 672-123-456-9910'
```

```
m.group(1) # Первая подгруппа в скобках  
# Результат: 'phone'
```

```
m.group(2) # Вторая подгруппа в скобках.  
# Результат: '672-123-456-9910'
```

```
m.group(1, 2) # Несколько аргументов дают кортеж  
# Результат: ('phone', '672-123-456-9910')
```

Именованные группы

```
match = re.search(r'My name is (?P<name>[A-Za-z ]+)', 'My name is John Smith')  
match.group('name')  
# Результат: 'John Smith'
```

```
match.group(1)  
# Результат: 'John Smith'
```

Создает группу захвата (capture group), на которую можно ссылаться как по имени, так и по индексу.

Группы без захвата (non-capturing groups)

Использование (?) создает группу, но сама группа не захватывается. Ее можно использовать как группу, но она не будет загрязнять ваше "групповое пространство".

```
re.match(r'(\d+)(\+(\d+))?', '11+22').groups()  
# Результат: ('11', '+22', '22')
```

```
re.match(r'(\d+)(?:\+(\d+))?', '11+22').groups()  
# Результат: ('11', '22')
```

В данном примере совпадают 11+22 или 11, но не 11+. Это происходит потому, что знак + и второй член сгруппированы. С другой стороны, знак + не захватывается.

75.10. Исключение специальных символов

Специальные символы не сопоставляются в прямом смысле:

```
match = re.search(r'[b]', 'a[b]c') match.group()  
# Результат: 'b'
```

Но если специальные символы экранированы, то они могут быть сопоставлены буквально:

```
match = re.search(r'\[b\]', 'a[b]c') match.group()  
# Результат: '[b]'
```

Для этого можно использовать функцию `re.escape()`:

```
re.escape('a[b]c')
# Результат: 'a\\[b\\]c'
match = re.search(re.escape('a[b]c'), 'a[b]c')
match.group()
# Результат: 'a[b]c'
```

Функция `re.escape()` экранирует все специальные символы, поэтому она полезна, если вы составляете регулярное выражение на основе пользовательского ввода:

```
username = 'A.C.' # предположим, что это пришло от пользователя
re.findall(r'Hi {}!'.format(username), 'Hi A.C.! Hi ABCD!')
# Результат: ['Hi A.C.!', 'Hi ABCD!']
re.findall(r'Hi {}!'.format(re.escape(username)), 'Hi A.C.! Hi ABCD!')
# Результат: ['Hi A.C.!']
```

75.11. Сопоставление выражения только в определенных местах

Часто требуется установить соответствие выражению только в *определенных* местах. Рассмотрим следующее предложение:

An apple a day keeps the doctor away (I eat an apple everyday).

Здесь “apple” встречается дважды, что можно решить с помощью так называемых *управляющих глаголов обратного хода* (backtracking control verbs), которые поддерживаются более новым модулем `regex`. Идея заключается в следующем:

forget_this | or this | and this as well | (but keep this)

В нашем примере это будет так:

```
import regex as re
string = "An apple a day keeps the doctor away (I eat an apple everyday).\"
rx = re.compile(r\"
    \\([^\"]*)\" (*SKIP)(*FAIL) # сопоставить все, что находится в круглых скобках, и “отбросить”
    | # или
    apple # сопоставить apple
    \", re.VERBOSE)
apples = rx.findall(string)
print(apples)
# только 1
```

Это соответствует слову “apple” только в том случае, если оно встречается за пределами круглых скобок.

Вот как это работает:

- При просмотре **слева направо** `regex`-движок поглощает все, что находится слева, а `(*SKIP)` выступает в роли “всегда истинного утверждения”. После этого он корректно завершает работу до `(*FAIL)` и выполняет обратный путь.
- Теперь он доходит до точки `(*SKIP)` **справа налево** (при обратном ходе), где запрещено двигаться дальше влево. Вместо этого движку предписывается отбросить все, что находится слева, и перейти в точку, где был вызван `(*SKIP)`.

75.12. Итерация по совпадениям с помощью `re.finditer`

Вы можете использовать `re.finditer` для итерационного перебора всех совпадений в строке. Это дает дополнительную информацию (по сравнению с `re.findall`), например, сведения о расположении совпадений в строке (индексы):

```
import re
text = 'You can try to find an ant in this string'
pattern = 'an?\\w' # найти 'an' как с последующим символом слова, так и без него
```

```
for match in re.finditer(pattern, text):
# Начальный индекс совпадения (целое число)
sStart = match.start()

# Конечный индекс совпадения (целое число)
sEnd = match.end()

# Полное совпадение (строка)
sGroup = match.group()

# Вывести соответствие
print('Match "{}" found at: {}'.format(sGroup, sStart,sEnd))
```

Результат:

```
Match "an" found at: [5,7] (Совпадение "an" найдено в: [5,7])
Match "an" found at: [20,22] (Совпадение "an" найдено в: [20,22])
Match "ant" found at: [23,26] (Совпадение "ant" найдено в: [23,26])
```

Глава 76. Копирование данных

76.1. Копирование словаря

Объект словаря имеет метод `copy`. Он выполняет неглубокое копирование словаря.

```
>>> d1 = {'1': []}
>>> d2 = d1.copy()
>>> d1 is d2
False
>>> d1[1] is d2[1]
True
```

76.2. Неглубокое копирование

Неглубокое копирование – это копирование коллекции без выполнения копирования ее элементов.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.copy(c)
>>> c is d
False
>>> c[0] is d[0]
True
```

76.3. Глубокое копирование

Если у вас есть вложенные списки, то желательно клонировать и вложенные списки. Это действие называется глубоким копированием.

```
>>> import copy
>>> c = [[1,2]]
>>> d = copy.deepcopy(c)
>>> c is d
False
>>> c[0] is d[0]
False
```

76.4. Неглубокое копирование списка

С помощью срезов можно создавать неглубокие копии списков.

```
>>> l1 = [1,2,3]
>>> l2 = l1[:] # Выполнить неглубокое копирование.
>>> l2
[1,2,3]
>>> l1 == l2
False
```

76.5. Копирование набора

Наборы также имеют метод `copy`. С помощью этого метода можно выполнить неглубокое копирование.

```
>>> s1 = {}
>>> s2 = s1.copy()
>>> s1 is s2
False
>>> s2.add(3)
>>> s1
{}
>>> s2
{3,}
```

Глава 77. Менеджеры контекста (инструкция `with`)

Несмотря на широкое распространение контекстных менеджеров Python, не все понимают, с какой целью они используются. Эти операторы, обычно используемые при чтении и записи файлов, помогают приложению экономить системную память и улучшают управление ресурсами, обеспечивая использование определенных ресурсов только для определенных процессов.

77.1. Введение в менеджеры контекста и инструкцию `with`

Менеджер контекста – это объект, который уведомляет о *начале* и *завершении* контекста (блока кода). Обычно он используется вместе с оператором `with`. Он берет на себя все заботы по уведомлению.

Например, файловые объекты являются менеджерами контекста. Когда контекст заканчивается, файловый объект автоматически закрывается:

```
open_file = open(filename)
with open_file:
    file_contents = open_file.read()
```

объект `open_file` был автоматически закрыт.

Приведенный выше пример обычно упрощается за счет использования ключевого слова `as`:

```
open(filename) as open_file:
    file_contents = open_file.read()
```

объект `open_file` был автоматически закрыт.

Все, что завершает выполнение блока, приводит к вызову метода `exit` менеджера контекста. Это включает исключения (exceptions) и может быть полезно, когда ошибка заставляет преждевременно выйти из открытого файла или соединения. Выход из скрипта без надлежащего закрытия файлов и соединений – плохая идея, которая может привести к потере данных или другим проблемам. Использование менеджера контекста позволяет гарантировать, что всегда будут приняты меры предосторожности, чтобы предотвратить повреждение или потерю данных таким образом. Эта возможность была добавлена в Python 2.5.

77.2. Написание собственного менеджера контекста

Менеджер контекста – это любой объект, реализующий два магических метода: `__enter__()` и `__exit__()` (хотя он может реализовывать и другие методы):

```
class AContextManager():
```

```
    def __enter__(self):
        print("Entered")
        # опционально возвращает объект
        return "A-instance"
```

```
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exited" + (" (with an exception)" if exc_type else ""))
        # возвращает True, если нужно подавить исключение
```

Если контекст завершается с исключением, то информация об этом исключении будет передана в виде тройки `exc_type, exc_value, traceback` (это те же переменные, которые возвращает функция `sys.exc_info()`). При нормальном выходе из программы все три аргумента будут равны `None`.

Если возникает исключение и оно передается в метод `__exit__`, то он может вернуть `True`, чтобы подавить исключение, или исключение будет повторно вызвано в конце функции `__exit__`.

```
with AContextManager() as a:
```

```
    print("a is %r" % a)
```

```
    # Вход
```

```
    # a is 'A-instance'
```

```
    # Выход
```

```
with AContextManager() as a:
```

```
    print("a is %d" % a)
```

```
    # Вход
```

```
    # Выход (с исключением)
```

```
    # Traceback (most recent call last):
```

```
    #   File "<stdin>", строка 2, in <module>
```

```
    # TypeError: %d format: %d format: a number is required, not str (требуется число, а не строка)
```

Обратите внимание, что во втором примере, несмотря на то, что исключение возникает в середине тела оператора `with`, обработчик `__exit__` все равно выполняется, прежде чем исключение распространится во внешнюю область.

Если вам нужен только метод `__exit__`, то можно вернуть экземпляр менеджера контекста:

```
class MyContextManager:
```

```
    def __enter__(self):
```

```
        return self
```

```
    def __exit__(self):
```

```
        print('something')
```

77.3. Написание собственного менеджера контекста с использованием синтаксиса генератора

Также можно написать менеджер контекста, используя синтаксис генератора, благодаря декоратору `contextlib.contextmanager`:

```
import contextlib
```

```
@contextlib.contextmanager
def context_manager(num):
    print('Enter')
    yield num + 1
    print('Exit')
```

```
with context_manager(2) as cm:
```

```
    # следующие инструкции выполняются при достижении точки "yield" менеджера контекста
    # 'cm' будет иметь значение, которое было получено
    print('Right in the middle with cm = {}'.format(cm))
```

производит:

```
Enter
Right in the middle with cm = 3
Exit
```

Декоратор упрощает задачу написания контекстного менеджера, преобразуя в него генератор. Все, что находится перед выражением `yield`, становится методом `__enter__`, полученное значение становится значением, возвращаемым генератором (которое может быть привязано к переменной при помощи выражения `with`), а все, что находится после выражения `yield`, становится методом `__exit__`. Если менеджеру контекста необходимо обработать исключение, то в генераторе можно написать блок `try...except...finally`, и любое исключение, возникшее в блоке `with`, будет обработано этим блоком исключений.

```
@contextlib.contextmanager
def error_handling_context_manager(num):
    print("Enter")
    try:
        yield num + 1
    except ZeroDivisionError:
        print("Caught error")
    finally:
        print("Cleaning up")
    print("Exit")
```

```
with error_handling_context_manager(-1) as cm:
```

```
    print("Dividing by cm = {}".format(cm))
    print(2 / cm)
```

Результат:

```
Enter
Dividing by cm = 0
Caught error
Cleaning up
Exit
```


77.4. Использование нескольких менеджеров контекста

Одновременно можно открыть несколько менеджеров контекста:

```
with open(input_path) as input_file, open(output_path, 'w') as output_file:
    # сделать что-то с обоими файлами.
    # например, скопировать содержимое input_file в output_file
    for line in input_file:
        output_file.write(line + '\n')
```

Это даст тот же эффект, что и вложенные контекстные менеджеры:

```
with open(input_path) as input_file:
    with open(output_path, 'w') as output_file:
        for line in input_file:
            output_file.write(line + '\n')
```

77.5. Назначение цели

Многие контекстные менеджеры при вводе возвращают объект. Вы можете присвоить этому объекту новое имя в with-выражении.

Например, использование соединения с базой данных в операторе with может привести к появлению объекта курсора:

```
with database_connection as cursor:
    cursor.execute(sql_query)
```

Файловые объекты возвращают сами себя, что позволяет в одном выражении и открыть объект файла, и использовать его в качестве менеджера контекста:

```
with open(filename) as open_file:
    file_contents = open_file.read()
```

77.6. Управление ресурсами

```
class File():
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        self.open_file = open(self.filename, self.mode)
        return self.open_file

    def __exit__(self, *args):
        self.open_file.close()
```

Метод `__init__()` настраивает объект, в данном случае задавая имя файла и режим его открытия. Метод `__enter__()` открывает и возвращает файл, а `__exit__()` просто закрывает его. Использование этих магических методов (`__enter__`, `__exit__`) позволяет реализовать объекты, которые можно легко использовать с `with`. Используем класс `File`:

```
for _ in range(10000):
    with File('foo.txt', 'w') as f:
        f.write('foo')
```

Глава 78. Специальная переменная `__name__`

Специальная переменная `__name__` используется для проверки того, был ли файл импортирован как модуль, а также для идентификации функции, класса, объекта модуля по их атрибуту `__name__`.

78.1. проверка `__name__ == '__main__'`

Имя специальной переменной `__name__` не задается пользователем. В основном она используется для проверки того, запускается ли модуль сам по себе или потому, что был выполнен `import`. Чтобы ваш модуль не запускал определенные части своего кода при импорте, проверьте, что `if __name__ == '__main__':`.

Пусть в файле **module1.py** будет всего одна строка:

```
import module2.py
```

И посмотрим, что произойдет в зависимости от **module2.py**

Ситуация 1 **module2.py**

```
print('hello')
```

Запуск **module1.py** выведет hello

Запуск **module2.py** выведет hello

Ситуация 2 **module2.py**

```
if __name__ == '__main__':  
    print('hello')
```

Запуск **module1.py** ничего не выведет

Запуск **module2.py** выведет hello

78.2. Использование в протоколировании

При настройке встроенной функции протоколирования (logging) обычно создается регистратор (logger) с именем текущего модуля (`__name__`):

```
logger = logging.getLogger(__name__)
```

Это означает, что в протоколах будет отображаться полное имя модуля, что облегчит понимание того, откуда поступили сообщения.

78.3. `function_class_or_module.__name__`

Специальный атрибут `__name__` функции, класса или модуля – это строка, содержащая его имя.

```
import os  
class C:  
    pass
```

```
def f(x):  
    x += 2  
    return x
```

```

print(f)
# <function f at 0x029976B0>
print(f.__name__)
# f

print(C)
# <class '__main__.C'>
print(C.__name__)
# C

print(os)
# <module 'os' from '/spam/eggs/'>
print(os.__name__)
# os

```

Атрибут `__name__` – не имя переменной, ссылающейся на класс, метод или функцию, а имя, присвоенное ей при создании.

```

def f():
    pass

print(f.__name__)
# f - как и ожидалось

g = f
print(g.__name__)
# f - несмотря на то, что переменная названа g, функция все равно называется f

```

Это может быть использовано, в частности, для отладки:

```

def enter_exit_info(func):
    def wrapper(*arg, **kw):
        print '-- entering', func.__name__
        res = func(*arg, **kw)
        print '-- exiting', func.__name__
        return res
    return wrapper

```

```

@enter_exit_info
def f(x):
    print 'In:', x
    res = x + 2
    print 'Out:', res
    return res

```

```

a = f(2)

```

```

# Результат:
# -- entering f
# In: 2
# Out: 4
# -- exiting f

```

Глава 79. Проверка существования пути и прав доступа

Параметр	Подробности
<code>os.F_OK</code>	Значение, которое нужно передать в качестве параметра режима функции <code>access()</code> для проверки существования пути
<code>os.R_OK</code>	Значение, включаемое в параметр режима функции <code>access()</code> для проверки читаемости пути
<code>os.W_OK</code>	Значение, включаемое в параметр режима функции <code>access()</code> для проверки возможности записи пути
<code>os.X_OK</code>	Значение, включаемое в параметр режима функции <code>access()</code> для определения возможности выполнения пути

79.1. Выполнение проверки с помощью `os.access`

`os.access` – лучшее решение для проверки того, существует ли каталог и доступен ли он для чтения и записи.

```
import os
path = "/home/myFiles/directory1"

## Проверка существования пути
os.access(path, os.F_OK)

## Проверьте, является ли путь доступным для чтения
os.access(path, os.R_OK)

## Проверьте, является ли путь доступным для записи
os.access(path, os.W_OK)

## Проверьте, является ли путь исполняемым
os.access(path, os.E_OK)
```

Также возможно выполнение всех проверок совместно:

```
os.access(path, os.F_OK & os.R_OK & os.W_OK & os.E_OK)
```

Все вышеперечисленное возвращает `True`, если доступ разрешен, и `False`, если не разрешен. Проверки доступны для Unix и Windows.

Глава 80. Создание пакетов Python

80.1. Введение

Для каждого пакета требуется файл `setup.py`, который его описывает. Рассмотрим следующую структуру каталогов для простого пакета:

```
+-- package_name
|   |
|   +-- __init__.py
|
+-- setup.py
```

В файле `init.py` содержится только строка `def foo(): return 100`. Следующий файл `setup.py` определяет пакет:

```
from setuptools import setup

setup(
    name='package_name',           # имя пакета
    version='0.1',                 # версия
    description='Package Description', # краткое описание
    url='http://example.com',      # URL пакета
    install_requires=[],           # список пакетов, от которых зависит данный пакет

    packages=['package_name'],     # Список имен модулей, которые
                                # этот пакет предоставляет.
```

`virtualenv` отлично подходит для тестирования установки пакетов без изменения других окружений Python:

```
$ virtualenv .virtualenv
...
$ source .virtualenv/bin/activate
$ python setup.py install
running install
...
Installed .../package_name-0.1-....egg
...
$ python
>>> import package_name
>>> package_name.foo()
100
```

80.2. Загрузка в каталог пакетов PyPI

После того как файл `setup.py` полностью готов к работе (см. “Введение”), очень просто загрузить свой пакет в PyPI.

Настройка файла `.pypirc`

В этом файле хранятся логины и пароли для аутентификации учетных записей. Обычно он хранится в домашнем каталоге.

```
# .pypirc file
[distutils]
index-servers =
    pypi
    pypitest

[pypi]
repository=https://pypi.python.org/pypi
username=your_username
password=your_password

[pypitest]
repository=https://testpypi.python.org/pypi
username=your_username
password=your_password
```

Для загрузки пакетов безопаснее использовать `twine`, поэтому убедитесь, что он установлен.

```
$ pip install twine
```

Регистрация и загрузка в testpypi (необязательно)

Примечание: PyPI не позволяет перезаписывать загруженные пакеты, поэтому целесообразно сначала протестировать свою установку на специальном тестовом сервере, например testpypi. Этот вариант будет рассмотрен далее. Перед загрузкой продумайте схему версионирования пакета, например, версионирование по календарю или семантическое версионирование.

Либо войдите в систему, либо создайте новую учетную запись на testpypi:

```
$ python setup.py register -r pypitest
```

находясь в корневом каталоге своего пакета:

```
$ twine upload dist/* -r pypitest
```

Теперь ваш пакет должен быть доступен через вашу учетную запись.

Тестирование

Создайте тестовое виртуальное окружение. Попробуйте установить свой пакет с помощью `pip install` из testpypi или PyPI.

```
# Использование virtualenv
$ mkdir testenv
$ cd testenv
$ virtualenv .virtualenv

$ source .virtualenv/bin/activate
# Тест из testpypi
(.virtualenv) pip install --verbose --extra-index-url https://testpypi.python.org/pypi
package_name

# или тест из PyPI
(.virtualenv) $ pip install package_name

(.virtualenv) $ python
Python 3.5.1 (default, Jan 27 2016, 19:16:39)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import package_name
>>> package_name.foo()
100
```

Если это удалось, то ваш пакет в наименьшей степени пригоден для импорта. Перед окончательной загрузкой в PyPI можно также протестировать API. Если при тестировании пакета произошел сбой, не волнуйтесь. Вы все равно можете исправить его, заново загрузить в testpypi и протестировать снова.

Регистрация и загрузка в PyPI

Убедитесь, что twine установлен:

```
$ pip install twine
```

Войдите в систему либо создайте новую учетную запись в PyPI:

```
$ python setup.py register -r pypi
```

```
$ twine upload dist/*
```

Вот и все! Ваш пакет загружен.

Если вы обнаружили ошибку, просто загрузите новую версию своего пакета.

Документация

Не забудьте включить в пакет хотя бы какую-нибудь документацию. В качестве языка форматирования PyPI по умолчанию использует reStructuredText.

Readme

Если в вашем пакете небольшое количество документации, включите в файл README.rst то, что может помочь другим пользователям. Когда файл будет готов, необходим еще один файл, чтобы указать PyPI отображать его. Создайте файл setup.cfg и поместите в него эти две строки:

```
[metadata]
description-file = README.rst
```

Обратите внимание, что если вы попытаетесь поместить в пакет текст в формате Markdown, PyPI прочтает его как чистый текст без какого-либо форматирования.

Лицензирование

Часто бывает более чем желательно поместить в пакет файл LICENSE.txt с одной из OpenSource-лицензий, чтобы сообщить пользователям, могут ли они использовать ваш пакет, например, в коммерческих проектах, или если ваш код пригоден для использования с их лицензией.

80.3. Создание исполняемого пакета

Если ваш пакет представляет собой не только библиотеку, но и содержит код, который может быть использован как демонстрационный пример либо самостоятельное приложение после установки пакета, поместите эту часть кода в файл __main__.py. Поместите этот файл в папку package_name. Так вы сможете запускать его непосредственно из консоли:

```
python -m package_name
```

Если в пакете отсутствует файл __main__.py, то он не будет запущен с помощью этой команды и будет выведена ошибка:

```
python: No module named package_name.__main__; 'package_name' is a package and cannot be directly
executed. (Нет модуля с именем package_name.main ; 'package_name' является пакетом и не может
быть выполнен напрямую).
```

Глава 81. Использование модуля “pip”: Менеджер пакетов PyPI

Иногда возникает необходимость использовать менеджер пакетов pip внутри Python, например, когда при импорте возникает ошибка ImportError и необходимо обработать это исключение. При распаковке в Windows Python_root/Scripts/pip.exe внутри находится файл __main__.py, в котором импортируется класс main из пакета pip. Это означает, что пакет pip используется всякий раз, когда вы применяете исполняемый файл pip. Об использовании pip в качестве исполняемого файла см. главу 82 “pip: Менеджер пакетов PyPI”.

81.1. Пример использования команд

```
import pip
command = 'install'
parameter = 'selenium'
second_param = 'numpy' # можете указать столько имен пакетов, сколько необходимо
switch = '-upgrade'
```

```
pip.main([command, parameter, second_param, switch])
```

Обязательными являются только необходимые параметры, поэтому допустимы как pip.main(['freeze']), так и pip.main(['freeze', "", "]).

Пакетная установка

В одном вызове можно передать много имен пакетов, но при неудачной установке или обновлении одного из них весь процесс установки останавливается и завершается со статусом '1'.

```
import pip

installed = pip.get_installed_distributions()
list = []
for i in installed:
    list.append(i.key)

pip.main(['install']+list+['--upgrade'])
```

Если вы не хотите останавливаться при неудаче некоторых установок, вызовите установку в цикле.

```
for i in installed:
    pip.main(['install']+i.key+['--upgrade'])
```

81.2. Обработка исключения ImportError

При использовании файла Python в качестве модуля нет необходимости постоянно проверять, установлен ли пакет, но для скриптов это все равно полезно.

```
if __name__ == '__main__':
    try:
        import requests
    except ImportError:
        print("To use this module you need 'requests' module")
        t = input("Install requests? y/n: ")
        if t == 'y':
            import pip
            pip.main(['install', 'requests'])
            import requests
            import os
            import sys
            pass
        else:
            import os
            import sys
            print("Some functionality can be unavailable.")
    else:
        import requests
        import os
        import sys
```

81.3. Принудительная установка

Многие пакеты, например версии 3.4, будут работать на 3.6 просто отлично, но если нет дистрибутивов для конкретной платформы, то они не могут быть установлены; однако есть обходной путь. В соглашениях об именовании файлов .whl (известных как wheels) определяется, можно ли установить пакет на указанную платформу. Например, scikit-learn-0.18.1-cp36-cp36m-win_amd64.whl[package_name]-[version]-[python interpreter]-[pythoninterpreter]-[Operating System].whl. Если имя .whl-файла изменено так, что платформа совпадает, pip попытается установить пакет, даже если платформа или версия Python не совпадают. Удаление платформы или интерпретатора из имени приведет к ошибке в новейшей версии модуля pip (kjfhkjdf.whl is not a valid wheel filename).

В качестве альтернативы .whl можно распаковать с помощью архиватора типа 7-zip. Обычно он содержит метапакет дистрибутива и папку с исходными файлами. Эти исходные файлы можно просто распаковать в каталог site-packages, если только в этом .whl не содержится инсталляционный скрипт – если да, то его необходимо запустить.

Глава 82. `pip`: менеджер пакетов PyPI

`pip` – наиболее распространенный менеджер пакетов Python Package Index, устанавливаемый по умолчанию с последними версиями Python.

82.1. Установка пакетов

Для установки последней версии пакета с именем `SomePackage`:

```
$ pip install SomePackage
```

Для установки определенной версии пакета:

```
$ pip install SomePackage==1.0.4
```

Чтобы указать минимальную версию для установки пакета:

```
$ pip install SomePackage>=1.0.4
```

Если в Linux/Unix команды выдают ошибку `permission denied`, используйте `sudo` с командами.

Установка из файлов `requirements`

```
$ pip install -r requirements.txt
```

Каждая строка файла `requirements` указывает на то, что должно быть установлено, и подобна аргументам для `pip install`. После установки пакета вы можете проверить его с помощью команды `freeze`:

```
$ pip freeze
```

82.2. Получение списка пакетов, установленных с помощью `pip`

Для получения списка установленных пакетов:

```
$ pip list
# пример вывода
docutils (0.9.1)
Jinja2 (2.6)
Pygments (1.5)
Sphinx (1.1.2)
```

Перечислить устаревшие пакеты и показать последнюю доступную версию:

```
$ pip list --outdated
# пример вывода
docutils (Current: 0.9.1 Latest: 0.10)
Sphinx (Current: 1.1.2 Latest: 1.1.3)
```

82.3. Обновление пакетов

Запустите

```
$ pip install --upgrade SomePackage
```

это обновит пакет `SomePackage` и все его зависимости. Кроме того, `pip` автоматически удаляет старую версию пакета перед обновлением.

Для обновления самого `pip` выполните следующие действия:

```
$ pip install --upgrade pip
```

на Unix или

```
$ python -m pip install --upgrade pip
```

для Windows.

82.4. Удаление пакетов

Чтобы удалить пакет, выполните:

```
$ pip uninstall SomePackage
```

82.5. Обновление всех устаревших пакетов в Linux

В настоящее время в `pip` нет возможности обновить все устаревшие пакеты сразу. Однако в среде Linux это можно сделать путем объединения команд:

```
pip list --outdated --local | grep -v '^\\-e' | cut -d = -f 1 | xargs -n1 pip install -U
```

Вышеприведенное берет все пакеты в локальной виртуальной среде и проверяет, не устарели ли они. Из этого списка она получает имя пакета и передает его команде `pip install -U`. В конце этого процесса все локальные пакеты должны быть обновлены.

82.6. Обновление всех устаревших пакетов под Windows

В настоящее время в `pip` нет возможности обновить все устаревшие пакеты сразу. Однако это можно сделать путем объединения команд:

```
for /F "delims= " %i in ('pip list --outdated --local') do pip install -U %i
```

Вышеприведенное берет все пакеты в локальной виртуальной среде и проверяет, не устарели ли они. Из этого списка она получает имя пакета и передает его команде `pip install -U`. В конце этого процесса все локальные пакеты должны быть обновлены.

82.7. Создание файла requirements.txt file всех пакетов в системе

`pip` помогает создавать файлы `requirements.txt`, предоставляя опцию `freeze`.

```
pip freeze > requirements.txt
```

В результате список всех пакетов и их версий, установленных в системе, будет сохранен в текущей папке в виде файла с именем `requirements.txt`.

82.8. Использование определенной версии Python с помощью pip

Если у вас установлены и Python 3, и Python 2, вы можете указать, какую версию Python вы хотите использовать в `pip`. Это полезно, когда пакеты поддерживают только Python 2 или 3 или когда вы хотите тестировать на обеих версиях.

Если вы хотите установить пакеты для Python 2, выполните одно из следующих действий:

```
pip install [package]
```

или:

```
pip2 install [package]
```

Если вы хотите установить пакеты для Python 3, сделайте это:

```
pip3 install [package]
```

Также можно вызвать установку пакета на конкретную установку Python с помощью команды:

```
\\path\\to\\that\\python.exe -m pip install some_package # Windows  
/usr/bin/python25 -m pip install some_package # OS-X/Linux
```

На платформах OS-X/Linux/Unix важно знать различие между системной версией Python (обновление которой может сделать вашу систему неработоспособной) и пользовательской версией (версиями) Python. В зависимости от того, какую версию вы пытаетесь обновить, вам может потребоваться предварительно включить эти команды с помощью `sudo` и ввести пароль.

Аналогичным образом в Windows некоторые установки Python, особенно те, которые являются частью другого пакета, могут оказаться установленными в системных каталогах – их

придется обновлять из командного окна, запущенного в режиме администратора. Если вам кажется, что это необходимо сделать, то очень хорошо проверить, какую установку python вы пытаетесь обновить с помощью такой команды, как `python -c "import sys;print(sys.path);"` или `py -3.5 -c "import sys;print(sys.path);"`. Вы также можете проверить, какой pip вы пытаетесь запустить с помощью `pip --version`.

В Windows, если у вас установлены и Python 2, и Python 3, и в вашем пути и ваш Python 3 выше, чем версия 3.4, то, скорее всего, в вашем системном пути также будет находиться `python launcher.py`. Тогда вы сможете проделать следующее:

```
py -3 -m pip install -U some_package
# Установка/обновление some_package до последней версии Python 3
py -3.3 -m pip install -U some_package
# Установка/обновление some_package до Python 3.3, если он присутствует
```

```
py -2 -m pip install -U some_package
# Установка/обновление some_package до последней версии python 2 - 64 bit, если присутствует
```

```
py -2.7-32 -m pip install -U some_package
# Установка/обновление some_package до python 2.7 - 32 бит, если присутствует
```

Если вы используете и поддерживаете несколько версий Python, настоятельно рекомендуется ознакомиться с виртуальными окружениями Python `virtualenv` или `venv`, которые позволяют изолировать версию Python и присутствующие пакеты.

82.9. Создание файла requirements.txt file пакетов только текущей virtualenv

pip помогает создавать файлы `requirements.txt`, предоставляя опцию `freeze`.

```
pip freeze --local > requirements.txt
```

Параметр `--local` выводит список только тех пакетов и версий, которые установлены локально в текущей виртуальной среде `virtualenv`. Глобальные пакеты не будут указаны.

82.10. Установка пакетов, еще не включенных в pip в качестве wheels

Многие пакеты еще не доступны в индексе пакетов Python (Python Package Index) в виде “колес” (wheels), но все равно отлично устанавливаются. Однако некоторые пакеты под Windows выдают ошибку `vcvarsall.bat not found`.

Проблема заключается в том, что пакет, который вы пытаетесь установить, содержит расширение C или C++ и *в настоящее время* не доступен в виде предварительно собранного “колеса” из индекса пакетов Python, `pypi`, а на Windows у вас нет цепочки инструментов, необходимой для сборки таких элементов.

Самый простой ответ – зайти на сайт Кристофа Гольке и найти *соответствующую* версию нужных библиотек. Соответственно, в имени пакета `-cpNN-` должно соответствовать вашей версии Python, т.е. если вы используете Windows 32 bit python *даже на Win64*, то в имени должно быть указано `-win32-`, а если 64 bit Python, то `-win_amd64-`, а затем должна соответствовать версия Python, т.е. для Python 3.4 в имени *должно быть* указано `-cp34-` и т. д. В принципе, это та “магия”, которую делает за вас pip на сайте `pypi`.

В качестве альтернативы необходимо получить соответствующий комплект средств разработки Windows (Windows development kit) для используемой версии Python, заголовки всех библиотек, с которыми взаимодействует пакет, возможно, заголовки Python для данной версии Python и т. д.

В Python 2.7 использовалась Visual Studio 2008, в Python 3.3 и 3.4 – Visual Studio 2010, а в Python 3.5+ – Visual Studio 2015.

- Установите Visual C++ Compiler Package for Python 2.7, который доступен на сайте Microsoft, или

- Установите Windows SDK for Windows 7 and .NET Framework 4 (v7.1), доступный на сайте Microsoft, или
- Установите Visual Studio 2015 Community Edition (или любую другую более позднюю версию), *убедившись, что вы выбрали опцию установки поддержки языков C и C++.*

Затем вам может потребоваться найти файлы заголовков в соответствующей ревизии для всех библиотек, на которые ссылается нужный вам пакет, и загрузить их в соответствующие места.

Наконец, вы можете позволить `pip` выполнить сборку – конечно, если пакет имеет зависимости, которых у вас еще нет, вам может потребоваться найти файлы заголовков и для них.

Альтернативы: Также стоит поискать на `ruり` или на сайте Кристофа более ранние версии искомого пакета на чистом Python или предварительно собранные для вашей платформы и версии Python и, возможно, использовать их, если они будут найдены, пока ваш пакет не станет доступен. Аналогичным образом, если вы используете самую последнюю версию Python, может оказаться, что сопровождающим пакета требуется некоторое время, чтобы догнать ее, поэтому для проектов, которым действительно необходим конкретный пакет, вам, возможно, придется использовать немного более старый Python на данный момент. Вы также можете проверить исходный сайт пакета на наличие форк-версии, которая доступна в виде предварительной сборки или чистого Python, и поискать альтернативные пакеты, которые предоставляют необходимую вам функциональность, но доступны – один из примеров, который приходит на ум, это активно поддерживаемый `Pillow`, заменяющий `PIL`, который в настоящее время не обновляется уже 6 лет и недоступен для Python 3.

После этого всем, кто столкнулся с подобной проблемой, хорошо было бы зайти на баг-трекер пакета и добавить или создать, если такового еще нет, тикет с вежливой просьбой к сопровождающим пакета предоставить `wheel` на `ruри` для вашей конкретной комбинации платформы и Python; если это будет сделано, то обычно со временем ситуация улучшается, некоторые сопровождающие пакетов не понимают, что они упустили ту или иную комбинацию.

Замечания по установке предварительных версий

`Pip` следует правилам семантического версионирования и по умолчанию отдает предпочтение выпущенным пакетам перед предварительными версиями. Так, если данный пакет был выпущен под номером `V0.98`, а также существует релиз-кандидат `V1.0-rc1`, то `pip install` по умолчанию установит `V0.98` – если вы хотите установить релиз-кандидат, *рекомендуется сначала протестировать его в виртуальной среде*, что можно сделать с помощью команды `–pip install –pre package-name` или `–pip install –pre – upgrade package-name`. Во многих случаях предварительные версии или релизы-кандидаты могут не иметь “колес”, собранных для всех комбинаций платформ и версий, поэтому вероятность возникновения описанных выше проблем больше.

Замечание по установке версий для разработчиков

Вы также можете использовать `pip` для установки версий пакетов из `github` и других мест. Очень маловероятно, что для них собраны “колеса”, поэтому любые подобные пакеты потребуют наличия инструментов сборки, и они могут быть сломаны в любой момент, поэтому пользователю **настоятельно** рекомендуется устанавливать такие пакеты только в виртуальной среде.

Для таких установок существует три варианта:

1. Загрузить сжатый снимок (`compressed snapshot`), в большинстве онлайн-систем контроля версий есть возможность загрузить сжатый снимок кода. Его можно загрузить вручную, а затем установить с помощью команды `pip install path/to/downloaded/file`. Обратите внимание, что для большинства форматов сжатия `pip` справится с распаковкой в область кэша и т. д.

2. Позвольте `pip` выполнить загрузку и установку за вас с помощью команды: `pip install URL/of/package/repository` – для корректной работы, особенно в корпоративной среде, вам также может потребоваться использовать `–trusted-host`, `–client-cert` и/или `–proxy` флаги:

```

> py -3.5-32 -m venv demo-pip
> demo-pip\Scripts\activate.bat
> python -m pip install -U pip
Collecting pip
Using cached pip-9.0.1-py2.py3-none-any.whl
Installing collected packages: pip
Found existing installation: pip 8.1.1
Uninstalling pip-8.1.1:
Successfully uninstalled pip-8.1.1
Successfully installed pip-9.0.1
> pip install git+https://github.com/sphinx-doc/sphinx/
Collecting git+https://github.com/sphinx-doc/sphinx/
Cloning https://github.com/sphinx-doc/sphinx/ to c:\users\steve-
~1\appdata\local\temp\pip-04yn9hpp-build
Collecting six>=1.5 (from Sphinx==1.7.dev20170506)
Using cached six-1.10.0-py2.py3-none-any.whl
Collecting Jinja2>=2.3 (from Sphinx==1.7.dev20170506)
Using cached Jinja2-2.9.6-py2.py3-none-any.whl
Collecting Pygments>=2.0 (from Sphinx==1.7.dev20170506)
Using cached Pygments-2.2.0-py2.py3-none-any.whl
Collecting docutils>=0.11 (from Sphinx==1.7.dev20170506)
Using cached docutils-0.13.1-py3-none-any.whl
Collecting snowballstemmer>=1.1 (from Sphinx==1.7.dev20170506)
Using cached snowballstemmer-1.2.1-py2.py3-none-any.whl
Collecting babel!=2.0,>=1.3 (from Sphinx==1.7.dev20170506)
Using cached Babel-2.4.0-py2.py3-none-any.whl
Collecting alabaster<0.8,>=0.7 (from Sphinx==1.7.dev20170506)
Using cached alabaster-0.7.10-py2.py3-none-any.whl
Collecting imagesize (from Sphinx==1.7.dev20170506)
Using cached imagesize-0.7.1-py2.py3-none-any.whl
Collecting requests>=2.0.0 (from Sphinx==1.7.dev20170506)
Using cached requests-2.13.0-py2.py3-none-any.whl
Collecting typing (from Sphinx==1.7.dev20170506)
Using cached typing-3.6.1.tar.gz
Requirement already satisfied: setuptools in f:\toolbuild\temp\demo-pip\lib\site-packages (from
Sphinx==1.7.dev20170506)
Collecting sphinxcontrib-websupport (from Sphinx==1.7.dev20170506)
Downloading sphinxcontrib_websupport-1.0.0-py2.py3-none-any.whl
Collecting colorama>=0.3.5 (from Sphinx==1.7.dev20170506)
Using cached colorama-0.3.9-py2.py3-none-any.whl
Collecting MarkupSafe>=0.23 (from Jinja2>=2.3->Sphinx==1.7.dev20170506)
Using cached MarkupSafe-1.0.tar.gz
Collecting pytz>=0a (from babel!=2.0,>=1.3->Sphinx==1.7.dev20170506)
Using cached pytz-2017.2-py2.py3-none-any.whl
Collecting sqlalchemy>=0.9 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading SQLAlchemy-1.1.9.tar.gz (5.2MB)
100% |#####| 5.2MB 220kB/s
Collecting whoosh>=2.0 (from sphinxcontrib-websupport->Sphinx==1.7.dev20170506)
Downloading Whoosh-2.7.4-py2.py3-none-any.whl (468kB)
100% |#####| 471kB 1.1MB/s
Installing collected packages: six, MarkupSafe, Jinja2, Pygments, docutils, snowballstemmer, pytz,
babel, alabaster, imagesize, requests, typing, sqlalchemy, whoosh, sphinxcontrib-websupport,
colorama, Sphinx
Running setup.py install for MarkupSafe ... done
Running setup.py install for typing ... done
Running setup.py install for sqlalchemy ... done
Running setup.py install for Sphinx ... done
Successfully installed Jinja2-2.9.6 MarkupSafe-1.0 Pygments-2.2.0 Sphinx-1.7.dev20170506

```

alabaster-0.7.10 babel-2.4.0 colorama-0.3.9 docutils-0.13.1 imagesize-0.7.1 pytz-2017.2
 requests-2.13.0 six-1.10.0 snowballstemmer-1.2.1 sphinxcontrib-websupport-1.0.0 sqlalchemy-1.1.9
 typing-3.6.1 whoosh-2.7.4

Обратите внимание на префикс `git+` в URL

3. Клонировать репозиторий с помощью `git`, `mercurial` или другого приемлемого инструмента, желательно `DVCS`, и используйте `pip install path/to/cloned/repo` – это позволит обработать любой файл `requires.txt` и выполнить сборку и настройку. Вы можете вручную изменить каталог на клонированный репозиторий и выполнить `pip install -r requires.txt`, а затем `python setup.py install`, чтобы получить тот же эффект. Большим преимуществом такого подхода является то, что, хотя первоначальная операция клонирования может занять больше времени, чем загрузка снимка, вы можете обновить его до последней версии с помощью, в случае `git`: `git pull origin master`, а если текущая версия содержит ошибки, вы можете использовать `pip uninstall package-name`, а затем использовать команды `git checkout` для возврата в историю репозитория к более ранней версии (или версиям) и повторить попытку.

Глава 83. Разбор аргументов командной строки

Большинство инструментов командной строки полагаются на аргументы, передаваемые программе при ее выполнении. Вместо того чтобы запрашивать ввод эти программы ожидают, что будут заданы данные или специальные метки (которые становятся булевыми значениями). Это позволяет как пользователю, так и другим программам запускать файл `Python`, передавая ему данные при запуске. В этой главе рассказывается и демонстрируется реализация и использование аргументов командной строки в `Python`.

83.1. Hello world в argparse

Следующая программа приветствует пользователя. Она принимает один позиционный аргумент – имя пользователя, и ей также можно передать приветствие.

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument('name',
                    help='name of user')

parser.add_argument('-g', '--greeting',
                    default='Hello',
                    help='optional alternate greeting')

args = parser.parse_args()

print("{greeting}, {name}!".format(
    greeting=args.greeting,
    name=args.name))

$ python hello.py --help
usage: hello.py [-h] [-g GREETING] name
```


positional arguments:

name name of user

optional arguments:

-h, --help show this help message and exit

-g GREETING, --greeting GREETING
 optional alternate greeting

\$ python hello.py world

Hello, world!

\$ python hello.py John -g Howdy

Howdy, John!

83.2. Использование аргументов командной строки с помощью argv

При вызове скрипта Python из командной строки пользователь может указать дополнительные **аргументы командной строки**, которые будут переданы скрипту. Эти аргументы будут доступны программисту из системной переменной `sys.argv` ("`argv`" – это традиционное название, используемое в большинстве языков программирования, и означает оно "**argument vector**" ("вектор аргументов")).

По условию первым элементом списка `sys.argv` является имя самого Python-скрипта, а остальные элементы – это лексемы, токены, передаваемые пользователем при вызове скрипта.

```
# cli.py
import sys
print(sys.argv)
```

```
$ python cli.py
=> ['cli.py']
```

```
$ python cli.py fizz
=> ['cli.py', 'fizz']
```

```
$ python cli.py fizz buzz
=> ['cli.py', 'fizz', 'buzz']
```

Вот еще один пример использования `argv`. Сначала мы удаляем начальный элемент `sys.argv`, поскольку он содержит имя скрипта. Затем мы объединяем остальные аргументы в одно предложение и, наконец, выводим это предложение, добавляя к нему имя пользователя, вошедшего в систему (чтобы эмулировать чат).

```
import getpass
import sys

words = sys.argv[1:]
sentence = " ".join(words)
print("[%s] %s" % (getpass.getuser(), sentence))
```

При "ручном" разборе множества непозиционных аргументов обычно используется алгоритм итерации по списку `sys.argv`. Один из способов заключается в том, чтобы пройти по списку и вытолкнуть каждый его элемент:

```
# перевернуть и скопировать sys.argv
argv = reversed(sys.argv)
# извлечь первый элемент
arg = argv.pop()
# прекратить итерацию, когда больше нет аргументов для pop()
while len(argv) > 0:
    if arg in ('-f', '--foo'):
        print('seen foo!')
```

```

elif arg in ('-b', '--bar'):
    print('seen bar!')
elif arg in ('-a', '--with-arg'):
    arg = arg.pop()
    print('seen value: {}'.format(arg))
# получить следующее значение
arg = argv.pop()

```

83.3. Задание взаимоисключающих аргументов с помощью argparse

Если необходимо, чтобы два или более аргументов были взаимоисключающими, можно воспользоваться функцией `argparse.ArgumentParser.add_mutually_exclusive_group()`. В приведенном ниже примере может существовать либо `foo`, либо `bar`, но не оба одновременно.

```

import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-f", "--foo")
group.add_argument("-b", "--bar")
args = parser.parse_args()
print "foo = ", args.foo
print "bar = ", args.bar

```

Если попытаться запустить скрипт, указав оба аргумента: `--foo` и `--bar`, то скрипт выдаст следующее сообщение:

```

error: argument -b/--bar: not allowed with argument -f/--foo (аргумент -b/--bar: не разрешен
с аргументом -f/--foo)

```

83.4. Базовый пример с использованием docopt

`docopt` перевернул представление о разборе аргументов командной строки. Вместо разбора аргументов вы просто пишете *строку использования* для вашей программы, а `docopt` *разбирает эту строку* и использует ее для извлечения аргументов командной строки.

```

Usage:
  script_name.py [-a] [-b] <path>

Options:
  -a      Print all the things.
  -b      Get more bees into the path.

```

```

from docopt import docopt

if __name__ == "__main__":
    args = docopt(__doc__)
    import pprint; pprint.pprint(args)

```

Пробные прогоны:

```

$ python script_name.py
Usage:
  script_name.py [-a] [-b] <path>
$ python script_name.py something
{'-a': False,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py something -a

```



```
{'-a': True,
 '-b': False,
 '<path>': 'something'}
$ python script_name.py -b something -a
{'-a': True,
 '-b': True,
 '<path>': 'something'}
```

83.5. Пользовательское сообщение об ошибке синтаксического анализатора с помощью argparse

Вы можете создавать сообщения об ошибках парсера в соответствии с потребностями вашего скрипта. Это делается с помощью функции `argparse.ArgumentParser.error`. В приведенном ниже примере показано, как скрипт выводит в `stderr` сообщение об использовании и ошибке, когда задано `-foo`, но не задано `-bar`.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-f", "--foo")
parser.add_argument("-b", "--bar")
args = parser.parse_args()
if args.foo and args.bar is None:
    parser.error("--foo requires --bar. You did not specify bar.")

print "foo =", args.foo
print "bar =", args.bar
```

Предположим, что имя вашего скрипта `sample.py`, и мы запускаем его: `python sample.py --foo ds_in_fridge`.

Скрипт выдаст следующее сообщение:

```
usage: sample.py [-h] [-f FOO] [-b BAR]
sample.py: error: --foo requires --bar. You did not specify bar.
```

83.6. Концептуальная группировка аргументов с помощью `argparse.add_argument_group()`

Когда вы создаете `argparse.ArgumentParser()` и запускаете свою программу с параметром `-h`, вы получаете автоматическое сообщение об использовании, объясняющее, с какими аргументами можно запускать вашу программу. По умолчанию позиционные аргументы и условные аргументы разделяются на две категории; например, вот небольшой скрипт (`example.py`) и вывод при запуске `python example.py -h`.

```
import argparse

parser = argparse.ArgumentParser(description='Простой пример')
parser.add_argument('name', help='Кого приветствовать', default='World')
parser.add_argument('--bar_this')
parser.add_argument('--bar_that')
parser.add_argument('--foo_this')
parser.add_argument('--foo_that')
args = parser.parse_args()

usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                  name
```

Простой пример

```
позиционные аргументы
имя          Кого приветствовать
```

необязательные аргументы:

```
-h, --help      show this help message and exit
--bar_this BAR_THIS
--bar_that BAR_THAT
--foo_this FOO_THIS
--foo_that FOO_THAT
```

Бывают ситуации, когда для удобства пользователя необходимо разделить аргументы на дополнительные концептуальные разделы. Например, все параметры ввода можно отнести к одной группе, а все параметры форматирования вывода – к другой. Приведенный выше пример можно изменить таким образом, чтобы отделить аргументы `--foo_*` от аргументов `--bar_*`.

```
import argparse
```

```
parser = argparse.ArgumentParser(description='Простой пример')
parser.add_argument('name', help='Кого приветствовать', default='World')
# Создать две группы аргументов
foo_group = parser.add_argument_group(title='Foo options')
bar_group = parser.add_argument_group(title='Bar options')
# Добавляем аргументы в эти группы
foo_group.add_argument('--bar_this')
foo_group.add_argument('--bar_that')
bar_group.add_argument('--foo_this')
bar_group.add_argument('--foo_that')
args = parser.parse_args()
```

В результате при запуске `python example.py -h` получается такой вывод:

```
usage: example.py [-h] [--bar_this BAR_THIS] [--bar_that BAR_THAT]
                  [--foo_this FOO_THIS] [--foo_that FOO_THAT]
                  name
```

Простой пример

```
позиционные аргументы
имя          Кого приветствовать
```

необязательные аргументы:

```
-h, --help      show this help message and exit
```

Foo options:

```
--bar_this BAR_THIS
--bar_that BAR_THAT
```

Bar options:

```
--foo_this FOO_THIS
--foo_that FOO_THAT
```

83.7. Расширенный пример с docopt и docopt_dispatch

Как и в случае с `docopt`, при использовании `docopt_dispatch` вы создаете свой `--help` в переменной `__doc__` вашего начального модуля. Там вы вызываете `dispatch` со строкой документации в качестве аргумента, чтобы он мог запустить по ней парсер.

Таким образом, вместо того, чтобы вручную обрабатывать аргументы (что обычно приводит к появлению высокоциклической структуры `if/else`), вы предоставляете диспетчеру возможность указать, как именно следует обрабатывать набор аргументов. Для этого и предназначен декоратор `dispatch.on`: вы сообщаете ему аргумент или последовательность аргументов, которые должны вызвать функцию, и эта функция будет выполнена с соответствующими значениями в качестве параметров.

"""Run something in development or production mode.

Usage: run.py --development <host> <port>
 run.py --production <host> <port>
 run.py items add <item>
 run.py items delete <item>

```
from docopt_dispatch import dispatch
```

```
@dispatch.on('--development')
def development(host, port, **kwargs):
    print('in *development* mode')
```

```
@dispatch.on('--production')
def development(host, port, **kwargs):
    print('in *production* mode')
```

```
@dispatch.on('items', 'add')
def items_add(item, **kwargs):
    print('adding item...')
```

```
@dispatch.on('items', 'delete')
def items_delete(item, **kwargs):
    print('deleting item...')
```

```
if __name__ == '__main__':
    dispatch(__doc__)
```

Глава 84. Библиотека подпроцессов

Параметр	Подробности
args	Один исполняемый файл или последовательность исполняемых файлов и аргументов – 'ls', ['ls', '-la']
shell	Запускать под управлением оболочки? Оболочка по умолчанию – /bin/sh на POSIX.
cwd	Рабочий каталог дочернего процесса.

84.1. Больше возможностей с помощью Popen

Использование subprocess.Popen дает более тонкий контроль над запущенными процессами, чем subprocess.call.

Запуск подпроцесса

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg1'])
```

Сигнатура функции Popen очень похожа на сигнатуру функции call, однако Popen возвращается немедленно, а не дожидается завершения подпроцесса, как это делает call.

Ожидание завершения подпроцесса

```
process = subprocess.Popen([r'C:\path\to\app.exe', 'arg1', '--flag', 'arg1'])
process.wait()
```

Чтение вывода из подпроцесса

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
# Это будет блокироваться до завершения процесса
stdout, stderr = process.communicate()
print stdout
print stderr
```

Интерактивный доступ к запущенным подпроцессам

Вы можете читать и писать на `stdin` и `stdout` даже тогда, когда подпроцесс еще не завершен. Это может быть полезно при автоматизации работы в другой программе.

Запись в подпроцесс

```
process = subprocess.Popen([r'C:\path\to\app.exe'], stdout = subprocess.PIPE, stdin = subprocess.PIPE)
process.stdin.write('line of input\n') # Запись входных данных
line = process.stdout.readline() # Читаем строку из stdout
# Выполняем логику на прочитанной строке.
```

Однако если вам нужен только один набор входных и выходных данных, а не динамическое взаимодействие, то следует использовать `communicate()` вместо того, чтобы напрямую обращаться к `stdin` и `stdout`.

Чтение потока из подпроцесса

В случае, если вы хотите увидеть вывод подпроцесса построчно, можно воспользоваться следующим фрагментом кода:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.readline()
```

В случае, если в выходных данных подкоманды отсутствует символ EOF, приведенный фрагмент не работает. Тогда можно прочитать вывод посимвольно следующим образом:

```
process = subprocess.Popen(<your_command>, stdout=subprocess.PIPE)
while process.poll() is None:
    output_line = process.stdout.read(1)
```

Число 1, указанное в качестве аргумента метода `read`, указывает на то, что метод `read` должен считывать по одному символу за раз. Вы можете задать чтение любого количества символов, используя другое число. Отрицательное число или 0 указывает `read` читать как одну строку до тех пор, пока не будет встречено EOF. В обоих приведенных фрагментах `process.poll()` является `None`, пока подпроцесс не завершится. Это используется для выхода из цикла, когда больше нет вывода для чтения. Аналогичная процедура может быть применена и к `stderr` подпроцесса.

84.2. Вызов внешних команд

Наиболее простым вариантом является использование функции `subprocess.call`. В качестве первого аргумента она принимает список. Первым элементом списка должно быть внешнее приложение, которое вы хотите вызвать. Остальные элементы списка являются аргументами, которые будут переданы этому приложению.

```
subprocess.call([r'C:\path\to\app.exe', 'arg1', '-flag', 'arg'])
```

Для команд оболочки установите значение `shell=True` и предоставьте команду в виде строки, а не списка.

```
subprocess.call('echo "Hello, world"', shell=True)
```

Обратите внимание, что две приведенные выше команды возвращают только статус выхода из подпроцесса. Кроме того, будьте внимательны при использовании `shell=True`, по-

скольку это создает проблемы с безопасностью. Если вы хотите иметь возможность получать стандартный вывод подпроцесса, то замените вызов `subprocess.call` на `subprocess.check_output`.

84.3. Как создать аргумент списка команд

Метод подпроцесса, позволяющий выполнять команды, требует наличия команды в виде списка (по крайней мере, при использовании `shell_mode=True`).

Правила создания списка не всегда просты, особенно при работе со сложными командами. К счастью, существует очень полезный инструмент, позволяющий это сделать: `shlex`. Самый простой способ создания списка для использования в качестве команды заключается в следующем:

```
import shlex
cmd_to_subprocess = shlex.split(command_used_in_the_shell)
```

Простой пример:

```
import shlex
shlex.split('ls --color -l -t -r')

out: ['ls', '--color', '-l', '-t', '-r']
```

Глава 85. Файл setup.py

Параметр	Подробности
<code>name</code>	Название вашего дистрибутива
<code>version</code>	Строка версии вашего дистрибутива
<code>packages</code>	Список пакетов Python (то есть каталогов, содержащих модули) для включения. Он может быть задан вручную, но обычно вместо этого используется вызов <code>setuptools.find_packages()</code>
<code>py_modules</code>	Список модулей Python верхнего уровня (то есть одиночных <code>.py</code> файлов), которые необходимо включить

85.1. Назначение файла setup.py

Сценарий установки является центром всей деятельности по сборке, распространению и установке модулей с помощью `Distutils`. Его назначение – корректная установка программного обеспечения.

Если вы хотите распространять модуль с названием `foo`, содержащийся в файле `foo.py`, то ваш сценарий установки может быть простым:

```
from distutils.core import setup
```

```
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Для создания исходного дистрибутива этого модуля необходимо создать установочный скрипт `setup.py`, содержащий приведенный выше код, и выполнить эту команду из терминала:

```
python setup.py sdist
```

sdist создаст архивный файл (например, tarball на Unix, ZIP на Windows), содержащий ваш скрипт установки setup.py и ваш модуль foo.py. Архив будет иметь имя foo-1.0.tar.gz (или .zip) и распакуется в каталог foo-1.0.

Если конечный пользователь захочет установить ваш модуль foo, ему достаточно загрузить foo-1.0.tar.gz (или .zip), распаковать его и – из каталога foo-1.0 – запустить:

```
python setup.py install
```

85.2. Использование метаданных системы управления версиями в файле setup.py

setuptools_scm – это официально утвержденный пакет, который может использовать метаданные Git или Mercurial для определения номера версии вашего пакета, а также найти Python-пакеты и пакетные данные для включения в него.

```
from setuptools import setup, find_packages
setup(
    setup_requires=['setuptools_scm'],
    use_scm_version=True,
    packages=find_packages(),
    include_package_data=True,
```

В данном примере используются обе возможности; чтобы использовать только метаданные SCM для версии, замените вызов find_packages() своим списком пакетов или, чтобы использовать только поиск пакетов, удалите use_scm_version=True.

85.3. Добавление скриптов командной строки в пакет Python

Скрипты командной строки внутри пакетов Python – обычное явление. Вы можете организовать свой пакет таким образом, чтобы при установке пакета пользователем скрипт был доступен по его пути. Допустим, у вас есть пакет greetings, в котором есть сценарий командной строки hello_world.py.

```
greetings/
greetings/
  __init__.py
  hello_world.py
```

Вы можете запустить этот скрипт, выполнив команду:

```
python greetings/greetings/hello_world.py
```

Однако, возможно, вы хотите запустить его следующим образом:

```
hello_world.py
```

Этого можно добиться, добавив scripts в setup() в файле setup.py следующим образом:

```
from setuptools import setup
setup(
    name='greetings',
    scripts=['hello_world.py']
```

Когда вы установите пакет greetings сейчас, hello_world.py будет добавлен в ваш путь. Другой возможностью было бы добавить точку входа:

```
entry_points={'console_scripts': ['greetings=greetings.hello_world:main']}
```

Таким образом, вам нужно просто запустить его как:

```
greetings
```

85.4. Добавление параметров установки

Как видно из предыдущих примеров, основное использование этого скрипта заключается в следующем:

```
python setup.py install
```

Но есть и более широкие возможности, например, установить пакет и иметь возможность изменять код и тестировать его без необходимости повторной установки. Для этого используется:

```
python setup.py develop
```

Если вы хотите выполнять специфические действия, например, компилировать документацию *Sphinx* или собирать *fortran*-код, вы можете создать свою собственную опцию, например такую:

```
cmdclasses = dict()

class BuildSphinx(Command):

    """ Build Sphinx documentation."""

    description = 'Build Sphinx documentation'
    user_options = []

    def initialize_options(self):
        pass

    def finalize_options(self):
        pass

    def run(self):
        import sphinx
        sphinx.build_main(['setup.py', '-b', 'html', './doc', './doc/_build/html'])
        sphinx.build_main(['setup.py', '-b', 'man', './doc', './doc/_build/man'])

cmdclasses['build_sphinx'] = BuildSphinx

setup(
    ...
    cmdclass=cmdclasses,
    ...
)
```

`initialize_options` и `finalize_options` будут выполняться до и после функции `run`, как это следует из их названий. После этого вы сможете вызвать:

```
python setup.py build_sphinx
```

Глава 86. Рекурсия

86.1. Что, как и когда делать с рекурсией

Рекурсия возникает, когда вызов функции приводит к повторному вызову этой же функции до завершения вызова исходной функции. Например, рассмотрим известное математическое выражение $x!$ (т.е. операцию факториала). Для всех неотрицательных целых чисел операция факториала определяется следующим образом:

- Если число равно 0, то ответом будет 1.
- В противном случае ответом будет это число, умноженное на факториал числа, на единицу меньшего этого числа.

В языке Python реализация операции факториала может быть оформлена в виде функции следующим образом:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Функции рекурсии иногда бывают очень сложны для понимания, поэтому рассмотрим их пошагово. Рассмотрим выражение `factorial(3)`. Это выражение и все вызовы функции создают новое **окружение**. По сути, окружение – это таблица, в которой идентификаторы (например, `n`, `factorial`, `print` и т. д.) сопоставляются с соответствующими значениями. В любой момент времени с помощью функции `locals()` можно получить доступ к текущему окружению. В первом вызове функции единственной локальной переменной, которая определяется, является `n = 3`. Поэтому вывод `locals()` покажет `{'n': 3}`. Поскольку `n == 3`, возвращаемое значение становится `n * factorial(n - 1)`.

На следующем этапе может возникнуть некоторая путаница. Глядя на наше новое выражение, мы уже знаем, что такое `n`. Однако мы еще не знаем, что такое `factorial(n - 1)`. Сначала `n - 1` оценивается в 2. Затем 2 передается в `factorial` как значение для `n`. Поскольку это новый вызов функции, создается вторая среда для хранения этого нового `n`. Пусть `A` – первая среда, а `B` – вторая среда. `A` по-прежнему существует и равно `{'n': 3}`, однако `B` (которая равна `{'n': 2}`) является текущим окружением. Если посмотреть на тело функции, то возвращаемое значение снова равно `n * factorial(n - 1)`. Не оценивая это выражение, подставим его в исходное выражение возврата. При этом мы мысленно отбрасываем `B`, поэтому не забываем подставлять `n` соответствующим образом (т.е. ссылки на `n` из `B` заменяются на `n - 1`, в котором используется `n` из `A`). Теперь исходное выражение возврата становится `n * ((n - 1) * factorial((n - 1) - 1))`. Потратьте секунду на то, чтобы убедиться, что вы понимаете, почему это так.

Теперь оценим часть `factorial((n - 1) - 1)`. Так как в `A n == 3`, то в `factorial` мы передаем 1. Следовательно, мы создаем новое окружение `C`, которое равно `{'n': 1}`. И снова возвращаемое значение равно `n * factorial(n - 1)`. Поэтому заменим `factorial((n - 1) - 1)` “исходного” выражения возврата аналогично тому, как мы корректировали исходное выражение возврата ранее. Теперь “исходное” выражение имеет вид `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))`.

Почти все готово. Теперь нужно вычислить `factorial((n - 2) - 1)`. На этот раз мы передаем 0. Поэтому эта оценка равна 1. Теперь выполним последнюю подстановку. Теперь “исходное” возвращаемое выражение имеет вид `n * ((n - 1) * ((n - 2) * 1))`. Если вспомнить, что исходное выражение возврата оценивается по `A`, то выражение становится `3 * ((3 - 1) * ((3 - 2) * 1))`. Это, конечно же, равно 6. Чтобы убедиться в правильности ответа, вспомним, что $3! = 3 * 2 * 1 = 6$. Прежде чем читать дальше, убедитесь, что вы полностью понимаете концепцию окружений и их применение к рекурсии.

Оператор `if n == 0: return 1` называется базовым. Это связано с тем, что в нем нет рекурсии. Базовый случай (base case) абсолютно необходим. Без него вы столкнетесь с бесконечной рекурсией. При этом, если у вас есть хотя бы один базовый случай, вы можете иметь сколько угодно случаев. Например, мы могли бы эквивалентно записать `factorial` в виде:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Возможны также случаи множественной рекурсии, но мы не будем их рассматривать, так как они относительно редки и часто сложны для восприятия.

Также возможны “параллельные” вызовы рекурсивных функций. Например, рассмотрим последовательность Фибоначчи, которая определяется следующим образом:

- Если число равно 0, то ответ равен 0.
- Если число равно 1, то ответ равен 1.
- В противном случае ответом будет сумма двух предыдущих чисел Фибоначчи.

Это можно определить следующим образом:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Не будем рассматривать эту функцию так подробно, как это было с `factorial(3)`, но конечное возвращаемое значение `fib(5)` эквивалентно следующему (синтаксически неверному) выражению:

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

Таким образом, получается $(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))$, что, конечно же, равно 5. Теперь рассмотрим еще несколько словарных терминов:

- Хвостовой вызов (Tail call) – это просто вызов рекурсивной функции, который является последней операцией, выполняемой перед возвратом значения. Чтобы было понятно, `return foo(n - 1)` – это вызов хвоста, а `return foo(n - 1) + 1` – нет (поскольку сложение является последней операцией).
- Оптимизация хвостовых вызовов (Tail call optimization, TCO) – это способ автоматического сокращения рекурсии в рекурсивных функциях.
- Устранение хвостового вызова (Tail call elimination, TCE) – это приведение хвостового вызова к выражению, которое может быть вычислено без рекурсии. TCE является разновидностью TCO.

Оптимизация хвостовых вызовов полезна по ряду причин:

- Интерпретатор может минимизировать объем памяти, занимаемый окружениями. Поскольку ни один компьютер не обладает неограниченной

памятью, чрезмерное количество рекурсивных вызовов функций приведет к переполнению стека.

- Интерпретатор может уменьшить количество переключений кадров стека.

В Python по ряду причин не реализована ТСО. Поэтому для преодоления этого ограничения требуются другие методы. Выбор метода зависит от конкретного случая. При определенной интуиции определения `factorial` и `fib` можно относительно легко преобразовать в итеративный код следующим образом:

```
def factorial(n):
    product = 1
    while n > 1:
        product *= n
        n -= 1
    return product
```

```
def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

Обычно это наиболее эффективный способ ручного устранения рекурсии, но для более сложных функций он может стать довольно сложным.

Другим полезным инструментом является декоратор `lru_cache` в Python, который может быть использован для сокращения числа избыточных вычислений.

Теперь вы имеете представление о том, как избежать рекурсии в Python, но когда *следует* использовать рекурсию? Ответ – “не часто”. Все рекурсивные функции могут быть реализованы итеративно. Просто нужно понять, как это сделать. Однако есть редкие случаи, когда рекурсия не помешает. В Python рекурсия распространена в тех случаях, когда ожидаемые входные данные не вызывают значительного количества вызовов рекурсивных функций.

Если рекурсия – это тема, которая вас интересует, то следует изучать функциональные языки, такие как Scheme или Haskell. В этих языках рекурсия гораздо полезнее.

Обратите внимание, что приведенный пример для последовательности Фибоначчи, хотя и хорошо показывает, как применять определение на языке Python и в дальнейшем использовать `lru_cache`, имеет не очень большое время работы, так как для каждого небазового случая выполняется 2 рекурсивных вызова. Количество обращений к функции растет экспоненциально до n .

Как ни странно, более рациональная реализация будет использовать линейную рекурсию:

```
def fib(n):
    if n <= 1:
        return (n, 0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Но в этом случае возникает проблема возврата *пары* чисел. Это подчеркивает, что некоторые функции действительно мало выигрывают от рекурсии.

86.2. Задача “исследования дерева” с помощью рекурсии

Допустим, у нас есть следующее дерево:

```
root
- A
  - AA
  - AB
- B
```

```
- BA
- BB
  - BBA
```

Теперь, если мы хотим перечислить все имена элементов, мы можем сделать это с помощью простого цикла `for`. Предполагается, что есть функция `get_name()`, возвращающая строку имени узла, функция `get_children()`, возвращающая список всех подузлов данного узла в дереве, и функция `get_root()`, получающая корневой узел.

```
root = get_root(tree)
for node in get_children(root):
    print(get_name(node))
    for child in get_children(node):
        print(get_name(child))
        for grand_child in get_children(child):
            print(get_name(grand_child))
# выводит: A, AA, AB, B, BA, BB, BBA
```

Это работает хорошо и быстро, но что если у вложенных узлов появятся свои собственные вложенные узлы? А у этих вложенных узлов могут быть еще вложенные узлы... Что делать, если заранее неизвестно, сколько их будет? Одним из способов решения этой проблемы является использование рекурсии.

```
def list_tree_names(node):
    for child in get_children(node):
        print(get_name(child))
        list_tree_names(node=child)

list_tree_names(node=get_root(tree))
# prints: A, AA, AB, B, BA, BB, BBA
```

Возможно, вы хотите не выводить, а возвращать плоский список всех имен узлов. Это можно сделать, передав в качестве параметра скользящий список.

```
def list_tree_names(node, lst=[]):
    for child in get_children(node):
        lst.append(get_name(child))
        list_tree_names(node=child, lst=lst)
    return lst

list_tree_names(node=get_root(tree))
# returns ['A', 'AA', 'AB', 'B', 'BA', 'BB', 'BBA']
```

86.3. Сумма чисел от 1 до n

Если нужно вычислить сумму чисел от 1 до n , где n – натуральное число, то можно выполнить $1 + 2 + 3 + 4 + \dots + (\text{несколько часов спустя}) + n$. В качестве альтернативы можно написать цикл `for`:

```
n = 0
for i in range(1, n+1):
    n += i
```

Или можно использовать рекурсию:

```
def recursion(n):
    if n == 1:
        return 1
    return n + recursion(n - 1)
```

Рекурсия имеет преимущества перед двумя вышеперечисленными методами. Рекурсия занимает меньше времени, чем выписывание $1 + 2 + 3$ для суммы от 1 до 3. Для `recursion(4)` рекурсия может быть использована для работы в обратном направлении:

Вызовы функций: ($4 \rightarrow 4 + 3 \rightarrow 4 + 3 + 2 \rightarrow 4 + 3 + 2 + 1 \rightarrow 10$)

тогда как цикл `for` работает строго вперед: ($1 \rightarrow 1 + 2 \rightarrow 1 + 2 + 3 \rightarrow 1 + 2 + 3 + 4 \rightarrow 10$). Иногда рекурсивное решение оказывается проще итеративного. Это наглядно видно при реализации реверсирования связного списка.

86.4. Увеличение максимальной глубины рекурсии

Существует предел глубины рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение `RuntimeError`:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Вот пример программы, которая вызовет эту ошибку:

```
def cursing(depth):
    try:
        cursing(depth + 1) # actually, re-cursing
    except RuntimeError as RE:
        print('Я рекурсировал {} раз!'.format(depth))
    cursing(0)
# Вывод: Я рекурсировал 1083 раз!
```

Можно изменить предел глубины рекурсии, используя

```
sys.setrecursionlimit(limit)
```

Проверить, каковы текущие параметры ограничения, можно, выполнив команду:

```
sys.getrecursionlimit()
```

Выполняя тот же метод с нашим новым пределом, получим

```
sys.setrecursionlimit(2000)
cursing(0)
# Вывод: Я рекурсировал 1997 раз!
```

Из Python 3.5 исключение представляет собой `RecursionError`, производное из `RuntimeError`.

86.5. Хвостовая рекурсия – плохая практика

Когда единственное, что возвращается из функции, – это рекурсивный вызов, это называется *хвостовой рекурсией*. Приведем пример обратного отсчета, написанный с использованием хвостовой рекурсии:

```
def countdown(n):
    if n == 0:
        print "Blastoff!"
    else:
        print n
        countdown(n-1)
```

Любое вычисление, которое может быть выполнено с помощью итерации, может быть выполнено и с помощью рекурсии. Вот версия `find_max`, написанная с использованием хвостовой рекурсии:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Хвостовая рекурсия считается плохой практикой в Python, поскольку компилятор Python не справляется с оптимизацией хвостовых рекурсивных вызовов. Рекурсивное решение в подобных случаях использует больше системных ресурсов, чем эквивалентное итеративное решение.

86.6. Оптимизация хвостовой рекурсии с помощью интроспекции стека

По умолчанию стек рекурсии в Python не может превышать 1000 кадров. Это можно изменить, задав параметр `sys.setrecursionlimit(15000)`, который работает быстрее, однако этот метод потребляет больше памяти. Вместо этого мы можем решить проблему хвостовой рекурсии с помощью интроспекции стека.

```
#!/usr/bin/env python2.4
# Эта программа демонстрирует декоратор Python, который реализует
# оптимизацию хвостовых вызовов (Tail call optimization, TCO) вызовов. Он
# делает это, бросая исключение, если он является своим прародителем, и перехватывая
# такие исключения для вызова стека.
```

```
import sys
class TailRecurseException:
    def init (self, args, kwargs):
        self.args = args
        self.kwargs = kwargs
```

```
def tail_call_optimized(g):
```

Эта функция украшает функцию оптимизации хвостовых вызовов. Для этого она бросает исключение, если функция является своим прародителем, и перехватывает такие исключения, чтобы симитировать оптимизацию хвостовых вызовов.
Эта функция не работает, если декорированная функция рекурсирует в нехвостовом контексте.

```
def func(*args, **kwargs):
    f = sys._getframe()
    if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
        raise TailRecurseException(args, kwargs)
    else:
        while 1:
            try:
                return g(*args, **kwargs)
            except TailRecurseException, e:
                args = e.args
                kwargs = e.kwargs
func.__doc__ = g.__doc__
return func
```

Для оптимизации рекурсивных функций мы можем использовать декоратор `@tail_call_optimized` для вызова нашей функции. Вот несколько распространенных примеров рекурсии с использованием описанного выше декоратора:

Пример с факториалом:

```
@tail_call_optimized
def factorial(n, acc=1):
    "calculate a factorial"
    if n == 0:
        return acc
    return factorial(n-1, n*acc)
```

```
print factorial(10000)
# выводит очень большое число,
# но не достигает предела рекурсии.
```

Пример Фибоначчи:

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
    if i == 0:
```

```

    return current
else:
    return fib(i - 1, next, current + next)

print fib(10000)
# также выводит очень большое число,
# но не достигает предела рекурсии.

```

Глава 87. Подсказки типов

87.1. Добавление типов в функцию

Рассмотрим пример функции, которая принимает два аргумента и возвращает значение, обозначающее их сумму:

```

def two_sum(a, b):
    return a + b

```

Глядя на этот код, нельзя уверенно и без сомнений указать тип аргументов функции `two_sum`. Он работает и в том, и в другом случае, когда ему передаются значения `int`:

```
print(two_sum(2, 1)) # 3
```

и со строками:

```
print(two_sum("a", "b")) # "ab"
```

и с другими значениями, такими как списки, кортежи и т. д.

В силу динамической природы типов Python, когда для данной операции применимо множество типов, любая программа проверки типов не сможет обоснованно утверждать, разрешен ли вызов данной функции или нет.

Чтобы помочь нашему средству проверки типов, мы можем дать ему подсказки в определении функции, указывающие на тип, который мы допускаем. Чтобы указать, что мы хотим разрешить только типы `int`, мы можем изменить определение функции на следующее:

```

def two_sum(a: int, b: int):
    return a + b

```

Аннотации следуют за именем аргумента с символом `:` в качестве разделителя.

Аналогично, чтобы указать, что разрешены только типы `str`, мы изменим нашу функцию, чтобы указать это:

```

def two_sum(a: str, b: str):
    return a + b

```

Кроме указания типа аргументов можно также указать возвращаемое значение вызова функции. Для этого после закрывающей круглой скобки в списке аргументов, *но* перед символом `:` в конце объявления функции добавляется символ `->`, за которым следует тип:

```

def two_sum(a: int, b: int) -> int:
    return a + b

```

Теперь мы указали, что возвращаемое значение при вызове `two_sum` должно иметь тип `int`. Аналогичным образом мы можем определить соответствующие значения для `str`, `float`, `list`, `set` и других.

Хотя подсказки типов в основном используются программами проверки типов и IDE, иногда может потребоваться их извлечение. Это можно сделать с помощью специального атрибута `__annotations__`:

```

two_sum.__annotations__
# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}

```

87.2. Функция NamedTuple

Создание именованного кортежа с подсказками типа осуществляется с помощью функции `NamedTuple` из модуля `typing`:

```
import typing
Point = typing.NamedTuple('Point', [('x', int), ('y', int)])
```

Заметим, что имя результирующего типа является первым аргументом функции, но его следует присвоить одноименной переменной, чтобы облегчить проверку типов.

87.3. Общие типы

`typing.TypeVar` - это фабрика общих типов. Его основная задача – служить в качестве параметра/заместителя для аннотаций общих функций/классов/методов:

```
import typing

T = typing.TypeVar("T")

def get_first_element(l: typing.Sequence[T]) -> T:
    """Получает первый элемент последовательности."""
    return l[0]
```

87.4. Переменные и атрибуты

Переменные аннотируются с помощью комментариев:

```
x = 3 # type: int
x = negate(x)
x = 'программа проверки типов (type-checker) может обнаружить эту ошибку'
```

Версия Python 3.x ≥ 3.6

Начиная с Python 3.6 появился новый синтаксис для аннотаций переменных. Приведенный выше код может иметь вид

```
x: int = 3
```

В отличие от комментариев, можно также просто добавить подсказку типа к переменной, которая ранее не была объявлена, не задавая ей значения:

```
y: int
```

Кроме того, если они используются на уровне модуля или класса, то подсказки о типе можно получить, используя `typing.get_type_hints(class_or_module)`:

```
class Foo:
    x: int
    y: str = 'abc'

print(typing.get_type_hints(Foo))
# ChainMap({'x': <class 'int'>, 'y': <class 'str'>}, {})
```

Кроме того, доступ к ним можно получить с помощью `__annotations__`:

```
x: int
print(__annotations__)
# {'x': <class 'int'>}

class C:
    s: str
print(C.__annotations__)
# {'s': <class 'str'>}
```

87.5. Члены и методы классов

```
class A:
    x = None # type: float
    def __init__(self, x: float) -> None:
        """
        self не должен быть аннотирован
        init должен быть аннотирован для возврата None
        """
        self.x = x

    @classmethod
    def from_int(cls, x: int) -> 'A':
        """
        cls не должен быть аннотирован
        Используйте прямую ссылку для обращения к текущему классу со строковым литералом 'A'
        """
        return cls(float(x))
```

Прямая ссылка (forward reference) на текущий класс необходима, поскольку аннотации оцениваются при определении функции. Прямые ссылки также могут использоваться при обращении к классу, при импортировании которого возникнет циклический импорт.

87.6. Подсказки типов для именованных аргументов

```
def hello_world(greeting: str = 'Hello'):
    print(greeting + ' world!')
```

Обратите внимание на пробелы вокруг знака равенства – это отличается от того, как обычно оформляются именованные аргументы.

Глава 88. Исключения

Ошибки, обнаруженные в процессе выполнения, называются исключениями (exceptions) и не являются безусловно фатальными. Большинство исключений не обрабатывается программами; можно написать программы, обрабатывающие отдельные исключения. В Python имеются специальные функции для работы с исключениями и логикой исключений. Кроме того, исключения имеют богатую иерархию типов, все они наследуются от типа `BaseException`.

88.1. Перехват исключений

Для перехвата исключений используйте `try...except`. Следует указывать исключение как можно более точно:

```
try:
    x = 5 / 0
except ZeroDivisionError as e:
    # `e` - объект исключения
    print("Получено деление на ноль! Исключение было:", e)
    # обработать случай исключения
    x = 0
finally:
    print("The END")
    # запускается независимо от выполнения.
```


Указанный класс исключений – в данном случае `ZeroDivisionError` – перехватывает любое исключение, относящееся к этому классу или к любому его подклассу. Например, `ZeroDivisionError` является подклассом `ArithmeticError`:

```
>>> ZeroDivisionError.__bases__
(<class 'ArithmeticError'>,)

```

Таким образом, следующий код также перехватит ошибку `ZeroDivisionError`:

```
try:
    5 / 0
except ArithmeticError:
    print("Получена арифметическая ошибка")

```

88.2. Не перехватывайте все подряд!

Часто возникает соблазн перехватить каждое исключение:

```
try:
    very_difficult_function()
except Exception:
    # log / попытка повторного подключения / благополучный выход
finally:
    print "The END"
    # запускается независимо от выполнения

```

и все, что включает в себя `BaseException` и все его дочерние элементы, включая `Exception`:

```
try:
    even_more_difficult_function()
except:
    pass # сделать все необходимое

```

В большинстве случаев это плохая практика. Она может перехватить больше, чем предполагалось, например, `SystemExit`, `KeyboardInterrupt` и `MemoryError` – каждая из которых, как правило, должна обрабатываться иначе, чем обычные системные или логические ошибки. Это также означает, что нет четкого понимания того, что внутренний код может сделать неправильно и как правильно восстановиться после этого состояния. Если вы будете перехватывать каждую ошибку, то не будете знать, какая ошибка произошла и как ее исправить.

Это чаще всего называют “маскировкой ошибок” (*bug masking*), и ее следует избегать. Пусть вместо тихого сбоя или, что еще хуже, сбоя на более глубоком уровне выполнения ваша программа аварийно завершается. (Представьте, что это транзакционная система.)

Обычно эти конструкции используются на самом внешнем уровне программы и фиксируют подробности ошибки, чтобы ее можно было устранить или обработать более конкретно.

88.3. Повторный вызов исключений

Иногда требуется перехватить исключение, чтобы проверить его, например, в целях протоколирования. После проверки вы хотите, чтобы исключение продолжало распространяться, как и раньше. В этом случае достаточно использовать оператор `raise` без параметров.

```
try:
    5 / 0
except ZeroDivisionError:
    print("Получена ошибка")
    raise

```

Однако следует помнить, что дальше в стеке вызывающей стороны исключение все равно может быть перехвачено и как-то обработано. Готовый вывод в этом случае может быть проблемным, поскольку он произойдет в любом случае (перехваченном или не перехваченном). Поэтому, возможно, лучше вызвать другое исключение, содержащее как ваш комментарий к ситуации, так и исходное исключение:

```
try:
    5 / 0
except ZeroDivisionError as e:
    raise ZeroDivisionError("Получена ошибка", e)
```

Однако это имеет тот недостаток, что трассировка исключений сводится именно к этому оператору `raise`, в то время как `raise` без аргумента сохраняет исходную трассировку исключений. В Python 3 можно сохранить исходный стек, используя синтаксис `raise-from`:

```
raise ZeroDivisionError("Получена ошибка") from e
```

88.4. Перехват множественных исключений

Существует несколько способов перехвата нескольких исключений.

Первый способ заключается в создании кортежа типов исключений, которые вы хотите перехватывать и обрабатывать аналогичным образом. В данном примере код будет игнорировать исключения `KeyError` и `AttributeError`.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except (KeyError, AttributeError) as e:
    print("Было перехвачено исключение KeyError или AttributeError")
```

Если вы хотите обрабатывать разные исключения разными способами, то для каждого типа можно выделить отдельный блок исключений. В данном примере мы по-прежнему перехватываем `KeyError` и `AttributeError`, но обрабатываем эти исключения по-разному.

```
try:
    d = {}
    a = d[1]
    b = d.non_existing_field
except KeyError as e:
    print("Произошла ошибка KeyError. Сообщение об ошибке:", e)
except AttributeError as e:
    print("Произошла ошибка AttributeError. Сообщение об ошибке:", e)
```

88.5. Иерархия исключений

Обработка исключений происходит на основе иерархии исключений, определяемой структурой наследования классов исключений. Например, `IOError` и `OSError` являются под-классами `EnvironmentError`. Код, перехватывающий `IOError`, не будет перехватывать `OSError`. Однако код, перехватывающий `EnvironmentError`, будет перехватывать и `IOError`, и `OSError`.

Иерархия встроенных исключений:

Python 2.x Версия ≥ 2.3:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StandardError
| +-- BufferError
| +-- ArithmeticError
| | +-- FloatingPointError
| | +-- OverflowError
| | +-- ZeroDivisionError
| +-- AssertionError
+-- AttributeError
```

```

|-- EnvironmentError
|-- IOError
|-- OSError
|-- WindowsError (Windows)
|-- VMSError (VMS)
|-- EOFError
|-- ImportError
|-- LookupError
|-- IndexError
|-- KeyError
|-- MemoryError
|-- NameError
|-- UnboundLocalError
|-- ReferenceError
|-- RuntimeError
|-- NotImplementedError
|-- SyntaxError
|-- IndentationError
|-- TabError
|-- SystemError
|-- TypeError
|-- ValueError
|-- UnicodeError
|-- UnicodeDecodeError
|-- UnicodeEncodeError
|-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning

```

Версия Python 3.x ≥ 3.0:

```

BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
|-- FloatingPointError
|-- OverflowError
|-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
+-- LookupError
|-- IndexError
|-- KeyError
+-- MemoryError
+-- NameError

```

```

| +-- UnboundLocalError
+-- OSError
| +-- BlockingIOError
| +-- ChildProcessError
| +-- ConnectionError
| | +-- BrokenPipeError
| | +-- ConnectionAbortedError
| | +-- ConnectionRefusedError
| | +-- ConnectionResetError
+-- FileExistsError
+-- FileNotFoundError
+-- InterruptedError
+-- IsADirectoryError
+-- NotADirectoryError
+-- PermissionError
+-- ProcessLookupError
+-- TimeoutError
+-- ReferenceError
+-- RuntimeError
| +-- NotImplementedError
| +-- RecursionError
+-- SyntaxError
| +-- IndentationError
| +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
| +-- UnicodeError
| +-- UnicodeDecodeError
| +-- UnicodeEncodeError
| +-- UnicodeTranslateError
+-- Warning
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
+-- ResourceWarning

```

88.6. Else

Код в блоке `else` будет выполняться только в том случае, если код в блоке `try` не вызвал исключений. Это полезно, если у вас есть код, который вы не хотите запускать, если возникло исключение, но не хотите, чтобы исключения, вызванные этим кодом, были перехвачены.

Например:

```

try:
    data = {'one': 1, 'two': 2}
    print(data['one'])
except KeyError as e:
    print('key not found')
else:
    raise ValueError()
# Выходные данные: 1
# Выходные данные: ValueError

```

Обратите внимание, что этот вид `else`: не может быть объединен с `if`, начинающим `else`-предложение с `elif`. Если у вас есть последующее `if`, то оно должно оставаться с отступом ниже `else`:, как в примере:

```
try:
except ...:
else:
    if ...:
    elif ...:
    else:
```

88.7. Вызов исключений

Если ваш код встречается условие, которое он не знает, как обработать, например, неправильный параметр, он должен вызвать соответствующее исключение.

```
def even_the_odds(odds):
    if odds % 2 != 1:
        raise ValueError("Не получено нечетное число")
    return odds + 1
```

88.8. Создание пользовательских типов исключений

Создайте класс, наследующий от `Exception`:

```
class FooException(Exception):
    pass
try:
    raise FooException("введите описание здесь")
except FooException:
    print("Возникло исключение FooException.")
```

или другой тип исключения:

```
class NegativeError(ValueError):
    pass
```

```
def foo(x):
    # функция, принимающая только положительные значения x
    if x < 0:
        raise NegativeError("Невозможно обрабатывать отрицательные числа")
    ... # остальная часть тела функции
try:
    result = foo(int(input("Введите целое положительное число: "))) # raw_input в Python 2.x
except NegativeError:
    print("Вы ввели отрицательное число!")
else:
    print("Результат: " + str(result))
```

88.9. Практические примеры обработки исключений

Ввод данных пользователем

Представьте, что вы хотите, чтобы пользователь ввел число с помощью функции `input`. Вы хотите убедиться, что вводимые данные являются числом. Для этого можно использовать `try/except`:

Версия Python 3.x ≥ 3.0:

```
while True:
    try:
        nb = int(input('Введите число: '))
        break
    except ValueError:
        print('Это не число, попробуйте еще раз.')
```

Примечание: в Python 2.x вместо этого используется `raw_input`; функция `input` в Python 2.x существует, но имеет другую семантику. В приведенном выше примере `input` также будет принимать такие выражения, как `2 + 2`, которые оцениваются в число. Если входные данные не удалось преобразовать в целое число, то выдается ошибка `ValueError`. Ее можно перехватить с помощью `except`. Если исключение не возникло, то `break` “выпрыгивает” из цикла. После завершения цикла `nb` содержит целое число.

Словари

Представьте, что вы выполняете итерацию по списку последовательных целых чисел `range(n)`, и у вас есть список словарей `d`, который содержит информацию о том, что нужно делать, когда вы встречаете некоторые конкретные целые числа, например, пропускать следующие за `d[i]`.

```
d = [{7: 3}, {25: 9}, {38: 5}]
```

```
for i in range(len(d)):
    do_stuff(i)
    try:
        dic = d[i]
        i += dic[i]
    except KeyError:
        i += 1
```

При попытке получить значение из словаря для несуществующего ключа будет выдана ошибка `KeyError`.

88.10. Исключения – это тоже объекты

Исключения – это обычные объекты Python, наследующие от встроенного типа `BaseException`. Скрипт на Python может использовать оператор `raise` для прерывания выполнения, в результате чего Python выводит трассировку стека вызовов на данный момент и представление экземпляра исключения. Например,

```
>>> def failing_function():
...     raise ValueError('Example error!')
>>> failing_function()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in failing_function
ValueError: Example error!
```

в котором говорится, что нашей функцией `failing_function()`, которая была выполнена в интерпретаторе, была вызвана ошибка `ValueError` с сообщением 'Example error!'

Вызывающий код может выбрать способ обработки всех типов исключений, которые могут быть вызваны:

```
>>> try:
...     failing_function()
... except ValueError:
...     print('Обработка ошибки')
Обработка ошибки
```

Объекты исключений можно получить, присвоив их в части `except...` кода обработки исключений:

```
>>> try:
...     failing_function()
... except ValueError as e:
...     print('Перехваченное исключение', repr(e))
Перехваченное исключение ValueError('Example error!')
```

Полный список встроенных исключений Python вместе с их описанием можно найти в официальной документации Python.

88.11. Выполнение кода очистки с помощью `finally`

Иногда требуется, чтобы что-то происходило независимо от того, какое исключение произошло, например, если необходимо очистить некоторые ресурсы. Блок `finally` предложения `try` будет выполнен независимо от того, были ли вызваны какие-либо исключения.

```
resource = allocate_some_expensive_resource()
try:
    do_stuff(resource)
except SomeException as e:
    log_error(e)
    raise # повторный вызов ошибки
finally:
    free_expensive_resource(resource)
```

Часто этот паттерн лучше обрабатывать с помощью менеджеров контекста (с использованием оператора `with`).

88.12. Создание цепочек исключений с использованием `raise from`

В процессе обработки исключения может возникнуть необходимость вызвать другое исключение. Например, при получении ошибки `IOError` во время чтения из файла вместо нее можно вызвать ошибку, специфичную для приложения, чтобы показать ее пользователям библиотеки.

Версия Python 3.x ≥ 3.0

Вы можете выстроить цепочку исключений, чтобы показать, как происходила обработка исключений:

```
>>> try:
...     5 / 0
... except ZeroDivisionError as e:
...     raise ValueError("Деление не удалось") from e
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Вышеуказанное исключение явилось непосредственной причиной следующего исключения:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ValueError: Деление не удалось
```

Глава 89. Вызов пользовательских ошибок и исключений

В Python имеется множество встроенных исключений, которые заставляют вашу программу выдавать ошибку, если что-то в ней идет не так. Однако иногда может потребоваться создание задуманных вами пользовательских исключений. Пользователь может определить такие исключения, создав новый класс. Этот класс исключений должен быть прямо или косвенно производным от класса `Exception`. Большинство встроенных исключений также являются производными от этого класса.

89.1. Пользовательское исключение

Здесь мы создали пользовательское исключение `CustomError`, которое является производным от класса `Exception`. Это новое исключение может быть вызвано, как и другие исключения, с помощью оператора `raise` с необязательным сообщением об ошибке.

```
class CustomError(Exception):
    pass

x = 1

if x == 1:
    raise CustomError('Это пользовательская ошибка')
```

Результат:

```
Traceback (most recent call last):
  File "error_custom.py", line 8, in
    raise CustomError('Это пользовательская ошибка')
__main__.CustomError: Это пользовательская ошибка
```

89.2. Перехват пользовательского исключения

В данном примере показано, как перехватить пользовательское исключение `Exception`:

```
class CustomError(Exception):
    pass

try:
    raise CustomError('можете меня поймать?')
except CustomError as e:
    print('Caught CustomError :{}'.format(e))
except Exception as e:
    print('Generic exception: {}'.format(e))

Результат:
Caught CustomError: можете меня поймать?
```

Глава 90. Исключения сообщества

В сообществе Stack Overflow участники часто задают вопросы об одних и тех же ошибках: `"ImportError: No module named '?????'"`, `"SyntaxError: invalid syntax"` или `"NameError: name '??' is not defined"`. Попытаемся сократить количество таких вопросов.

90.1. Прочие ошибки

`AssertionError`

Оператор `assert` существует практически во всех языках программирования. При выполнении:

```
assert condition
```


или:

```
assert condition, message
```

это эквивалентно следующему:

```
if __debug__:
    if not condition: raise AssertionError(message)
```

Утверждения (assertions) могут содержать необязательное сообщение, и вы можете отключить их, когда закончите отладку.

Примечание: встроенная переменная **debug** имеет значение True в обычных условиях, False – при запросе оптимизации (опция командной строки -O). Присвоение переменной debug является недопустимым. Значение встроенной переменной определяется при запуске интерпретатора.

KeyboardInterrupt

Ошибка, возникающая при нажатии пользователем клавиши прерывания, обычно "Ctrl" + "C" или "del".

ZeroDivisionError

Вы пытались вычислить 1/0, что дает неопределенный результат. Посмотрите этот пример:

Python 2.x версии $\geq 2.0 \leq 2.7$:

```
div = float(raw_input("Делители: "))
for x in xrange(div+1): #включает само число и ноль
    if div/x == div//x:
        print x, "является делителем", div
```

Версия Python 3.x ≥ 3.0 :

```
div = int(input("Делители: "))
for x in range(div+1): #включает само число и ноль
    if div/x == div//x:
        print(x, "является делителем", div)
```

При этом возникает ошибка ZeroDivisionError, поскольку цикл for присваивает x это значение. Должно быть так:

Python 2.x версии $\geq 2.0 \leq 2.7$:

```
div = float(raw_input("Делители: "))
for x in xrange(1,div+1): #включает само число, но не ноль
    if div/x == div//x:
        print x, "является делителем", div
```

Версия Python 3.x ≥ 3.0 :

```
div = int(input("Делители: "))
for x in range(1,div+1): ##включает само число, но не ноль
    if div/x == div//x:
        print(x, "является делителем", div)
```

90.2. Ошибка "NameError. name '??' is not defined"

Возникает при попытке использования переменной, метода или функции, которые не инициализированы (по крайней мере, не были инициализированы ранее). Другими словами, она возникает, когда запрашиваемое локальное или глобальное имя не найдено. Возможно, вы неправильно указали имя объекта или забыли что-то импортировать. Также возможно, что он находится в другой области видимости. Об этом мы расскажем в отдельных примерах.

Не прописано в коде

Возможно, вы забыли провести инициализацию, особенно это касается констант:

```
foo # Эта переменная не определена
bar() # Эта функция не определена
```

Возможно, определение произведено позже:

```
baz()
```

```
def baz():
    pass
```

или не был осуществлен импорт:

```
#needs import math
```

```
def sqrt():
    x = float(input("Value: "))
    return math.sqrt(x)
```

Области видимости Python и правило LEGB

Так называемое правило LEGB говорит об областях видимости Python. Его название происходит от различных областей, упорядоченных по соответствующим приоритетам:

Local → Enclosed → Global → Built-in.

- **Local** – Локальные: Переменные, не объявленные глобальными или не назначенные в функции.
- **Enclosing** – Охватывающие: Переменные, определяемые в функции, которая обернута внутри другой функции.
- **Global** – Глобальные: Переменные, объявленные глобальными, или назначенные на верхнем уровне ячейки.
- **Built-in** – Встроенные: Переменные, предварительно назначенные в модуле встроенных имен.

В качестве примера:

```
for i in range(4):
    d = i * 2
    print(d)
```

Переменная `d` доступна, так как цикл `for` не помечает новую область видимости, но если бы это произошло, то мы бы получили ошибку, и поведение цикла было бы аналогичным:

```
def noaccess():
    for i in range(4):
        d = i * 2
    noaccess()
    print(d)
```

NameError: имя 'd' не определено

90.3. Ошибки типов данных

Эти исключения возникают в тех случаях, когда тип некоторого объекта должен быть другим.

TypeError: [definition/method] принимает ? позиционных аргументов, но было задано ?

Функция или метод были вызваны с большим (или меньшим) количеством аргументов, чем те, которые она может принять.

Пример

Если задано больше аргументов:

```
def foo(a): return a
foo(a,b,c,d) # Определены a, b, c, d
```

Если задано меньше аргументов:

```
def foo(a,b,c,d): return a += b + c + d
foo(a) # a определено
```

Примечание: если необходимо использовать неизвестное количество аргументов, можно использовать `*args` или `**kwargs`.

TypeError: неподдерживаемые типы операндов для [operand]: '???' и '???'

Некоторые типы не могут работать вместе, в зависимости от операнда.

Пример

Операнд `+` используется для конкатенации и сложения, но ни один из них нельзя использовать для обоих типов. Например, попытка создать набор (`set`) путем конкатенации `'set1'` и `'tuple1'` дает ошибку. Код:

```
set1, tuple1 = {1,2}, (3,4)
a = set1 + tuple1
```

Некоторые типы (например, `int` и `string`) оба используют `+`, но для разных целей:

```
b = 400 + 'foo'
```

Но вы можете, например, добавить числа типа `float` к `int`:

```
d = 1 + 1.0
```

TypeError: '??' object is not iterable/subscriptable:

Для того чтобы объект был итерируемым, он может принимать последовательные индексы, начиная с нуля, до тех пор, пока индексы не перестанут быть действительными и не возникнет ошибка `IndexError` (более технически: он должен иметь метод `__iter__`, который возвращает `__iterator__` или который определяет метод `__getitem__`, который делает то, о чем говорилось ранее).

Пример

Здесь мы указываем, что `bar` является нулевым элементом 1. Бессмысленность:

```
foo = 1
bar = foo[0]
```

Или другой вариант: `for` пытается для `x` установить `amount[0]`, первый элемент итерируемого объекта, но не может, так как `amount` – это целое число:

```
amount = 10
for x in amount: print(x)
```

TypeError: '??' object is not callable

Вы определили переменную и вызываете ее позже (как это делается с функцией или методом).

Пример

```
foo = "notAFunction"
foo()
```

90.4. Синтаксическая ошибка в хорошем коде

В подавляющем большинстве случаев ошибка `SyntaxError`, указывающая на строку, означает, что проблема есть *в строке перед ней* (в данном примере это пропущенная скобка):

```
def my_print():
    x = (1 + 1
    print(x)
```

Возвращает

```
File "<input>", line 3
print(x)
    ^
```

`SyntaxError: invalid syntax`

Как видно из примера, наиболее распространенной причиной возникновения этой проблемы является несовпадение скобок или круглых скобок. В Python 3 есть одна существенная оговорка для операторов печати:

Python 3.x Version ≥ 3.0 :

```
>>> print "hello world"
File "<stdin>", line 1
print "hello world"
    ^
```

Так как оператор `print` был заменен функцией `print()`, то:

`print("hello world")` # Обратите внимание, что это работает и в Py2, и в Py3

90.5. Ошибки отступа (или ошибки синтаксиса отступов)

В большинстве других языков отступы не обязательны, но в Python (и других языках: ранние версии FORTRAN, Makefiles, Whitespace (эзотерический язык) и т. д.) это не так, что может сбить с толку, если вы привыкли к другому языку, если вы копировали код из примера в свой собственный, или просто если вы новичок.

`IndentationError/SyntaxError: unexpected indent` (неожиданный отступ)

Это исключение возникает, когда уровень отступа увеличивается без причины.

В примере ниже нет причин для повышения уровня отступа:

Версия Python 2.x $\geq 2.0 \leq 2.7$:

```
print "This line is ok"
    print "This line isn't ok"
```

Версия Python 3.x ≥ 3.0 :

```
print("This line is ok")
    print("This line isn't ok")
```

Здесь две ошибки: описанная выше и то, что отступ не соответствует ни одному уровню отступа. Однако показана только одна:

Версия Python 2.x $\geq 2.0 \leq 2.7$:

```
print "This line is ok"
    print "This line isn't ok"
```

Версия Python 3.x ≥ 3.0 :

```
print("This line is ok")
    print("This line isn't ok")
```

IndentationError/SyntaxError: unindent does not match any outer indentation level (отмена отступа не соответствует ни одному внешнему уровню отступа)

В этом случае чаще всего недостаточно уменьшен уровень отступа. Пример:

Версия Python 2.x $\geq 2.0 \leq 2.7$:

```
def foo():
    print "This should be part of foo()"
    print "ERROR!"
print "This is not a part of foo()"
```

Версия Python 3.x ≥ 3.0 :

```
print("This line is ok")
print("This line isn't ok")
```

IndentationError: expected an indented block (ожидается блок с отступами)

После двоеточия (и затем новой строки) уровень отступа должен увеличиваться. Эта ошибка возникает, если этого не произошло. Пример:

```
if ok:
doStuff()
```

Примечание: используйте ключевое слово `pass` (которое ничего не делает), чтобы просто поставить `if`, `else`, `except`, `class`, `method` или `definition`, но не говорить, что произойдет, если вызов/условие будет истинным (а сделать это позже, или в случае `except`: просто ничего не делать):

```
def foo():
    pass
```

IndentationError: inconsistent use of tabs and spaces in indentation (непоследовательное использование табуляции и пробелов в отступах)

```
def foo():
    if ok:
        return "Two != Four != Tab"
        return "Мне все равно, я делаю все, что хочу"
```

Чтобы избежать этой ошибки, не используйте табуляции. Это не рекомендуется PEP8, руководством по стилю для Python.

1. Установите в редакторе **4 пробела** для отступа.
2. Выполните поиск и замену, чтобы заменить все табуляции на 4 пробела.
3. Убедитесь, что ваш редактор настроен на **отображение табуляции** в виде 8 пробелов, чтобы вы могли легко понять эту ошибку и исправить ее.

Глава 91. urllib

91.1. HTTP GET

Версия Python 2.x ≤ 2.7 :

Python 2

```
import urllib
response = urllib.urlopen('http://stackoverflow.com/documentation/')
```

Использование `urllib.urlopen()` вернет объект ответа, с которым можно работать как с файлом.

```
print response.code
# Выводит: 200
```

`response.code` представляет собой возвращаемое значение `http`, где 200 – OK, 404 – NotFound и т. д.

```
print response.read()
'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack. etc'
```

Методы `response.read()` и `response.readlines()` могут быть использованы для чтения собственно `html`-файла, возвращаемого из запроса. Эти методы работают аналогично `file.read*`.

Версия Python 3.x ≥ 3.0:

Python 3

```
import urllib.request
```

```
print(urllib.request.urlopen("http://stackoverflow.com/documentation/"))
# Выводит: <http.client.HTTPResponse at 0x7f37a97e3b00>
```

```
response = urllib.request.urlopen("http://stackoverflow.com/documentation/")
```

```
print(response.code)
# Выводит: 200
print(response.read())
# Выводит: b'<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Documentation - Stack Overflow</title>
```

Модуль был обновлен для Python 3.x, но варианты использования остались в основном теми же. `urllib.request.urlopen` вернет аналогичный, подобный файлу объект.

91.2. HTTP POST

Для POST-данных передайте закодированные аргументы запроса в качестве данных в функцию `urlopen()`.

Версия Python 2.x ≤ 2.7:

Python 2

```
import urllib
query_params = {'username':'stackoverflow', 'password':'me.me'}
encoded_params = urllib.urlencode(query_params)
response = urllib.urlopen("https://stackoverflow.com/users/login", encoded_params)
response.code
# Вывод: 200
response.read()
# Вывод: '<!DOCTYPE html>\r\n<html>\r\n<head>\r\n\r\n<title>Log In - Stack Overflow'
```

Версия Python 3.x ≥ 3.0:

Python 3

```
import urllib
query_params = {'username':'stackoverflow', 'password':'me.me'}
encoded_params = urllib.parse.urlencode(query_params).encode('utf-8')
response = urllib.request.urlopen("https://stackoverflow.com/users/login", encoded_params)
response.code
# Вывод: 200
response.read()
# Вывод: b'<!DOCTYPE html>\r\n<html>....etc'
```

91.3. Декодирование полученных байтов в соответствии с кодировкой типа содержимого

Полученные байты должны быть декодированы с помощью правильной кодировки символов, чтобы быть интерпретированными как текст:

Версия Python 3.x ≥ 3.0:

```
import urllib.request

response = urllib.request.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Версия Python 2.x ≤ 2.7:

```
import urllib2
response = urllib2.urlopen("http://stackoverflow.com/")
data = response.read()

encoding = response.info().getencoding()
html = data.decode(encoding)
```

Глава 92. Веб-скрапинг с помощью Python

Веб-скрапинг (также известен как веб-скрейпинг или скрепинг) – это автоматизированный программный процесс, с помощью которого можно постоянно извлекать (“соскребать”, *scrape*) данные с веб-страниц. Также известный как screen scraping или web harvesting, веб-скрапинг позволяет мгновенно получать данные с любой общедоступной веб-страницы. На некоторых сайтах веб-скрапинг может быть незаконным.

92.1. Скрапинг с использованием фреймворка Scrapy

Сначала необходимо создать новый проект Scrapy. Укажите каталог, в котором будет храниться ваш код, и запустите его:

```
scrapy startproject projectName
```

Для того чтобы выполнить скрапинг, нам нужен “паук” (spider). Пауки определяют, каким образом будет осуществляться скрапинг с определенного сайта. Вот код паука, который переходит по ссылкам на вопросы, получившие наибольшее количество голосов на StackOverflow, и считывает некоторые данные с каждой страницы:

```
import scrapy

class StackOverflowSpider(scrapy.Spider):
    name = 'stackoverflow' # каждый паук имеет уникальное имя
    start_urls = ['http://stackoverflow.com/questions?sort=votes']
    # парсинг начинается с определенного набора url

    def parse(self, response):
        # для каждого запроса, выдаваемого этим генератором, его ответ отправляется в parse_question
        for href in response.css('.question-summary h3 a::attr(href)'):
            # выполняем скрапинг с использованием селекторов css для поиска url вопросов
```

```

full_url = response.urljoin(href.extract())
yield scrapy.Request(full_url, callback=self.parse_question)

def parse_question(self, response):
    yield {
        'title': response.css('h1 a::text').extract_first(),
        'votes': response.css('.question .vote-count-post::text').extract_first(),
        'body': response.css('.question .post-text').extract_first(),
        'tags': response.css('.question .post-tag::text').extract(),
        'link': response.url,
    }

```

Сохраните классы-пауки в каталоге `projectName\spiders`. В данном случае – `projectName\spiders\stackoverflow_spider.py`.

Теперь вы можете использовать своего паука. Например, попробуйте выполнить (в каталоге проекта):

```
scrapy crawl stackoverflow
```

92.2. Скрапинг с использованием Selenium WebDriver

Некоторые веб-сайты “не любят” скрапинга. В таких случаях может потребоваться имитация работы реального пользователя с браузером. Selenium запускает и управляет веб-браузером.

```
from selenium import webdriver
```

```
browser = webdriver.Firefox() # запуск браузера Firefox
```

```
browser.get('http://stackoverflow.com/questions?sort=votes') # загрузка url
```

```
title = browser.find_element_by_css_selector('h1').text # заголовок страницы (первый элемент h1)
```

```
questions = browser.find_elements_by_css_selector('.question-summary') # список вопросов
```

```
for question in questions: # итерация по вопросам
```

```
    question_title = question.find_element_by_css_selector('.summary h3 a').text
```

```
    question_excerpt = question.find_element_by_css_selector('.summary .excerpt').text
```

```
    question_vote = question.find_element_by_css_selector('.stats .vote .votes .vote-count-post').text
```

```
    print "%s\n%s\n%s votes\n———\n" % (question_title, question_excerpt, question_vote)
```

Но Selenium способен на гораздо большее. Он может изменять cookies браузера, заполнять формы, имитировать щелчки мыши, делать скриншоты веб-страниц и запускать пользовательский JavaScript.

92.3. Базовый пример использования запросов и lxml для скрапинга некоторых данных

```
# Для совместимости с Python 2.
```

```
from __future__ import print_function
```

```
import lxml.html
```

```
import requests
```

```
def main():
```

```
    r = requests.get("https://httpbin.org")
```

```
    html_source = r.text
```

```
    root_element = lxml.html.fromstring(html_source)
```



```
# Обратите внимание, что root_element.xpath() дает *список* результатов.
# XPath задает путь к нужному нам элементу.
page_title = root_element.xpath('/html/head/title/text()')[0]
print(page_title)

if __name__ == '__main__':
    main()
```

92.4. Поддержание сессии веб-скрапинга с помощью запросов

Для сохранения cookies и других параметров целесообразно поддерживать сессию веб-скрапинга. Кроме того, это может привести к *повышению производительности*, поскольку requests.Session повторно использует базовое TCP-соединение с хостом:

```
import requests

with requests.Session() as session:
    # для всех запросов сессии теперь устанавливается заголовок User-Agent
    session.headers = {'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36'}

    # установить файлы cookie
    session.get('http://httpbin.org/cookies/set?key=value')

    # получить файлы cookie
    response = session.get('http://httpbin.org/cookies')
    print(response.text)
```

92.5. Скрапинг с использованием BeautifulSoup4

```
from bs4 import BeautifulSoup
import requests

# Использование модуля requests для получения страницы
res = requests.get('https://www.codechef.com/problems/easy')

# Создать объект BeautifulSoup
page = BeautifulSoup(res.text, 'lxml') # текстовое поле содержит исходник страницы

# Теперь используйте CSS-селектор, чтобы получить таблицу, содержащую список проблем
datatable_tags = page.select('table.dataTable') # Проблемы в тэге <table>,
                                                # с классом dataTable
# Извлекаем из списка первый тег, поскольку именно он нам нужен
datatable = datatable_tags[0]
# Теперь, поскольку нам нужны имена проблем, они содержатся в тегах <b>, которые
# непосредственно вложены в теги <a>
prob_tags = datatable.select('a > b')
prob_names = [tag.getText().strip() for tag in prob_tags]

print prob_names
```

92.6. Простое скачивание веб-контента с помощью urllib.request

Для скачивания веб-контента можно использовать модуль стандартной библиотеки urllib.request:

```
from urllib.request import urlopen

response = urlopen('http://stackoverflow.com/questions?sort=votes')
data = response.read()
```

```
# Полученные байты обычно должны быть декодированы в соответствии
# с набором символов response
encoding = response.info().get_content_charset()
html = data.decode(encoding)
```

Аналогичный модуль имеется и в Python 2.

92.7. Модификация пользовательского агента Scrapy

Иногда пользовательский агент по умолчанию Scrapy ("Scrapy/VERSION (+http://scrapy.org)") блокируется хостом. Чтобы изменить пользовательский агент по умолчанию, откройте файл settings.py, откомментируйте и измените следующую строку на нужную вам.

```
#USER_AGENT = 'projectName (+http://www.yourdomain.com)'
```

Например:

```
USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_4) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/51.0.2704.103 Safari/537.36'
```

92.8. Скрапинг с использованием инструмента командной строки cURL

Импортируйте:

```
from subprocess import Popen, PIPE
from lxml import etree
from io import StringIO
```

Скачивание:

```
user_agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/55.0.2883.95 Safari/537.36'
url = 'http://stackoverflow.com'
get = Popen(['curl', '-s', '-A', user_agent, url], stdout=PIPE)
result = get.stdout.read().decode('utf8')
```

-s: тихая загрузка

-A: флаг пользовательского агента

Парсинг:

```
tree = etree.parse(StringIO(result), etree.HTMLParser())
divs = tree.xpath('//div')
```

Глава 93. Парсинг HTML

93.1. Использование CSS-селекторов в библиотеке BeautifulSoup

Библиотека BeautifulSoup имеет ограниченную поддержку CSS-селекторов, но охватывает наиболее часто используемые из них. Используйте метод SELECT() для выбора нескольких элементов и select_one() для выбора одного элемента.

Пример:

```
from bs4 import BeautifulSoup
```

```
data = """
<ul>
<li class="item">item1</li>
```

```

<li class="item">item2</li>
<li class="item">item3</li>
</ul>

```

```
soup = BeautifulSoup(data, "html.parser")
```

```

for item in soup.select("li.item"):
    print(item.get_text())

```

Выводит на экран:

```

item1
item2
item3

```

93.2. Библиотека PyQuery

PyQuery – это jquery-подобная библиотека для Python. В ней очень хорошо реализована поддержка селекторов CSS.

```
from pyquery import PyQuery
```

```

html = """
<h1>Sales</h1>
<table id="table">
<tr>
  <td>Lorem</td>
  <td>46</td>
</tr>
<tr>
  <td>Ipsum</td>
  <td>12</td>
</tr>
<tr>
  <td>Dolor</td>
  <td>27</td>
</tr>
<tr>
  <td>Sit</td>
  <td>90</td>
</tr>
</table>

```

```
doc = PyQuery(html)
```

```
title = doc('h1').text()
```

```
print title
```

```
table_data = []
```

```
rows = doc('#table > tr')
```

```
for row in rows:
```

```
    name = PyQuery(row).find('td').eq(0).text()
```

```
    value = PyQuery(row).find('td').eq(1).text()
```

```
    print "%s\t %s" % (name, value)
```

93.3. Нахождение текста после элемента в библиотеке BeautifulSoup

Представьте, что у вас есть следующий HTML:

```
<div>
  <label>Name:</label>
  John Smith
</div>
```

Необходимо расположить текст “John Smith” после элемента label. В этом случае можно найти элемент label по тексту, а затем использовать свойство `.next_sibling`:

```
from bs4 import BeautifulSoup
```

```
data = """
<div>
  <label>Name:</label>
  John Smith
</div>
"""
```

```
soup = BeautifulSoup(data, "html.parser")
```

```
label = soup.find("label", text="Name:")
print(label.next_sibling.strip())
```

В результате на экран будет выведено: John Smith.

Глава 94. Работа с XML

94.1. Открытие и чтение с помощью библиотеки ElementTree

Импортируйте объект ElementTree, откройте соответствующий файл .xml и получите корневой тег:

```
import xml.etree.ElementTree as ET
tree = ET.parse("yourXMLfile.xml")
root = tree.getroot()
```

Существует несколько способов поиска по дереву. Первый – итерационный:

```
for child in root:
    print(child.tag, child.attrib)
```

или можно ссылаться на конкретные места, как на список:

```
print(root[0][1].text)
```

Для поиска конкретных тегов по имени используйте команду `.find` или `.findall`:

```
print(root.findall("myTag"))
print(root[0].find("myOtherTag"))
```

94.2. Создание и построение XML-документов

Импортируйте модуль Element Tree:

```
import xml.etree.ElementTree as ET
```

Функция `Element()` используется для создания элементов XML:

```
p=ET.Element('parent')
```

Функция `SubElement()` используется для создания подэлементов заданного элемента:

```
c = ET.SubElement(p, 'child1')
```

Функция `dump()` используется для сброса элементов xml.

```
ET.dump(p)
```

Выходные данные будут выглядеть следующим образом:

```
#<parent><child1 /></parent>
```

Если необходимо сохранить данные в файл, создайте xml-дерево с помощью функции `ElementTree()`, а для сохранения в файл используйте метод `write()`.

```
tree = ET.ElementTree(p)
```

```
tree.write("output.xml")
```

Функция `Comment()` используется для вставки комментариев в xml-файл.

```
comment = ET.Comment('user comment')
```

```
p.append(comment) # этот комментарий будет добавлен к родительскому элементу
```

94.3. Изменение XML-файла

Импортируйте модуль `Element Tree` и откройте файл xml, получите элемент xml:

```
import xml.etree.ElementTree as ET
```

```
tree = ET.parse('sample.xml')
```

```
root=tree.getroot()
```

```
element = root[0] # получение первого дочернего элемента корневого элемента
```

Объектом элемента можно манипулировать, изменяя его поля, добавляя и изменяя атрибуты, добавляя и удаляя дочерние элементы:

```
element.set('attribute_name', 'attribute_value') # установка атрибута для xml-элемента
```

```
element.text="string_text"
```

Если необходимо удалить элемент, используйте метод `Element.remove()`:

```
root.remove(element)
```

Метод `ElementTree.write()` используется для вывода xml-объекта в файл xml.

```
tree.write('output.xml')
```

94.4. Поиск в XML с помощью XPath

Начиная с версии 2.7 в `ElementTree` улучшена поддержка XPath-запросов. XPath – это синтаксис, позволяющий осуществлять навигацию по xml подобно тому, как язык SQL используется для поиска в базе данных. Функции `find` и `findall` поддерживают XPath. В данном примере будет использован приведенный ниже xml:

```
<Catalog>
  <Books>
    <Book id="1" price="7.95">
      <Title>Do Androids Dream of Electric Sheep?</Title>
      <Author>Philip K. Dick</Author>
    </Book>
    <Book id="5" price="5.95">
      <Title>The Colour of Magic</Title>
      <Author>Terry Pratchett</Author>
    </Book>
    <Book id="7" price="6.95">
      <Title>The Eye of The World</Title>
      <Author>Robert Jordan</Author>
    </Book>
```

```
</Books>
</Catalog>
```

Поиск по всем книгам:

```
import xml.etree.cElementTree as ET
tree = ET.parse('sample.xml')
tree.findall('Books/Book')
```

Поиск книги с названием = 'The Colour of Magic':

```
tree.find("Books/Book[Title='The Colour of Magic']")
# всегда используйте ' ' в правой части сравнения
```

Поиск книги с id = 5:

```
tree.find("Books/Book[@id='5']")
# поиск с xml-атрибутами должен иметь '@' перед именем
```

Поиск второй книги:

```
tree.find("Books/Book[2]")
# индексы начинаются с 1, а не с 0
```

Поиск последней книги:

```
tree.find("Books/Book[last()]")
# 'last' - единственная xpath-функция, допустимая в ElementTree
```

Поиск всех авторов:

```
tree.findall("./Author")
# поиск с // должен использовать относительный путь
```

94.5. Открытие и чтение больших XML-файлов с помощью инкрементного парсинга

Иногда мы не хотим загружать весь файл XML, чтобы получить необходимую информацию. В таких случаях полезно иметь возможность постепенно загружать нужные разделы, а затем удалять их по завершении работы. С помощью инкрементного парсинга можно редактировать дерево элементов, которое сохраняется при разборе XML.

Импортируйте объект ElementTree:

```
import xml.etree.ElementTree as ET

Откройте файл .xml и выполните итерацию по всем элементам:

for event, elem in ET.iterparse("yourXMLfile.xml"):
    ... do something ...
```

В качестве альтернативы можно искать только специфические события, такие как начальные/конечные теги или пространства имен. Если эта опция опущена (как указано выше), то возвращаются только "конечные" события:

```
events=("start", "end", "start-ns", "end-ns")
for event, elem in ET.iterparse("yourXMLfile.xml", events=events):
    ... do something ...
```

Вот полный пример, показывающий, как очистить элементы из дерева памяти, когда мы с ними закончили работу:

```
for event, elem in ET.iterparse("yourXMLfile.xml", events=("start", "end")):
    if elem.tag == "record_tag" and event == "end":
        print elem.text
        elem.clear()
    ... do something else ...
```

Глава 95. Python Requests Post

Рассмотрим использование модуля Python Requests в контексте метода HTTP POST и соответствующей ему функции Requests.

95.1. Простая операция Post

```
from requests import post
```

```
foo = post('http://httpbin.org/post', data = {'key': 'value'})
```

Выполняет простую операцию HTTP POST. Передаваемые данные могут быть самых разных форматов, однако наиболее распространены пары “ключ-значение”.

Заголовки

Можно просматривать заголовки:

```
print(foo.headers)
```

Пример ответа:

```
{'Content-Length': '439', 'X-Processed-Time': '0.000802993774414', 'X-Powered-By': 'Flask',  
'Server': 'meinheld/0.6.1', 'Connection': 'keep-alive', 'Via': '1.1 vegur', 'Access-Control-Allow-Credentials': 'true', 'Date': 'Sun, 21 May 2017 20:56:05 GMT', 'Access-Control-Allow-Origin': '*',  
'Content-Type': 'application/json'}
```

Заголовки также могут быть подготовлены перед отправкой:

```
headers = {'Cache-Control': 'max-age=0',  
          'Upgrade-Insecure-Requests': '1',  
          'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like  
          Gecko) Chrome/54.0.2840.99 Safari/537.36',  
          'Content-Type': 'application/x-www-form-urlencoded',  
          'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',  
          'Referer': 'https://www.groupon.com/signup',  
          'Accept-Encoding': 'gzip, deflate, br',  
          'Accept-Language': 'es-ES,es;q=0.8'}
```

```
foo = post('http://httpbin.org/post', headers=headers, data = {'key': 'value'})
```

Кодирование

Аналогичным образом можно установить и просмотреть кодировку:

```
print(foo.encoding)  
'utf-8'  
foo.encoding = 'ISO-8859-1'
```

Верификация SSL

По умолчанию запросы проверяют SSL-сертификаты доменов. Эта возможность может быть переопределена:

```
foo = post('http://httpbin.org/post', data = {'key': 'value'}, verify=False)
```

Перенаправление

Будет выполняться любое перенаправление (например, с http на https), что также может быть изменено:

```
foo = post('http://httpbin.org/post', data = {'key': 'value'}, allow_redirects=False)
```

Если операция post была перенаправлена, то к этому значению можно получить доступ:

```
print(foo.url)
```

Можно просмотреть полную историю перенаправлений:

```
print(foo.history)
```

95.2. Данные, закодированные в форме

```
from requests import post
```

```
payload = {'key1': 'value1',  
          'key2': 'value2'  
}
```

```
foo = post('http://httpbin.org/post', data=payload)
```

Для передачи закодированных в форме данных с помощью операции `post` эти данные должны быть структурированы в виде словаря и предоставлены в качестве параметра.

Если данные не кодируются в форме, просто передайте строку или целое число в параметр `data`. Для того чтобы Python Requests автоматически отформатировал данные, в параметр `json` следует передать словарь:

```
from requests import post  
payload = {'key1': 'value1', 'key2': 'value2'}  
foo = post('http://httpbin.org/post', json=payload)
```

95.3. Загрузка файлов

В модуле Requests необходимо предоставить только дескриптор файла, в отличие от содержимого, получаемого с помощью функции `.read()`:

```
from requests import post  
files = {'file': open('data.txt', 'rb')}  
foo = post('http://http.org/post', files=files)
```

Также могут быть заданы имя файла, тип содержимого и заголовки:

```
files = {'file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel', {'Expires': '0'})}  
foo = requests.post('http://httpbin.org/post', files=files)
```

Строки также могут быть отправлены в виде файла, если они указаны в качестве параметра `files`.

Несколько файлов

Несколько файлов могут поставляться почти так же, как и один:

```
multiple_files = [  
    ('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),  
    ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))  
]  
  
foo = post('http://httpbin.org/post', files=multiple_files)
```

95.4. Ответы

Коды ответов могут быть просмотрены из `post`-операции:

```
from requests import post  
  
foo = post('http://httpbin.org/post', data={'data': 'value'})  
print(foo.status_code)
```

Возвращенные данные

Доступ к возвращаемым данным:

```
foo = post('http://httpbin.org/post', data={'data': 'value'})  
print(foo.text)
```


Сырые ответы

В тех случаях, когда необходимо получить доступ к базовому объекту urllib3 response. HTTPResponse, это можно сделать следующим образом:

```
foo = post('http://httpbin.org/post', data={'data': 'value'})
res = foo.raw

print(res.read())
```

95.5. Аутентификация

Простая HTTP-аутентификация

Может быть реализована следующим образом:

```
from requests import post
foo = post('http://natas0.natas.labs.overthewire.org', auth=('natas0', 'natas0'))
```

Технически это означает следующее:

```
from requests import post
from requests.auth import HTTPBasicAuth
foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPBasicAuth('natas0', 'natas0'))
```

Аутентификация по протоколу HTTP Digest

Такая аутентификация осуществляется очень похожим образом, но в запросах для этого предусмотрен другой объект:

```
from requests import post
from requests.auth import HTTPDigestAuth
foo = post('http://natas0.natas.labs.overthewire.org', auth=HTTPDigestAuth('natas0', 'natas0'))
```

Пользовательская аутентификация

В некоторых случаях встроенных механизмов аутентификации может быть недостаточно. К примеру, сервер соглашается принять аутентификацию, если отправитель имеет правильную строку user-agent, определенное значение заголовка и предоставляет правильные учетные данные через HTTP Basic Authentication. Для этого необходимо подготовить собственный класс аутентификации, подклассифицирующий AuthBase, который является базовым для реализаций аутентификации запросов:

Это можно использовать с помощью следующего кода:

```
from requests.auth import AuthBase
from requests.auth import _basic_auth_str
from requests._internal_utils import to_native_string

class CustomAuth(AuthBase):

    def __init__(self, secret_header, user_agent, username, password):
        # устанавливаем здесь любые данные, связанные с авторизацией
        self.secret_header = secret_header
        self.user_agent = user_agent
        self.username = username
        self.password = password

    def __call__(self, r):
        # модифицировать и вернуть запрос
        r.headers[self.secret_header] = self.secret_header
        r.headers['User-Agent'] = self.user_agent
        r.headers['Authorization'] = _basic_auth_str(self.username, self.password)

        return r
```

Это можно использовать с помощью следующего кода:

```
foo = get('http://test.com/admin', auth=CustomAuth('SecretHeader', 'CustomUserAgent', 'user',
'password' ))
```

95.6. Прокси-серверы

Каждая операция POST-запроса может быть настроена на использование прокси-серверов.

Прокси-серверы HTTP/S

```
from requests import post
```

```
proxies = {
    'http': 'http://192.168.0.128:3128',
    'https': 'http://192.168.0.127:1080',
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Таким образом может быть обеспечена базовая аутентификация HTTP (HTTP Basic Authentication):

```
proxies = {'http': 'http://user:pass@192.168.0.128:312'}
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Прокси-серверы SOCKS

Для использования прокси-серверов SOCKS требуются сторонние зависимости `requests[socks]`, после установки которых такие прокси-серверы используются аналогично HTTPBasicAuth:

```
proxies = {
    'http': 'socks5://user:pass@host:port',
    'https': 'socks5://user:pass@host:port'
}
```

```
foo = requests.post('http://httpbin.org/post', proxies=proxies)
```

Глава 96. Распространение кода

96.1. py2app

Для использования фреймворка `py2app` необходимо сначала установить его. Для этого откройте терминал и введите следующую команду:

```
sudo easy_install -U py2app
```

Вы также можете установить пакеты с помощью `pip`:

```
pip install py2app
```

Затем создайте установочный файл для вашего Python-скрипта:

```
py2applet --make-setup MyApplication.py
```

Отредактируйте параметры установочного файла по своему усмотрению, по умолчанию используется этот вариант:

```
#!/usr/bin/env python
'''
This is a setup.py script generated by py2applet
'''
```

Usage:

```
python setup.py py2app
```

```
from setuptools import setup
```

```
APP = ['test.py']
DATA_FILES = []
OPTIONS = {'argv_emulation': True}
```

```
setup(
    app=APP,
    data_files=DATA_FILES,
    options={'py2app': OPTIONS},
    setup_requires=['py2app'],
```

Чтобы добавить пиктограмму (иконку), которая должна иметь расширение .icns, или включить изображения в приложение в качестве ссылок, измените параметры, как показано на рисунке:

```
DATA_FILES = ['myInsertedImage.jpg']
OPTIONS = {'argv_emulation': True, 'iconfile': 'myCoolIcon.icns'}
```

Наконец, введите в терминал:

```
python setup.py py2app
```

Скрипт должен быть запущен, и вы обнаружите готовое приложение в папке dist. Для дополнительной настройки можно использовать следующие опции:

optimize (-O)	уровень оптимизации: -O1 для "python -O", -O2 для "python -OO", и -OO для отключения [по умолчанию: -OO]
includes (-i)	список модулей для включения, разделенный запятыми
packages (-p)	список пакетов для включения, разделенный запятыми
extension	Расширение пакета [по умолчанию: .app для приложения, .plugin для плагина]
extra-scripts	разделенный запятыми список дополнительных скриптов для включения в приложение или плагин.

96.2. cx_Freeze

Установите cx_Freeze, распакуйте папку и запустите из нее эти команды:

```
python setup.py build
sudo python setup.py install
```

Создайте новый каталог для вашего Python-скрипта и создайте в этом же каталоге файл setup.py со следующим содержанием:

```
application_title = "My Application" # Используйте собственное имя приложения
main_python_file = "my_script.py"   # Ваш python-скрипт
```

```
import sys
```

```
from cx_Freeze import setup, Executable
```

```
base = None
if sys.platform == "win32":
    base = "Win32GUI"
```

```
includes = ["atexit", "re"]
```

```
setup(
    name = application_title,
    version = "0.1",
```

```
description = "Your Description",
options = {"build_exe" : {"includes" : includes }},
executables = [Executable(main_python_file, base = base)])
```

Теперь запустите файл setup.py из терминала:

```
python setup.py bdist_mac
```

Примечание: на ОС El Capitan эту операцию необходимо выполнить от имени root с отключенным режимом SIP.

Глава 97. Объекты свойств

97.1. Использование декоратора @property для свойств чтения и записи

Если вы хотите использовать @property для реализации пользовательского поведения при установке и получении, используйте этот паттерн:

```
class Cash(object):
    def __init__(self, value):
        self.value = value
    @property
    def formatted(self):
        return '${:.2f}'.format(self.value)
    @formatted.setter
    def formatted(self, new):
        self.value = float(new[1:])
```

Пример:

```
>>> wallet = Cash(2.50)
>>> print(wallet.formatted)
$2.50
>>> print(wallet.value)
2.5
>>> wallet.formatted = '$123.45'
>>> print(wallet.formatted)
$123.45
>>> print(wallet.value)
123.45
```

97.2. Еще об использовании декоратора @property

Декоратор @property может быть использован для определения методов в классе, которые действуют как атрибуты. Например, это может быть полезно при раскрытии информации, которая может потребовать первоначального (ресурсоемкого) поиска и простого извлечения впоследствии. Дан некоторый модуль foobar.py:

```
class Foo(object):
    def __init__(self):
        self.__bar = None

    @property
    def bar(self):
        if self.__bar is None:
            self.__bar = some_expensive_lookup_operation()
        return self.__bar
```

Затем

```
>>> from foobar import Foo
>>> foo = Foo()
>>> print(foo.bar) # Это займет некоторое время, так как после инициализации bar равен None
42
>>> print(foo.bar) # Это гораздо быстрее, так как bar теперь имеет значение
42
```

97.3. Переопределение только функций getter, setter или deleter объекта свойства

При наследовании от класса со свойством можно предоставить новую реализацию одной или нескольких функций получения, установки или удаления (getter, setter, deleter) свойства, обратившись к объекту свойства *в родительском классе*:

```
class BaseClass(object):
    @property
    def foo(self):
        return some_calculated_value()

    @foo.setter
    def foo(self, value):
        do_something_with_value(value)

class DerivedClass(BaseClass):
    @BaseClass.foo.setter
    def foo(self, value):
        do_something_different_with_value(value)
```

Также можно добавить функции setter или deleter туда, где их раньше не было.

97.4. Использование свойств без декораторов

Использование синтаксиса декораторов (с символом @) удобно, но в то же время несколько скрытно. Свойства можно использовать напрямую, без декораторов. Это показано в следующем примере Python 3.x:

```
class A:
    p = 1234
    def getX (self):
        return self._x

    def setX (self, value):
        self._x = value

    def getY (self):
        return self._y

    def setY (self, value):
        self._y = 1000 + value # Странно, но возможно

    def getY2 (self):
        return self._y

    def setY2 (self, value):
        self._y = value

    def getT (self):
        return self._t
```

```
def setT (self, value):
    self._t = value

def getU (self):
    return self._u + 10000

def setU (self, value):
    self._u = value - 5000

x, y, y2 = property (getX, setX), property (getY, setY), property (getY2, setY2)
t = property (getT, setT)
u = property (getU, setU)

A.q = 5678

class B:
    def getZ (self):
        return self.z_

    def setZ (self, value):
        self.z_ = value

    z = property (getZ, setZ)

class C:
    def __init__ (self):
        self.offset = 1234

    def getW (self):
        return self.w_ + self.offset

    def setW (self, value):
        self.w_ = value - self.offset

    w = property (getW, setW)

a1 = A ()
a2 = A ()

a1.y2 = 1000
a2.y2 = 2000

a1.x = 5
a1.y = 6

a2.x = 7
a2.y = 8

a1.t = 77
a1.u = 88

print (a1.x, a1.y, a1.y2)
print (a2.x, a2.y, a2.y2)
print (a1.p, a2.p, a1.q, a2.q)

print (a1.t, a1.u)

b = B ()
c = C ()
```

```

b.z = 100100
c.z = 200200
c.w = 300300

print (a1.x, b.z, c.z, c.w)

c.w = 400400
c.z = 500500
b.z = 600600

print (a1.x, b.z, c.z, c.w)

```

Глава 98. Перегрузка

98.1. Перегрузка операторов

Ниже перечислены операторы, которые могут быть перегружены в классах, а также необходимые определения методов и пример использования оператора в выражении.

N.B. Использование `other` в качестве имени переменной не является обязательным, но считается нормой.

Оператор	Метод	Выражение
+ Сложение	<code>__add__(self, other)</code>	<code>a1 + a2</code>
- Вычитание	<code>__sub__(self, other)</code>	<code>a1 - a2</code>
* Умножение	<code>__mul__(self, other)</code>	<code>a1 * a2</code>
@ Умножение матриц	<code>__matmul__(self, other)</code>	<code>a1 @ a2</code> (Python 3.5)
/ Деление	<code>__div__(self, other)</code>	<code>a1 / a2</code> (только Python 2)
/ Деление	<code>__truediv__(self, other)</code>	<code>a1 / a2</code> (Python 3)
// Целочисленное деление	<code>__floordiv__(self, other)</code>	<code>a1 // a2</code>
% Остаток от деления	<code>__mod__(self, other)</code>	<code>a1 % a2</code>
** Возведение в степень	<code>__pow__(self, other[, modulo])</code>	<code>a1 ** a2</code>
<< Побитовый сдвиг влево	<code>__lshift__(self, other)</code>	<code>a1 << a2</code>
>> Побитовый сдвиг вправо	<code>__rshift__(self, other)</code>	<code>a1 >> a2</code>
& Побитовое И	<code>__and__(self, other)</code>	<code>a1 & a2</code>
^ Побитовое XOR	<code>__xor__(self, other)</code>	<code>a1 ^ a2</code>
(побитовое ИЛИ)	<code>__or__(self, other)</code>	<code>a1 a2</code>
- Отрицание (арифметическое)	<code>__neg__(self)</code>	<code>-a1</code>
+ Положительное	<code>__pos__(self)</code>	<code>+a1</code>
~ Побитовое НЕ	<code>__invert__(self)</code>	<code>~a1</code>
< Меньше	<code>__lt__(self, other)</code>	<code>a1 < a2</code>
<= Меньше или равно	<code>__le__(self, other)</code>	<code>a1 <= a2</code>
= Равно	<code>__eq__(self, other)</code>	<code>a1 == a2</code>
!= Не равно	<code>__ne__(self, other)</code>	<code>a1 != a2</code>
> Больше	<code>__gt__(self, other)</code>	<code>a1 > a2</code>
>= Больше или равно	<code>__ge__(self, other)</code>	<code>a1 >= a2</code>
[index] оператор индекса	<code>__getitem__(self, index)</code>	<code>a1[index]</code>
in оператор "in"	<code>__contains__(self, other)</code>	<code>a2 in a1</code>
(*args, ...) вызов	<code>__call__(self, *args, **kwargs)</code>	<code>a1(*args, **kwargs)</code>

Необязательный параметр `modulo` для `__pow__` используется только встроенной функцией `pow`.

Каждый из методов, соответствующих *бинарному* оператору, имеет соответствующий “правый”, “right” метод, начинающийся с `_r`, например `__radd__`:

```
class A:
    def __init__(self, a):
        self.a = a
    def __add__(self, other):
        return self.a + other
    def __radd__(self, other):
        print("radd")
        return other + self.a
```

```
A(1) + 2 # Вывод: 3
2 + A(1) # выводит radd. Вывод: 3
```

а также соответствующая *inplace*-версия, начинающаяся с `i`:

```
class B:
    def __init__(self, b):
        self.b = b
    def __iadd__(self, other):
        self.b += other
        print("iadd")
        return self
```

```
b = B(2)
b.b      # Вывод: 2
b += 1   # выводит iadd
b.b      # Вывод: 3
```

Поскольку в этих методах нет ничего особенного, многие другие части языка, части стандартной библиотеки и даже сторонние модули сами добавляют “магические” методы, например, методы приведения объекта к типу или проверки свойств объекта. Например, встроенная функция `str()` вызывает метод `__str__` объекта, если он существует. Некоторые из этих вариантов использования перечислены ниже.

Функция	Метод	Выражение
Приведение к <code>int</code>	<code>__int__(self)</code>	<code>int(a1)</code>
Абсолютная функция	<code>__abs__(self)</code>	<code>abs(a1)</code>
Приведение к <code>str</code>	<code>__str__(self)</code>	<code>str(a1)</code>
Приведение к <code>Unicode</code>	<code>__unicode__(self)</code>	<code>unicode(a1)</code> (только для Python 2)
Строковое представление	<code>__repr__(self)</code>	<code>repr(a1)</code>
Приведение к <code>bool</code>	<code>__nonzero__(self)</code>	<code>bool(a1)</code>
Форматирование строк	<code>__format__(self, formatstr)</code>	<code>"Hi {abc}".format(a1)</code>
Хеширование	<code>__hash__(self)</code>	<code>hash(a1)</code>
Длина	<code>__len__(self)</code>	<code>len(a1)</code>
Переворачивание	<code>__reversed__(self)</code>	<code>reversed(a1)</code>
Округление в меньшую сторону	<code>__floor__(self)</code>	<code>math.floor(a1)</code>
Округление в большую сторону	<code>__ceil__(self)</code>	<code>math.ceil(a1)</code>

Существуют также специальные методы `__enter__` и `__exit__` для менеджеров контекста и многие другие.

98.2. “Магические” методы, или Dunder-методы

“Магические” методы (также называемые Dunder-методы, Dunder – аббревиатура от double-underscore, “двойное подчеркивание”) в Python служат целям, аналогичным перегрузке операторов в других языках. Они позволяют классу определить свое поведение при использовании его в качестве операнда в унарных или бинарных операторных выражениях. Они также служат реализациями, вызываемыми некоторыми встроенными функциями.

Рассмотрим данную реализацию двумерных векторов.

```
import math

class Vector(object):
    # инстанцирование
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # унарное отрицание (-v)
    def __neg__(self):
        return Vector(-self.x, -self.y)

    # сложение (v + u)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # вычитание (v - u)
    def __sub__(self, other):
        return self + (-other)

    # равенство (v == u)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # abs(v)
    def __abs__(self):
        return math.hypot(self.x, self.y)

    # str(v)
    def __str__(self):
        return '<{0.x}, {0.y}>'.format(self)

    # repr(v)
    def __repr__(self):
        return 'Vector({0.x}, {0.y})'.format(self)
```

Теперь можно естественным образом использовать экземпляры класса Vector в различных выражениях.

```
v = Vector(1, 4)
u = Vector(2, 0)

u + v          # Vector(3, 4)
print(u + v)   # "<3, 4>" (неявное преобразование строки)
u - v          # Vector(1, -4)
u == v         # False
u + v == v + u # True
abs(u + v)     # 5.0
```

98.3. Типы контейнеров и последовательностей

Возможна эмуляция контейнерных типов, поддерживающих доступ к значениям по ключу или индексу.

Рассмотрим эту наивную реализацию разреженного списка, в котором для экономии памяти хранятся только ненулевые элементы.

```
class sparselist(object):
    def __init__(self, size):
        self.size = size
        self.data = {}

    # l[index]
    def __getitem__(self, index):
        if index < 0:
            index += self.size
        if index >= self.size:
            raise IndexError(index)
        try:
            return self.data[index]
        except KeyError:
            return 0.0

    # l[index] = value
    def __setitem__(self, index, value):
        self.data[index] = value

    # del l[index]
    def __delitem__(self, index):
        if index in self.data:
            del self.data[index]

    # value in l
    def __contains__(self, value):
        return value == 0.0 or value in self.data.values()

    # len(l)
    def __len__(self):
        return self.size

    # for value in l ...
    def __iter__(self):
        return (self[i] for i in range(self.size)) # используйте xrange для python2

Тогда мы можем использовать sparselist, подобно обычному списку.

l = sparselist(10 ** 6)      # список с 1 млн элементов
0 in l                       # True
10 in l                      # False

l[12345] = 10
10 in l                      # True
l[12345]                     # 10

for v in l:
    pass # 0, 0, 0, ... 10, 0, 0 ... 0
```

98.4. Вызываемые типы

```
class adder(object):
    def __init__(self, first):
        self.first = first
```

```
# a(...)
def __call__(self, second):
    return self.first + second
```

```
add2 = adder(2)
add2(1) # 3
add2(2) # 4
```

98.5. Обработка нереализованного поведения

Если в вашем классе не реализован специальный перегруженный оператор для указанных типов аргументов, то он должен возвращать `NotImplemented` (обратите внимание, что это специальная константа, а не то же самое, что `NotImplementedError`). Это позволит Python вернуться к попыткам использовать другие методы для выполнения операции:

Если возвращается `NotImplemented`, то интерпретатор пытается выполнить отраженную операцию для другого типа, либо другой запасной вариант, в зависимости от оператора. Если все попытки возвращают `NotImplemented`, то интерпретатор выдаст соответствующее исключение.

Например, при заданных $x + y$, если $x.__add__(y)$ возвращается нереализованным, вместо него делается попытка вычисления $y.__radd__(x)$.

```
class NotAddable(object):

    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return NotImplemented

class Addable(NotAddable):

    def __add__(self, other):
        return Addable(self.value + other.value)

    __radd__ = __add__
```

Поскольку речь идет об отраженном методе, следует реализовать `__add__` и `__radd__`, чтобы получить ожидаемое поведение во всех случаях; к счастью, поскольку в этом простом примере они оба выполняют одно и то же действие, мы можем воспользоваться коротким путем.

Использование:

```
>>> x = NotAddable(1)
>>> y = Addable(2)
>>> x + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NotAddable' and 'NotAddable'
>>> y + y
<so.Addable object at 0x1095974d0>
>>> z = x + y
>>> z
<so.Addable object at 0x109597510>
>>> z.value
3
```

Глава 99. Полиморфизм

99.1. “Утиная типизация”

Полиморфизм без наследования в виде т. н. “утиной типизации” доступен в Python благодаря системе динамической типизации. Это означает, что до тех пор, пока классы содержат одинаковые методы, интерпретатор Python не делает различий между ними, поскольку единственная проверка вызовов происходит во время выполнения.

```
class Duck:
    def quack(self):
        print("Кря!")
    def feathers(self):
        print("У утки белые и серые перья.")

class Person:
    def quack(self):
        print("Человек имитирует утку.")
    def feathers(self):
        print("Человек берет с земли перо и показывает его.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()

donald = Duck()
john = Person()
in_the_forest(donald)

in_the_forest(john)
На выходе получаем:

Кря!
Утка имеет белые и серые перья.
Человек имитирует утку.
Человек берет с земли перо и показывает его.
```

99.2. Базовый полиморфизм

Полиморфизм – это возможность выполнять действие над объектом независимо от его типа. Обычно это реализуется путем создания базового класса и двух или более подклассов, реализующих методы с одинаковой сигнатурой. Любая другая функция или метод, манипулирующая этими объектами, может вызывать те же методы независимо от типа объекта, с которым она работает, без необходимости предварительной проверки типа. В объектно-ориентированной терминологии, когда класс X расширяет класс Y, то Y называется суперклассом или базовым классом, а X – подклассом или производным классом.

```
class Shape:
    """
    Это родительский класс, который предназначен для наследования другими классами
    """

    def calculate_area(self):
        """
        Этот метод должен переопределяться в подклассах.
        Если подкласс не реализует его, но он вызывается, то будет выдано
        сообщение NotImplemented.
        """
```

```
"""
```

```
raise NotImplemented
```

```
class Square(Shape):
```

```
"""
```

```
Этот класс является подклассом класса Shape и представляет собой квадрат
```

```
"""
```

```
side_length = 2 # в данном примере длина сторон равна 2 единицам
```

```
def calculate_area(self):
```

```
"""
```

```
Этот метод переопределяет Shape.calculate_area(). Когда у объекта типа Square  
вызывается метод calculate_area(), то будет вызван именно этот метод, а не версия  
родительского класса.
```

```
Он выполняет вычисления, необходимые для данной формы - квадрата,  
и возвращает результат.
```

```
"""
```

```
return self.side_length * 2
```

```
class Triangle(Shape):
```

```
"""
```

```
Это также подкласс класса Shape, и он представляет собой треугольник
```

```
"""
```

```
base_length = 4
```

```
height = 3
```

```
def calculate_area(self):
```

```
"""
```

```
Этот метод также переопределяет Shape.calculate_area() и выполняет вычисление  
площади треугольника, возвращая результат.
```

```
"""
```

```
return 0.5 * self.base_length * self.height
```

```
def get_area(input_obj):
```

```
"""
```

```
Эта функция принимает входной объект и вызывает метод calculate_area() этого объекта.  
Обратите внимание, что тип объекта не указывается. Это может быть объект  
Square, Triangle или Shape. """
```

```
print(input_obj.calculate_area())
```

```
# Создать по одному объекту каждого класса
```

```
shape_obj = Shape()
```

```
square_obj = Square()
```

```
triangle_obj = Triangle()
```

```
# Теперь передадим каждый объект по очереди в функцию get_area() и посмотрим результат
```

```
get_area(shape_obj)
```

```
get_area(square_obj)
```

```
get_area(triangle_obj)
```

```
В результате получаем:
```

```
None
```

```
4
```

```
6.0
```

Что происходит без полиморфизма?

Без полиморфизма перед выполнением действия над объектом может потребоваться проверка типа, чтобы определить правильный метод для вызова. Следующий “контрпример” выполняет ту же задачу, что и предыдущий код, но без использования полиморфизма, и функции `get_area()` приходится выполнять больше работы.

```
class Square:

    side_length = 2

    def calculate_square_area(self):
        return self.side_length ** 2

class Triangle:

    base_length = 4
    height = 3

    def calculate_triangle_area(self):
        return (0.5 * self.base_length) * self.height

def get_area(input_obj):

    # Обратите внимание на проверки типов, которые теперь необходимы. Такие проверки типов
    # могут стать очень сложными, что приведет
    # к дублированию и сложностям в сопровождении кода.

    if type(input_obj).__name__ == "Square":
        area = input_obj.calculate_square_area()

    elif type(input_obj).__name__ == "Triangle":
        area = input_obj.calculate_triangle_area()

    print(area)

# Создать по одному объекту каждого класса
square_obj = Square()
triangle_obj = Triangle()

# Теперь передадим каждый объект по очереди в функцию get_area() и посмотрим результат
get_area(square_obj)
get_area(triangle_obj)

В результате получаем:

4
6.0
```

Важное замечание

Обратите внимание, что классы, используемые в примере без полиморфизма, являются классами “нового стиля” и неявно наследуются от класса `object`, если используется Python 3. Полиморфизм будет работать как в Python 2.x, так и в 3.x, но код “контрпримера” вызовет исключение при запуске в интерпретаторе Python 2.x, поскольку `type(input_obj).name` вернет “экземпляр” вместо имени класса, если они явно не наследуются от объекта, в результате чего `area` никогда не будет присвоена.

Глава 100. Переопределение методов

100.1. Переопределение базовых методов

Приведем пример базового переопределения в Python (для наглядности и совместимости с Python 2 и 3 используется класс нового стиля и `print с ()`):

```
class Parent(object):
    def introduce(self):
        print("Hello!")

    def print_name(self):
        print("Parent")
```

```
class Child(Parent):
    def print_name(self):
        print("Child")
```

```
p = Parent()
c = Child()
```

```
p.introduce()
p.print_name()
```

```
c.introduce()
c.print_name()
```

```
$ python basic_override.py
Hello!
Parent
Hello!
Child
```

Когда создается класс `Child`, он наследует методы класса `Parent`. Это означает, что все методы, которые есть у родительского класса, будут и у дочернего класса. В примере `introduce` определен для класса `Child`, поскольку он определен для `Parent`, несмотря на то, что он не был определен явно в определении класса `Child`.

В данном примере переопределение происходит, когда `Child` определяет свой собственный метод `print_name`. Если бы этот метод не был объявлен, то `c.print_name()` вывел бы "Parent". Однако `Child` переопределил родительское определение метода `print_name`, и теперь при вызове `c.print_name()` выводится слово "Child".

Глава 101. Пользовательские методы

101.1. Создание пользовательских объектов метода

Объекты пользовательских методов могут быть созданы при получении атрибута класса (возможно, через экземпляр этого класса), если этот атрибут является объектом пользовательской функции, несвязанным объектом пользовательского метода или объектом метода класса.

```
class A(object):
    # func: Объект функции, определяемый пользователем
    #
```

```
# Обратите внимание, что func - это объект функции, когда он определен,
# и несвязанный (unbound) объект метода, когда он извлекается.
def func(self):
    pass
```

```
# classMethod: Метод класса
@classmethod
def classMethod(self):
    pass
```

```
class B(object):
    # unboundMeth: Объект несвязанного пользовательского метода
    #
    # Parent.func здесь является несвязанным объектом пользовательского метода,
    # потому что он извлекается.
    unboundMeth = A.func

a = A()
b = B()
```

```
print A.func
# результат: <unbound method A.func>
print a.func
# результат: <bound method A.func of <__main__.A object at 0x10e9ab910>>
print B.unboundMeth
# результат: <unbound method A.func>
print b.unboundMeth
# результат: <unbound method A.func>
print A.classMethod
# результат: <bound method type.classMethod of <class '__main__.A'>
print a.classMethod
# результат: <bound method type.classMethod of <class '__main__.A'>
```

Если атрибут является пользовательским объектом метода, то новый объект метода создается только в том случае, если класс, из которого он извлекается, совпадает с классом, хранящимся в исходном объекте метода, или является производным классом от него; в противном случае исходный объект метода используется как есть.

```
# Parent: Класс, хранящийся в исходном объекте метода
class Parent(object):
    # func: Функция, лежащая в основе исходного объекта метода
    def func(self):
        pass
    func2 = func
```

```
# Child: Производный класс от класса Parent
class Child(Parent):
    func = Parent.func
```

```
# AnotherClass: Другой класс, не подкласс и не имеющий подклассов
class AnotherClass(object):
    func = Parent.func
```

```
print Parent.func is Parent.func          # False, создан новый объект
print Parent.func2 is Parent.func2        # False, создан новый объект
print Child.func is Child.func            # False, создан новый объект
print AnotherClass.func is AnotherClass.func # True, используется исходный объект
```


101.2. Пример с использованием Turtle Graphics

Ниже приведен пример использования пользовательской функции, которую можно легко вызвать несколько раз в сценарии.

```
import turtle, time, random # сообщим python, что нам нужно 3 разных модуля
turtle.speed(0) # установим скорость рисования на самую быструю
turtle.colormode(255) # специальный режим цвета
turtle.pensize(4) # размер линий, которые будут нарисованы
def triangle(size): # Это наша собственная функция, в скобках указана переменная, которая
# будет использоваться только в этой функции. Эта функция создает правильный треугольник
    turtle.forward(size) # для начала этой функции мы продвигаемся вперед, величина, на которую
# нужно продвинуться, является переменной size
    turtle.right(90) # поворот вправо на 90 градусов
    turtle.forward(size) # переход вперед, снова с переменной
    turtle.right(135) # снова поворот вправо
    turtle.forward(size * 1.5) # закрыть треугольник. Благодаря теореме Пифагора мы знаем,
# что эта линия должна быть в 1.5 раза длиннее двух других (если они равны)
while(1): #бесконечный цикл
    turtle.setpos(random.randint(-200, 200), random.randint(-200, 200)) # установить точку рисования
# в случайную позицию (x,y)
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
# рандомизация цвета RGB
    triangle(random.randint(5, 55)) # используем нашу функцию, поскольку она имеет только одну
# переменную, мы можем просто поместить значение в круглые скобки. Значение, которое
# будет отправлено, случайное в диапазоне от 5 до 55, в конечном счете это просто изменит
# размер треугольника.
    turtle.pencolor(random.randint(1, 255), random.randint(1, 255), random.randint(1, 255))
# снова рандомизация цвета
```

Глава 102. Строковые представления экземпляров классов: `__str__` и `__repr__` методы

102.1. Мотивация

Допустим, что вы создали свой первый класс в Python – маленький аккуратный класс, инкапсулирующий игральную карту:

```
class Card:
    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips
```

В других местах вашего кода вы создаете несколько экземпляров этого класса:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs', 4)
six_of_hearts = Card('Hearts', 6)
```

Вы даже создали список карт, чтобы представить “руку”:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

Теперь, во время отладки, вы хотите увидеть, как выглядит ваша рука, поэтому вы делаете то, что приходит в голову само собой, и пишете:

```
print(my_hand)
```

Но в ответ вы получаете кучу белиберды:

```
<__main__.Card instance at 0x0000000002533788>,  
<__main__.Card instance at 0x00000000025B95C8>,  
<__main__.Card instance at 0x00000000025FF508>]
```

Сбитый с толку, вы попытаетесь просто напечатать одну карту:

```
print(ace_of_spades)
```

И снова получается странный вывод:

```
<__main__.Card instance at 0x0000000002533788>
```

Не бойтесь. Мы разберемся с этим. Однако сначала необходимо понять, что здесь происходит. Когда вы написали `print(ace_of_spades)`, вы сказали Python, что хотите, чтобы он вывел информацию об экземпляре `Card`, который в вашем коде называется `ace_of_spades`. И, честно говоря, так он и вывел. Этот вывод состоит из типа объекта и его идентификатора. Только второй части (шестнадцатеричного числа) достаточно, чтобы однозначно идентифицировать объект на момент вызова `print`.

На самом деле все происходило так: вы просили Python “выразить словами” сущность этого объекта, а затем отобразить ее вам. Более явная версия того же механизма может быть такой:

```
string_of_card = str(ace_of_spades)  
print(string_of_card)
```

В первой строке вы пытаетесь превратить экземпляр `Card` в строку, а во второй – вывести его на экран.

Проблема

Проблема, с которой вы столкнулись, связана с тем, что, сообщив Python все, что ему нужно знать о классе `Card` для создания карт, вы не сообщили ему, каким образом экземпляры `Card` должны быть преобразованы в строки. А поскольку он этого не знал, то, когда вы (неявно) написали `str(ace_of_spades)`, он выдал вам то, что вы видели, – общее представление экземпляра `Card`.

Решение (часть 1)

Но мы можем указать Python, как мы хотим, чтобы экземпляры наших пользовательских классов преобразовывались в строки. И сделать это можно с помощью метода “dunder” (double-underscore) или, другими словами, “магического” метода – `__str__`. Всякий раз, когда вы говорите Python создать строку из экземпляра класса, он будет искать метод `__str__` в классе и вызовет его. Рассмотрим следующий, обновленный вариант нашего класса `Card`:

```
class Card:  
    def __init__(self, suit, pips):  
        self.suit = suit  
        self.pips = pips  
  
    def __str__(self):  
        special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}  
  
        card_name = special_names.get(self.pips, str(self.pips))  
  
        return "%s of %s" % (card_name, self.suit)
```

Здесь мы определили метод `__str__` в нашем классе `Card`, который после простого поиска карт по словарю **возвращает** строку, отформатированную так, как мы решили.

(Обратите внимание, что слово “возвращает” выделено жирным шрифтом, чтобы подчеркнуть важность возврата строки, а не просто ее печати. Напечатать ее, казалось бы, можно, но тогда при выполнении команды `str(ace_of_spades)` будет напечатана карта, даже не имея вызова функции `print` в основной программе. Поэтому, чтобы внести ясность, убедитесь, что `__str__` возвращает строку).

Так как `__str__` является методом, его первым аргументом будет `self`, и он не должен ни принимать, ни передавать дополнительные аргументы.

Возвращаясь к нашей проблеме отображения карты в более удобном для пользователя виде, если мы снова выполним:

```
ace_of_spades = Card('Spades', 1)
print(ace_of_spades)
```

мы увидим, что наш результат будет гораздо лучше:

```
Ace of Spades
```

Отлично, мы закончили, верно? Но для того чтобы подстраховаться, давайте еще раз проверим, что мы решили первую возникшую проблему, выведя на печать `hand` – список экземпляров `Card`. Перепроверяем следующий код:

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
print(my_hand)
```

И, к нашему удивлению, мы снова получаем эти забавные шестнадцатеричные коды:

```
[<__main__.Card instance at 0x00000000026F95C8>,
<__main__.Card instance at 0x000000000273F4C8>,
<__main__.Card instance at 0x0000000002732E08>]
```

Что происходит? Мы сказали Python, как мы хотим, чтобы наши экземпляры `Card` отображались, но почему он забыл?

Решение (часть 2)

Когда Python хочет получить строковое представление элементов в списке, механизм за кулисами немного меняется. Оказывается, Python не заботится о `__str__` для этой цели. Вместо этого он ищет другой метод, `__repr__`, и если он не найден, то возвращается к шестнадцатеричному представлению.

То есть следует создать два метода для выполнения одного и того же действия? Один для случая, когда я хочу вывести карту саму по себе, а другой – когда она находится в каком-то контейнере?

Нет, но сначала давайте посмотрим, как *выглядел бы* наш класс, если бы мы реализовали и `__str__`, и `__repr__` методы:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (S)" % (card_name, self.suit)

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s (R)" % (card_name, self.suit)
```

Здесь реализация двух методов абсолютно одинакова, за исключением того, что для различия между этими двумя методами к строкам, возвращаемым `__str__`, добавляется символ (S), а к строкам, возвращаемым `__repr__`, добавляется символ (R). Обратите внимание, что, как и в нашем методе `__str__`, метод `__repr__` не принимает никаких аргументов и **возвращает** строку. Теперь можно увидеть, какой метод отвечает за каждый случай:

```
ace_of_spades = Card('Spades', 1)
four_of_clubs = Card('Clubs', 4)
six_of_hearts = Card('Hearts', 6)
```

```
my_hand = [ace_of_spades, four_of_clubs, six_of_hearts]
```

```
print(my_hand)           # [Ace of Spades (R), 4 of Clubs (R), 6 of Hearts (R)]

print(ace_of_spades)     # Ace of Spades (S)
```

Как уже говорилось, метод `__str__` был вызван, когда мы передали наш экземпляр `Card` в `print`, а метод `__repr__` вызывался, когда мы передавали в `print` *список наших экземпляров*.

Здесь стоит отметить, что подобно тому, как мы можем явно создать строку из экземпляра пользовательского класса с помощью функции `str()`, как мы это делали ранее, мы также можем явно создать **строковое представление** нашего класса с помощью встроенной функции `repr()`.

Например:

```
str_card = str(four_of_clubs)
print(str_card)           # 4 of Clubs (S)

repr_card = repr(four_of_clubs)
print(repr_card)          # 4 of Clubs (R)
```

И, кроме того, в случае необходимости мы *могли бы* вызывать методы напрямую (хотя это кажется несколько непонятным и ненужным):

```
print(four_of_clubs.__str__())    # 4 of Clubs (S)

print(four_of_clubs.__repr__())   # 4 of Clubs (R)
```

По поводу дублирования функций...

Разработчики Python поняли, что в случае, если вы хотите, чтобы из `str()` и `repr()` возвращались одинаковые строки, вам придется функционально дублировать методы – а это никому не нравится.

Поэтому существует механизм, устраняющий необходимость в этом. Оказывается, если класс реализует метод `__repr__`, но не реализует метод `__str__` и вы передаете экземпляр этого класса в `str()` (неявно или явно), то Python вернется к вашей реализации `__repr__` и использует его. Итак, для наглядности рассмотрим следующий вариант класса `Card`:

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    def __repr__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)
```

Обратите внимание, что в этой версии реализован *только* метод `__repr__`. Тем не менее вызов `str()` приводит к удобной для пользователя версии:

```
print(six_of_hearts)       # 6 of Hearts (неявное преобразование)
print(str(six_of_hearts))  # 6 of Hearts (явное преобразование)

как и вызовы repr():

print([six_of_hearts])     #[6 of Hearts] (неявное преобразование)
print(repr(six_of_hearts)) # 6 of Hearts (явное преобразование)
```

Резюме

Для того чтобы дать возможность экземплярам класса “проявлять себя” удобным для пользователя образом, необходимо реализовать, по крайней мере, метод `__repr__` вашего класса. Во время своего выступления Раймонд Хеттингер сказал, что обеспечение реализации классами метода `__repr__` является одной из первых вещей, на которые он обращает внимание при проверке кода на Python, и сейчас уже должно быть понятно, почему. Количество информации, которую *можно* было бы добавить в отладочные утверждения, отчеты о сбоях

или лог-файлы с помощью простого метода, просто ошеломляет по сравнению с той мизерной и зачастую не слишком полезной информацией (тип, id), которая предоставляется по умолчанию.

Если вам нужны *разные* представления, например внутри контейнера, вы будете реализовывать и `__repr__`, и `__str__` методы. (Подробнее о том, как можно использовать эти два метода по-разному, будет сказано ниже.)

102.2. Реализация обоих методов, `__repr__()` в стиле `eval-round-trip`

```
class Card:
    special_names = {1:'Ace', 11:'Jack', 12:'Queen', 13:'King'}

    def __init__(self, suit, pips):
        self.suit = suit
        self.pips = pips

    # Вызывается, когда экземпляр преобразуется в строку с помощью функции str()
    # Примеры:
    # print(card1)
    # print(str(card1))
    def __str__(self):
        card_name = Card.special_names.get(self.pips, str(self.pips))
        return "%s of %s" % (card_name, self.suit)

    # Вызывается, когда экземпляр преобразуется в строку с помощью repr()
    # Примеры:
    # print([card1, card2, card3])
    # print(repr(card1))
    def __repr__(self):
        return "Card(%s,%d)" % (self.suit, self.pips)
```

Глава 103. Отладка

103.1. При помощи IPython и ipdb

Если установлен IPython (или Jupyter), то отладчик можно вызвать с помощью:

```
import ipdb
ipdb.set_trace()
```

Код завершает работу и выводит на экран:

```
/home/usr/ook.py(3)<module>()
  1 import ipdb
  2 ipdb.set_trace()
--> 3 print("Hello world!")
```

```
ipdb>
```

что означает необходимость редактирования кода. Существует более простой способ:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

Это приведет к вызову отладчика при возникновении неперехваченного исключения.

103.2. Отладчик Python. Пошаговая отладка с помощью pdb

Стандартная библиотека Python включает интерактивную библиотеку отладки `pdb`. Она обладает широкими возможностями, наиболее часто используемой из которых является возможность “пошагового выполнения” программы.

Для немедленного перехода к пошаговой отладке используйте:

```
python -m pdb <my_file.py>
```

Это приведет к запуску отладчика с первой строки программы. Обычно для отладки требуется выделить определенный участок кода. Для этого мы импортируем библиотеку `pdb` и с помощью функции `set_trace()` прервем поток этого проблемного примера кода.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # Что здесь не так? Подсказка: 2 != 3

print divide(1, 2)
```

Запуск этой программы приводит к запуску интерактивного отладчика.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Часто эта команда используется в одной строке, поэтому она может быть закомментирована одним символом `#`.

```
import pdf; pdb.set_trace()
```

В приглашении (Pdb) можно вводить команды. Эти команды могут быть командами отладчика или Python. Для печати переменных можно использовать команду `p` из отладчика или `print` из Python.

```
(Pdb) p a
1
(Pdb) print a
1
```

Для просмотра списка всех локальных переменных используйте

```
locals
```

Полезные команды отладчика, которые необходимо знать:

```
b | : установить точку останова на строке *n* или функции с именем *f*.
# b 3
# b divide
b: показать все точки останова.
c: продолжить работу до следующей точки останова.
s: перешагнуть через эту строку (перейдет в функцию).
n: перешагнуть через эту строку (перепрыгнет через функцию).
r: продолжить работу до возврата текущей функции.
l: перечислить окно кода вокруг этой строки.
p : печать переменной с именем *var*.
# p x
q: выход из отладчика.
bt: печать трассировки стека вызовов текущего выполнения
up: переместить область видимости вверх по стеку вызовов функций к вызывающей функции
down: Переместить область видимости на один уровень вниз по стеку вызовов функций
step: Выполнить программу до следующей строки выполнения в программе, затем вернуть управление обратно отладчику
next: выполнение программы до следующей строки в текущей функции, затем возврат управления обратно в отладчик
```

return: выполнение программы до возврата текущей функции, затем возврат управления обратно в отладчик
 continue: продолжить выполнение программы до следующей точки останова (или повторного вызова set_trace si)

Отладчик также может оценивать Python в интерактивном режиме:

```
-> return a/b
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Примечание. Если имена переменных совпадают с командами отладчика, используйте восклицательный знак '!' перед переменной, чтобы явно сослаться на переменную, а не на команду отладчика. Например, часто бывает так, что вы используете имя переменной 'c' для счетчика и хотите вывести его в отладчике. Простая команда 'c' продолжит выполнение до следующей точки останова. Вместо этого используйте команду '!c', чтобы вывести значение переменной следующим образом:

```
(Pdb) !c
4
```

103.3. Удаленный отладчик

Иногда требуется отладить Python-код, выполняемый другим процессом, в таких случаях на помощь приходит rpdb. Это обертка вокруг pdb, которая перенаправляет stdin и stdout в обработчик сокетов. По умолчанию он открывает отладчик на порту 4444.

Использование:

```
# В файле Python, который вы хотите отлаживать
import rpdb
rpdb.set_trace()
```

А затем для подключения к этому процессу необходимо выполнить в терминале следующее.

```
# Вызов терминала для просмотра вывода
$ nc 127.0.0.1 4444
```

и вы получите pdb подсказку:

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Глава 104. Чтение и запись в формате CSV

104.1. Использование pandas

Запись CSV из dict или DataFrame:

```
import pandas as pd

d = {'a': (1, 101), 'b': (2, 202), 'c': (3, 303)}
pd.DataFrame.from_dict(d, orient="index")
df.to_csv("data.csv")
```

Чтение CSV как DataFrame и преобразование его в dict:

```
df = pd.read_csv("data.csv")
d = df.to_dict()
```

104.2. Запись файла TSV

Python:

```
import csv
```

```
with open('/tmp/output.tsv', 'wt') as out_file:
    tsv_writer = csv.writer(out_file, delimiter='\t')
    tsv_writer.writerow(['name', 'field'])
    tsv_writer.writerow(['Dijkstra', 'Computer Science'])
    tsv_writer.writerow(['Shelah', 'Math'])
    tsv_writer.writerow(['Aumann', 'Economic Sciences'])
```

Файл вывода:

```
$ cat /tmp/output.tsv
```

name	field
Dijkstra	Computer Science
Shelah	Math
Aumann	Economic Sciences

Глава 105. Запись в формат CSV из строки или списка

Параметр

Подробности

`open ("/path/", "mode")`

Укажите путь к вашему CSV файлу

`open (path, "mode")`

Укажите режим открытия файла (чтение, запись и т. д.)

`csv.writer(file, delimiter)`

Передайте сюда открытый CSV файл

`csv.writer(file, delimiter='')`

Укажите символ или шаблон разделителя

Запись в файл .csv в большинстве случаев не отличается от записи в обычный файл и является достаточно простой задачей. Далее будет рассказано о наиболее простом и эффективном подходе к решению проблемы.

105.1. Пример базовой записи

```
import csv
```

```
#—— В этой функции мы будем писать в CSV ———
def csv_writer(data, path):
```

Открыть CSV-файл, путь к которому мы передали.

```
with open(path, "wb") as csv_file:
```

```
    writer = csv.writer(csv_file, delimiter=',')
```

```
    for line in data:
```

```
        writer.writerow(line)
```



```
#— Определим здесь наш список и вызовем функцию ———
if __name__ == "__main__":

    data = наш список, который мы хотим записать.
    Разделим его так, чтобы получился список списков.

    data = ["first_name,last_name,age".split(","),
            "John,Doe,22".split(","),
            "Jane,Doe,31".split(","),
            "Jack,Reacher,27".split(",")
            ]

    # Путь к CSV-файлу, в который мы хотим записать данные.
    path = "output.csv"
    csv_writer(data, path)
```

105.2. Добавление строки в качестве новой строки в CSV-файл

```
def append_to_csv(input_string):
    with open("fileName.csv", "a") as csv_file:
        csv_file.write(input_row + "\n")
```

Глава 106. Динамическое выполнение кода с помощью функций `exec` и `eval`

Аргумент	Подробности
<code>expression</code>	Код выражения в виде строки или объекта <code>code</code>
<code>object</code>	Код высказывания в виде строки или объекта <code>code</code>
<code>globals</code>	Словарь, используемый для глобальных переменных. Если не определено в <code>locals</code> , то этот словарь также используется для <code>locals</code> . Если опущено, то используются <code>globals()</code> вызывающей области видимости.
<code>locals</code>	Объект <i>отображения</i> (<i>mapping</i>), используемый для локальных переменных. Если это значение опущено, то используется объект, переданный для глобальных переменных. Если оба параметра опущены, то используются <code>globals()</code> и <code>locals()</code> вызывающей области видимости для <code>globals</code> и <code>locals</code> соответственно.

106.1. Выполнение кода, предоставленного недоверенным пользователем, с использованием `exec`, `eval` или `ast.literal_eval`

Невозможно использовать `eval` или `exec` для безопасного выполнения кода от **недоверенного пользователя**. Даже `ast.literal_eval` подвержен сбоям в парсере. Иногда можно защититься от выполнения вредоносного кода, но это не исключает возможности сбоев в парсере или токенизаторе.

Для оценки кода недоверенного пользователя необходимо обратиться к какому-либо стороннему модулю или, возможно, написать собственный парсер и собственную виртуальную машину на языке Python.

106.2. Оценка строки, содержащей литерал Python, с помощью ast.literal_eval

Если у вас есть строка, содержащая литералы Python, такие как strings, floats и т. д., вы можете использовать ast.literal_eval для оценки ее значения вместо eval. При этом добавляется возможность использования только определенного синтаксиса.

```
>>> import ast
>>> code = """(1, 2, {'foo': 'bar'})"""
>>> object = ast.literal_eval(code)
>>> object
(1, 2, {'foo': 'bar'})
>>> type(object)
<class 'tuple'>
```

Однако **это небезопасно** для выполнения кода, предоставленного недоверенным пользователем, и можно легко вывести интерпретатор из строя с тщательно обработанными входными данными.

```
>>> import ast
>>> ast.literal_eval('(' * 1000000)
[5] 21358 segmentation fault (core dumped) python3
```

Здесь на вход подается строка из (), повторяющаяся миллион раз, что приводит к аварийному завершению работы парсера CPython. Разработчики CPython не рассматривают ошибки в парсере как проблемы безопасности.

106.3. Оценка утверждений с помощью функции exec

```
>>> code = """for i in range(5):\n print('Hello world!')"""
>>> exec(code)
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

106.4. Оценка выражения с помощью функции eval

```
>>> expression = '5 + 3 * a'
>>> a = 5
>>> result = eval(expression)
>>> result
20
```

106.5. Предварительная компиляция выражения для его многократной оценки

Встроенная функция compile может быть использована для предварительной компиляции выражения в объект кода, который затем может быть передан в eval. Это позволит ускорить повторное выполнение оцениваемого кода. Третьим параметром для compile должна быть строка 'eval'.

```
>>> code = compile('a * b + c', '<string>', 'eval')
>>> code
<code object <module> at 0x7f0e51a58830, file "<string>", line 1>
>>> a, b, c = 1, 2, 3
>>> eval(code)
5
```

106.6. Оценка выражения с помощью eval с использованием пользовательских глобальных переменных

```
>>> variables = {'a': 6, 'b': 7}
>>> eval('a * b', globals=variables)
42
```

В этом коде невозможно случайно сослаться на имена, заданные “снаружи”:

```
>>> eval('variables')
{'a': 6, 'b': 7}
>>> eval('variables', globals=variables)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'variables' is not defined
```

Использование defaultdict позволяет, например, установить нулевое значение для нео-
пределенных переменных:

```
>>> from collections import defaultdict
>>> variables = defaultdict(int, {'a': 42})
>>> eval('a * c', globals=variables) # обратите внимание, что 'c' не определено явно
0
```

Глава 107. PyInstaller – распространение кода Python

107.1. Установка и настройка

Pyinstaller – это обычный пакет Python. Он может быть установлен с помощью pip:

```
pip install pyinstaller
```

Установка в Windows

Для Windows необходимым условием является наличие pywin32 или pyriwin32. Последний устанавливается автоматически при установке pyinstaller с помощью pip.

Установка в Mac OS X

PyInstaller работает с Python 2.7, поставляемым по умолчанию с текущей версией Mac OS X. Если предполагается использовать более поздние версии Python или основные пакеты, такие как PyQt, Numpy, Matplotlib и т.п., то рекомендуется устанавливать их с помощью MacPorts или Homebrew.

Установка из архива

Если pip недоступен, загрузите сжатый архив с PyPI. Чтобы протестировать версию разработки, загрузите сжатый архив из ветки *develop* на странице загрузки PyInstaller. Раскройте архив и найдите скрипт setup.py. Выполните `python setup.py install` с правами администратора для установки или обновления PyInstaller.

Проверка установки

После успешной установки команда pyinstaller должна быть в системном пути для всех платформ. Убедитесь в этом, набрав в командной строке `pyinstaller --version`. В результате будет выведена текущая версия pyinstaller.

107.2. Использование Pyinstaller

В простейшем случае достаточно перейти в каталог, в котором находится ваш файл, и ввести:

```
pyinstaller myfile.py
```

Pyinstaller анализирует file и создает:

- файл **myfile.spec** в том же каталоге, что и myfile.py
- Папка **build** в том же каталоге, что и myfile.py
- Папка **dist** в том же каталоге, что и myfile.py
- Файлы логов (журнал) в папке **build**.

Поставляемое в комплекте приложение (bundled app) находится в папке **dist**.

Опции

Существует несколько опций, которые могут быть использованы в pyinstaller. Приложение можно запустить, открыв 'dist\myfile\myfile.exe'.

107.3. Объединение в одну папку

Если PyInstaller используется без каких-либо опций для компоновки myscript.py, то по умолчанию на выходе получается одна папка (с именем myscript), содержащая исполняемый файл с именем myscript (myscript.exe в Windows) вместе со всеми необходимыми зависимыми файлами.

Приложение может быть распространено путем сжатия папки в zip-архив. Режим One Folder может быть явно задан с помощью опции -D или --onedir.

```
pyinstaller myscript.py -D
```

Преимущества

Одним из основных преимуществ компоновки в одну папку является удобство отладки проблем. Если какие-либо модули не импортируются, это можно проверить, просмотрев папку.

Еще одно преимущество ощущается при обновлениях. Если в коде произошло несколько изменений, но используемые зависимые файлы точно такие же, дистрибьюторы могут просто поставлять исполняемый файл (который меньше, чем вся папка).

Недостатки

Единственным недостатком этого метода является то, что пользователю приходится искать исполняемый файл среди большого количества других файлов.

Также пользователи могут удалять или изменять другие файлы, что может привести к некорректной работе приложения.

107.4. Объединение в один файл

```
pyinstaller myscript.py -F
```

Для создания одного файла можно использовать опции -F или --onefile. В этом случае программа объединяется в один файл myscript.exe. Этот вариант работает медленнее, чем компоновка в одну папку. Кроме того, единственный файл труднее отлаживать.

Глава 108. Визуализация данных с помощью Python

108.1. Seaborn

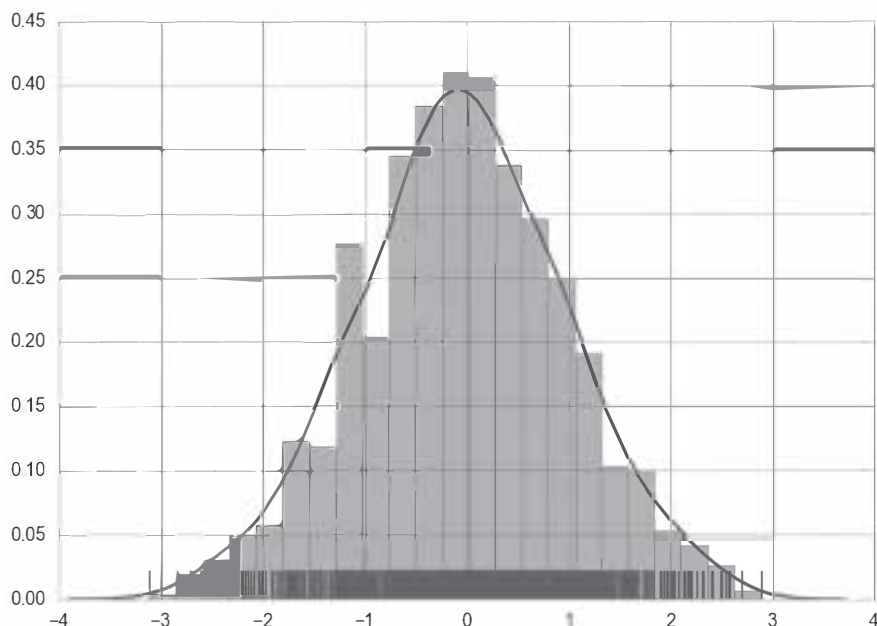
Seaborn – это “обертка” вокруг Matplotlib, позволяющая легко создавать распространенные статистические графики. Список поддерживаемых графиков включает графики одномерных и двумерных распределений, графики регрессии, а также ряд методов построения графиков категориальных переменных. Полный список графиков, которые предоставляет Seaborn, приведен в справке по API.

Создание графиков в Seaborn сводится к вызову соответствующей графической функции. Ниже приведен пример создания гистограммы, оценки плотности ядра и графика ковры для случайно сгенерированных данных.

```
import numpy as np # numpy используется при построении графиков
import seaborn as sns # распространенная форма импорта seaborn

# Генерировать нормально распределенные данные
data = np.random.randn(1000)

# Построить гистограмму с наложением графиков типов rugplot и kde
sns.distplot(data, kde=True, rug=True)
```

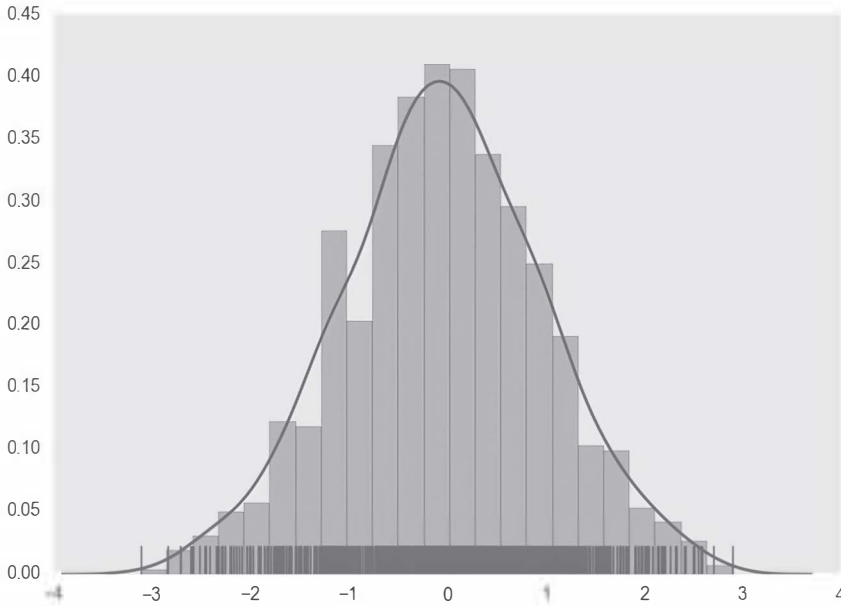


Стиль графика также может управляться с помощью декларативного синтаксиса.

```
# Использование ранее проведенного импорта и данных
```

```
# Использовать темный фон без сетки.
sns.set_style('dark')
```

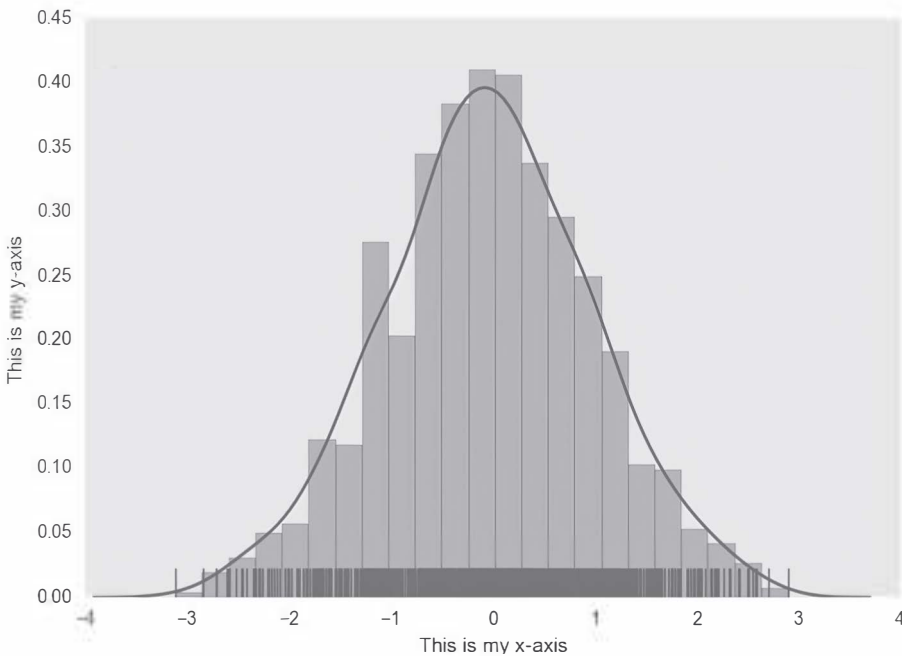
```
# Создать график заново
sns.distplot(data, kde=True, rug=True)
```



Кроме того, к графикам Seaborn можно применять обычные команды библиотеки `matplotlib`. Вот пример добавления заголовков осей к ранее созданной гистограмме.

```
# Используем те же данные и стиль
# Доступ к командам matplotlib
import matplotlib.pyplot as plt
```

```
# Ранее созданный график
sns.distplot(data, kde=True, rug=True)
# Установка меток осей
plt.xlabel('This is my x-axis')
plt.ylabel('This is my y-axis')
```



108.2. Matplotlib

Matplotlib – это библиотека математического построения графиков для Python, предоставляющая множество различных функций.

Matplotlib предоставляет два различных метода построения графиков, хотя по большей части они взаимозаменяемы:

- Интерфейс `pyplot` – прямой и простой в использовании, позволяет строить сложные графики в стиле, близком к стилю пакета `MATLAB`.
- `matplotlib` позволяет пользователю управлять различными аспектами (осями, линиями, тиками и т. д.) непосредственно с помощью объектно-ориентированной системы. Это сложнее, но позволяет полностью контролировать построение графика.

Ниже приведен пример использования интерфейса `pyplot` для построения графика некоторых сгенерированных данных:

```
import matplotlib.pyplot as plt
```

```
# Сгенерировать некоторые данные для построения графика.
```

```
x = [0, 1, 2, 3, 4, 5, 6]
```

```
y = [i**2 for i in x]
```

```
# Построить график данных x, y с некоторыми ключевыми аргументами, управляющими  
# стилем построения.
```

```
# Используйте две разные команды plot для построения как точек (scatter), так и линии (plot).  
plt.scatter(x, y, c='blue', marker='x', s=100) # Создание синих маркеров формы "x" и размера 100  
plt.plot(x, y, color='red', linewidth=2) # Создаем красную линию с шириной линии 2.
```

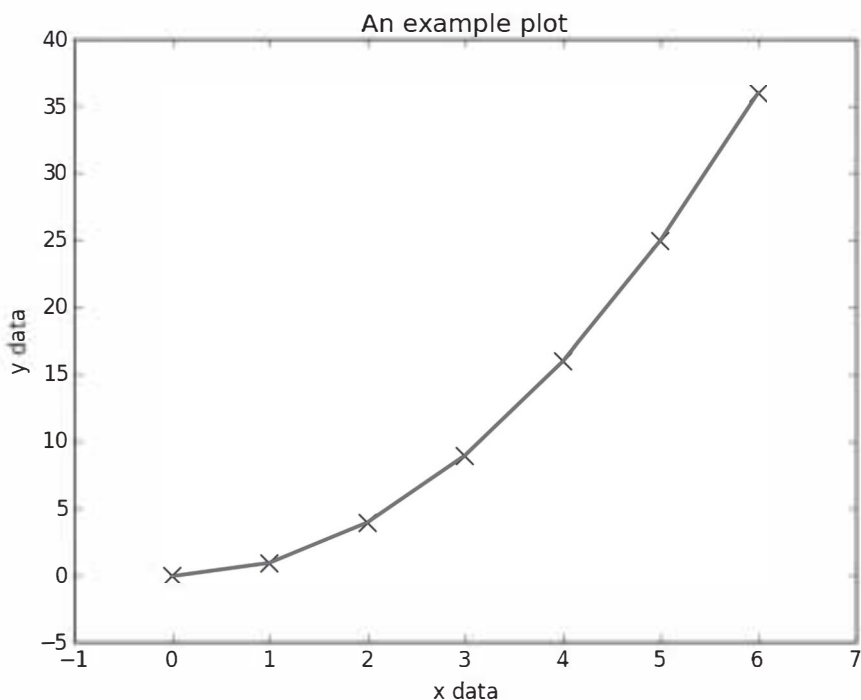
```
# Добавьте текст на оси и заголовок.
```

```
plt.xlabel('x data')
```

```
plt.ylabel('y data') plt.title('An example plot')
```

```
# Сформировать график и показать его пользователю.
```

```
plt.show()
```



Заметим, что `plt.show()` может вызывать проблемы в некоторых средах из-за запуска `matplotlib.pyplot` в интерактивном режиме, и если это так, то блокировку можно явно отменить, передав необязательный аргумент `plt.show(block=True)`, чтобы снять эту проблему.

108.3. Plotly

Plotly – это современная платформа для построения графиков и визуализации данных. Полезная для построения разнообразных графиков, особенно в области наук о данных, **Plotly** доступна в виде библиотеки для **Python**, R, JavaScript, Julia и MATLAB. Ее можно использовать как веб-приложение. Пользователи могут установить библиотеку `plotly` и использовать ее в оффлайн-режиме после аутентификации. Кроме того, графики можно строить и с использованием **Jupyter Notebooks**.

Для использования этой библиотеки требуется учетная запись с именем пользователя и паролем. Она предоставляет рабочую область (`workspace`) для сохранения графиков и данных в облаке.

Бесплатная версия библиотеки имеет несколько ограниченные возможности и рассчитана на создание 250 графиков в день. Платная версия имеет все возможности, неограниченное количество скачек графиков и более приватное хранилище данных. Более подробную информацию можно получить на главной странице сайта. Пример графика из документации:

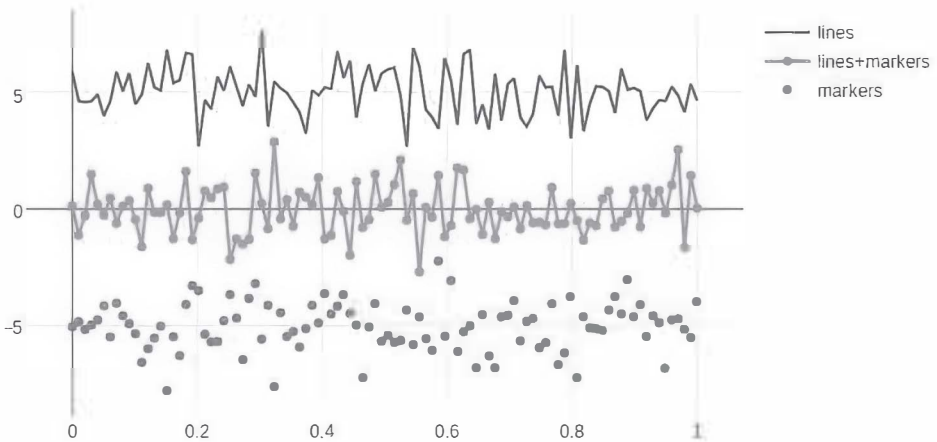
```
import plotly.graph_objs as go
import plotly as ply

# Создание случайных данных с помощью numpy
import numpy as np

N = 100
random_x = np.linspace(0, 1, N)
random_y0 = np.random.randn(N)+5
random_y1 = np.random.randn(N)
random_y2 = np.random.randn(N)-5

# Создаем графики
trace0 = go.Scatter(
    x = random_x,
    y = random_y0,
    mode = 'lines',
    name = 'lines'
)
trace1 = go.Scatter(
    x = random_x,
    y = random_y1,
    mode = 'lines+markers',
    name = 'lines+markers'
)
trace2 = go.Scatter(
    x = random_x,
    y = random_y2,
    mode = 'markers',
    name = 'markers'
)
data = [trace0, trace1, trace2]

ply.offline.plot(data, filename='line-mode')
```

108.4. MayaVI

MayaVI – это инструмент трехмерной визуализации научных данных. В нем используется набор инструментов визуализации или VTK (Visualization Tool Kit). Используя возможности VTK, MayaVI способен создавать разнообразные трехмерные графики и диаграммы. Он доступен как в виде отдельного программного приложения, так и в виде библиотеки. Подобно Matplotlib, эта библиотека предоставляет объектно-ориентированный интерфейс языка программирования для создания графиков без необходимости знания VTK.

MayaVI доступен только в серии версий Python 2.7х! Надеемся, что в скором времени он будет доступен и для Python серии 3х. (Хотя замечены некоторые успехи при использовании его зависимостей в Python 3.)

Здесь приведен пример графика, созданного с помощью MayaVI, из официальной документации:

```
# Автор: Gael Varoquaux <gael.varoquaux@normalesup.org>
# Copyright (c) 2007, Enthought, Inc.
# Лицензия: BSD Style.
```

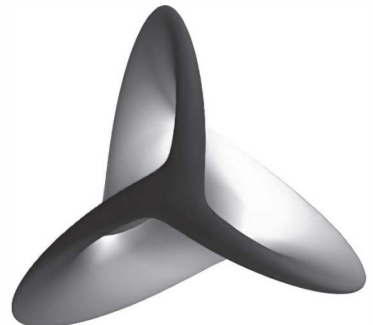
```
from numpy import sin, cos, mgrid, pi, sqrt
from mayavi import mlab
```

```
mlab.figure(fgcolor=(0, 0, 0), bgcolor=(1, 1, 1))
u, v = mgrid[- 0.035:pi:0.01, - 0.035:pi:0.01]
```

```
X = 2 / 3. * (cos(u) * cos(2 * v)
    + sqrt(2) * sin(u) * cos(v)) * cos(u) / (sqrt(2) -
    sin(2 * u) * sin(3 * v))
Y = 2 / 3. * (cos(u) * sin(2 * v) -
    sqrt(2) * sin(u) * sin(v)) * cos(u) / (sqrt(2)
    - sin(2 * u) * sin(3 * v))
Z = -sqrt(2) * cos(u) * cos(u) / (sqrt(2) - sin(2 * u) * sin(3 * v))
S = sin(u)
```

```
mlab.mesh(X, Y, Z, scalars=S, colormap='YlGnBu', )
```

```
# Хороший вид спереди
mlab.view(0, - 5.0, 4)
mlab.show()
```



Глава 109. Интерпретатор (консоль командной строки)

109.1. Получение общей помощи

Если функция `help` вызывается в консоли без аргументов, то Python представляет интерактивную справочную консоль, в которой можно узнать о модулях Python, символах, ключевых словах и многом другом.

```
>>> help()
```

Добро пожаловать в справочную утилиту Python 3.4!

Если вы впервые используете Python, то вам обязательно следует ознакомиться с учебным пособием, размещенным в Интернете по адресу <http://docs.python.org/3.4/tutorial/>.

Введите имя любого модуля, ключевого слова или темы, чтобы получить справку по написанию программ на Python и использованию модулей Python. Чтобы выйти из этой справочной утилиты и вернуться в интерпретатор, достаточно набрать `"quit"`.

Чтобы получить список доступных модулей, ключевых слов, символов или тем, введите `"modules"`, `"keywords"`, `"symbols"` или `"topics"`. Каждый модуль также имеет однострочное краткое описание своей работы; чтобы получить список модулей, название или краткое описание которых содержит заданную строку, например `"spam"`, введите `"modules spam"`.

109.2. Ссылка на последнее выражение

Чтобы получить в консоли значение последнего результата из последнего выражения, используйте знак подчеркивания `_`.

```
>>> 2 + 2
4
>>> _
4
>>> _ + 6
10
```

Это значение магического подчеркивания обновляется только при использовании выражения Python, результатом которого является значение. Выполнение функций или циклов `for` не приводит к изменению значения. Если выражение вызывает исключение, то никаких изменений для `_` не произойдет.

```
>>> "Hello, {0}".format("World")
'Hello, World'
>>> _
'Hello, World'
>>> def wontchangethings():
...     pass
>>> _
'Hello, World'
>>> 27 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> _
'Hello, World'
```

Помните, что эта магическая переменная доступна только в интерактивном интерпретаторе Python. Запущенные скрипты этого не сделают.

109.3. Открытие консоли Python

Консоль для основной версии Python обычно можно открыть, набрав `py` в консоли Windows или набрав `python` на других платформах.

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Если у вас несколько версий, то по умолчанию их исполняемые файлы будут сопоставлены с `python2` или `python3` соответственно. Это, конечно, зависит от наличия исполняемых файлов Python в `PATH`.

109.4. Переменная PYTHONSTARTUP

Для консоли Python можно установить переменную окружения `PYTHONSTARTUP`. При каждом входе в консоль Python будет выполняться эта переменная, что позволяет добавлять в консоль дополнительные функции, например, автоматический импорт часто используемых модулей.

Если переменная `PYTHONSTARTUP` была установлена в местоположение файла, содержащего это:

```
print("Welcome!")
```

тогда открытие консоли Python приведет к появлению такого дополнительного вывода:

```
$ py
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Welcome!
>>>
```

109.5. Аргументы командной строки

В Python существует множество переключателей командной строки, которые можно передавать в `py`. Их можно найти, выполнив команду `py --help`, которая на Python 3.4 выдает следующее сообщение:

Python Launcher

```
usage: py [launcher-arguments] [python-arguments] script [script-arguments]
```

Аргументы программы запуска:

- 2 : Запуск последней версии Python 2.x
- 3 : Запуск последней версии Python 3.x
- X.Y : Запуск указанной версии Python
- X.Y-32: Запуск указанной 32-разрядной версии Python

Ниже приведен текст справки из языка Python:

использование: `G:\Python34\python.exe [option] ... [-c cmd | -m mod | file | -] [arg] ...` Опции и аргументы (и соответствующие переменные окружения):

- b : выдать предупреждения о `str(bytes_instance)`, `str(bytearray_instance)` и сравнении байтов/байтовых массивов с `str`. (-bb: выдать ошибки).
- B : не записывать файлы `.py[co]` при импорте; также `PYTHONDONTWRITEBYTECODE=x`
- c cmd : программа передается в виде строки (завершает список опций)
- d : вывод отладки парсера (синтаксического анализатора); также `PYTHONDEBUG=x`
- E : игнорировать переменные окружения `PYTHON*` (например, `PYTHONPATH`)
- h : вывести это справочное сообщение и выйти (также `-help`)
- i : проверка в интерактивном режиме после выполнения скрипта; вызывает приглашение, даже если `stdin` не является терминалом; также `PYTHONINSPECT=x`
- I : изолировать Python от окружения пользователя (подразумевает -E и -s)
- m mod : запуск библиотечного модуля как скрипта (завершает список опций)
- O : немного оптимизировать генерируемый байткод; также `PYTHONOPTIMIZE=x`

-OO : удаление dos-строк в дополнение к оптимизации -O
 -q : не печатать сообщения о версии и авторских правах при интерактивном запуске
 -s : не добавлять каталог сайта пользователя в sys.path; также PYTHONNOUSERSITE
 -S : не подразумевать 'import site' при инициализации
 -u : небуферизованные двоичные stdout и stderr, stdin всегда буферизован;
 также PYTHONUNBUFFERED=x
 Подробности о внутренней буферизации, связанной с '-u', см. на основной странице.
 -v : verbose (трассировка операторов импорта); также можно многократно указать
 PYTHONVERBOSE=x для увеличения объема информации
 -V : вывести номер версии Python и выйти (также --version)
 -W arg : управление предупреждениями; arg - это action:message:category:module:lineno
 также PYTHONWARNINGS=arg
 -x : пропускает первую строку исходного текста, позволяя использовать не-Unix-формы #!cmd
 -X opt : установить специфическую опцию реализации файла : программа, прочитанная из
 файла сценария
 - : программа, прочитанная из stdin (по умолчанию; интерактивный режим, если tty) arg ...:
 аргументы, переданные программе в sys.argv[1:]

Другие переменные окружения:

PYTHONSTARTUP: файл, выполняемый при интерактивном запуске (по умолчанию отсутствует)
 PYTHONPATH : ':'-разделенный список каталогов с префиксом к пути поиска модулей по
 умолчанию. В результате получается sys.path.

PYTHONHOME : альтернативный каталог (или ;). По умолчанию для поиска модулей
 используется путь \\lib.

PYTHONCASEOK : игнорирование регистра в операторах 'import' (Windows).

PYTHONIOENCODING: Кодировка[errors], используемая для stdin/stdout/stderr.

PYTHONFAULTHANDLER: дамп трассировки Python при фатальных ошибках.

PYTHONHASHSEED: если эта переменная имеет значение 'random', то для засева хешей
 объектов str, bytes и datetime используется случайная величина. Она также может быть
 задана целым числом в диапазоне [0,4294967295] для получения хеш-значений
 с предсказуемым seed.

109.6. Получение справки об объекте

В консоли Python добавлена новая функция help, с помощью которой можно получить
 информацию о функции или объекте. Для функции help выводит ее сигнатуру (аргументы)
 и docstring, если она есть.

```
>>> help(print)
Помощь по встроенной функции print в модуле builtins:
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

Выводит значения в поток или по умолчанию в sys.stdout.

Необязательные аргументы в виде ключевых слов:

file: файлоподобный объект (поток); по умолчанию - текущий sys.stdout.

sep: строка, вставляемая между значениями, по умолчанию пробел.

end: строка, добавляемая после последнего значения, по умолчанию - новая строка.

flush: следует ли принудительно очищать поток.

Для объекта в справке выводятся erodocstring и различные функции-члены, которыми об-
 ладает объект.

```
>>> x = 2
>>> help(x)
Справка по объекту int:
```

```
class int(object)
| int(x=0) -> integer
| int(x, base=10) -> integer|
```

| Преобразовать число или строку в целое число или вернуть 0, если аргументы не заданы.

Если `x` - число, то возвращается `x.int()`. Для чисел с плавающей точкой происходит усечение до нуля.

Если `x` не является числом или задано основание, то `x` должен быть строкой, байтом или экземпляром байтового массива, представляющим целочисленный литерал в заданном основании. Перед литералом может стоять знак `'+'` или `'-'`, и он может быть окружен пробелами. По умолчанию основание равно 10.

Допустимыми основаниями являются 0 и 2-36.

Base 0 означает интерпретацию основания из строки как целочисленного литерала.

```
>>> int('0b100', base=0)
```

```
4
```

Методы определены здесь:

```
__abs__(self, /)
```

```
abs(self)
```

```
__add__(self, value, /)
```

```
Возвращает self+value...
```

Глава 110. *args и **kwargs

110.1. Использование **kwargs при написании функций

Определить функцию, принимающую произвольное количество ключевых (именованных) аргументов, можно с помощью двойной звездочки `**` перед именем параметра:

```
def print_kwargs(**kwargs):
    print(kwargs)
```

При вызове метода Python построит словарь всех аргументов ключевого слова и сделает его доступным в теле функции:

```
print_kwargs(a="two", b=3)
# выведет: "{a: 'two', b=3}"
```

Обратите внимание, что параметр `**kwargs` в определении функции всегда должен быть последним, и он будет соответствовать только тем аргументам, которые были переданы после предыдущих.

```
def example(a, **kw):
    print kw
```

```
example(a=2, b=3, c=4) # => {'b': 3, 'c': 4}
```

Внутри тела функции работа с `kwargs` осуществляется так же, как и со словарем; для доступа к отдельным элементам `kwargs` достаточно пройти по ним циклом, как это делается с обычным словарем:

```
def print_kwargs(**kwargs):
    for key in kwargs:
        print("key = {0}, value = {1}".format(key, kwargs[key]))
```

Теперь вызов `print_kwargs(a="two", b=1)` приводит к следующему результату:

```
print_kwargs(a="two", b=1)
key = a, value = "two"
key = b, value = 1
```

110.2. Использование *args при написании функций

При написании функции можно использовать звездочку *, чтобы собрать все позиционные (т.е. неименованные) аргументы в кортеж:

```
def print_args(farg, *args):
    print("formal arg: %s" % farg)
    for arg in args:
        print("another positional arg: %s" % arg)
```

Метод вызова:

```
print_args(1, "two", 3)
```

В этом вызове farg будет присвоен, как всегда, а два других будут переданы в кортеж args в порядке их поступления.

110.3. Заполнение значений kwarg с помощью словаря

```
def foobar(foo=None, bar=None):
    return "{} {}".format(foo, bar)
```

```
values = {"foo": "foo", "bar": "bar"}
```

```
foobar(**values) # "foobar"
```

110.4. Аргументы только для ключевых слов (keyword-only arguments) и аргументы, требующие ключевых слов (keyword-required arguments)

Python 3 позволяет определить аргументы функции, которые могут быть назначены только по ключевому слову, даже без значений по умолчанию. Это делается с помощью символа * для потребления дополнительных позиционных параметров без задания параметров ключевого слова. Все аргументы после * являются аргументами только для ключевого слова (т.е. непозиционными, (keyword-only, non-positional)). Обратите внимание, что если аргументам, относящимся только к ключевым словам, не задано значение по умолчанию, то они все равно требуются при вызове функции.

```
def print_args(arg1, *args, keyword_required, keyword_only=True):
    print("первый позиционный аргумент: {}".format(arg1))
    for arg in args:
        print("другой позиционный аргумент: {}".format(arg))
    print("keyword_required value: {}".format(keyword_required))
    print("keyword_only value: {}".format(keyword_only))
```

```
print(1, 2, 3, 4) # TypeError: print_args() missing 1 required keyword-only argument: 'keyword_required'
print(1, 2, 3, keyword_required=4)
# первый позиционный аргумент: 1
# другой позиционный аргумент: 2
# другой позиционный аргумент: 3
# keyword_required value: 4
# keyword_only value: True
```

110.5. Использование **kwargs при вызове функций

Для присвоения значений параметрам функции можно использовать словарь, в котором в качестве ключей используются имена параметров, а к каждому ключу привязано значение этих аргументов:

```
def test_func(arg1, arg2, arg3): # Обычная функция с тремя аргументами
    print("arg1: %s" % arg1)
    print("arg2: %s" % arg2)
    print("arg3: %s" % arg3)
```

```
# Обратите внимание, что словари неупорядочены, поэтому мы можем
# поменять местами arg2 и arg3. Имеют значение только имена.
kwargs = {"arg3": 3, "arg2": "2"}

# Привязать первый аргумент (т.е. arg1) к 1, а для привязки остальных
# использовать словарь kwargs
test_var_args_call(1, **kwargs)
```

110.6. **kwargs и значения по умолчанию

Чтобы использовать значения по умолчанию с **kwargs:

```
def fun(**kwargs):
    print kwargs.get('value', 0)
    fun()
# print 0
fun(value=1)
# print 1
```

110.7. Использование *args при вызове функций

Эффект использования оператора * на аргументе при вызове функции заключается в распаковке списочного или кортежного аргумента:

```
def print_args(arg1, arg2):
    print(str(arg1) + str(arg2))

a = [1,2]
b = tuple([3,4])

print_args(*a) # 12
print_args(*b) # 34
```

Обратите внимание, что длина аргумента со звездочкой должна быть равна количеству аргументов функции. Распространенной идиомой в Python является использование оператора распаковки * с функцией zip, чтобы обратить ее действие:

```
a = [1,3,5,7,9]
b = [2,4,6,8,10]

zipped = zip(a,b)
# [(1,2), (3,4), (5,6), (7,8), (9,10)]

zip(*zipped)
# (1,3,5,7,9), (2,4,6,8,10)
```

Глава 111. Сборка мусора

111.1. Повторное использование примитивов

Интересный момент, который может помочь оптимизировать ваши приложения, заключается в том, что примитивы на самом деле также пересчитываются. Для всех целых чисел от -5 до 256 Python всегда повторно использует один и тот же объект:

```
>>> import sys
>>> sys.getrefcount(1)
797
>>> a = 1
>>> b = 1
```



```
>>> sys.getrefcount(1)
799
```

Обратите внимание, что счетчик ссылок (refcount) увеличивается, что означает, что а и b ссылаются на один и тот же базовый объект, когда они ссылаются на примитив 1. Однако для больших чисел Python фактически не использует базовый объект повторно:

```
>>> a = 999999999
>>> sys.getrefcount(999999999)
3
>>> b = 999999999
>>> sys.getrefcount(999999999)
3
```

Поскольку счетчик ссылок для 999999999 не меняется при присвоении его а и b, можно сделать вывод, что они ссылаются на два разных базовых объекта, хотя им присвоен один и тот же примитив.

111.2. Последствия команды del

Удаление имени переменной из области видимости с помощью del v, удаление объекта из коллекции с помощью del v[item] или del[i:j], удаление атрибута с помощью del v.name или любой другой способ удаления ссылок на объект сами по себе *не приводят* к вызову деструктора или освобождению памяти. Объекты уничтожаются только тогда, когда количество их ссылок достигает нуля.

```
>>> import gc
>>> gc.disable() # отключить сборщик мусора
>>> class Track:
    def init(self):
        print("Initialized")
    def del(self):
        print("Destructed")
>>> def bar():
    return Track()
>>> t = bar()
Initialized
>>> another_t = t # присвоить другую ссылку
>>> print("...")

>>> del t          # еще не уничтожен - another_t все еще ссылается на него
>>> del another_t # конечная ссылка исчезла, объект уничтожен
Destructed
```

111.3. Подсчет ссылок

В подавляющем большинстве случаев управление памятью в Python осуществляется с помощью подсчета ссылок.

Каждый раз, когда на объект ссылаются (например, присваивают переменной), количество ссылок на него автоматически увеличивается. При исключении ссылки (например, при выходе переменной из области видимости) количество ссылок автоматически уменьшается.

Когда счетчик ссылок достигает нуля, объект **немедленно уничтожается** и память сразу же освобождается. Таким образом, для большинства случаев сборщик мусора даже не нужен.

```
>>> import gc; gc.disable() # отключить сборщик мусора
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("Destructed")
```



```
>>> def foo():
    Track()
    # уничтожается немедленно, так как больше не имеет ссылок
    print("--")
    t = Track()
    # на переменную есть ссылка, поэтому она еще не уничтожена
    print("--")
    # переменная уничтожается при выходе из функции
>>> foo()
Initialized
Destructed
--
Initialized
--
Destructed
```

Для дальнейшей демонстрации концепции ссылок:

```
>>> def bar():
    return Track()
>>> t = bar()
Initialized
>>> another_t = t          # присвоить другую ссылку
>>> print("...")

>>> t = None               # еще не уничтожен - на него по-прежнему ссылается another_t
>>> another_t = None       # конечная ссылка исчезла, объект уничтожен
Destructed
```

111.4. Сборщик мусора для ссылочных циклов

Единственный случай, когда сборщик мусора необходим, – это *цикл ссылок*. Простейшим примером цикла ссылок является цикл, в котором А ссылается на В, В ссылается на А, и больше ничто не ссылается ни на А, ни на В. Ни А, ни В недоступны из любой точки программы, поэтому они могут быть безопасно уничтожены, но их счетчики ссылок равны 1, и поэтому они не могут быть освобождены только алгоритмом подсчета ссылок.

```
>>> import gc; gc.disable() # отключить сборщик мусора
>>> class Track:
    def __init__(self):
        print("Initialized")
    def __del__(self):
        print("Destructed")
>>> A = Track()
Initialized
>>> B = Track()
Initialized
>>> A.other = B
>>> B.other = A
>>> del A; del B # объекты не уничтожаются из-за ссылочного цикла
>>> gc.collect() # запустить сбор
Destructed
Destructed
4
```

Цикл ссылок может быть произвольной длины. Если А указывает на В указывает на С указывает на ... указывает на Z, который указывает на А, то ни А, ни Z не будут собраны, вплоть до фазы сборки мусора:

```
>>> objs = [Track() for _ in range(10)]
Initialized
Initialized
```

```

Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
Initialized
>>> for i in range(len(objs)-1):
...   objs[i].other = objs[i + 1]

>>> objs[-1].other = objs[0] # завершить цикл
>>> del objs                # теперь никто не может ссылаться на objs - они все еще не уничтожены
>>> gc.collect()
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
Destructed
20

```

111.5. Принудительное удаление объектов

Как в Python 2, так и в Python 3 можно принудительно освободить (деаллоцировать) объекты, даже если их счетчик ссылок `refcount` не равен 0. В обеих версиях для этого используется модуль `ctypes`.

ПРЕДУПРЕЖДЕНИЕ: в результате этого ваша среда Python *станет нестабильной* и склонной к аварийному завершению работы! Использование этого метода также может привести к проблемам безопасности (что весьма маловероятно). Деаллокация объектов производится только в том случае, если вы уверены, что никогда больше не обратитесь к ним.

Версия Python 3.x ≥ 3.0:

```

import ctypes
deallocated = 12345
ctypes.pythonapi._Py_Dealloc(ctypes.py_object(deallocated))

```

Версия Python 2.x ≥ 2.3:

```

import ctypes, sys
deallocated = 12345
(ctypes.c_char * sys.getsizeof(deallocated)).from_address(id(deallocated))[4] = '\x00' * 4

```

После выполнения любая ссылка на деаллоцированный объект приведет к тому, что Python либо выдаст неопределенное поведение, либо аварийно завершится – без трассировки (`traceback`). Вероятно, все-таки была причина, по которой сборщик мусора не удалил этот объект... При деаллокации `None` выдается специальное сообщение `Fatal Python error: deallocating None` и происходит аварийное завершение.

111.6. Просмотр счетчика ссылок (`refcount`) объекта

```

>>> import sys
>>> a = object()
>>> sys.getrefcount(a)
2
>>> b = a

```

```
>>> sys.getrefcount(a)
3
>>> del b
>>> sys.getrefcount(a)
2
```

111.7. Не ждите, пока сборщик мусора наведет порядок

Тот факт, что сборка мусора будет очищаться, не означает, что следует ждать, пока цикл сборки мусора очистится. В частности, не следует ждать, пока сборка мусора закроет дескрипторы файлов, соединения с базами данных и открытые сетевые соединения.

Например, в следующем коде предполагается, что файл будет закрыт на следующем цикле сборки мусора, если `f` была последней ссылкой на файл.

```
>>> f = open("test.txt")
>>> del f
```

Более явным способом очистки является вызов `f.close()`. Можно сделать это еще более “элегантно”, с помощью оператора `with` (с помощью менеджера контекста):

```
>>> with open("test.txt") as f:
...     pass
...     # сделать что-нибудь с f
>>> # теперь объект f все еще существует, но он закрыт
```

Оператор `with` позволяет узнать, как долго файл остается открытым. Кроме того, оператор `with` всегда закрывает файл, даже если в блоке `while` возникло исключение.

111.8. Управление сборкой мусора

Существует два подхода влияния на выполнение очистки памяти. Один из них – повлиять на частоту выполнения автоматического процесса, а другой – ручной запуск очистки.

Сборщиком мусора можно управлять, настраивая пороговые значения сбора, которые влияют на частоту запуска сборщика. В Python используется система управления памятью на основе поколений (`generations`). Новые объекты сохраняются в самом новом поколении – **generation0**, и с каждой новой сборкой объекты переходят в более старшие поколения. После достижения последнего поколения – **generation2** – они больше не продвигаются.

Изменить пороговые значения можно с помощью следующего фрагмента кода:

```
import gc
gc.set_threshold(1000, 100, 10) # Значения приведены только для демонстрации
```

Первый аргумент представляет собой порог для сбора **generation0**. Каждый раз, когда количество **аллокаций** превышает количество **деаллокаций** на 1000, будет вызван сборщик мусора.

Для оптимизации процесса старые поколения не очищаются при каждом запуске. Второй и третий аргументы являются **необязательными** и определяют частоту очистки старших поколений. Если **generation0** было обработано 100 раз без очистки **generation1**, то будет обработано **generation1**. Аналогично объекты в **generation2** будут обрабатываться только в том случае, если объекты в **generation1** были очищены 10 раз, не затрагивая **generation2**.

Один из случаев, когда ручная установка пороговых значений полезна, – это когда программа аллоцирует много мелких объектов, не деаллоцируя их, что приводит к слишком частому запуску сборщика мусора. Несмотря на то, что сборщик довольно быстр, при работе с огромным количеством объектов он создает проблемы с производительностью. В любом случае не существует универсальной стратегии выбора пороговых значений, и она зависит от конкретного случая.

Ручное срабатывание коллекции может быть выполнено, как в следующем фрагменте кода:

```
import gc
gc.collect()
```

Сборка мусора автоматически запускается на основе количества аллокаций и деаллокаций, а не на основе потребляемой или доступной памяти. Следовательно, при работе с большими объектами память может быть исчерпана до того, как сработает автоматическая очистка. В этом случае целесообразно вызывать сборщик мусора вручную.

Хотя это и возможно, но не приветствуется. Оптимальным вариантом является избегание утечек памяти. В любом случае в больших проектах обнаружение утечки памяти может оказаться сложной задачей, и ручной запуск сборки мусора может быть использован в качестве быстрого решения до дальнейшей отладки.

Для длительно работающих программ сборка мусора может запускаться по времени или по событию. В качестве примера первого варианта можно привести веб-сервер, который запускает сборку мусора после определенного количества запросов. В качестве второго примера можно привести веб-сервер, который запускает сборку мусора при получении определенного типа запроса.

Глава 112. Сериализация данных при помощи модуля Pickle

Параметр	Подробности
object	Объект, подлежащий хранению
file	Открытый файл, который будет содержать объект
protocol	Протокол, используемый для Pickle (необязательный параметр)
buffer	Байтовый объект, содержащий сериализованный объект

112.1. Использование модуля Pickle для сериализации и десериализации объекта

Модуль pickle реализует алгоритм превращения произвольного объекта Python в последовательность байтов. Этот процесс также называется **сериализацией** объекта. Поток байтов, представляющий объект, может быть передан или сохранен, а затем реконструирован для создания нового объекта с теми же характеристиками. Для простейшего кода мы используем функции `dump()` и `load()`.

Для сериализации объекта

```
import pickle

# Произвольная коллекция объектов, поддерживаемых pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Применим Pickle к словарю 'data' с использованием самого высокого из доступных протоколов.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

Для десериализации объекта

```
import pickle
```

```
with open('data.pickle', 'rb') as f:
    # Версия используемого протокола определяется автоматически, поэтому нам не
    # нужно ее указывать.
    data = pickle.load(f)
```

Pickle и байтовые объекты

Также возможна сериализация в байтовые объекты и десериализация из них с помощью функций `dumps` и `loads`, которые эквивалентны функциям `dump` и `load`.

```
serialized_data = pickle.dumps(data, pickle.HIGHEST_PROTOCOL)
# type(serialized_data) - это байты

deserialized_data = pickle.loads(serialized_data)
# deserialized_data == данные
```

112.2. Данные для Pickle

Не все данные подходят для модуля `Pickle`, а некоторые данные не следует обрабатывать этим модулем по другим причинам.

То, что будет направляться в модуль, может быть определено в методе `__getstate__`. Этот метод должен возвращать то, что можно направлять в `Pickle`. Метод `__setstate__` получит то, что создал `__getstate__` и должен инициализировать объект.

```
class A(object):
    def __init__(self, important_data):
        self.important_data = important_data

    # Добавить данные, которые нельзя направлять в Pickle:
    self.func = lambda: 7

    # Добавить данные, которые никогда не направлять в Pickle,
    # так как они быстро заканчиваются::
    self.is_up_to_date = False

    def __getstate__(self):
        return [self.important_data] # только это необходимо

    def __setstate__(self, state):
        self.important_data = state[0]

    self.func = lambda: 7 # просто некая жестко закодированная функция, которую
    # не надо направлять в Pickle

    self.is_up_to_date = False # даже если это было до направления в Pickle
```

Теперь можно сделать следующее:

```
>>> a1 = A('very important')
>>>
>>> s = pickle.dumps(a1) # calls a1.__getstate__()
>>>
>>> a2 = pickle.loads(s) # calls a1.__setstate__(['very important'])
>>> a2
<__main__.A object at 0x0000000002742470>
>>> a2.important_data
'very important'
>>> a2.func()
7
```

Реализация здесь направляет в `Pickle` список с одним значением: `[self.important_data]`. Это был просто пример, `__getstate__` мог бы возвращать все, что можно направлять в `Pickle`, если

бы `__setstate__` знал, как сделать обратное. Хорошей альтернативой является словарь всех значений: `{'important_data': self.important_data}`.

Конструктор не вызывается! Обратите внимание, что в предыдущем примере экземпляра `a2` был создан в `pickle.loads` без вызова `A.__init__`, поэтому `A.__setstate__` должен был инициализировать все то, что инициализировал бы `__init__`, если бы он был вызван.

Глава 113. Двоичные данные

113.1. Форматирование списка значений в байтовый объект

```
from struct import pack
```

```
print(pack('I3c', 123, b'a', b'b', b'c')) # b'\x00\x00\x00abc'
```

113.2. Распаковка байтового объекта в соответствии со строкой формата

```
from struct import unpack
```

```
print(unpack('I3c', b'\x00\x00\x00abc')) # (123, b'a', b'b', b'c')
```

113.3. Упаковка структуры

Модуль `struct` предоставляет возможность упаковывать объекты Python в виде непрерывного “куска байтов” (chunk of bytes) или разъединять кусок байтов на структуры Python.

Функция `pack` принимает строку формата и один или несколько аргументов и возвращает двоичную строку. Это очень похоже на форматирование строки, за исключением того, что на выходе получается не строка, а кусок байтов.

```
import struct
import sys
print "Native byteorder: ", sys.byteorder
# Если порядок байтов не указан, то используется
# собственный порядок байтов (Native byteorder)
buffer = struct.pack("ihb", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
print "Byte chunk unpacked: ", struct.unpack("ihb", buffer)
# Последний элемент в виде unsigned short вместо unsigned char (2 байта)
buffer = struct.pack("ihh", 3, 4, 5)
print "Byte chunk: ", repr(buffer)
```

Выход:

```
Native byteorder: little Byte chunk: '\x03\x00\x00\x04\x00\x05' Byte chunk unpacked: (3, 4, 5) Byte
chunk: '\x03\x00\x00\x04\x00\x05\x00'
```

Можно использовать сетевой порядок байтов с данными, полученными из сети, или упаковать данные для отправки их в сеть.

```
import struct
# Если порядок байтов не указан, то используется
# собственный порядок байтов (Native byteorder)
buffer = struct.pack("hhh", 3, 4, 5)
print "Byte chunk native byte order: ", repr(buffer)
```

```
buffer = struct.pack("!hhh", 3, 4, 5)
print "Byte chunk network byte order: ", repr(buffer)
```

Выход:

```
Byte chunk native byte order: '\x03\x00\x04\x00\x05\x00'
Byte chunk network byte order: '\x00\x03\x00\x04\x00\x05'
```

Вы можете оптимизировать работу, избежав накладных расходов на выделение нового буфера, предоставив ранее созданный буфер.

```
import struct
from ctypes import create_string_buffer
bufferVar = create_string_buffer(8)
bufferVar2 = create_string_buffer(8)
# Мы используем уже созданный буфер
# предоставляем формат, буфер, смещение и данные
struct.pack_into("hhh", bufferVar, 0, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar.raw)
struct.pack_into("hhh", bufferVar2, 2, 3, 4, 5)
print "Byte chunk: ", repr(bufferVar2.raw)
```

Выход:

```
Byte chunk: '\x03\x00\x04\x00\x05\x00\x00\x00'
Byte chunk: '\x00\x00\x03\x00\x04\x00\x05\x00'
```

Глава 114. Идиомы

114.1. Инициализация ключей словаря

Предпочтителен метод `dict.get`, если вы не уверены в наличии ключа. Он позволяет вернуть значение по умолчанию, если ключ не найден. Традиционный метод с использованием `dict[key]` вызовет исключение `KeyError`.

Вместо использования:

```
def add_student():
    try:
        students['count'] += 1
    except KeyError:
        students['count'] = 1
```

осуществите:

```
def add_student():
    students['count'] = students.get('count', 0) + 1
```

114.2. Переключение переменных

Для переключения значений двух переменных можно использовать распаковку кортежей.

```
x = True
y = False
x, y = y, x
x
# False
y
# True
```

114.3. Использование проверки значения истинности

Python неявно преобразует любой объект в булево значение для тестирования, поэтому используйте его везде, где это возможно.

```
# Хорошие примеры, использующие неявное тестирование истинности
```

```
if attr:
```

```
    # сделать что-нибудь
```

```
if not attr:
```

```
    # сделать что-нибудь
```

```
# Неудачные примеры, использующие конкретные типы
```

```
если attr == 1:
```

```
    # сделать что-нибудь
```

```
if attr == True:
```

```
    # сделать что-нибудь
```

```
if attr != ":
```

```
    # сделать что-нибудь
```

```
# Если вы хотите проверить наличие None, используйте 'is' или 'is not'.
```

```
if attr is None:
```

```
    # сделать что-нибудь
```

Как правило, в этом случае код получается более читабельным, а при работе с неожиданными типами – более безопасным.

114.4. Тест на функцию `__main__`, чтобы избежать неожиданного выполнения кода

Хорошей практикой является проверка переменной `__name__`, вызывающей программы перед выполнением своего кода.

```
import sys
```

```
def main():
```

```
    # Ваш код начинается здесь
```

```
    # Не забудьте указать код возврата
```

```
    return 0
```

```
if __name__ == "__main__":
```

```
    sys.exit(main())
```

Использование этого паттерна гарантирует, что ваш код будет выполняться только тогда, когда вы этого ожидаете; например, при явном запуске вашего файла:

```
python my_program.py
```

Однако польза от этого будет в том случае, если вы решите импортировать свой файл в другую программу (например, если вы пишете его как часть библиотеки). Тогда вы можете импортировать файл, а “ловушка” `__main__` будет гарантировать, что никакой код не будет выполнен неожиданно:

```
# Новый файл программы
```

```
import my_program # main() не выполняется
```

```
# Но вы можете запустить main() явно, если действительно хотите, чтобы она выполнялась:
```

```
my_program.main()
```


Глава 115. Сериализация данных

Параметр Подробности

protocol При использовании pickle или cPickle – это метод, которым объекты сериализуются/несериализуются (Serialized/Unserialized). Вероятно, вы захотите использовать pickle.HIGHEST_PROTOCOL, что означает самый новый метод.

115.1. Сериализация с использованием формата JSON

JSON – это формат, который может использоваться практически с любым языком программирования; он широко используется для сериализации данных. Поддерживаемые в нем типы данных: int, float, boolean, string, list и dict. Далее будет приведен пример, демонстрирующий базовое использование JSON:

```
import json

families = ([ 'John' ], [ 'Mark', 'David', { 'name': 'Avraham' } ])

# Выгрузка в строку
json_families = json.dumps(families)
# [ "[\"John\"]", "[\"Mark\", \"David\", {\"name\": \"Avraham\"}]" ]

# Выгрузка в файл
with open('families.json', 'w') as json_file:
    json.dump(families, json_file)

# Загрузка из строки
json_families = json.loads(json_families)

# Загрузка из файла
with open('families.json', 'r') as json_file:
    json_families = json.load(json_file)
```

Подробную информацию о JSON см. в соответствующей посвященной ему главе.

115.2. Сериализация с использованием модуля Pickle

Приведем пример, демонстрирующий базовое использование модуля Pickle:

```
# Импортирование Pickle
try:
    import cPickle as pickle # Python 2
except ImportError:
    import pickle # Python 3

# Создание Python-объекта:
class Family(object):
    def __init__(self, names):
        self.sons = names

    def __str__(self):
        return ' '.join(self.sons)

my_family = Family([ 'John', 'David' ])

# Выгрузка в строку
pickle_data = pickle.dumps(my_family, pickle.HIGHEST_PROTOCOL)
```

```
# Выгрузка в файл
with open('family.p', 'w') as pickle_file:
    pickle.dump(families, pickle_file, pickle.HIGHEST_PROTOCOL)
```

```
# Загрузка из строки
my_family = pickle.loads(pickle_data)
```

```
# Загрузка из файла
with open('family.p', 'r') as pickle_file:
    my_family = pickle.load(pickle_file)
```

Подробную информацию о модуле Pickle см. в соответствующей главе.

ПРЕДУПРЕЖДЕНИЕ: В официальной документации по Pickle указано, что никаких гарантий безопасности нет. Не загружайте данные, в происхождении которых вы не уверены.

Глава 116. Многопроцессорная обработка

116.1. Запуск двух простых процессов

Простым примером использования нескольких процессов могут служить два процесса (workers), которые выполняются отдельно. В следующем примере запускаются два процесса:

- `countUp()` отсчитывает 1 единицу вверх, каждую секунду.
- `countDown()` отсчитывает 1 единицу вниз, каждую секунду.

```
import multiprocessing
import time
from random import randint
```

```
def countUp():
    i = 0
    while i <= 3:
        print('Up:\t{}'.format(i))
        time.sleep(randint(1, 3)) # сон 1, 2 или 3 секунды
        i += 1
```

```
def countDown():
    i = 3
    while i >= 0:
        print('Down:\t{}'.format(i))
        time.sleep(randint(1, 3)) # сон 1, 2 или 3 секунды
        i -= 1
```

```
if __name__ == '__main__':
    # Инициировать процессы (workers).
    workerUp = multiprocessing.Process(target=countUp)
    workerDown = multiprocessing.Process(target=countDown)
```

```
# Запуск процессов (workers).
workerUp.start()
workerDown.start()
```

```
# Присоединиться к workers. Это приведет к блокировке главного (родительского) процесса
# до завершения работы workers.
workerUp.join()
workerDown.join()
```

Выходные данные выглядят следующим образом:

```
Up: 0
Down: 3
Up: 1
Up: 2
Down: 2
Up: 3
Down: 1
Down: 0
```

116.2. Использование класса Pool and функции pool.map

```
from multiprocessing import Pool
```

```
def cube(x):
    return x ** 3
```

```
if __name__ == "__main__":
    pool = Pool(5)
    result = pool.map(cube, [0, 1, 2, 3])
```

Pool – это класс, который управляет несколькими workers (процессами) “за кулисами” и позволяет вам, программисту, использовать их.

Pool(5) создает новый Pool с 5 процессами, а pool.map работает так же, как и map, но использует несколько процессов (количество определяется при создании пула). Аналогичных результатов можно добиться, используя map_async, apply и apply_async.

Глава 117. Многопоточность

Потоки позволяют программам Python выполнять несколько функций одновременно, в отличие от выполнения последовательности команд по отдельности. В этой главе объясняются принципы работы потоков и демонстрируется их использование.

117.1. Основы многопоточности

С помощью модуля threading можно запустить новый поток выполнения, создав новый поток threading.Thread и назначив ему функцию для выполнения:

```
import threading

def foo():
    print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

Параметр target ссылается на запускаемую функцию (или вызываемый объект). Поток не начнет выполнение до тех пор, пока не будет вызван start на объекте Thread.

Запуск потока

```
my_thread.start() # выводит 'Hello threading!'
```

Теперь, когда поток my_thread запущен и завершен, повторный вызов start приведет к ошибке RuntimeError. Если вы хотите запустить свой поток в качестве демона (daemon), то, передав kwarg daemon=True или установив значение my_thread.daemon равным True перед вызовом start(), вы заставите ваш поток работать в фоновом режиме в качестве демона.

Присоединение к потоку

В тех случаях, когда вы разделили одно большое задание на несколько маленьких и хотите выполнять их одновременно, но при этом необходимо дождаться завершения всех заданий, прежде чем продолжить выполнение, вам подойдет метод `Thread.join()`.

Допустим, вы хотите загрузить несколько страниц сайта и скомпилировать их в одну страницу. Это можно сделать следующим образом:

```
import requests
from threading import Thread
from queue import Queue

q = Queue(maxsize=20)
def put_page_to_q(page_num):
    q.put(requests.get('http://some-website.com/page_%s.html' % page_num))

def compile(q):
    # магическая функция, которой для выполнения необходимы все страницы
    if not q.full():
        raise ValueError
    else:
        print("Done compiling!")

threads = []
for page_num in range(20):
    t = Thread(target=requests.get, args=(page_num,))
    t.start()
    threads.append(t)
# Далее объединяем все потоки, чтобы убедиться, что все потоки завершили работу, прежде чем
# продолжить. join() - блокирующий вызов (если не указано иное с помощью
# kwarg blocking=False при вызове join).for t in threads:
    t.join()

# Вызовем compile(), поскольку все потоки завершены
compile(q)
```

Создание класса пользовательских потоков

Используя класс `threading.Thread`, мы можем подклассифицировать новый пользовательский класс `Thread`. При этом мы должны переопределить метод `run` в подклассе.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start() # запуск автоматически вызывает метод Thread class run
    # print 'Основная программа продолжает выполняться в фоновом режиме.'
    t.join()
    print("Основная программа продолжает выполняться в фоновом режиме.")
```

117.2. Общение между потоками

В коде имеется несколько потоков, и необходимо обеспечить безопасное взаимодействие между ними. Для этого можно использовать `Queue` из библиотеки `queue`.

```
from queue import Queue
from threading import Thread
```

```
# создать производителя данных
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# создать потребителя
def consumer(input_queue):
    while True:
        # получение данных
        data = input_queue.get()

        # сделать что-нибудь с данными

        # указать на то, что данные были израсходованы
        input_queue.task_done()
```

Создание потоков производителя и потребителя с общей очередью:

```
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

117.3. Создание пула процессов (workers)

Использование threading и queue:

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue
```

```
def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Запуск клиентских процессов (client workers)
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()
```

```
# Запуск сервера
sock = socket(AF_INET, SOCK_STREAM)
sock.bind(addr)
sock.listen(5)
while True:
    client_sock, client_addr = sock.accept()
    q.put((client_sock, client_addr))
```

```
echo_server(('', 15000), 128)
```

Использование concurrent.futures.ThreadPoolExecutor:

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor
```

```
def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
```

```

sock.bind(addr)
sock.listen(5)
while True:
    client_sock, client_addr = sock.accept()
    pool.submit(echo_client, client_sock, client_addr)

```

```
echo_server(("15000"))
```

Этот пример взят из книги “Python Cookbook, 3rd edition”, by David Beazley and Brian K. Jones (O’Reilly). Copyright 2013 David Beazley and Brian Jones, ISBN 978-1-449-34037-7.

117.4. Продвинутое использование многопоточности

В этой главе будут приведены наиболее сложные примеры, реализованные с использованием многопоточности.

Усовершенствованный вывод (регистратор)

Поток, который выводит все, что получает, и модифицирует вывод в соответствии с шириной терминала. Приятно то, что при изменении ширины терминала модифицируется и вывод “уже написанного”.

```
#!/usr/bin/env python2
```

```

import threading
import Queue
import time
import sys
import subprocess
from backports.shutil_get_terminal_size import get_terminal_size

```

```

printq = Queue.Queue()
interrupt = False
lines = []

```

```
def main():
```

```

    ptt = threading.Thread(target=printer) # Включает вывод (printer)
    ptt.daemon = True
    ptt.start()

```

```
    # Просто пример того, что нужно выводить
```

```

    for i in xrange(1,100):
        printq.put(''.join([str(x) for x in range(1,i)])) # Фактический способ отправки материала на вывод
        time.sleep(.5)

```

```
def split_line(line, cols):
```

```

    if len(line) > cols:
        new_line = ""
        ww = line.split()
        i = 0
        while len(new_line) <= (cols - len(ww[i]) - 1):
            new_line += ww[i] + ' '
            i += 1
        print len(new_line)
    if new_line == "":
        return (line, "")

```

```

    return (new_line, ''.join(ww[i:]))
else:
    return (line, "")

```

```
def printer():
    while True:
        cols, rows = get_terminal_size() # Получение размеров терминала
        msg = '#' + '.' * (cols - 2) + '#\n' # Создание
        try:
            new_line = str(printq.get_nowait())
            if new_line != '!@#EXIT#@!': # Хороший способ отключить поток вывода
                lines.append(new_line)
                printq.task_done()
            else:
                printq.task_done()
                sys.exit()
        except Queue.Empty:
            pass

    # Построение нового сообщения для отображения и разбиения слишком длинных строк
    for line in lines:
        res = line # Ниже приводится разбиение строк, которые
                  # длиннее, чем столбцы.
        while len(res) != 0:
            toprint, res = split_line(res, cols)
            msg += '\n' + toprint

    # Очистить оболочку и вывести новый вывод
    subprocess.check_call('clear') # Сохранить оболочку чистой
    sys.stdout.write(msg)
    sys.stdout.flush()
    time.sleep(.5)
```

117.5. Останавливаемый поток с циклом while

```
import threading
import time

class StoppableThread(threading.Thread):
    """Класс Thread с методом stop().
    Сам поток должен регулярно проверять состояние stopped()."""

    def __init__(self):
        super(StoppableThread, self).__init__()
        self._stop_event = threading.Event()

    def stop(self):
        self._stop_event.set()

    def join(self, *args, **kwargs):
        self.stop()
        super(StoppableThread, self).join(*args, **kwargs)

    def run():
        while not self._stop_event.is_set():
            print("еще работает!")
            time.sleep(2)
        print("stopped!")
```

Глава 118. Процессы и потоки

Большинство программ выполняются построчно, одновременно запуская только один процесс. Потоки позволяют нескольким процессам работать независимо друг от друга. Использование потоков на нескольких процессорах позволяет программам выполнять несколько процессов одновременно. В этой теме описываются реализация и использование потоков в Python.

118.1. Глобальная блокировка интерпретатора

Производительность многопоточности в Python часто может снижаться из-за глобальной блокировки интерпретатора (Global Interpreter Lock, GIL). Одним словом, несмотря на то, что в программе на Python может быть несколько потоков, в каждый момент времени, независимо от количества процессоров, параллельно может выполняться только одна инструкция байткода.

Многопоточность в случаях, когда операции блокируются внешними событиями, например доступом к сети, может быть весьма эффективной:

```
import threading
import time

def process():
    time.sleep(2)

start = time.time()
process()
print("Один прогон занял %.2fs" % (time.time() - start))

start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Четыре прогона заняли %.2fs" % (time.time() - start))

# Вывод: Один прогон занял 2.00s
# Вывод: Четыре прогона заняли 2.00s
```

Обратите внимание, что, несмотря на то, что каждый процесс занимает 2 секунды, все четыре процесса вместе смогли эффективно работать параллельно, заняв в общей сложности 2 секунды.

Однако многопоточность в случаях, когда в коде Python выполняются интенсивные вычисления – например, большое количество вычислений, – не приводит к значительному улучшению, и даже может быть медленнее, чем при параллельной работе:

```
import threading
import time

def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
```



```

for i in range(100000):
    result = otherfunc(result, somefunc(i))

start = time.time()
process()
print("Один прогон занял %.2fs" % (time.time() - start))

start = time.time()
threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()
print("Четыре прогона заняли %.2fs" % (time.time() - start))

# Вывод: Один прогон занял 2.05s
# Вывод: Четыре прогона заняли 14.42s

```

В последнем случае многопроцессорность может быть очень эффективной, так как несколько процессов могут выполнять несколько инструкций одновременно:

```

import multiprocessing
import time

def somefunc(i):
    return i * i

def otherfunc(m, i):
    return m + i

def process():
    for j in range(100):
        result = 0
        for i in range(100000):
            result = otherfunc(result, somefunc(i))

start = time.time()
process()
print("Один прогон занял %.2fs" % (time.time() - start))

start = time.time()
processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()
print("Четыре прогона заняли %.2fs" % (time.time() - start))

# Вывод: Один прогон занял 2.07s
# Вывод: Четыре прогона заняли 2.30s

```

118.2. Работа с несколькими потоками

Используйте `threading.Thread` для запуска функции в другом потоке.

```

import threading
import os

def process():
    print("Pid is %s, thread id is %s" % (os.getpid(), threading.current_thread().name))

```

```

threads = [threading.Thread(target=process) for _ in range(4)]
for t in threads:
    t.start()
for t in threads:
    t.join()

```

```

# Вывод: Pid is 11240, thread id is Thread-1
# Вывод: Pid is 11240, thread id is Thread-2
# Вывод: Pid is 11240, thread id is Thread-3
# Вывод: Pid is 11240, thread id is Thread-4

```

118.3. Работа с несколькими процессами

Используйте `multiprocessing.Process` для запуска функции в другом процессе. Интерфейс аналогичен интерфейсу `threading.Thread`:

```

import multiprocessing
import os

def process():
    print("Pid is %s" % (os.getpid(),))

processes = [multiprocessing.Process(target=process) for _ in range(4)]
for p in processes:
    p.start()
for p in processes:
    p.join()

# Вывод: Pid is 11206
# Вывод: Pid is 11207
# Вывод: Pid is 11208
# Вывод: Pid is 11209

```

118.4. Совместное использование состояния потоками

Поскольку все потоки выполняются в одном и том же процессе, они имеют доступ к одним и тем же данным. Однако одновременный доступ к общим данным должен быть защищен блокировкой, чтобы избежать проблем с синхронизацией.

```

import threading

obj = {}
obj_lock = threading.Lock()

def objify(key, val):
    print("Obj имеет %d значений(-я)" % len(obj))
    with obj_lock:
        obj[key] = val
    print("Obj теперь имеет %d значений" % len(obj))

ts = [threading.Thread(target=objify, args=(str(n), n)) for n in range(4)]
for t in ts:
    t.start()
for t in ts:
    t.join()
print("Obj окончательный результат:")
import pprint; pprint.pprint(obj)

# Вывод: Obj имеет 0 значений(-я)
# Вывод: Obj имеет 0 значений(-я)
# Вывод: Obj теперь имеет 1 значений(-я)
# Вывод: Obj теперь имеет 2 значений(-я) Obj имеет 2 значений(-я)

```

```
# Вывод: Obj теперь имеет 3 значений(-я)
# Вывод:
# Вывод: Obj имеет 3 значений(-я)
# Вывод: Obj теперь имеет 4 значений(-я)
# Вывод: Obj окончательный результат:
# Вывод: {'0': 0, '1': 1, '2': 2, '3': 3}
```

118.5. Совместное использование состояния процессами

Код, выполняющийся в разных процессах, по умолчанию не использует одни и те же данные. Однако модуль `multiprocessing` содержит примитивы для совместного использования значений несколькими процессами.

```
import multiprocessing

plain_num = 0
shared_num = multiprocessing.Value('d', 0)
lock = multiprocessing.Lock()

def increment():
    global plain_num
    with lock:
        # модификации обычных переменных не видны в разных процессах
        plain_num += 1
        # multiprocessing.Value модификации
        shared_num.value += 1

ps = [multiprocessing.Process(target=increment) for n in range(4)]
for p in ps:
    p.start()
for p in ps:
    p.join()

print("plain_num is %d, shared_num is %d" % (plain_num, shared_num.value))

# Вывод: plain_num равен 0, shared_num равен 4
```

Глава 119. Параллелизм в Python

119.1. Многопроцессорный модуль

```
from __future__ import print_function
import multiprocessing

def countdown(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

if __name__ == "__main__":
    p1 = multiprocessing.Process(target=countdown, args=(10,))
    p1.start()

    p2 = multiprocessing.Process(target=countdown, args=(20,))
    p2.start()
```

```
p1.join()
p2.join()
```

Здесь каждая функция выполняется в новом процессе. Поскольку код выполняется в новом экземпляре виртуальной машины Python VM, то отсутствует GIL и обеспечивается параллелизм при работе на нескольких ядрах. Метод `Process.start` запускает этот новый процесс и выполняет функцию, переданную в аргументе `target`, с аргументами `args`. Метод `Process.join` ожидает окончания выполнения процессов `p1` и `p2`.

Новые процессы запускаются по-разному в зависимости от версии Python и платформы, на которой выполняется код, например:

- Для создания нового процесса Windows использует `spawn`.
- В Unix-системах и версиях ранее 3.3 процессы создаются с помощью `fork`. Заметим, что этот метод не соответствует POSIX-стандарту использования и, следовательно, приводит к неожиданному поведению, особенно при взаимодействии с другими многопроцессорными библиотеками.
- В Unix версии 3.4+ можно выбрать запуск новых процессов с помощью `fork`, `forkserver` или `spawn`, используя метод `multiprocessing.set_start_method` в начале программы. Методы `forkserver` и `spawn` медленнее, чем `forking`, но позволяют избежать некоторых неожиданностей.

Использование POSIX fork:

После вилки (`fork`) в многопоточной программе дочерняя программа может безопасно вызывать только `async-signal-safe` функции до тех пор, пока не вызовет `execve`.

При использовании вилки `fork` будет запущен новый процесс с точно таким же состоянием для всех текущих мьютексов (взаимоисключений, `mutual exception`, кратко – `mutex`), но будет запущен только `MainThread`. Это небезопасно, так как может привести к возникновению “условий гонки”, например:

- Если вы используете блокировку (`Lock`) в `MainThread` и передаете ее другому потоку, который в какой-то момент должен ее заблокировать. Если `fork` произойдет одновременно, то новый процесс начнется с заблокированной блокировкой, которая никогда не будет освобождена, поскольку второго потока в этом новом процессе не существует.

На самом деле в “чистом” Python такого поведения быть не должно, поскольку мультипроцессинг справляется с этим должным образом, но при взаимодействии с другими библиотеками такое поведение может привести к краху системы (например, с `numpy/accelerated` на macOS).

119.2. Модуль потокообразования (threading)

```
from __future__ import print_function
import threading
def counter(count):
    while count > 0:
        print("Count value", count)
        count -= 1
    return

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
t2 = threading.Thread(target=countdown,args=(20,))
t2.start()
```

В некоторых реализациях Python, таких как CPython, истинный параллелизм не достигается с помощью потоков из-за использования так называемой GIL, или глобальной блокировки интерпретатора.

119.3. Передача данных между многопроцессорными процессами

Поскольку работа с данными между двумя потоками очень чувствительна (одновременное чтение и одновременная запись могут противоречить друг другу, вызывая “условия гонки”), для облегчения передачи данных между потоками был создан набор уникальных объектов. Между потоками можно использовать любую “операцию над атомами”, но всегда безопасно использовать Queue.

```
import multiprocessing
import queue
my_Queue=multiprocessing.Queue()
#Создает очередь с неопределенным максимальным размером
#это может быть опасно, так как очередь становится все больше и больше
#копирование данных в/из каждого потока чтения/записи будет занимать много времени
```

Большинство специалистов советуют при использовании очереди всегда помещать данные очереди в блок try: except:, а не использовать empty. Однако для приложений, где не имеет значения, пропустите ли вы цикл сканирования (данные могут быть помещены в очередь, пока она переходит из состояния queue.Empty==True в состояние queue.Empty==False), обычно лучше поместить доступ на чтение и запись в так называемый блок Iftry, поскольку оператор ‘if’ технически более производителен, чем перехват исключения.

```
import multiprocessing
import queue
"""Импорт необходимых стандартных библиотек Python, мультипроцессинга для классов
и очереди для предоставляемых ею исключений из очереди"""
def Queue_Iftry_Get(get_queue, default=None, use_default=False, func=None, use_func=False):
    """Этот глобальный метод для блока Iftry предоставляется для его повторного использования
    и стандартной функциональности, if также позволяет сэкономить на производительности,
    в отличие от перехвата исключения, который требует больших затрат.
    Он также позволяет пользователю указать функцию для использования исходящих данных,
    и значение по умолчанию, которое возвращается, если функция не может вернуть значение
    из очереди"""
    if get_queue.empty():
        if use_default:
            return default
    else:
        try:
            value = get_queue.get_nowait()
        except queue.Empty:
            if use_default:
                return default
        else:
            if use_func:
                return func(value)
            else:
                return value
def Queue_Iftry_Put(put_queue, value):
    """Этот глобальный метод для блока Iftry предоставляется из-за его повторного
    использования и стандартной функциональности, If также позволяет сэкономить на
    производительности, в отличие от перехвата исключения, который требует больших затрат.
    Возвращает True, если помещение значения в очередь прошло успешно. В противном случае -
    false"""
    if put_queue.full():
        return False
    else:
        try:
            put_queue.put_nowait(value)
        except queue.Full:
            return False
        else:
            return True
```

Глава 120. Параллельные вычисления

120.1. Использование модуля многопроцессорной обработки для распараллеливания задач

```
import multiprocessing
```

```
def fib(n):
    """Вычисление чисел Фибоначчи неэффективным способом было выбрано
    для замедления работы процессора."""
    if n <= 2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
p = multiprocessing.Pool()
print(p.map(fib,[38,37,36,35,34,33]))
```

Результат: [39088169, 24157817, 14930352, 9227465, 5702887, 3524578]

Поскольку выполнение каждого вызова `fib` происходит параллельно, время выполнения полного примера на двухпроцессорном компьютере оказывается в 1,8 раза быстрее, чем при последовательном выполнении.

120.2. Использование C-расширения для распараллеливания задач

Идея нижеприведенного кода заключается в том, чтобы перенести вычислительно интенсивные задания на язык C (с помощью специальных макросов), не зависящий от Python, и чтобы код на языке C освобождал GIL в процессе работы.

```
#include "Python.h"
```

```
PyObject *pyfunc(PyObject *self, PyObject *args) {
```

```
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
```

```
    Py_END_ALLOW_THREADS
```

```
}
```

120.3. Использование родительских и дочерних скриптов для параллельного выполнения кода

child.py

```
import time
```

```
def main():
    print "starting work"
    time.sleep(1)
    print "work work work work work"
    time.sleep(1)
    print "done working"
```

```
if __name__ == '__main__':
    main()
```

parent.py

```
import os

def main():
    for i in range(5):
        os.system("python child.py &")

if __name__ == '__main__':
    main()
```

Это удобно для параллельных, независимых задач HTTP-запросов/ответов или выбора/вставки в базу данных. Аргументы командной строки также могут быть переданы скрипту **child.py**. Синхронизация между скриптами может быть достигнута за счет того, что все скрипты регулярно проверяют отдельный сервер.

120.4. Использование модуля PyPar для распараллеливания

PyPar – это библиотека, использующая интерфейс передачи сообщений (message passing interface, MPI) для обеспечения параллелизма в Python. Простой пример на PyPar выглядит следующим образом:

```
import pypar as pp

ncpus = pp.size()
rank = pp.rank()
node = pp.get_processor_name()

print 'I am rank %d of %d on node %s' % (rank, ncpus, node)

if rank == 0:
    msh = 'P0'
    pp.send(msg, destination=1)
    msg = pp.receive(source=rank-1)
    print 'Processor 0 received message "%s" from rank %d' % (msg, rank-1)
else:
    source = rank-1
    destination = (rank+1) % ncpus
    msg = pp.receive(source)
    msg = msg + 'P' + str(rank)
    pypar.send(msg, destination)
pp.finalize()
```

Глава 121. Сокеты

Параметр	Описание
socket.AF_UNIX	UNIX Socket
socket.AF_INET	IPv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	TCP
socket.SOCK_DGRAM	UDP

Многие языки программирования используют сокеты для обмена данными между процессами или устройствами. В данной теме рассматривается правильное использование модуля `sockets` в Python для облегчения отправки и получения данных по распространенным сетевым протоколам.

121.1. Сырые сокеты (Raw Sockets) в Linux

Сначала отключается автоматическая проверка контрольных сумм сетевой карты:

```
sudo ethtool -K eth1 tx off
```

Затем отправьте свой пакет, используя сокет `SOCK_RAW`:

```
#!/usr/bin/env python
from socket import socket, AF_PACKET, SOCK_RAW
s = socket(AF_PACKET, SOCK_RAW)
s.bind(("eth1", 0))

# Мы собираем здесь фрейм ethernet,
# но вместо него можно использовать что угодно.
# Обратите внимание на модуль 'struct' для более
# гибкой упаковки/распаковки двоичных данных
# и 'binascii' для 32-битного CRC
src_addr = "\x01\x02\x03\x04\x05\x06"
dst_addr = "\x01\x02\x03\x04\x05\x06"
payload = ("["*30)+"PAYLOAD"+("]"*30)
checksum = "\x1a\x2b\x3c\x4d"
ethertype = "\x08\x01"

s.send(dst_addr+src_addr+ethertype+payload+checksum)
```

121.2. Передача данных по протоколу UDP

UDP – это протокол без установления соединения. Сообщения другим процессам или компьютерам отправляются без установления какого-либо соединения. Автоматического подтверждения того, что сообщение получено, не происходит. UDP обычно используется в приложениях, чувствительных к задержкам, или в приложениях, посылающих широковещательные сообщения по сети.

Следующий код отправляет сообщение процессу, прослушивающему порт 6667 на `localhost`, используя UDP. *Обратите внимание, что после отправки нет необходимости “закрывать” сокет, поскольку UDP не имеет соединений.*

```
from socket import socket, AF_INET, SOCK_DGRAM
s = socket(AF_INET, SOCK_DGRAM)
msg = ("Hello you there!").encode('utf-8')
# socket.sendto() принимает на входе байты, поэтому сначала необходимо закодировать строку.
s.sendto(msg, ('localhost', 6667))
```

121.3. Получение данных по протоколу UDP

UDP является протоколом без соединений. Это означает, что пиры (`peers`), посылающие сообщения, не должны устанавливать соединение перед отправкой сообщений. `socket.recvfrom` возвращает кортеж (`msg` [сообщение, полученное сокетом], `addr` [адрес отправителя]).

UDP-сервер, использующий только модуль `socket`:

```
from socket import socket, AF_INET, SOCK_DGRAM
sock = socket(AF_INET, SOCK_DGRAM)
sock.bind(('localhost', 6667))
```



```
while True:
    msg, addr = sock.recvfrom(8192) # Это количество байтов для максимального чтения
    print("Got message from %s: %s" % (addr, msg))
```

Ниже приведена альтернативная реализация с использованием `socketserver.UDPServer`:

```
from socketserver import BaseRequestHandler, UDPServer

class MyHandler(BaseRequestHandler):
    def handle(self):
        print("Получено соединение от: %s" % self.client_address)
        msg, sock = self.request
        print("Оно сообщает: %s" % msg)
        sock.sendto("Получил ваше сообщение!".encode(), self.client_address) # Отправить ответ

serv = UDPServer(('localhost', 6667), MyHandler)
serv.serve_forever()
```

По умолчанию сокеты блокируются. Это означает, что выполнение скрипта будет ждать, пока сокет не получит данные.

121.4. Отправка данных по протоколу TCP

Передача данных через Интернет осуществляется с помощью нескольких модулей. Модуль `sockets` обеспечивает низкоуровневый доступ к базовым операциям операционной системы, отвечающим за отправку или получение данных от других компьютеров или процессов. Следующий код отправляет байтовую строку `b'Hello'` на TCP-сервер, прослушивающий порт 6667 на хосте `localhost`, и закрывает соединение после завершения:

```
from socket import socket, AF_INET, SOCK_STREAM
s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 6667)) # Адрес прослушивающего TCP-сервера
s.send(b'Hello')
s.close()
```

По умолчанию вывод через сокет блокируется, то есть программа будет ждать в вызовах `connect` и `send` до тех пор, пока действие не будет “завершено”. Для `connect` это означает, что сервер действительно принимает соединение. Для `send` это означает только то, что операционная система имеет достаточно места в памяти, чтобы поставить данные в очередь для последующей отправки. После использования сокеты всегда должны быть закрыты.

121.5. Многопоточный сервер TCP Socket Server

При запуске без аргументов эта программа запускает сервер TCP-сокетов, который прослушивает соединения с адресом `127.0.0.1` на порту 5000. Сервер обрабатывает каждое соединение в отдельном потоке.

При запуске с аргументом `-s` эта программа подключается к серверу, считывает список клиентов и выводит его на экран. Список клиентов передается в виде JSON-строки. Имя клиента может быть задано с помощью аргумента `-n`. Передавая разные имена, можно наблюдать эффект на списке клиентов.

client_list.py

```
import argparse
import json
import socket
import threading

def handle_client(client_list, conn, address):
    name = conn.recv(1024)
    entry = dict(zip(['name', 'address', 'port'], [name, address[0], address[1]]))
    client_list[name] = entry
```

```
conn.sendall(json.dumps(client_list))
conn.shutdown(socket.SHUT_RDWR)
conn.close()
```

```
def server(client_list):
    print "Запуск сервера..."
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('127.0.0.1', 5000))
    s.listen(5)
    while True:
        (conn, address) = s.accept()
        t = threading.Thread(target=handle_client, args=(client_list, conn, address))
        t.daemon = True
        t.start()
```

```
def client(name):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(('127.0.0.1', 5000))
    s.send(name)
    data = s.recv(1024)
    result = json.loads(data)
    print json.dumps(result, indent=4)
```

```
def parse_arguments():
    parser = argparse.ArgumentParser()
    parser.add_argument('-c', dest='client', action='store_true')
    parser.add_argument('-n', dest='name', type=str, default='name')
    result = parser.parse_args()
    return result
```

```
def main():
    client_list = dict()
    args = parse_arguments()
    if args.client:
        client(args.name)
    else:
        try:
            server(client_list)
        except KeyboardInterrupt:
            print "Прерывание клавиатуры"
```

```
if __name__ == '__main__':
    main()
```

Вывод сервера

```
$ python client_list.py
Starting server...
```

Вывод у клиента

```
$ python client_list.py -c -n name1
{
  "name1": {
    "address": "127.0.0.1",
    "port": 62210,
    "name": "name1"
  }
}
```

Буферы приема ограничены 1024 байтами. Если строковое представление JSON списка клиентов превышает этот размер, оно будет усечено. Это приведет к возникновению следующего исключения:

```
ValueError: Unterminated string starting at: line 1 column 1023 (char 1022) (Непрерванная строка, начинающаяся в: строка 1 столбец 1023 (char 1022))
```

Глава 122. Веб-сокеты

122.1. Простое эхо с помощью aiohttp

aiohttp обеспечивает асинхронные веб-сокеты.

Версия Python 3.x ≥ 3.5:

```
import asyncio
from aiohttp import ClientSession

with ClientSession() as session:
    async def hello_world():

        websocket = await session.ws_connect("wss://echo.websocket.org")

        websocket.send_str("Hello, world!")

        print("Received:", (await websocket.receive()).data)

        await websocket.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(hello_world())
```

122.2. Класс-обертка с aiohttp

aiohttp.ClientSession может быть использован в качестве родительского для пользовательского класса веб-сокета.

Версия Python 3.x ≥ 3.5:

```
import asyncio
from aiohttp import ClientSession

class EchoWebSocket(ClientSession):

    URL = "wss://echo.websocket.org"

    def __init__(self):
        super().__init__()
        self.websocket = None

    async def connect(self):
        """Подключение к WebSocket."""
        self.websocket = await self.ws_connect(self.URL)

    async def send(self, message):
        """Отправить сообщение в WebSocket."""
        assert self.websocket is not None, "Сначала вы должны подключиться!"
        self.websocket.send_str(message)
        print("Отправлено:", message)
```

```

async def receive(self):
    """Получить одно сообщение от WebSocket."""
    assert self.websocket is not None, "Сначала вы должны подключиться!"
    return (await self.websocket.receive()).data

async def read(self):
    """Чтение сообщений из WebSocket."""
    assert self.websocket is not None, "Сначала вы должны подключиться!"

    while self.websocket.receive():
        message = await self.receive()
        print("Получено:", message)
        if message == "Echo 9!":
            break

async def send(websocket):
    for n in range(10):
        await websocket.send("Echo {}!".format(n))
        await asyncio.sleep(1)

loop = asyncio.get_event_loop()

with EchoWebSocket() as websocket:

    loop.run_until_complete(websocket.connect())

    tasks = (
        send(websocket),
        websocket.read()
    )

    loop.run_until_complete(asyncio.wait(tasks))

    loop.close()

```

122.3. Использование пакета Autobahn в качестве фабрики веб-сокетов

Пакет Autobahn может быть использован для создания фабрик веб-сокет-серверов на языке Python. Для установки, как правило, достаточно воспользоваться командой терминала (для Linux):

```
sudo pip install autobahn
```

(Для Windows):

```
python -m pip install autobahn
```

Затем в Python-скрипте можно создать простой эхо-сервер:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
class MyServerProtocol(WebSocketServerProtocol):
```

"При создании серверного протокола

Класс, определяемый пользователем и наследующий WebSocketServerProtocol, должен переопределить события onMessage, onConnect и т. д. для обеспечения функциональности, определяемой пользователем.

Эти события, по сути, определяют протокол вашего сервера"

```
def onMessage(self, payload, isBinary):
```

"Процедура onMessage вызывается, когда сервер получает сообщение.

Он имеет необходимую полезную нагрузку в виде аргументов и bool isBinary. Полезная нагрузка - это собственно содержимое "сообщения", а isBinary - это просто флаг, сообщающий пользователю, что полезная нагрузка содержит двоичные данные. Обычно в других случаях предполагается, что полезная нагрузка представляет собой строку.

```

В данном примере полезная нагрузка возвращается отправителю дословно."
self.sendMessage(payload,isBinary)
if __name__=='__main__':
    try:
        import asyncio
    except ImportError:
        "Trollius = 0.3 was renamed"
        import trollius as asyncio
    from autobahn.asyncio.websocket import WebSocketServerFactory
    factory=WebSocketServerFactory()
    "Инициализируем фабрику websocket и устанавливаем протокол на определенный
    выше протокол (класс, который наследуется от
    autobahn.asyncio.websocket.WebSocketServerProtocol)"
    factory.protocol=MyServerProtocol
    "Эту строку можно рассматривать как "связывание" методов
    onConnect, onMessage и т. д., которые были описаны в классе MyServerProtocol,
    задавая серверу функциональность, т.е. протокол"
    loop=asyncio.get_event_loop()
    coro=loop.create_server(factory,'127.0.0.1',9000)
    server=loop.run_until_complete(coro)
    "Запускать сервер в бесконечном цикле"
    try:
        loop.run_forever()
    except KeyboardInterrupt:
        pass
    finally:
        server.close()
        loop.close()

```

В данном примере сервер создается на localhost (127.0.0.1) на порту 9000. Это IP-адрес и порт прослушивания. Это важная информация, так как с ее помощью можно определить LAN-адрес компьютера и пробросить порт от модема, через любые маршрутизаторы, которые у вас есть, к компьютеру. Затем, используя google для изучения IP-адреса WAN, можно спроектировать свой сайт так, чтобы он отправлял сообщения WebSocket на IP-адрес WAN, на порт 9000 (в данном примере).

Важно, чтобы переадресация портов осуществлялась от модема обратно, т.е. если к модему последовательно подключены маршрутизаторы, войдите в настройки конфигурации модема, переадресуйте порт с модема на подключенный маршрутизатор и так далее, пока на конечный маршрутизатор, к которому подключен ваш компьютер, не будет переадресована информация, получаемая на порт 9000 модема (в данном примере).

Глава 123. Сокеты и шифрование/дешифрование сообщений между клиентом и сервером

В целях обеспечения безопасности используется криптография. Примеров шифрования/дешифрования в Python с использованием IDEA encryption MODE CTR не так много. Цель нижеприведенной документации – расширение и реализация схемы цифровой подписи RSA в межстанционной связи; использование хеширования для обеспечения целостности сообщения, то есть SHA-1; разработка простого протокола транспортировки ключей; шифрование ключа с помощью IDEA.

123.1. Реализация на стороне сервера

```

import socket
import hashlib
import os
import time
import itertools
import threading
import sys
import Crypto.Cipher.AES as AES
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

# адрес сервера и номер порта вводятся от администратора
host= raw_input("Server Address - > ")
port = int(input("Port - > "))
# boolean для проверки сервера и порта
check = False
done = False

def animate():
    for c in itertools.cycle(['.', '..', '...', '....', '.....', '.....']):
        if done:
            break
        sys.stdout.write('\rПРОВЕРКА IP-АДРЕСА И НЕИСПОЛЬЗУЕМОГО ПОРТА '+c)
        sys.stdout.flush()
        time.sleep(0.1)
    sys.stdout.write('\r ---СЕРВЕР ЗАПУЩЕН. ОЖИДАНИЕ КЛИЕНТА---\n')
try:
    # настройка сокета
    server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    server.bind((host,port))
    server.listen(5)
    check = True
except BaseException:
    print " ---Проверьте адрес сервера или порт---"
    check = False

if check is True:
    # server Quit
    shutdown = False
    # printing "Сообщение о запуске сервера"
    thread_load = threading.Thread(target=animate)
    thread_load.start()

    time.sleep(4)
    done = True
    #привязка клиента и адреса
    client,address = server.accept()
    print ("КЛИЕНТ СОЕДИНЕН. АДРЕС КЛИЕНТА ->",address)
    print ("\n ---ОЖИДАНИЕ ОТКРЫТОГО КЛЮЧА И ХЕША ОТКРЫТОГО КЛЮЧА---\n")

    #сообщение клиента(открытый ключ)
    getpbk = client.recv(2048)

    #преобразование строки в КЛЮЧ
    server_public_key = RSA.importKey(getpbk)

    #хеширование открытого ключа на стороне сервера для проверки хеша, полученного от клиента
    hash_object = hashlib.sha1(getpbk)
    hex_digest = hash_object.hexdigest()

```

```

if getpbk != "":
    print(getpbk)
    client.send("YES")
    gethash = client.recv(1024)
    print("\n---ХЕШ ОТКРЫТОГО КЛЮЧА--- \n"+gethash)
if hex_digest == gethash:
    # создание сеансового ключа
    key_128 = os.urandom(16)
    #шифрование сеансового ключа CTR MODE
    en = AES.new(key_128,AES.MODE_CTR,counter = lambda:key_128)
    encrypto = en.encrypt(key_128)
    #хеширование sha1
    en_object = hashlib.sha1(encrypto)
    en_digest = en_object.hexdigest()

    print ("\n---СЕАНСОВЫЙ КЛЮЧ---\n"+en_digest)

#шифрование ключа сеанса и открытого ключа
E = server_public_key.encrypt(encrypto,16)
print ("\n---ЗАШИФРОВАННЫЙ ХЕШИРОВАННЫЙ КЛЮЧ И СЕАНСОВЫЙ КЛЮЧ---\n"+str(E))
print ("\n---HANDSHAKE COMPLETE---")
client.send(str(E))
while True:
    #сообщение от клиента
    newmess = client.recv(1024)
    #декодирование сообщения из шестнадцатичного для расшифровки
    #только зашифрованной версии сообщения
    decoded = newmess.decode("hex")
    #используя в качестве ключа en_digest(session_key)
    key = en_digest[:16]
    print ("\nЗАШИФРОВАННОЕ СООБЩЕНИЕ ОТ КЛИЕНТА -> "+newmess)
    #дешифровка сообщения от клиента
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print ("\n**Новое сообщение** "+time.ctime(time.time())+" > "+dMsg+"\n")
    mess = raw_input("\nСообщение клиенту -> ")
    if mess != "":
        ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda : key)
        eMsg = ideaEncrypt.encrypt(mess)
        eMsg = eMsg.encode("hex").upper()
        if eMsg != "":
            print ("ЗАШИФРОВАННОЕ СООБЩЕНИЕ КЛИЕНТУ-> " + eMsg)
            client.send(eMsg)
    client.close()
else:
    print ("\n---ХЕШ ОТКРЫТОГО КЛЮЧА НЕ СОВПАДАЕТ ----\n")

```

123.2. Реализация на стороне клиента

```

import time
import socket
import threading
import hashlib
import itertools
import sys
from Crypto import Random
from Crypto.PublicKey import RSA
from CryptoPlus.Cipher import IDEA

```

```

#анимация загрузки
done = False
def animate():

```

```

for c in itertools.cycle(['...', '.....', '.....', '.....']):
    if done:
        break
    sys.stdout.write('\rПОДТВЕРЖДЕНИЕ СОЕДИНЕНИЯ С СЕРВЕРОМ '+c)
    sys.stdout.flush()
    time.sleep(0.1)

#открытый (public) ключ и закрытый (private) ключ
random_generator = Random.new().read
key = RSA.generate(1024, random_generator)
public = key.publickey().exportKey()
private = key.exportKey()

#хеширование открытого ключа
hash_object = hashlib.sha1(public)
hex_digest = hash_object.hexdigest()

#Установка сокета
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#хост и порт вводятся пользователем
host = raw_input("Адрес подключаемого сервера -> ")
port = int(input("Порт сервера -> "))
#привязка адреса и порта
server.connect((host, port))
# вывод "Server Started Message"
thread_load = threading.Thread(target=animate)
thread_load.start()

time.sleep(4)
done = True

def send(t, name, key):
    mess = raw_input(name + " : ")
    key = key[:16]
    #слияние сообщения и имени
    whole = name + " : " + mess
    ideaEncrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    eMsg = ideaEncrypt.encrypt(whole)
    #преобразование зашифрованного сообщения в 16-ричное для возможности его чтения
    eMsg = eMsg.encode("hex").upper()
    if eMsg != "":
        print("\nЗАШИФРОВАННОЕ СООБЩЕНИЕ СЕРВЕРУ-> "+eMsg)
    server.send(eMsg)
def recv(t, key):
    newmess = server.recv(1024)
    print("\nЗАШИФРОВАННОЕ СООБЩЕНИЕ ОТ СЕРВЕРА-> " + newmess)
    key = key[:16]
    decoded = newmess.decode("hex")
    ideaDecrypt = IDEA.new(key, IDEA.MODE_CTR, counter=lambda: key)
    dMsg = ideaDecrypt.decrypt(decoded)
    print("\n**Новое сообщение ОТ сервера** " + time.ctime(time.time()) + " : " + dMsg + "\n")

while True:
    server.send(public)
    confirm = server.recv(1024)
    if confirm == "YES":
        server.send(hex_digest)

```



```

#сообщение connected
msg = server.recv(1024)
en = eval(msg)
decrypt = key.decrypt(en)
# хеширование sha1
en_object = hashlib.sha1(decrypt)
en_digest = en_object.hexdigest()

print ("\n---ЗАШИФРОВАННЫЙ ОТКРЫТЫЙ КЛЮЧ И СЕАНСОВЫЙ КЛЮЧ ОТ СЕРВЕРА---")
print (msg)
print ("\n---РАСШИФРОВАННЫЙ СЕАНСОВЫЙ КЛЮЧ---")
print (en_digest)
print ("\n---HANDSHAKE COMPLETE---\n")
alais = raw_input("\nВаше имя -> ")

while True:
    thread_send = threading.Thread(target=send,args=("---Отправка сообщения---",alais,en_digest))
    thread_rcv = threading.Thread(target=recv,args=("---Получение сообщения---",en_digest))
    thread_send.start()
    thread_rcv.start()

    thread_send.join()
    thread_rcv.join()
    time.sleep(0.5)
time.sleep(60)
server.close()

```

Глава 124. Сетевое взаимодействие на языке Python

124.1. Создание простого Http-сервера

Для совместного использования файлов или размещения простых сайтов (http и javascript) в локальной сети можно использовать встроенный в Python модуль SimpleHTTPServer. Python должен находиться в вашей переменной Path. Перейдите в папку, где находятся ваши файлы, и введите:

Для Python 2:

```
$ python -m SimpleHTTPServer <portnumber>
```

Для Python 3:

```
$ python3 -m http.server <portnumber>
```

Если номер порта не указан, то по умолчанию используется порт 8000. Поэтому вывод будет таким:

```
Serving HTTP on 0.0.0.0 port 8000 ... (Обслуживание HTTP на 0.0.0.0 порту 8000 ..._
```

Вы можете получить доступ к своим файлам через любое устройство, подключенное к локальной сети, набрав

```
http://hostipaddress:8000/.
```

hostipaddress – это ваш локальный IP-адрес, который, вероятно, начинается с 192.168.x.x. Чтобы завершить работу с модулем, просто нажмите ctrl+c.

124.2. Создание TCP-сервера

Вы можете создать TCP-сервер, используя библиотеку `socketserver`. Приведем пример простого эхо-сервера. Сторона сервера:

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('соединение от:', self.client_address)
        while True:
            msg = self.request.recv(8192)
            if not msg:
                break
            self.request.send(msg)

if __name__ == '__main__':
    server = TCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

Клиентская сторона:

```
from socket import socket, AF_INET, SOCK_STREAM
sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('localhost', 5000))
sock.send(b'Monty Python')
sock.recv(8192) # возвращает b'Monty Python'
```

`socketserver` позволяет сравнительно легко создавать простые TCP-серверы. Однако следует иметь в виду, что по умолчанию серверы являются однопоточными и могут обслуживать только одного клиента одновременно. Если необходимо обслуживать несколько клиентов, то вместо этого следует инстанцировать `ThreadingTCPServer`.

```
from socketserver import ThreadingTCPServer

if __name__ == '__main__':
    server = ThreadingTCPServer(('', 5000), EchoHandler)
    server.serve_forever()
```

124.3. Создание UDP-сервера

UDP-сервер легко создается с помощью библиотеки `socketserver`:

```
import time
from socketserver import BaseRequestHandler, UDPServer

class CtimeHandler(BaseRequestHandler):
    def handle(self):
        print('connection from: ', self.client_address)
        # Получить сообщение и клиентский сокет
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    server = UDPServer(('', 5000), CtimeHandler)
    server.serve_forever()
```

Проверка:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> sock = socket(AF_INET, SOCK_DGRAM)
```

```
>>> sick.sendto(b", ('localhost', 5000))
0
>>> sock.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 5000))
```

124.4. Запуск простого HttpServer в потоке и открытие браузера

Полезно, если ваша программа выводит веб-страницы по ходу работы:

```
from http.server import HTTPServer, CGIHTTPRequestHandler
import webbrowser
import threading

def start_server(path, port=8000):
    """Запуск простого веб-сервера, обслуживающего путь на порту"""
    os.chdir(path)
    httpd = HTTPServer(("", port), CGIHTTPRequestHandler)
    httpd.serve_forever()

# Запуск сервера в новом потоке
port = 8000
daemon = threading.Thread(name='daemon_server',
                           target=start_server,
                           args=('.', port))
daemon.setDaemon(True) # Устанавливается в качестве демона, поэтому он будет завершен,
# как только завершится основной поток.
daemon.start()

# Открыть браузер
webbrowser.open('http://localhost:{}'.format(port))
```

124.5. Простейший пример клиент-сервер сокетов на Python

Сторона сервера:

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind(('localhost', 8089))
serversocket.listen(5) # стать серверным сокетом, максимум 5 соединений

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print(buf)
    break
```

Сторона клиента:

```
import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')
```

Глава 125. Python HTTP Server

125.1. Запуск простого HTTP-сервера

Версия Python 2.x ≥ 2.3:

```
python -m SimpleHTTPServer 9000
```

Версия Python 3.x ≥ 3.0:

```
python -m http.server 9000
```

Выполнение этой команды обслуживает файлы текущего каталога на порту 9000. Если в качестве аргумента не указан номер порта, то сервер будет работать на порту 8000 по умолчанию. Флаг `-m` будет искать в `sys.path` соответствующий файл `.py` для запуска в качестве модуля. Если вы хотите обслуживать только `localhost`, то вам необходимо написать собственную программу на языке Python, например такую:

```
import sys
import BaseHTTPServer
from SimpleHTTPServer import SimpleHTTPRequestHandler
```

```
HandlerClass = SimpleHTTPRequestHandler
ServerClass = BaseHTTPServer.HTTPServer
Protocol      = "HTTP/1.0"
```

```
if sys.argv[1:]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ('127.0.0.1', port)
```

```
HandlerClass.protocol_version = Protocol
httpd = ServerClass(server_address, HandlerClass)
```

```
sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()
```

125.2. Обслуживание файлов

Предположим, что у вас есть следующий каталог файлов:



Настроить веб-сервер для обслуживания этих файлов можно следующим образом:

Версия Python 2.x ≥ 2.3:

```
import SimpleHTTPServer
import SocketServer
```

```
PORT = 8000
```

```
handler = SimpleHTTPServer.SimpleHTTPRequestHandler
httpd = SocketServer.TCPServer(("localhost", PORT), handler)
```

```
print "Serving files at port {}".format(PORT)
httpd.serve_forever()
```

Версия Python 3.x ≥ 3.0:

```
import http.server
import socketserver

PORT = 8000

handler = http.server.SimpleHTTPRequestHandler
httpd = socketserver.TCPServer(("", PORT), handler)
print("serving at port", PORT)
httpd.serve_forever()
```

Модуль `SocketServer` предоставляет классы и функциональные возможности для создания сетевого сервера. Класс `TCPServer`, входящий в состав `SocketServer`, устанавливает сервер, использующий протокол TCP. Конструктор принимает кортеж, представляющий адрес сервера (т.е. IP-адрес и порт) и класс, обрабатывающий запросы к серверу. Класс `SimpleHTTPRequestHandler` модуля `SimpleHTTPServer` позволяет обслуживать файлы, находящиеся в текущем каталоге.

Сохраните скрипт в том же каталоге и запустите его. Запустите HTTP-сервер:

Версия Python 2.x ≥ 2.3:

```
python -m SimpleHTTPServer 8000
```

Версия Python 3.x ≥ 3.0:

```
python -m http.server 8000
```

Флаг `-m` будет искать в `sys.path` соответствующий файл `.py` для запуска в качестве модуля. Откройте `localhost:8000` в браузере, и он выдаст следующее:

Directory listing for /

- [facade.py](#)
- [factory.py](#)
- [server.py](#)

125.3. Базовая обработка GET, POST, PUT с помощью BaseHTTPRequestHandler

```
# from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer # python2
from http.server import BaseHTTPRequestHandler, HTTPServer # python3
class HandleRequests(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("received get request")

    def do_POST(self):
        """Reads post request body"""
        self._set_headers()
```

```

content_len = int(self.headers.getheader('content-length', 0))
post_body = self.rfile.read(content_len)
self.wfile.write("received post request:<br>{}".format(post_body))

```

```

def do_PUT(self):
    self.do_POST()

```

```

host = "
port = 80
HTTPServer((host, port), HandleRequests).serve_forever()

```

Пример вывода с помощью cURL:

```

$ curl http://localhost/
received get request%

```

```

$ curl -X POST http://localhost/
received post request:<br>%

```

```

$ curl -X PUT http://localhost/
received post request:<br>%

```

```

$ echo 'hello world' | curl -data-binary @- http://localhost/
received post request:<br>hello world

```

125.4. Программное API SimpleHTTPServer

Что произойдет, если мы выполним команду `Python -m SimpleHTTPServer 9000`? Для ответа на этот вопрос необходимо разобраться в конструкции `SimpleHTTPServer` и `BaseHTTPServer`. Сначала Python вызывает модуль `SimpleHTTPServer` с 9000 в качестве аргумента. Рассмотрим код `SimpleHTTPServer`:

```

def test(HandlerClass = SimpleHTTPRequestHandler,
        ServerClass = BaseHTTPServer.HTTPServer):
    BaseHTTPServer.test(HandlerClass, ServerClass)
if __name__ == '__main__':
    test()

```

Функция `test` вызывается вслед за обработчиками запросов и классом `ServerClass`. Теперь вызывается `BaseHTTPServer.test`:

```

def test(HandlerClass = BaseHTTPRequestHandler,
        ServerClass = HTTPServer, protocol="HTTP/1.0"):
    """Тестирование класса обработчика HTTP-запросов.

```

Запускается HTTP-сервер на порту 8000 (или на первом аргументе командной строки).

```


```

```

if sys.argv[1]:
    port = int(sys.argv[1])
else:
    port = 8000
server_address = ("", port)

```

```

HandlerClass.protocol_version = protocol
httpd = ServerClass(server_address, HandlerClass)

```

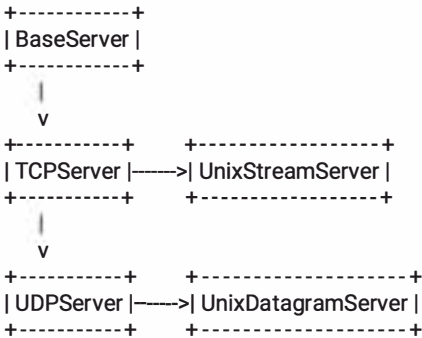
```

sa = httpd.socket.getsockname()
print "Serving HTTP on", sa[0], "port", sa[1], "..."
httpd.serve_forever()

```

Поэтому здесь номер порта, переданный пользователем в качестве аргумента, разбирается (парсится) и привязывается к адресу хоста. Далее выполняются основные шаги программирования сокетов с заданным портом и протоколом. В завершение запускается сервер сокетов.

Это базовый обзор наследования от класса `SocketServer` к другим классам:



Глава 126. Микровебфреймворк Flask

Flask – это микровебфреймворк на языке Python, используемый для запуска таких крупных сайтов, как Pinterest, Twilio и LinkedIn. В данной теме рассказывается и демонстрируется разнообразие возможностей Flask как для фронт-, так и для бэк-энд веб-разработки.

126.1. Файлы и шаблоны

Вместо того чтобы вводить нашу HTML-разметку в операторы возврата, мы можем использовать функцию `render_template()`:

```

from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route("/about")
def about():
    return render_template("about-us.html")

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)

```

При этом будет использоваться наш файл шаблона `about-us.html`. Для того чтобы наше приложение могло найти этот шаблон, мы должны организовать наш каталог в следующем формате:

```

- application.py
/templates
- about-us.html
- login-form.html
/static
/styles
- about-style.css
- login-style.css
/scripts
- about-script.js
- login-script.js

```

Самое главное, что ссылки на эти файлы в HTML должны выглядеть следующим образом:

```
<link rel="stylesheet" type="text/css", href="{{url_for('static', filename='styles/about-style.css')}}">.
```

который направит приложение на поиск файла `about-style.css` в папке `styles`, расположенной в папке `static`. Такой же формат пути применяется ко всем ссылкам на изображения, стили, скрипты или файлы.

126.2. Основы

Ниже приведен пример базового сервера:

```
# Импортирует класс Flask
from flask import Flask
# Создает приложение и проверяет, является ли оно основным или импортированным
app = Flask(__name__)

# Указывает, какой URL запускает hello_world()
@app.route('/')
# Функция, выполняемая на маршруте индекса
def hello_world():
    # Возвращает текст, который будет выведен на экран
    return "Hello World!"

# Если этот скрипт не является импортом
if __name__ == "__main__":
    # Запустить приложение до тех пор, пока оно не будет остановлено
    app.run()
```

Запуск этого скрипта (с установкой всех необходимых зависимостей) должен запустить локальный сервер. В качестве хоста используется адрес `127.0.0.1`, известный как **localhost**. По умолчанию этот сервер работает на порту **5000**. Чтобы получить доступ к веб-серверу, откройте браузер и введите URL `localhost:5000` или `127.0.0.1:5000` (разницы нет). В настоящее время доступ к веб-серверу может получить только ваш компьютер.

`app.run()` имеет три параметра: **host**, **port** и **debug**. По умолчанию хост равен `127.0.0.1`, но установка его в значение `0.0.0.0` сделает ваш веб-сервер доступным с любого устройства в вашей сети, используя ваш частный IP-адрес в URL. Порт по умолчанию равен `5000`, но если параметр установлен в значение `80`, то пользователям не нужно будет указывать номер порта, так как браузеры используют порт `80` по умолчанию. Что касается опции отладки, то в процессе разработки (никогда в производстве) этот параметр лучше установить в значение `True`, поскольку при внесении изменений в проект Flask сервер будет перезапускаться.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80, debug=True)
```

126.3. Маршрутизация URL-адресов

В Flask маршрутизация URL традиционно выполняется с помощью декораторов. Эти декораторы могут использоваться как для статической маршрутизации, так и для маршрутизации URL с параметрами. В следующем примере представим, что этот Flask-скрипт запускает сайт `www.example.com`.

```
@app.route("/")
def index():
    return "Вы перешли на сайт www.example.com"

@app.route("/about")
def about():
    return "Вы перешли на сайт www.example.com/about"

@app.route("/users/guido-van-rossum")
return "Вы перешли на сайт to www.example.com/guido-van-rossum"
```


Последний маршрут показывает, что при наличии URL-адреса с /users/ и именем профиля мы можем вернуть профиль. Поскольку включать @app.route() для каждого пользователя было бы нерационально и беспорядочно, Flask предлагает принимать параметры из URL:

```
@app.route("/users/<username>")
def profile(username):
    return "Добро пожаловать в профиль " + username

cities = ["ОМАНА", "MELBOURNE", "NEPAL", "STUTTGART", "LIMA", "CAIRO", "SHANGHAI"]

@app.route("/stores/locations/<city>")
def storefronts(city):
    if city in cities:
        return "Да! Мы находимся в " + city
    else:
        return "Нет. Мы не находимся в " + city
```

126.4. HTTP-методы

Два наиболее распространенных HTTP-метода – **GET** и **POST**. В зависимости от используемого метода HTTP Flask может запускать различный код с одного и того же URL. Например, в веб-сервисе с учетными записями удобнее всего направлять страницу регистрации и процесс регистрации через один и тот же URL. При этом GET-запрос, тот самый, который выполняется при открытии URL в браузере, должен отображать форму входа в систему, а POST-запрос (несущий данные для входа) должен обрабатываться отдельно. Также создается маршрут для обработки HTTP-методов DELETE и PUT.

```
@app.route("/login", methods=["GET"])
def login_form():
    return "Это форма логина"
@app.route("/login", methods=["POST"])
def login_auth():
    return "Обработка ваших данных"
@app.route("/login", methods=["DELETE", "PUT"])
def deny():
    return "Этот метод не разрешен"
```

Чтобы немного упростить код, мы можем импортировать пакет request из flask.

```
from flask import request
```

```
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "Этот метод не разрешен"
    elif request.method == "GET":
        return "Это форма логина"
    elif request.method == "POST":
        return "Этот метод не разрешен"
```

Для получения данных из POST-запроса мы должны использовать пакет request:

```
from flask import request
@app.route("/login", methods=["GET", "POST", "DELETE", "PUT"])
def login():
    if request.method == "DELETE" or request.method == "PUT":
        return "Этот метод не разрешен"
    elif request.method == "GET":
        return "Это форма логина"
    elif request.method == "POST":
        return "Имя пользователя было " + request.form["username"] + " и пароль был " + request.form["password"]
```

126.5. Jinja Templating

Подобно Meteor.js, Flask хорошо интегрируется с сервисами фронт-энд шаблонизации. По умолчанию Flask использует шаблоны Jinja Templating. Шаблоны позволяют использовать в HTML небольшие фрагменты кода, такие как условия или циклы.

Когда мы выводим шаблон, любые параметры, помимо имени шаблона, передаются в службу шаблонизации HTML. В следующем маршруте в HTML передаются имя пользователя и дата подключения (из другой функции).

```
@app.route("/users/<username>")
def profile(username):
    joinedDate = get_joined_date(username) # Код этой функции неактуален
    awards = get_awards(username) # Код этой функции неактуален
    # joinDate - строка, а awards - массив строк
    return render_template("profile.html", username=username, joinDate=joinDate, awards=awards)
```

При выводе этого шаблона он может использовать переменные, переданные ему из функции `render_template()`. Вот содержимое файла `profile.html`:

```
<!DOCTYPE html>
<html>
  <head>
    # if username
    <title>Profile of {{ username }}</title>
    # else
    <title>No User Found</title>
    # endif
  <head>
  <body>
    {% if username %}
    <h1>{{ username }} joined on the date {{ date }}</h1>
    {% if len(awards) > 0 %}
    <h3>{{ username }} has the following awards:</h3>
    <ul>
      {% for award in awards %}
      <li>{{ award }}</li>
    {% endfor %}
    </ul>
    {% else %}
    <h3>{{ username }} has no awards</h3>
    {% endif %}
    {% else %}
    <h1>No user was found under that username</h1>
    {% endif %}
    {# This is a comment and doesn't affect the output #}
  </body>
</html>
```

Используются следующие разделители:

- `{% ... %}` обозначает высказывание
- `{{ ... }}` обозначает выражение, в котором выводится шаблон
- `{# ... #}` обозначает комментарий (не включается в вывод шаблона)
- `{# ... ##}` подразумевает, что оставшаяся часть строки должна быть интерпретирована как утверждение

126.6. Объект запроса

Объект `request` предоставляет информацию о запросе, который был сделан к маршруту. Чтобы использовать этот объект, он должен быть импортирован из модуля `flask`:

```
from flask import request
```

Параметры URL

В предыдущих примерах использовались `request.method` и `request.form`, однако мы также можем использовать свойство `request.args` для получения словаря ключей/значений в параметрах URL.

```
@app.route("/api/users/<username>")
def user_api(username):
    try:
        token = request.args.get("key")
        if key == "pA55w0Rd":
            if isUser(username): # Код этого метода не имеет значения
                joined = joinDate(username) # Код этого метода не имеет значения
                return "User " + username + " joined on " + joined
            else:
                return "Пользователь не найден"
        else:
            return "Неверный ключ"
    # Если нет параметра key
    except KeyError:
        return "Ключ не предоставлен"
```

Для корректной аутентификации в этом контексте потребуется следующий URL (заменяя имя пользователя на любое имя пользователя):

`www.example.com/api/users/guido-van-rossum?key=pa55w0Rd`

Загрузка файлов

Если загрузка файлов была частью формы, отправленной в POST-запросе, то файлы могут быть обработаны с помощью объекта запроса:

```
@app.route("/upload", methods=["POST"])
def upload_file():
    f = request.files["wordlist-upload"]
    f.save("/var/www/uploads/" + f.filename) # Сохранить с исходным именем файла
```

Cookies

Запрос может также включать в себя файлы cookie в словаре, аналогичном параметрам URL.

```
@app.route("/home")
def home():
    try:
        username = request.cookies.get("username")
        return "Ваше сохраненное имя пользователя это " + username
    except KeyError:
        return "cookies имени пользователя не найдены")
```

Глава 127. Использование библиотеки AMQPStorm в брокере сообщений RabbitMQ

127.1. Как получать сообщения из RabbitMQ

Начните с импорта библиотеки.

```
from amqpstorm import Connection
```

При получении (consuming) сообщений сначала необходимо определить функцию для обработки входящих сообщений. Это может быть любая вызываемая функция, которая должна принимать объект сообщения или кортеж сообщений (в зависимости от параметра `to_tuple`, заданного в `start_consuming`).

Помимо обработки данных из входящего сообщения нам также необходимо подтвердить (Acknowledge) или отклонить (Reject) сообщение. Это очень важно, поскольку нам необходимо сообщить RabbitMQ, что мы правильно приняли и обработали сообщение.

```
def on_message(message):
    """Эта функция вызывается при получении сообщения.

    :param message: Доставленное сообщение.
    :return:
    """
    print("Сообщение:", message.body)

    # Подтвердить, что мы обработали сообщение без каких-либо проблем.
    message.ack()

    # Отклонить сообщение.
    # message.reject()

    # Отклонить сообщение и поместить его обратно в очередь.
    # message.reject(requeue=True)
```

Далее необходимо настроить соединение с сервером RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

После этого необходимо настроить канал. Каждое соединение может иметь несколько каналов, и в общем случае при выполнении многопоточных задач рекомендуется (но не обязательно) иметь один канал на поток.

```
channel = connection.channel()
```

После того как канал настроен, необходимо сообщить RabbitMQ о том, что мы хотим начать потреблять сообщения. В этом случае для обработки всех потребляемых сообщений мы будем использовать нашу ранее созданную функцию `on_message`. Очередь, которую мы будем прослушивать на сервере RabbitMQ – это `simple_queue`, и мы также сообщаем RabbitMQ, что будем подтверждать все входящие сообщения, как только закончим их обработку.

```
channel.basic.consume(callback=on_message, queue='simple_queue', no_ack=False)
```

Наконец, нам нужно запустить цикл ввода-вывода, чтобы начать обработку сообщений, доставленных сервером RabbitMQ.

```
channel.start_consuming(to_tuple=False)
```

127.2. Как публиковать сообщения в RabbitMQ

Начните с импорта библиотеки.

```
from amqpstorm import Connection
from amqpstorm import Message
```

Далее нам необходимо открыть соединение с сервером RabbitMQ.

```
connection = Connection('127.0.0.1', 'guest', 'guest')
```

После этого необходимо настроить канал. Каждое соединение может иметь несколько каналов, и в общем случае при выполнении многопоточных задач рекомендуется (но не обязательно) иметь один канал на поток.

```
channel = connection.channel()
```

После того как канал настроен, можно приступить к подготовке сообщения.

```
# Свойства сообщения
properties = {
    'content_type': 'text/plain',
    'headers': {'key': 'value'}
```

```
# Создать сообщение
message = Message.create(channel=channel, body='Hello World!', properties=properties)

Теперь мы можем опубликовать сообщение, просто вызвав publish и указав ключ маршрутизации (routing_key). В данном случае мы собираемся отправить сообщение в очередь с именем simple_queue.

message.publish(routing_key='simple_queue')
```

127.3. Как создать очередь с задержкой в RabbitMQ

Сначала нам необходимо настроить два основных канала, один для основной очереди, другой для очереди задержки. В этом примере в конце включено несколько дополнительных флагов, которые не являются обязательными, но делают код более надежным – это confirm delivery, delivery_mode и durable. Более подробную информацию о них можно найти в руководстве по RabbitMQ.

После настройки каналов мы добавляем привязку к основному каналу, которую можно использовать для отправки сообщений из канала задержки в нашу основную очередь.

```
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')

Далее нам необходимо настроить канал задержки для пересылки сообщений в основную очередь по истечении срока их хранения.

delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000,
    'x-dead-letter-exchange': 'amq.direct',
    'x-dead-letter-routing-key': 'hello'
})
```

- x-message-ttl (Message - Time To Live) (Сообщение - "Время жизни")

Обычно эта функция используется для автоматического удаления старых сообщений из очереди по истечении определенного времени, но добавив два необязательных аргумента, мы можем изменить это поведение, и вместо этого параметр будет определять в миллисекундах, сколько времени сообщения будут находиться в очереди задержки.

- x-dead-letter-routing-key

Эта переменная позволяет нам передавать сообщения в другую очередь по истечении срока их действия, вместо того чтобы по умолчанию удалять их полностью.

- x-dead-letter-exchange

Эта переменная определяет, через какой Exchange передавать сообщение из hello_delay в очередь hello.

Публикация в очередь задержки

Когда настройка всех основных параметров Pika закончена, вы просто отправляете сообщение в очередь задержки, используя базовую публикацию.

```
delay_channel.basic.publish(exchange="",
                            routing_key='hello_delay',
                            body='test',
                            properties={'delivery_mod': 2})
```

После выполнения скрипта в модуле управления RabbitMQ вы увидите следующие очереди.

Overview					Messages			Message rates			
Name	Exclusive	Parameters	Policy	Status	Ready	Unacked	Total	Incoming	deliver	/ get	ack
hello		D		Idle	0	0	0				
hello_delay		TTL DLX DLK D		Idle	0	0	0	0.00/s			

Пример

```
from amqpstorm import Connection
```

```
connection = Connection('127.0.0.1', 'guest', 'guest')
# Создать обычный канал типа "Hello World".
channel = connection.channel()
channel.confirm_deliveries()
channel.queue.declare(queue='hello', durable=True)
```

```
# Нужно привязать этот канал к обмену, который будет использоваться для передачи
# сообщений из нашей очереди задержки.
channel.queue.bind(exchange='amq.direct', routing_key='hello', queue='hello')
```

```
# Создаем наш канал задержки.
delay_channel = connection.channel()
delay_channel.confirm_deliveries()
```

```
# Объявляем задержку и маршрутизацию для нашего канала задержки.
delay_channel.queue.declare(queue='hello_delay', durable=True, arguments={
    'x-message-ttl': 5000, # Задержка до передачи сообщения в миллисекундах.
    'x-dead-letter-exchange': 'amq.direct', # Обмен, используемый для передачи сообщения из А в В.
    'x-dead-letter-routing-key': 'hello' # Имя очереди, в которую мы хотим передать сообщение.
})
```

```
delay_channel.basic.publish(exchange="",
                           routing_key='hello_delay',
                           body='test',
                           properties={'delivery_mode': 2})

print("[x] Sent")
```

Глава 128. Дескриптор

128.1. Простой дескриптор

Существует два различных типа дескрипторов. Дескрипторы данных определяются как объекты, которые имеют метод `__get__()` и `__set__()`, в то время как дескрипторы без данных имеют только метод `__get__()`. Это различие важно при рассмотрении переопределений и пространства имен словаря экземпляра. Если дескриптор данных и запись в словаре экземпляра имеют одинаковые имена, то приоритет будет отдан дескриптору данных. Однако если вместо этого дескриптор, не содержащий данных, и запись в словаре экземпляра имеют одинаковые имена, то приоритет будет иметь запись в словаре экземпляра.

Чтобы сделать дескриптор данных, доступный только для чтения, необходимо определить оба метода `get()` и `set()`, причем метод `set()` должен вызывать ошибку `AttributeError`. Для превращения в дескриптор данных достаточно дезактивировать метод `set()` с помощью заполнителя, вызывающего исключение.

```
descr.__get__(self, obj, type=None) --> value
descr.__set__(self, obj, value) --> None
descr.__delete__(self, obj) --> None
```

Реализованный пример:

```
class DescPrinter(object):
    """Дескриптор данных, регистрирующий активность."""
    _val = 7
```

```

def __get__(self, obj, objtype=None):
    print('Getting ...')
    return self._val

def __set__(self, obj, val):
    print('Setting', val)
    self._val = val

def __delete__(self, obj):
    print('Deleting ...')
    del self._val

class Foo():
    x = DescPrinter()

i = Foo()
i.x
# Получение ...
# 7

i.x = 100
# Установка 100
i.x
# Получение ...
# 100

del i.x
# Удаление ...
i.x
# Получение ...
# 7

```

128.2. Двусторонние преобразования

Объекты-дескрипторы могут позволить связанным с ними атрибутам объекта автоматически реагировать на изменения.

Предположим, что мы хотим смоделировать осциллятор с заданной частотой (в герцах) и периодом (в секундах). При обновлении частоты мы хотим, чтобы обновлялся период, а при обновлении периода – частота:

```

>>> oscillator = Oscillator(freq=100.0)      # Установить частоту 100,0 Гц
>>> oscillator.period                        # Период равен 1 / частота, т.е. 0,01 секунды
0.01
>>> oscillator.period = 0.02                # Установить период 0,02 секунды
>>> oscillator.freq                          # Частота перестраивается автоматически
50.0
>>> oscillator.freq = 200.0                  # Установить частоту 200,0 Гц
>>> oscillator.period                        # Период регулируется автоматически
0.005

```

Мы выбираем одно из значений (частота, в герцах) в качестве “якоря”, т.е. того, которое может быть задано без преобразования, и пишем для него класс-дескриптор:

```

class Hertz(object):
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = float(value)

```

Период, в секундах (“другое” значение), определяется в терминах “якоря”. Мы пишем класс дескриптора, который выполняет наши преобразования:

```
class Second(object):
    def __get__(self, instance, owner):
        # При чтении периода пересчитать из частоты
        return 1 / instance.freq

    def __set__(self, instance, value):
        # При установке периода обновить частоту
        instance.freq = 1 / float(value)
```

Теперь мы можем написать класс Oscillator:

```
class Oscillator(object):
    period = Second() # Установить другое значение в качестве атрибута класса

    def __init__(self, freq):
        self.freq = Hertz() # Установить значение якоря как атрибут экземпляра
        self.freq = freq # Присвоить переданное значение - self.period будет скорректирован
```

Глава 129. Временный файл NamedTemporaryFile

Параметр	Описание
mode	режим открытия файла, по умолчанию=w+b
delete	удалить файл при закрытии, по умолчанию=True
suffix	суффикс имени файла, по умолчанию=""
prefix	префикс имени файла, по умолчанию 'tmp'
dir	каталог размещения временного файла, по умолчанию =None
buffsize	по умолчанию =-1, (используется значение ОС по умолчанию)

129.1. Создание известного постоянного временного файла и запись в него

Вы можете создавать временные файлы, которые имеют видимое имя в файловой системе, доступ к которому можно получить через свойство name. В Unix-системах такой файл может быть настроен на удаление при закрытии (задается параметром delete, по умолчанию – True) или может быть открыт позже.

В следующем примере создается и открывается именованный временный файл и в него записывается ‘Hello World!’. Доступ к пути временного file можно получить через name, в данном примере он сохраняется в переменной path и выводится для пользователя. После закрытия файл снова открывается, содержимое временного файла считывается и выводится для пользователя.

```
import tempfile

with tempfile.NamedTemporaryFile(delete=False) as t:
    t.write('Hello World!')
```



```
path = t.name
print path
```

```
with open(path) as t:
    print t.read()
```

Вывод:

```
/tmp/tmp6pireJ
Hello World!
```

Глава 130. Ввод, подмножество и вывод внешних файлов данных с помощью библиотеки Pandas

В этом разделе демонстрируется базовый код для чтения, подстановки и записи внешних данных с помощью библиотеки Pandas.

130.1. Базовый код для импорта, подмножества и записи внешних файлов данных с помощью библиотеки Pandas

```
# Вывести рабочий каталог
```

```
import os
print os.getcwd()
# C:\Python27\Scripts
```

```
# Установить рабочий каталог
```

```
os.chdir('C:/Users/general1/Documents/simple Python files')
print os.getcwd()
# C:\Users\general1\Documents\simple Python files
```

```
# загрузить pandas
```

```
import pandas as pd
```

```
# прочитать файл данных csv с именем 'small_dataset.csv', содержащий 4 строки и 3 переменные
my_data = pd.read_csv("small_dataset.csv")
```

```
my_data
```

```
#   x  y  z
# 0  1  2  3
# 1  4  5  6
# 2  7  8  9
# 3 10 11 12
```

```
my_data.shape    # количество строк и столбцов в наборе данных
# (4, 3)
```

```
my_data.shape[0] # количество строк в наборе данных
# 4
```

```
my_data.shape[1] # количество столбцов в наборе данных
# 3
```

В Python используется 0-ориентированная индексация. Первая строка или столбец
в наборе данных располагается # в позиции 0. В R первая строка или столбец в наборе
данных располагается # в позиции 1.

Выбрать первые две строки

```
my_data[0:2]
```

```
# x y z
```

```
#0 1 2 3
```

```
#1 4 5 6
```

Выбрать вторую и третью строки

```
my_data[1:3]
```

```
# x y z
```

```
#1 4 5 6
```

```
#2 7 8 9
```

Выберите третью строку

```
my_data[2:3]
```

```
# x y z
```

```
#2 7 8 9
```

Выбрать первые два элемента первого столбца

```
my_data.iloc[0:2, 0:1]
```

```
# x
```

```
#0 1
```

```
#1 4
```

Выбор первого элемента переменных y и z

```
my_data.loc[0, ['y', 'z']]
```

```
# y 2
```

```
# z 3
```

Выделить первые три элемента переменных y и z

```
my_data.loc[0:2, ['y', 'z']]
```

```
# y z
```

```
#0 2 3
```

```
#1 5 6
```

```
#2 8 9
```

Записать первые три элемента переменных y и z

во внешний файл. Здесь index = 0 означает, что имена строк не записываются.

```
my_data2 = my_data.loc[0:2, ['y', 'z']]
```

```
my_data2.to_csv('my.output.csv', index = 0)
```

Глава 131. Распаковка файлов

Для извлечения или распаковки архивов tarball, ZIP или gzip в Python предоставляются модули tarfile, zipfile и gzip соответственно. Модуль tarfile в Python предоставляет функцию TarFile.extractall(path=".", members=None) для извлечения из tar-файла. Модуль Python zipfile предоставляет функцию ZipFile.extractall([path[, members[, pwd]]) для разархивирования ZIP-файлов. Наконец, модуль gzip в Python предоставляет для распаковки соответствующего формата класс GzipFile.

131.1. Использование Python ZipFile.extractall() для распаковки ZIP-файла

```
file_unzip = 'filename.zip'
unzip = zipfile.ZipFile(file_unzip, 'r')
unzip.extractall()
unzip.close()
```

131.2. Использование Python TarFile.extractall() для распаковки tarball-архива

```
file_untar = 'filename.tar.gz'
untar = tarfile.TarFile(file_untar)
untar.extractall()
untar.close()
```

Глава 132. Работа с ZIP-архивами

132.1. Изучение содержимого ZIP-файла

Существует несколько способов проверки содержимого zip-файла. Можно использовать `printdir`, чтобы просто получить разнообразную информацию, отправляемую в `stdout`:

```
with zipfile.ZipFile(filename) as zip:
    zip.printdir()
# Вывод:
# File Name          Modified              Size
# pyexpat.pyd         2016-06-25 22:13:34  157336
# python.exe          2016-06-25 22:13:34   39576
# python3.dll         2016-06-25 22:13:34   51864
# python35.dll        2016-06-25 22:13:34  3127960
```

Мы также можем получить список имен с помощью метода `namelist`. Здесь мы просто выводим список:

```
with zipfile.ZipFile(filename) as zip:
    print(zip.namelist())
# Вывод: ['pyexpat.pyd', 'python.exe', 'python3.dll', 'python35.dll', ... etc. ...]
```

Вместо `namelist` можно вызвать метод `infolist`, который возвращает список объектов `ZipInfo`, содержащих дополнительную информацию о каждом файле, например метку времени и размер:

```
with zipfile.ZipFile(filename) as zip:
    info = zip.infolist()
    print(zip[0].filename)
    print(zip[0].date_time)
    print(info[0].file_size)
# Вывод: pyexpat.pyd
# Вывод: (2016, 6, 25, 22, 13, 34)
# Вывод: 157336
```

132.2. Открытие ZIP-файлов

Для начала импортируем модуль `zipfile` и задаем имя файла.

```
import zipfile
filename = 'zipfile.zip'
```

Работа с zip-архивами очень похожа на работу с file, вы создаете объект, открывая zip-архив, который позволяет вам работать с ним, прежде чем снова закрыть его.

```
zip = zipfile.ZipFile(filename)
print(zip)
# <zipfile.ZipFile object at 0x000000002E51A90>
zip.close()
```

В Python 2.7 и в Python 3 версий выше 3.2 можно использовать менеджер контекста with. Открываем файл в режиме чтения, а затем выводим список имен файлов:

```
with zipfile.ZipFile(filename, 'r') as z:
    print(z)
# <zipfile.ZipFile object at 0x000000002E51A90>
```

132.3. Извлечение содержимого ZIP-файла в каталог

Извлечение всего содержимого zip-файла:

```
import zipfile
with zipfile.ZipFile('zipfile.zip','r') as zfile:
    zfile.extractall('path')
```

Если необходимо извлечь отдельные элементы, используйте метод extract, который принимает в качестве входного параметра список имен и путь:

```
import zipfile
f=open('zipfile.zip','rb')
zfile=zipfile.ZipFile(f)
for cont in zfile.namelist():
    zfile.extract(cont,path)
```

132.4. Создание новых архивов

Для создания нового архива откройте zip-файл в режиме записи.

```
import zipfile
new_arch=zipfile.ZipFile("filename.zip",mode="w")
```

Для добавления элементов в этот архив используйте метод write().

```
new_arch.write('filename.txt','filename_in_archive.txt') # первый параметр - имя файла,
# второй - имя файла в архиве, по умолчанию будет взято имя файла, если оно не указано
new_arch.close()
```

Если необходимо записать в архив строку байтов, то можно воспользоваться методом writestr().

```
str_bytes="string buffer"
new_arch.writestr('filename_string_in_archive.txt',str_bytes)
new_arch.close()
```

Глава 133. Работа с GZip

Этот модуль предоставляет простой интерфейс для сжатия и распаковки файлов подобно тому, как это делают GNU-программы gzip и gunzip. Сжатие данных обеспечивается модулем zlib. Модуль gzip предоставляет класс GzipFile, который создан по образцу объекта файла в Python. Класс GzipFile читает и записывает файлы в формате gzip, автоматически сжимая или распаковывая данные так, чтобы они выглядели как обычный объект файла.

133.1. Чтение и запись файлов GNU zip

```
import gzip
import os

outfilename = 'example.txt.gz'
output = gzip.open(outfilename, 'wb')
try:
    output.write('Contents of the example file go here.\n')
finally:
    output.close()

print outfilename, 'contains', os.stat(outfilename).st_size, 'bytes of compressed data'
os.system('file -b --mime %s' % outfilename)
```

Сохраните его под именем 1gzip_write.py1 и запустите через терминал:

```
$ python gzip_write.py
application/x-gzip; charset=binary
example.txt.gz contains 68 bytes of compressed data
```

Глава 134. Стек

Стек – это контейнер с объектами, которые чаще всего вставляются и вынимаются по принципу “последним пришел – первым ушел” (LIFO, last in first out). В pushdown-стеках разрешены только две операции: заталкивание, или проталкивание (**push**) элемента в стек и выталкивание, или удаление (pop) элемента из стека. Стек является структурой данных с ограниченным доступом – **добавлять и удалять элементы из стека можно только сверху**. Вот структурное определение стека: стек либо пуст, либо состоит из вершины и остальной части, которая является стеком.

134.1. Создание класса Stack с объектом List

Используя объект list, можно создать полнофункциональный общий стек с такими вспомогательными методами, как чтение головного элемента (peeking) и проверка пустоты стека.

```
#определение класса стека
class Stack:
    def __init__(self):
        self.items = []

    #метод для проверки того, пуст стек или нет
    def isEmpty(self):
        return self.items == []

    #метод для проталкивания элемента
    def push(self, item):
        self.items.append(item)

    #метод для выталкивания элемента
    def pop(self):
        return self.items.pop()

    #проверить, какой элемент находится на вершине стека, не удаляя его
    def peek(self):
        return self.items[-1]
```

```
#метод для получения размера
def size(self):
    return len(self.items)
```

```
#для просмотра всего стека
def fullStack(self):
    return self.items
```

Пример:

```
stack = Stack()
print('Текущий стек:', stack.fullStack())
print('Стек пуст?:', stack.isEmpty())
print('Проталкиваем integer 1')
stack.push(1)
print('Проталкиваем строку "Я обычный стек!"')
stack.push('Я обычный стек!')
print('Проталкиваем integer 3')
stack.push(3)
print('Текущий стек:', stack.fullStack())
print('Вытолкнутый элемент:', stack.pop())
print('Текущий стек:', stack.fullStack())
print('Стек пуст?:', stack.isEmpty())
```

Выход:

```
Текущий стек: []
Стек пуст?: True
Проталкиваем integer 1
Проталкиваем string "Я обычный стек!"
Проталкиваем integer 3
Текущий стек: [1, 'Я обычный стек!', 3]
Вытолкнутый элемент: 3
Текущий стек: [1, 'Я обычный стек!']
Стек пуст?: False
```

134.2. Разбор (парсинг) круглых скобок

Стеки часто используются для синтаксического анализа (парсинга). Простой задачей синтаксического анализа является проверка соответствия строки круглых скобок.

Например, строка (()) является совпадающей, так как внешние и внутренние скобки образуют пары. (<>) не совпадает, так как последняя круглая скобка не имеет партнера. ([]) также не совпадает, так как пары должны быть либо полностью внутри, либо снаружи других пар.

```
def checkParenth(str):
    stack = Stack()
    pushChars, popChars = "<{([", ">)}]"
    for c in str:
        if c in pushChars:
            stack.push(c)
        elif c in popChars:
            if stack.isEmpty():
                return False
            else:
                stackTop = stack.pop()
                # Проверяет, совпадает ли открывающая скобка с закрывающей.
                balancingBracket = pushChars[popChars.index(c)]
                if stackTop != balancingBracket:
                    return False
        else:
            return False

    return not stack.isEmpty()
```

Глава 135. Работа в обход глобальной блокировки интерпретатора (GIL)

135.1. Multiprocessing.Pool

Когда возникает вопрос, как использовать потоки в Python, чаще всего звучит совет: “Не надо с ними связываться. Вместо этого используйте процессы”. Модуль мультипроцессинга позволяет создавать процессы с помощью синтаксиса, аналогичного созданию потоков, но лучше использовать их удобный объект `Pool`.

Используя код, который впервые применил Дэвид Бизли, чтобы показать опасность использования потоков в обход GIL, перепишем его с использованием `multiprocessing.Pool`:

Вот код Дэвида Бизли, демонстрирующий проблемы с потоками и GIL:

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

t1 = Thread(target=countdown,args=(COUNT/2,))
t2 = Thread(target=countdown,args=(COUNT/2,))
start = time.time()
t1.start();t2.start()
t1.join();t2.join()
end = time.time()
print end-start
```

Перепишем его с использованием `multiprocessing.Pool`:

```
import multiprocessing
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

start = time.time()
with multiprocessing.Pool as pool:
    pool.map(countdown, [COUNT/2, COUNT/2])

    pool.close()
    pool.join()

end = time.time()
print(end-start)
```

Вместо создания потоков здесь создаются новые процессы. Поскольку каждый процесс является собственным интерпретатором, проблем с GIL не возникает. `multiprocessing.Pool` откроет столько процессов, сколько ядер имеется на машине, хотя в приведенном примере ему потребуется только два. В реальных условиях необходимо, чтобы список имел длину не меньше, чем количество процессоров на машине. `Pool` будет запускать функцию, которую вы укажете ему выполнить с каждым аргументом, вплоть до количества создаваемых процессов. После завершения работы функции все оставшиеся в списке функции будут запущены на этом процессе. Даже при использовании оператора `with`, если не закрыть и не присоединить пул, процессы продолжат существовать. Для очистки ресурсов следует всегда закрывать и присоединять пулы.

135.2. Использование Cython nogil

Cython – это альтернативный интерпретатор Python. Он использует GIL, но позволяет отключить его. В качестве примера, используя код, который впервые использовал Дэвид Бизли, чтобы показать опасность использования потоков в обход GIL (см. предыдущую главу), перепишем его, используя ключевое слово `nogil` в Cython:

```
from threading import Thread
import time
def countdown(n):
    while n > 0:
        n -= 1

COUNT = 10000000

with nogil:
    t1 = Thread(target=countdown,args=(COUNT/2,))
    t2 = Thread(target=countdown,args=(COUNT/2,))
    start = time.time()
    t1.start();t2.start()
    t1.join();t2.join()

end = time.time()
print end-start
```

Все очень просто, если использовать Cython. Обратите внимание: в документации сказано, что необходимо следить за тем, чтобы не изменять никакие объекты Python:

Код в теле утверждения не должен манипулировать объектами Python каким-либо образом и не должен вызывать ничего, что манипулирует объектами Python без предварительного повторного обращения к GIL. Cython не проверяет это.

Глава 136. Развертывание

136.1. Загрузка пакета Conda

Перед началом работы необходимо, чтобы в вашей системе был установлен кроссплатформенный дистрибутив Anaconda на системном аккаунте на Binstar. Если вы не используете Anaconda 1.6+, установите клиент командной строки `binstar`:

```
$ conda install binstar
$ conda update binstar
```

Если вы не используете Anaconda, то Binstar также доступен на `rupi`:

```
$ pip install binstar
```

Теперь мы можем войти в систему:

```
$ binstar login
```

Протестируйте свой вход в систему с помощью команды `whoami`:

```
$ binstar whoami
```

Будем загружать пакет с простой функцией 'hello world'. Чтобы проследить за этим, начнем с получения демонстрационного пакета с Github:

```
$ git clone https://github.com/<NAME>/<Package>
```

Это небольшой каталог, который выглядит следующим образом:

```
package/
  setup.py
```



```
test_package/
  __init__.py
  hello.py
  bld.bat
  build.sh
  meta.yaml
```

Setup.py – это стандартная сборка Python, а hello.py содержит нашу единственную функцию hello_world().

bld.bat, build.sh и meta.yaml – это скрипты и метаданные для пакета Conda. Более подробную информацию об этих трех файлах и их назначении можно найти на странице сборки Conda.

Теперь создадим пакет, выполнив команду:

```
$ conda build test_package/
```

Это все, что требуется для создания пакета Conda. Последний шаг – загрузка в binstar путем копирования и вставки последней строки вывода после выполнения команды build test_package/ в Conda. Это может выглядеть следующим образом:

```
$ binstar upload /home/xavier/anaconda/conda-bld/linux-64/test_package-0.1.0-py27_0.tar.bz2
```

Поскольку вы создаете пакет и релиз впервые, вам будет предложено заполнить несколько текстовых полей, что можно сделать и через веб-приложение. На экране появится сообщение об успешной загрузке вашего пакета Conda в Binstar.

Глава 137. Модуль logging

137.1. Введение в использование модуля logging

Функции и классы этого модуля реализуют гибкую систему регистрации событий для приложений и библиотек. Все модули Python могут участвовать в протоколировании, поэтому журнал вашего приложения (application log) может включать ваши собственные сообщения, интегрированные с сообщениями сторонних модулей. Итак, начнем:

Пример конфигурации непосредственно в коде:

```
import logging

logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
    '%(asctime)s %(name)-12s %(levelname)-8s %(message)s')
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.debug('this is a %s test', 'debug')
```

Пример вывода:

```
2016-07-26 18:53:55,332 root    DEBUG  this is a debug test
```

Пример конфигурации через файл INI:

Предположим, что файл имеет имя logging_config.ini.

```
[loggers]
keys=root

[handlers]
keys=stream_handler
```

```
[formatters]
keys=formatter
```

```
[logger_root]
level=DEBUG
handlers=stream_handler
```

```
[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)
```

```
[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

Затем используйте `logging.config.fileConfig()` в коде:

```
import logging
from logging.config import fileConfig

fileConfig('logging_config.ini')
logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

Пример конфигурирования с использованием словаря

Начиная с версии Python 2.7 можно использовать словарь с деталями конфигурации. PEP 391 содержит список обязательных и необязательных элементов конфигурационного словаря.

```
import logging
from logging.config import dictConfig

logging_config = dict(
    version = 1,
    formatters = {
        'f': {'format':
            '%(asctime)s %(name)-12s %(levelname)-8s %(message)s'}
    },
    handlers = {
        'h': {'class': 'logging.StreamHandler',
            'formatter': 'f',
            'level': logging.DEBUG}
    },
    root = {
        'handlers': ['h'],
        'level': logging.DEBUG,
    },
)

dictConfig(logging_config)

logger = logging.getLogger()
logger.debug('often makes a very good meal of %s', 'visiting tourists')
```

137.2. Протоколирование исключений

Если вы хотите регистрировать исключения, то можно и нужно использовать метод `logging.exception(msg)`:

```
>>> import logging
>>> logging.basicConfig()
```

```
>>> try:
...     raise Exception('foo')
... except:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Не передавайте исключение в качестве аргумента:

Так как `logging.exception(msg)` ожидает аргумент `msg`, распространенной ошибкой является передача исключения в вызов протоколирования в таком виде:

```
>>> try:
...     raise Exception('foo')
... except Exception as e:
...     logging.exception(e)
...
ERROR:root:foo
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: foo
```

Хотя на первый взгляд может показаться, что это правильно, на самом деле это проблематично из-за того, как исключения и различные кодировки работают вместе в модуле протоколирования:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception(e)
...
Traceback (most recent call last):
  File ".../python2.7/logging/__init__.py", line 861, in emit
    msg = self.format(record)
  File ".../python2.7/logging/__init__.py", line 734, in format
    return fmt.format(record)
  File ".../python2.7/logging/__init__.py", line 469, in format
    s = self._fmt % record.__dict__
UnicodeEncodeError: 'ascii' codec can't encode characters in position 1-2: ordinal not in range(128)
Logged from file <stdin>, line 4
```

Попытка зарегистрировать исключение, содержащее символы Unicode, таким способом будет безуспешной. Он скроет стек-трассировку исходного исключения, переопределив его на новое, которое будет вызвано при форматировании вашего вызова `logging.exception(e)`.

Очевидно, что в собственном коде вы можете знать о кодировке в исключениях. Однако в библиотеках сторонних разработчиков этот вопрос может решаться по-другому.

Правильное использование:

Если вместо исключения передать сообщение и позволить Python самому “сделать свое волшебство”, то все будет работать:

```
>>> try:
...     raise Exception(u'föö')
... except Exception as e:
...     logging.exception('bar')
...
ERROR:root:bar
Traceback (most recent call last):
```

```
File "<stdin>", line 2, in <module>
Exception: f\xfb\xfb
```

Как видно, в этом случае мы не используем `e`, а вызов `logging.exception(...)` магическим образом форматирует последнее исключение.

Регистрация исключений с уровнями журнала, отличными от ERROR

Если вы хотите регистрировать исключение не на уровне ERROR, а на другом уровне, можно использовать аргумент `exc_info` регистраторов (логгеров) по умолчанию:

```
logging.debug('exception occurred', exc_info=1)
logging.info('exception occurred', exc_info=1)
logging.warning('exception occurred', exc_info=1)
```

Доступ к сообщению об исключении

Следует помнить, что библиотеки могут выдавать исключения с сообщениями в виде любой из форм представления Unicode или байт-строки (utf-8, если повезет). Если вам действительно нужно получить доступ к тексту исключения, то единственный надежный способ, который всегда будет работать, – это использовать `repr(e)` или форматирование строки `%r`:

```
>>> try:
...     raise Exception(u'föo')
... except Exception as e:
...     logging.exception('received this exception: %r' % e)
ERROR:root:received this exception: Exception(u'f\xfb\xfb')
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: f\xfb\xfb
```

Глава 138. Стандарт Web Server Gateway Interface (WSGI)

Параметр	Описание
<code>start_response</code>	Функция, используемая для обработки запуска

138.1. Объект (метод) сервера

Нашему серверному объекту передается параметр `'application'`, который может быть любым вызываемым объектом приложения. Он записывает сначала заголовки, а затем тело данных, возвращаемых нашим приложением, в стандартный вывод системы.

```
import os, sys
```

```
def run(application):
    environ['wsgi.input'] = sys.stdin
    environ['wsgi.errors'] = sys.stderr
```

```
    headers_set = []
    headers_sent = []
```

```
    def write(data):
```

```
        """
        Записывает данные заголовка из 'start_response()', а также данные тела из 'response'
        в стандартный вывод системы.
        """
```

```

"""
if not headers_set:
    raise AssertionError("write() before start_response()")

elif not headers_sent:
    status, response_headers = headers_sent[:] = headers_set
    sys.stdout.write('Status: %s\r\n' % status)
    for header in response_headers:
        sys.stdout.write('%s: %s\r\n' % header)
    sys.stdout.write('\r\n')

sys.stdout.write(data)
sys.stdout.flush()

def start_response(status, response_headers):
    """ Устанавливает заголовки для ответа, возвращаемого этим сервером. """
    if headers_set:
        raise AssertionError("Заголовки уже установлены!")

    headers_set[:] = [status, response_headers]
    return write

# Это самая важная часть нашего 'server object'
# Наш результат будет сгенерирован приложением 'application', переданным этому методу
# в качестве параметра
result = application(environ, start_response)
try:
    for data in result:
        if data:
            write(data)          # Тело не пусто, отправьте его данные в 'write()'
        if not headers_sent:
            write("")             # Тело пустое, передаем пустую строку в 'write()'

```

Глава 139. События, посылаемые сервером (SSE)

Server Sent Events (SSE) – это однонаправленное соединение между сервером и клиентом (обычно веб-браузером), которое позволяет серверу передавать (“проталкивать”, “push”) информацию клиенту. Технология похожа на технологии веб-сокетов и “длинного опроса” (Long Polling). Основное различие заключается в том, что SSE является однонаправленным, только сервер может посылать информацию клиенту, а при использовании веб-сокетов оба могут посылать информацию друг другу. Считается, что SSE гораздо проще в использовании и реализации, чем веб-сокеты.

139.1. Flask SSE

```

@route("/stream")
def stream():
    def event_stream():
        while True:
            if message_to_send:
                yield "data:
                    {}\n\n".format(message_to_send)"

    return Response(event_stream(), mimetype="text/event-stream")

```

139.2. Asyncio SSE

В данном примере используется SSE-библиотека `asyncio`:

```
import asyncio
import sse

class Handler(sse.Handler):
    @asyncio.coroutine
    def handle_request(self):
        yield from asyncio.sleep(2)
        self.send('foo')
        yield from asyncio.sleep(2)
        self.send('bar', event='wakeup')

start_server = sse.serve(Handler, 'localhost', 8888)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Глава 140. Альтернативы оператору switch из других языков

140.1. Используйте то, что предлагает Python: конструкция if/else

Если вам нужна конструкция `switch/case`, то наиболее простым способом будет использование старых добрых `if/else`:

```
def switch(value):
    if value == 1:
        return "one"
    if value == 2:
        return "two"
    if value == 42:
        return "ответ на вопрос о жизни, Вселенной и всем остальном"
    raise Exception("Дело не найдено!")
```

Это может выглядеть излишним и не всегда красивым, но это, безусловно, самый надежный способ, и он работает:

```
>>> switch(1)
one
>>> switch(2)
two
>>> switch(3)
Exception: Дело не найдено!
>>> switch(42)
ответ на вопрос о жизни, Вселенной и всем остальном
```

140.2. Использование словаря функций

Другой прямой путь – это создание словаря функций:

```
switch = {
    1: lambda: 'one',
    2: lambda: 'two',
    42: lambda: 'ответ о жизни, Вселенной и всем остальном',
```

затем добавляется функция по умолчанию:

```
def default_case():
    raise Exception("Дело не найдено")
```

и с помощью метода get словаря получаем функцию, заданную значением для проверки, и можем запустить ее. Если значение не существует в словаре, то выполняется default_case.

```
>>> switch.get(1, default_case())
one
>>> switch.get(2, default_case())
two
>>> switch.get(3, default_case())
```

```
Exception: Дело не найдено!
```

```
>>> switch.get(42, default_case())
ответ о жизни, Вселенной и всем остальном
```

Также можно сделать “синтаксический сахар”, чтобы переключатель выглядел красивее:

```
def run_switch(value):
    return switch.get(value, default_case())
```

```
>>> run_switch(1)
one
```

140.3. Использование интроспекции классов

Для имитации структуры switch/case можно использовать класс. Ниже показано использование интроспекции класса (с помощью функции getattr(), разрешающей строку в связанный метод на экземпляре) для разрешения части “case”. Затем этот интроспективный метод алиасится на метод __call__ для перегрузки оператора ().

```
class SwitchBase:
    def switch(self, case):
        m = getattr(self, 'case_{}'.format(case), None)
        if not m:
            return self.default
        return m

    __call__ = switch
```

Затем, чтобы все выглядело красивее, мы подклассифицируем класс SwitchBase (но это можно сделать и в одном классе), и там все case определяем как методы:

```
class CustomSwitcher:
    def case_1(self):
        return 'one'

    def case_2(self):
        return 'two'

    def case_42(self):
        return 'ответ на вопрос о жизни, Вселенной и всем остальном!'

    def default(self):
        raise Exception("Not a case!")
```

и тогда мы сможем его использовать:

```
>>> switch = CustomSwitcher()
>>> print(switch(1))
```

```

one
>>> print(switch(2))
two
>>> print(switch(3))

```

```

Exception: Not a case!

```

```

>>> print(switch(42))
ответ на вопрос о жизни, Вселенной и всем остальном!

```

140.4. Использование менеджера контекста

Другой способ, удобный для чтения и элегантный, но гораздо менее эффективный, чем структура `if/else`, заключается в создании класса, который будет считывать и хранить значение, с которым нужно сравнить, и представлять себя в контексте как вызываемый класс, который будет возвращать `true`, если он соответствует сохраненному значению:

```

class Switch:
    def __init__(self, value):
        self._val = value
    def __enter__(self):
        return self
    def __exit__(self, type, value, traceback):
        return False # Позволяет выполнить обратную трассировку
    def __call__(self, cond, *mconds):
        return self._val in (cond,)+mconds

```

определение случаев практически совпадает с реальной конструкцией `switch/case` (показанной внутри нижеприведенной функции):

```

def run_switch(value):
    with Switch(value) as case:
        if case(1):
            return 'one'
        if case(2):
            return 'two'
        if case(3):
            return 'ответ на вопрос о жизни, Вселенной и всем остальном!'
        # default
        raise Exception('Not a case!')

```

Таким образом, исполнение будет следующим:

```

>>> run_switch(1)
one
>>> run_switch(2)
two
>>> run_switch(3)

```

```

Exception: Not a case!

```

```

>>> run_switch(42)
ответ на вопрос о жизни, Вселенной и всем остальном!

```

Nota Bene:

Это решение предлагается в качестве модуля `switch`, доступного на `ru.py`.

Глава 141. Деструктуризация списка, упаковка и распаковка

141.1. Присваивание деструктуризации

В присваиваниях можно разбить итерируемый объект на значения, используя синтаксис распаковки:

Раскладывание на значения

```
a, b = (1, 2)
print(a)
# Выводит: 1
print(b)
# Выводит: 2
```

Если попытаться распаковать больше, чем длина итерируемого объекта, то будет выдана ошибка:

```
a, b, c = [1]
# Raises: ValueError: not enough values to unpack (expected 3, got 1) (недостаточно значений для распаковки (ожидалось 3, получено 1))
```

Версия Python 3.x > 3.0

Деструктуризация в виде списка

Распаковать список неизвестной длины можно с помощью следующего синтаксиса:

```
head, *tail = [1, 2, 3, 4, 5]
```

Здесь мы извлекаем первое значение в виде скаляра, а остальные значения – в виде списка:

```
print(head)
# Выводит: 1
print(tail)
# Выводит: [2, 3, 4, 5]
```

Что эквивалентно:

```
l = [1, 2, 3, 4, 5]
head = l[0]
tail = l[1:]
```

Он также работает с несколькими элементами или элементами, находящимися в конце списка:

```
a, b, *other, z = [1, 2, 3, 4, 5]
print(a, b, z, other)
# Выводит: 1 2 5 [3, 4]
```

Игнорирование значений в деструктурирующих присваиваниях

Если вас интересует только данное значение, вы можете использовать `_`, чтобы указать то, что вас не интересует. **Примечание:** при этом `_` все равно будет установлено, просто большинство людей не используют его в качестве переменной.

```
a, _ = [1, 2]
print(a)
# Выводит: 1
a, _, c = (1, 2, 3)
print(a)
# Выводит: 1
print(c)
# Выводит: 3
```

Версия Python 3.x > 3.0

Игнорирование списков в деструктурирующих присваиваниях

Наконец, с помощью синтаксиса `*` в присваивании можно игнорировать множество значений:

```
a, *_ = [1, 2, 3, 4, 5]
print(a)
# Выводит: 1
```

что не очень интересно, так как вместо этого можно использовать индексацию списка. Это удобно для работы с первым и последним значениями в одном присваивании:

```
a, *_ , b = [1, 2, 3, 4, 5]
print(a, b)
# Prints: 1 5
```

или извлечении нескольких значений сразу:

```
a, _, b, _, c, *_ = [1, 2, 3, 4, 5, 6]
print(a, b, c)
# Prints: 1 3 5
```

141.2. Аргументы функции упаковки

Можно определить ряд обязательных аргументов функции:

```
def fun1(arg1, arg2, arg3):
    return (arg1, arg2, arg3)
```

что делает ее вызываемой только при наличии трех аргументов:

```
fun1(1, 2, 3)
```

и можно определить аргументы как необязательные, используя значения по умолчанию:

```
def fun2(arg1='a', arg2='b', arg3='c'):
    return (arg1, arg2, arg3)
```

поэтому функцию можно вызывать разными способами, например:

```
fun2(1)           → (1,b,c)
fun2(1, 2)        → (1,2,c)
fun2(arg2=2, arg3=3) → (a,2,3)
```

Но вы также можете использовать синтаксис деструктуризации для *упаковки* аргументов, так что вы можете присваивать переменные, используя список или словарь.

Упаковка списка аргументов

Предположим, что у вас есть список значений:

```
l = [1,2,3]
```

Вызвать функцию со списком значений в качестве аргумента можно с помощью синтаксиса `*`:

```
fun1(*l)
# Возвращает: (1,2,3)
fun1(*['w', 't', 'f'])
# Возвращает: ('w','t','f')
```

Если только вы не предоставите список, длина которого соответствует количеству аргументов:

```
fun1(*['oops'])
# Raises: TypeError: fun1() missing 2 required positional arguments: 'arg2' and 'arg3' (в fun1()
отсутствует 2 необходимых позиционных аргумента: 'arg2' и 'arg3')
```

Упаковка именованных аргументов

Можно также упаковывать аргументы с помощью словаря. Вы можете использовать оператор **, чтобы указать Python распаковать словарь в качестве значений параметров:

```
d = {
    'arg1': 1,
    'arg2': 2,
    'arg3': 3
}
fun1(**d)
# Возвращает: (1, 2, 3)
```

если функция имеет только позиционные аргументы (без значений по умолчанию), то необходимо, чтобы словарь содержал все ожидаемые параметры и не имел лишних, иначе будет выдана ошибка:

```
fun1(**{'arg1':1, 'arg2':2})
# Вызывает: TypeError: fun1() missing 1 required positional argument: 'arg3' (у fun1() отсутствует
1 требуемый позиционный аргумент: 'arg3')
fun1(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Вызывает: TypeError: fun1() got an unexpected keyword argument 'arg4' (fun1() получила
неожиданный аргумент в виде ключевого слова 'arg4')
```

Для функций, имеющих необязательные аргументы, аргументы можно упаковать в словарь аналогичным образом:

```
fun2(**d)
# Возвращает: (1, 2, 3)
```

Но там можно не указывать значения, так как они будут заменены значениями по умолчанию:

```
fun2(**{'arg2': 2})
# Возвращает: ('a', 2, 'c')
```

При этом, как и ранее, нельзя задавать дополнительные значения, не являющиеся существующими параметрами:

```
fun2(**{'arg1':1, 'arg2':2, 'arg3':3, 'arg4':4})
# Вызывает: TypeError: fun2() got an unexpected keyword argument 'arg4' (fun2() получила
неожиданный аргумент в виде ключевого слова 'arg4')
```

В реальных условиях функции могут иметь как позиционные, так и необязательные аргументы, и это работает одинаково:

```
def fun3(arg1, arg2='b', arg3='c')
    return (arg1, arg2, arg3)
```

можно вызвать функцию, используя только итерируемый объект:

```
fun3(*[1])
# Возвращает: (1, 'b', 'c')
fun3(*[1,2,3])
# Возвращает: (1, 2, 3)
```

или словарь:

```
fun3(**{'arg1':1})
# Возвращает: (1, 'b', 'c')
fun3(**{'arg1':1, 'arg2':2, 'arg3':3})
# Возвращает: (1, 2, 3)
```

или вы можете использовать оба варианта в одном вызове:

```
fun3(*[1,2], **{'arg3':3})
# Возвращает: (1,2,3)
```

Однако следует помнить, что для одного и того же аргумента нельзя указывать несколько значений:

```
fun3(*[1,2], **{'arg2':42, 'arg3':3})
# Вызывает: TypeError: fun3() got multiple values for argument 'arg2' (fun3() получил несколько значений для аргумента 'arg2')
```

141.3. Распаковка аргументов функции

Если вы хотите создать функцию, которая может принимать любое количество аргументов и при этом не задавать позицию или имя аргумента во время “компиляции”, то это возможно, и вот как следует это сделать:

```
def fun1(*args, **kwargs):
    print(args, kwargs)
```

Параметры `*args` и `**kwargs` – это специальные параметры, которые устанавливаются в виде кортежа и словаря соответственно:

```
fun1(1,2,3)
# Выводит: (1, 2, 3) {}
fun1(a=1, b=2, c=3)
# Выводит: () {'a': 1, 'b': 2, 'c': 3}
fun1('x', 'y', 'z', a=1, b=2, c=3)
# Выводит: ('x', 'y', 'z') {'a': 1, 'b': 2, 'c': 3}
```

Если вы просмотрите достаточно много примеров кода на языке Python, то быстро обнаружите, что подобное широко используется при передаче аргументов в другую функцию. Например, если вы хотите расширить строковый класс:

```
class MyString(str):
    def __init__(self, *args, **kwarg):
        print('Constructing MyString')
        super(MyString, self).__init__(*args, **kwarg)
```

Глава 142. Доступ к исходному коду Python и байт-коду

142.1. Отображение байт-кода функции

Интерпретатор Python компилирует код в байт-код перед его выполнением на виртуальной машине Python (см. также главу “Что такое байткод Python?”).

Вот как посмотреть байт-код функции на Python:

```
import dis

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)

# Вывести дизассемблированный байт-код функции.
dis.dis(fib)
```

Функция `dis.dis` в модуле `dis` вернет декомпилированный байт-код переданной ей функции.

142.2. Отображение исходного кода объекта

Объекты, не являющиеся встроенными

Для вывода исходного кода объекта Python используйте команду `inspect`. Обратите внимание, что это не работает ни для встроенных объектов, ни для объектов, создаваемых в интерактивном режиме. Для них потребуются другие методы, о которых будет рассказано далее. Вот как вывести исходный код метода `randint` из модуля `random`:

```
import random
import inspect
print(inspect.getsource(random.randint))
# Выходные данные:
# def randint(self, a, b):
#     """Возвращает случайное целое число в диапазоне [a, b], включая обе конечные точки.
#     """
#
#     return self.randrange(a, b+1)
```

Чтобы просто вывести строку документации:

```
print(inspect.getdoc(random.randint))
# Вывод:
# Возвращает случайное целое число в диапазоне [a, b], включая обе конечные точки.
```

Вывести полный путь к файлу, в котором определен метод `random.randint`:

```
print(inspect.getfile(random.randint))
# c:\Python35\lib\random.py
print(random.randint.__code__.co_filename) # эквивалентно приведенному выше
# c:\Python35\lib\random.py
```

Объекты, создаваемые в интерактивном режиме

Если объект создается в интерактивном режиме, то команда `inspect` не может предоставить исходный код, но вы можете использовать вместо нее `dill.source.getsource`:

```
# определить новую функцию в интерактивной оболочке
def add(a, b):
    return a + b
print(add.__code__.co_filename) # Output: <stdin>
```

```
import dill
print(dill.source.getsource(add))
# def add(a, b):
#     return a + b
```

Встроенные объекты

Исходный код встроенных функций Python написан на языке C и может быть получен только путем просмотра исходного кода Python (размещенного на Mercurial или загружаемого с сайта <https://www.python.org/downloads/source/>).

```
print(inspect.getsource(sorted)) # выдает ошибку типа TypeError
type(sorted) # <class 'builtin_function_or_method'>
```

142.3. Исследование кодового объекта функции

CPython позволяет получить доступ к объекту кода для объекта функции. Объект `__code__` содержит исходный “сырой” байт-код (`co_code`) функции, а также другую информацию, например, имена констант и переменных.

```
def fib(n):
    if n <= 2: return 1
```

```

    return fib(n-1) + fib(n-2)
dir(fib.__code__)

def fib(n):
    if n <= 2: return 1
    return fib(n-1) + fib(n-2)
dir(fib.__code__)

```

Глава 143. Миксины

143.1. Миксин

Миксин (Mixin) – это набор свойств и методов, которые могут использоваться в разных классах, но *не* происходят из базового класса. В языках объектно-ориентированного программирования для наделения объектов разных классов одинаковой функциональностью обычно используется *наследование*; если набор объектов обладает некоторой способностью, то эта способность помещается в базовый класс, от которого *наследуют* оба объекта.

Например, у вас есть классы Car, Boat и Plane. Объекты всех этих классов обладают способностью путешествовать, поэтому они получают функцию travel. В этом сценарии все они путешествуют одним и тем же основным способом: получают маршрут и двигаются по нему. Чтобы реализовать эту функцию, можно вывести все классы из класса Vehicle и поместить функцию в этот общий класс:

```

class Vehicle(object):
    """Общий класс транспортного средства (vehicle)."""

    def init (self, position):
        self.position = position

    def travel(self, destination):
        route = calculate_route(from=self.position, to=destination)
        self.move_along(route)

class Car(Vehicle):
    pass

class Boat(Vehicle):
    pass

class Plane(Vehicle):
    pass

```

С помощью этого кода можно вызвать путешествие на автомобиле (car.travel("Montana")), лодке (boat.travel("Hawaii")) и самолете (plane.travel("France")).

Однако что если у вас есть функциональность, недоступная базовому классу? Например, вы хотите дать Car радио и возможность использовать его для воспроизведения песни на радиостанции с помощью функции play_song_on_station, но у вас также есть класс Clock, который тоже может использовать радио. Car и Clock могли бы иметь общий базовый класс (Machine). Однако Boat и Plane не могут (по крайней мере, в данном примере) проигрывать песни. Как же решить проблему, не дублируя код? Можно использовать миксин. В Python присвоить классу миксин можно просто добавив его в список подклассов, например так:

```
class Foo(main_super, mixin): ...
```

Класс Foo унаследует все свойства и методы main_super, а также свойства и методы mixin.

Так, чтобы предоставить классам Car и Clock возможность использовать радио, можно переопределить Car из предыдущего примера и написать следующее:

```
class RadioUserMixin(object):
    def __init__(self):
        self.radio = Radio()
    def play_song_on_station(self, station):
        self.radio.set_station(station)
        self.radio.play_song()

class Car(Vehicle, RadioUserMixin):
    # ...

class Clock(Vehicle, RadioUserMixin):
    # ...
```

Теперь вы можете вызвать `car.play_song_on_station(98.7)` и `clock.play_song_on_station(101.3)`, но не что-то вроде `boat.play_song_on_station(100.5)`.

Важным моментом в использовании миксинов является то, что они позволяют добавлять функциональность к совершенно разным объектам, не имеющим общего “главного” подкласса с этой функциональностью, но тем не менее имеющим общий код для нее. Без миксинов сделать что-то вроде приведенного выше примера было бы гораздо сложнее и/или потребовалось бы некоторое дублирование.

143.2. Переопределение методов в миксинах

Миксины – это своего рода классы, которые используются для “подмешивания” (“mix in”) в класс дополнительных свойств и методов. Обычно это не вызывает затруднений, поскольку во многих случаях классы-миксины не переопределяют методы друг друга или базового класса. Но если вы переопределите методы или свойства в своих миксинах, это может привести к неожиданным результатам, поскольку в Python иерархия классов строится справа налево. Например, возьмем следующие классы:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class BaseClass(object):
    def test(self):
        print "Base"

class MyClass(BaseClass, Mixin1, Mixin2):
    pass
```

В данном случае класс Mixin2 является базовым классом, расширенным Mixin1 и, наконец, BaseClass. Таким образом, если мы выполним следующий фрагмент кода:

```
>>> x = MyClass()
>>> x.test()
Base
```

мы увидим, что возвращаемый результат относится к классу Base. Это может привести к неожиданным ошибкам в логике вашего кода, и это необходимо учитывать и иметь в виду.

Глава 144. Доступ к атрибутам

144.1. Базовый доступ к атрибутам с использованием точечной нотации

Рассмотрим пример класса.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

```
book1 = Book(title="Right Ho, Jeeves", author="P.G. Wodehouse")
```

В Python доступ к атрибуту title класса можно получить, используя точечную нотацию.

```
>>> book1.title
'P.G. Wodehouse'
```

Если атрибут не существует, Python выдает ошибку:

```
>>> book1.series
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Book' object has no attribute 'series' (у объекта 'Book' отсутствует атрибут 'series')
```

144.2. Методы setter, getter и свойства

В целях инкапсуляции данных иногда требуется иметь атрибут, значение которого поступает из других атрибутов или, вообще говоря, значение которого должно быть вычислено в данный момент. Стандартный способ решения этой ситуации – создание метода, называемого getter или setter.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
```

В приведенном примере легко увидеть, что произойдет, если мы создадим новую книгу (Book), содержащую название (title) и автора (author). Если все книги, которые мы собираемся добавить в нашу библиотеку, имеют авторов и названия, то мы можем обойтись без методов getter и setter и использовать точечную нотацию. Однако предположим, что у некоторых книг нет автора, и мы хотим установить для них значение "Unknown". Или если у них несколько авторов, и мы хотим вернуть список авторов. В этом случае мы можем создать getter и setter для атрибута author.

```
class P:
    def __init__(self, title, author):
        self.title = title
        self.setAuthor(author)

    def get_author(self):
        return self.author

    def set_author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Данная схема не рекомендуется. Одна из причин заключается в том, что здесь есть подвох: предположим, что мы создали наш класс с атрибутом public и без методов. Программисты уже много раз пользовались этим и писали такой код:


```
>>> book = Book(title="Древний манускрипт", author="Какой-то парень")
>>> book.author = "" #Какой-то парень не писал эту книгу!
```

Проблема в том, что `author` не является атрибутом. В Python есть решение этой проблемы – свойства. Метод для получения свойств декорируется символом `@property` перед заголовком. Метод, который мы хотим использовать как `setter`, украшается `@attributeName.setter` перед ним.

Исходя из этого, мы получили наш новый обновленный класс.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    @property
    def author(self):
        return self.__author

    @author.setter
    def author(self, author):
        if not author:
            self.author = "Unknown"
        else:
            self.author = author
```

Обратите внимание, что обычно Python не позволяет иметь несколько методов с одним и тем же именем и разным количеством параметров. Однако в данном случае Python позволяет это сделать благодаря использованию декораторов. Тестируем код:

```
>>> book = Book(title="Древний манускрипт", author="Какой-то парень")
>>> book.author = "" #Какой-то парень не писал эту книгу!
>>> book.author
Unknown
```

Глава 145. Пакет ArcPy

145.1. Использование `createDissolvedGDB` для создания gdb-файла в рабочей области

```
def createDissolvedGDB(workspace, gdbName):
    gdb_name = workspace + "/" + gdbName + ".gdb"

    if arcpy.Exists(gdb_name):
        arcpy.Delete_management(gdb_name)
        arcpy.CreateFileGDB_management(workspace, gdbName, "")
    else:
        arcpy.CreateFileGDB_management(workspace, gdbName, "")

    return gdb_name
```

145.2. Печать значения одного поля для всех строк класса признаков в файле `geodatabase` с помощью функции `SearchCursor`

Для печати тестового поля (`TestField`) из тестового класса характеристик (`TestFC`) в тестовой базе геоданных (`Test.gdb`), расположенной во временной папке (`C:\Temp`):

```
with arcpy.da.SearchCursor(r"C:\Temp\Test.gdb\TestFC", ["TestField"]) as cursor:
    for row in cursor:
        print row[0]
```

Глава 146. Абстрактные базовые классы (abc)

146.1. Установка метакласса ABCMeta

Абстрактные классы – это классы, которые предназначены для наследования, но не реализуют конкретные методы, оставляя только сигнатуры методов, которые должны реализовывать подклассы. Абстрактные классы полезны для определения и реализации абстракций классов на высоком уровне, подобно концепции интерфейсов в типизированных языках, без необходимости реализации методов.

Один из концептуальных подходов к определению абстрактного класса заключается в том, чтобы заглушить методы класса, а при обращении к ним выдавать ошибку `NotImplementedError`. Это не позволяет дочерним классам обращаться к родительским методам, не переопределив их сначала. Например:

```
class Fruit:

    def check_ripeness(self):
        raise NotImplementedError("метод check_ripeness не реализован!")

class Apple(Fruit):
    pass

a = Apple()
a.check_ripeness() # вызывает ошибку NotImplementedError
```

Создание абстрактного класса предотвращает неправильное использование методов, которые не переопределены, и, конечно, поощряет определение методов в дочерних классах, но не принуждает к их определению. С помощью модуля `abc` мы можем запретить инстанцирование дочерних классов, если они не переопределяют методы абстрактных классов своих родителей и предков:

```
from abc import ABCMeta

class AbstractClass(object):
    # атрибут метакласса всегда должен быть задан как переменная класса
    __metaclass__ = ABCMeta

    # декоратор abstractmethod регистрирует этот метод как неопределенный
    @abstractmethod
    def virtual_method_subclasses_must_define(self):
        # Можно оставить полностью пустым или предоставить базовую реализацию
        # Обратите внимание, что обычно пустая интерпретация неявно возвращает `None`,
        # но при регистрации это поведение больше не применяется
```

Теперь можно просто подклассифицировать и переопределять:

```
class Subclass(AbstractClass):
    def virtual_method_subclasses_must_define(self):
        return
```

146.2. Зачем и как использовать ABCMeta и @abstractmethod

Абстрактные базовые классы (ABC) устанавливают, какие производные классы реализуют те или иные методы базового класса.

Чтобы понять, как это работает и почему мы должны это использовать, давайте рассмотрим пример, который понравился бы Гвидо ван Россуму. Допустим, у нас есть базовый класс `MontyPython` с двумя методами (`joke` и `punchline`), которые должны быть реализованы всеми производными классами.

```
class MontyPython:
    def joke(self):
        raise NotImplementedError()

    def punchline(self):
        raise NotImplementedError()

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

При инстанцировании объекта и вызове двух его методов мы получим ошибку (как и следовало ожидать) с методом `punchline()`.

```
>>> sketch = ArgumentClinic()
>>> sketch.punchline()
NotImplementedError
```

Однако это все равно позволяет нам инстанцировать объект класса `ArgumentClinic` без возникновения ошибки. Фактически мы не получаем ошибки до тех пор, пока не найдем функцию `punchline()`.

Избежать этого можно, используя модуль `Abstract Base Class (ABC)`. Посмотрим, как это работает, на том же примере:

```
from abc import ABCMeta, abstractmethod

class MontyPython(metaclass=ABCMeta):
    @abstractmethod
    def joke(self):
        pass

    @abstractmethod
    def punchline(self):
        pass

class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"
```

На этот раз при попытке инстанцировать объект из неполного класса мы сразу же получаем ошибку `TypeError`!

```
>>> c = ArgumentClinic()
TypeError:
"Can't instantiate abstract class ArgumentClinic with abstract methods punchline"
```

В этом случае достаточно просто завершить класс, чтобы избежать ошибок типа `TypeError`:

```
class ArgumentClinic(MontyPython):
    def joke(self):
        return "Hahahahahah"

    def punchline(self):
        return "Send in the constable!"
```

На этот раз при инстанцировании объекта все работает.

Глава 147. Плагины и расширения

147.1. Миксины

В объектно-ориентированном языке программирования миксин (mixin) – это класс, который содержит методы для использования другими классами без необходимости быть родительским классом этих классов. То, как другие классы получают доступ к методам миксина, зависит от языка.

Миксины обеспечивают механизм множественного наследования, позволяя нескольким классам использовать общую функциональность, но без сложной семантики множественного наследования. Они полезны, когда программист хочет разделить функциональность между разными классами. Вместо того чтобы повторять один и тот же код снова и снова, общая функциональность может быть просто сгруппирована в миксин и затем унаследована в каждом классе, которому она требуется.

Когда мы используем более одного миксина, важен порядок следования:

```
class Mixin1(object):
    def test(self):
        print "Mixin1"

class Mixin2(object):
    def test(self):
        print "Mixin2"

class MyClass(Mixin1, Mixin2):
    pass
```

В данном примере мы вызываем класс MyClass и метод test:

```
>>> obj = MyClass()
>>> obj.test()
Mixin1
```

В результате должен получиться Mixin1, так как порядок следования – слева направо. Это может привести к неожиданным результатам при добавлении суперклассов. Поэтому лучше использовать обратный порядок:

```
class MyClass(Mixin2, Mixin1):
    pass
```

Результатом будет:

```
>>> obj = MyClass()
>>> obj.test()
Mixin2
```

Миксины могут быть использованы для создания пользовательских плагинов.

Версия Python 3.x ≥ 3.0:

```
class Base(object):
    def test(self):
        print("Base.")

class PluginA(object):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(object):
    def test(self):
```

```

    super().test()
    print("Plugin B.")

plugins = PluginA, PluginB

class PluginSystemA(PluginA, Base):
    pass

class PluginSystemB(PluginB, Base):
    pass

PluginSystemA().test()
# Base.
# Plugin A.

PluginSystemB().test()
# Base.
# Plugin B.

```

147.2. Плагины с пользовательскими классами

В Python 3.6 в PEP 487 был добавлен специальный метод `__init_subclass__`, который упрощает и расширяет настройку классов без использования метаклассов. Следовательно, эта возможность позволяет создавать простые плагины. Продемонстрируем эту возможность, модифицируя предыдущий пример:

Версия Python 3.x ≥ 3.6:

```

class Base:
    plugins = []

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        cls.plugins.append(cls)

    def test(self):
        print("Base.")

class PluginA(Base):
    def test(self):
        super().test()
        print("Plugin A.")

class PluginB(Base):
    def test(self):
        super().test()
        print("Plugin B.")

```

Результаты:

```

PluginA().test()
# Base.
# Plugin A.

PluginB().test()
# Base.
# Plugin B.

Base.plugins
# [__main__.PluginA, __main__.PluginB]

```

Глава 148. Неизменяемые типы данных (int, float, str, tuple и frozenset)

148.1. Отдельные символы строк не подлежат присвоению

```
foo = "bar"
foo[0] = "c" # Error
```

Значение неизменяемой переменной не может быть изменено после ее создания.

148.2. Индивидуальные члены кортежа не могут быть присвоены

```
foo = ("bar", 1, "Hello!")
foo[1] = 2 # ERROR!!
```

Вторая строка вернет ошибку, так как члены кортежа после создания не могут быть назначены. Это связано с неизменяемостью кортежа.

148.3. Frozenset является неизменяемым и не подлежит присваиванию

```
foo = frozenset(["bar", 1, "Hello!"])
foo[2] = 7 # ERROR
foo.add(3) # ERROR
```

Вторая строка вернет ошибку, так как созданные члены frozenset не могут быть присвоены. Третья строка вернет ошибку, так как frozenset не поддерживает функции, которые могут манипулировать элементами.

Глава 149. Несовместимости при переходе с Python 2 на Python 3

В отличие от большинства языков программирования, Python поддерживает две основные версии. С 2008 года, когда был выпущен Python 3, многие успели перейти на него, но многие по-прежнему используют Python 2. Поэтому в данной главе мы рассмотрим важные различия между Python 2 и Python 3.

149.1. Целочисленное деление

Стандартный **символ деления** (/) действует по-разному в Python 3 и Python 2 при применении к целым числам. При делении целого числа на другое целое число в Python 3 операция деления x / y представляет собой *истинное деление* (используется метод `truediv`) и выдает результат в виде числа с плавающей точкой. Та же операция в Python 2 представляет собой *классическое деление*, при котором результат округляется в меньшую сторону по направлению к отрицательной бесконечности.

Например:

Код	результат в Python 2	результат в Python 3
3 / 2	1	1.5
2 / 3	0	0.6666666666666666
-3 / 2	2	-1.5

Такое округления в сторону нуля было признано устаревшим в Python 2.2, но осталось в Python 2.7 для обеспечения обратной совместимости и было окончательно убрано в Python 3.

Примечание: для получения результата в виде числа с плавающей точкой (без округления в меньшую сторону) в Python 2 можно указать один из операндов с десятичной точкой. Приведенный выше пример с $2/3$, который дает 0 в Python 2, можно записать как $2/3.0$, или $2.0/3$, или $2.0/3.0$, чтобы получить 0,6666666666666666.

Код	результат в Python 2	результат в Python 3
$3.0 / 2.0$	1.5	1.5
$2 / 3.0$	0.6666666666666666	0.6666666666666666
$-3.0 / 2$	1.5	-1.5

Существует также оператор деления `//`, который работает одинаково в обеих версиях: он округляет до ближайшего меньшего целого числа (хотя при использовании с числами типа `float` (числами с плавающей точкой) и возвращается `float`). В обеих версиях используется метод `__floordiv__`.

Код	результат в Python 2	результат в Python 3
$3 // 2$	1	1
$2 // 3$	0	0
$-3 // 2$	-2	-2
$3.0 // 2.0$	1.0	1.0
$2.0 // 3$	0.0	0.0
$-3 // 2.0$	-2.0	-2.0

Можно явно обеспечить истинное деление или деление с округлением в меньшую сторону с помощью собственных функций в модуле `operator`:

```
from operator import truediv, floordiv
assert truediv(10, 8) == 1.25      # эквивалентно '/' в Python 3
assert floordiv(10, 8) == 1        # эквивалентно '//'
```

Использование операторной функции для каждого деления, хотя и является понятным и явным, может быть утомительным. Изменение поведения оператора `/` часто оказывается предпочтительнее. Обычной практикой является устранение типичного поведения при делении путем добавления оператора `from __future__ import division` в качестве первого утверждения в каждом модуле:

```
# должно быть первым утверждением в модуле
from __future__ import division
```

Код	результат в Python 2	результат в Python 3
$3 / 2$	1.5	1.5
$2 / 3$	0.6666666666666666	0.6666666666666666
$-3 / 2$	-1.5	-1.5

Использование `from __future__ import division` гарантирует, что оператор `/` обеспечивает истинное деление и обеспечивает его только в пределах модулей, содержащих `__future__ import`, поэтому нет веских причин не включать его во всех новых модулях.

Примечание: в некоторых других языках программирования используется *округление в сторону нуля* (усечение), а не *округление в сторону отрицательной бесконечности*, как в Python (т.е. в этих языках $-3 / 2 == -1$). Такое поведение может привести к путанице при переносе или сравнении кода.

Примечание по поводу операндов типа float: В качестве альтернативы `from __future__ import division` можно использовать обычный символ деления `/` и убедиться, что хотя бы один из операндов имеет тип `float`: `3 / 2.0 == 1.5`. Однако это можно считать плохой практикой. Слишком легко написать `average = sum(items) / len(items)` и забыть привести один из аргументов к типу `float`. Кроме того, такие случаи часто могут остаться незамеченными при тестировании, например, если тестирование проводится на массиве, содержащем числа с плавающей точкой, а в процессе работы вы получаете массив целых чисел. Кроме того, если этот же код использовать в Python 3, то программы, ожидающие, что `3 / 2 == 1` будет равно `True`, будут работать некорректно.

Более подробное обоснование того, почему оператор деления был изменен в Python 3 и почему следует избегать деления в старом стиле, см. в PEP 238.

149.2. Распаковка итерируемых объектов

Версия Python 3.x ≥ 3.0

В Python 3 можно распаковать итерируемый объект, не зная точного количества элементов в нем, и даже привязывать переменную к конечному элементу итерируемого объекта. Для этого необходимо указать переменную, которая может собирать список значений. Для этого перед ее именем ставится звездочка. Например, распакуем список:

```
first, second, *tail, last = [1, 2, 3, 4, 5]
print(first)      # Результат: 1
print(second)     # Результат: 2
print(tail)       # Результат: [3, 4]
print(last)       # Результат: 5
```

Примечание: при использовании синтаксиса `*variable` переменная всегда будет списком, даже если исходный тип не был списком. Она может содержать ноль или более элементов в зависимости от количества элементов в исходном списке.

```
first, second, *tail, last = [1, 2, 3, 4]
print(tail)      # Результат: [3]
```

```
first, second, *tail, last = [1, 2, 3]
print(tail)      # Результат: []
print(last)      # Результат: 3
```

Аналогично, распаковка строки `str`:

```
begin, *tail = "Hello"
print(begin)   # Результат: 'H'
print(tail)    # Результат: ['e', 'l', 'l', 'o']
```

Пример распаковки типа `date`; `_` используется в данном примере как отбрасываемая переменная (нас интересует только значение года `year`):

```
person = ('John', 'Doe', (10, 16, 2016))
*_, (*_, year_of_birth) = person
print(year_of_birth)      # Результат: 2016
```

Следует отметить, что поскольку звездочка `*` потребляет переменное количество элементов, нельзя использовать две `*` для одного итерируемого объекта в присваивании – не будет понятно, сколько элементов входит в первую распаковку, а сколько во вторую:

```
*head, *tail = [1, 2]
# Out: SyntaxError: two starred expressions in assignment
(в присваивании два выражения со звездочками)
```

Версия Python 3.x ≥ 3.5

До сих пор мы обсуждали распаковку в присваиваниях. В Python 3.5 использование `*` и `**` было расширено. Теперь в одном выражении можно выполнять несколько операций распаковки:


```
{*range(4), 4, *(5, 6, 7)}
# Результат: {0, 1, 2, 3, 4, 5, 6, 7}
```

Версия Python 2.x ≥ 2.0:

Также возможна распаковка итерируемого объекта в аргументы функции:

```
iterable = [1, 2, 3, 4, 5]
print(iterable)    # Результат: [1, 2, 3, 4, 5]
print(*iterable)   # Результат: 1 2 3 4 5
```

Версия Python 3.x ≥ 3.5

При распаковке словаря используются две звездочки ** (PEP 448):

```
tail = {'y': 2, 'z': 3}
{'x': 1, **tail}    # Результат: {'x': 1, 'y': 2, 'z': 3}
```

Это позволяет как переопределять старые значения, так и объединять словари.

```
dict1 = {'x': 1, 'y': 1}
dict2 = {'y': 2, 'z': 3}
{**dict1, **dict2}
# Результат: {'x': 1, 'y': 2, 'z': 3}
```

Версия Python 3.x ≥ 3.0:

В Python 3 удалена распаковка кортежей в функциях (подробное обоснование см. в PEP 3113.). Таким образом, в Python 3 не работает следующее:

```
# Работает в Python 2, но выдает синтаксическую ошибку в Python 3:
map(lambda (x, y): x + y, zip(range(5), range(5)))
# То же самое справедливо и для не-лямбд:
def example((x, y)):
    pass
```

```
# Работает как в Python 2, так и в Python 3:
map(lambda x: x[0] + x[1], zip(range(5), range(5)))
# И не-лямбда тоже:
def working_example(x_y):
    x, y = x_y
    pass
```

149.3. Строки: байты и Unicode

Версия Python 2.x ≤ 2.7:

В Python 2 существует два вида строк: состоящие из байтов с типом `str` и состоящие из текста с типом `Unicode`. В Python 2 объект типа `str` всегда является последовательностью байтов, но обычно используется как для текстовых, так и для двоичных данных. Строковый литерал интерпретируется как байтовая строка.

```
s = 'Cafe'          # type(s) == str
```

Существуют два исключения: вы можете явно выделить *текстовый литерал Unicode*, предварительно обозначив его символом `u`:

```
s = u'Café'         # type(s) == unicode
b = 'Lorem ipsum'   # type(b) == str
```

В качестве альтернативы можно указать, что строковые литералы всего модуля должны создавать *текстовые литералы Unicode*:

```
from __future__ import unicode_literals
```

```
s = 'Café'          # type(s) == unicode
b = 'Lorem ipsum'   # type(b) == unicode
```

Для проверки того, является ли переменная строкой (Unicode или байтовой строкой), можно использовать:

```
isinstance(s, basestring)
```

Версия Python 3.x ≥ 3.0:

В Python 3 тип `str` является текстовым типом Unicode.

```
s = 'Cafe' # type(s) == str
```

```
s = 'Caf ' # type(s) == str (обратите внимание на акцентированное завершающее e)
```

Кроме того, в Python 3 добавлен объект `bytes`, который подходит для двоичных объектов или записи в независимые от кодировки файлы. Чтобы создать объект `bytes`, можно предварительно записать `b` в строковый литерал или вызвать метод `encode` строки:

Или, если вам действительно нужна байтовая строка:

```
s = b'Cafe' # type(s) == bytes
```

```
s = 'Caf '.encode() # type(s) == bytes
```

Для проверки того, является ли значение строкой, используйте:

```
isinstance(s, str)
```

Версия Python 3.x ≥ 3.3:

Для облегчения совместимости между кодовыми базами Python 2 и Python 3 можно предварять строковые литералы символом `u`. Поскольку в Python 3 все строки по умолчанию являются Unicode, добавление к строковому литералу символа `u` не имеет никакого эффекта:

```
u'Cafe' == 'Cafe'
```

Однако необработанная строка Unicode из Python 2 с префиксом `u` не поддерживается:

```
>>> ur'Caf '
```

```
File "<stdin>", line 1
```

```
ur'Caf '
```

```
^
```

```
SyntaxError: invalid syntax
```

Обратите внимание, что для преобразования текстового объекта Python 3 (`str`) в байтовое представление этого текста (`bytes`) необходимо его закодировать. По умолчанию в этом методе используется кодировка UTF-8. С помощью `decode` можно запросить у объекта `bytes`, какой текст Unicode он представляет:

```
>>> b.decode()
```

```
'Caf '
```

Версия Python 2.x ≥ 2.6

В то время как тип `bytes` существует как в Python 2, так и в Python 3, тип `unicode` есть только в Python 2. Чтобы использовать неявные Unicode-строки Python 3 в Python 2, добавьте в верхнюю часть кода следующую фразу:

```
from __future__ import unicode_literals
```

```
print(repr("hi"))
```

```
# u'hi'
```

Версия Python 3.x ≥ 3.0:

Еще одно важное отличие заключается в том, что индексирование байтов в Python 3 приводит к выводу `int` следующим образом:

```
b"abc"[0] == 97
```

В то время как нарезка размером один приводит к получению объекта длиной 1 байт:

```
b"abc"[0:1] == b"a"
```

Кроме того, в Python 3 устранилось необычное поведение с Unicode, т.е. реверсирование байтовых строк в Python 2.

149.4. Оператор Print и функция Print

В Python 2 print – это оператор:

Версия Python 2.x ≤ 2.7:

```
print "Hello World"
print                                     # вывести новую строку
print "No newline",                     # добавить запятую для удаления новой строки
print >>sys.stderr, "Error"            # вывод в stderr
print("hello")                          # вывести "hello", так как ("hello") == "hello"
print()                                 # вывести пустой кортеж "()".
print 1, 2, 3                           # вывести аргументы, разделенные пробелами: "1 2 3"
print(1, 2, 3)                          # вывести кортеж "(1, 2, 3)"
```

В Python 3 функция print() является функцией, имеющей ключевые слова-аргументы для общего использования:

Версия Python 3.x ≥ 3.0

```
print "Hello World"                    # SyntaxError
print("Hello World")
print()                                # вывести новую строку (необходимо использовать круглые скобки)
print("No newline", end="")             # end указывает, что добавлять (по умолчанию - новая строка)
print("Error", file=sys.stderr)         # file указывает выходной буфер
print("Comma", "separated", "output", sep=",") # sep задает разделитель
print("A", "B", "C", sep="")           # нулевая строка для sep: выводится как ABC
print("Flush this", flush=True)         # очистка выходного буфера, добавлена в Python 3.3
print(1, 2, 3)                          # печать аргументов, разделенных пробелами: "1 2 3"
print((1, 2, 3))                        # вывести кортеж "(1, 2, 3)".
```

Функция печати имеет следующие параметры:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

sep – это то, что разделяет объекты, передаваемые на печать. Например:

```
print('foo', 'bar', sep='~') # вывод: foo~bar
print('foo', 'bar', sep='.') # вывод: foo.bar
```

end – это то, чем заканчивается оператор печати. Например:

```
print('foo', 'bar', end='!') # вывод: foo bar!
```

Повторная печать после оператора печати, не завершающего новую строку, будет выведена на ту же строку:

```
print('foo', end='~')
print('bar')
# вывод: foo~bar
```

Примечание: для обеспечения совместимости функция print доступна и в Python 2.6 и далее; однако ее нельзя использовать, если только разбор оператора print не отключен с помощью функции

```
from __future__ import print_function
```

Эта функция имеет точно такой же формат, как и функция Python 3, за исключением того, что в ней отсутствует параметр flush. Обоснование см. в PEP 3105.

149.5. Отличия между функциями range и xrange

В Python 2 функция range возвращает список, а xrange создает специальный объект xrange, представляющий собой неизменяемую последовательность, которая, в отличие от других встроенных типов последовательностей, не поддерживает нарезку и не имеет методов index и count:

Версия Python 2.x ≥ 2.3:

```
print(range(1, 10))           # Вывод: [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(isinstance(range(1, 10), list)) # Вывод: True
print(xrange(1, 10))         # Вывод: xrange(1, 10)
print(isinstance(xrange(1, 10), xrange)) # Вывод: True
```

В Python 3 xrange был расширен до последовательности range, которая, таким образом, теперь создает объект range. В настоящее время xrange не существует:

Версия Python 3.x ≥ 3.0:

```
print(range(1, 10))
# Вывод: range(1, 10)
print(isinstance(range(1, 10), range))
# Вывод: True
# print(xrange(1, 10)) # Выходные данные будут иметь вид:
# Traceback (last recent call last):
#   File "<stdin>", line 1, in <module>
# NameError: имя 'xrange' не определено
```

Кроме того, начиная с Python 3.2 range также поддерживает нарезку, index и count:

```
print(range(1, 10)[3:7])    # Вывод: range(3, 7)
print(range(1, 10).count(5)) # Вывод: 1
print(range(1, 10).index(7)) # Вывод: 6
```

Преимущество использования специального типа последовательности вместо списка заключается в том, что интерпретатору не нужно выделять память под список и заполнять его:

Версия Python 2.x ≥ 2.3:

```
# range(1000000000000000000)
# Результатом будет:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# MemoryError
print(xrange(1000000000000000000))
# Вывод: xrange(1000000000000000000)
```

Поскольку последнее поведение в целом желательно, ранее было убрано в Python 3. Если вы все же хотите иметь список в Python 3, вы можете просто использовать конструктор list() на объекте range:

Версия Python 3.x ≥ 3.0:

```
print(list(range(1, 10)))
# Вывод: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Совместимость

Для поддержания совместимости между версиями Python 2.x и Python 3.x можно использовать модуль builtins из внешнего пакета future для достижения *прямой и обратной совместимости*:

Версия Python 2.x ≥ 2.0

```
#прямая совместимость
from builtins import range

for i in range(10**8):
    pass
```

Версия Python 3.x ≥ 3.0:

```
#обратная совместимость
from past.builtins import xrange
```

```
for i in xrange(10**8):
    pass
```

Функция `range` в библиотеке `future` поддерживает нарезку, `index` и `count` во всех версиях Python, как встроенный метод в Python 3.2+.

149.6. Возникновение и обработка исключений

Вот пример синтаксиса Python 2, обратите внимание на запятые в строках `raise` и `except`:

Версия Python 2.x ≥ 2.3:

```
try:
    raise IOError, "input/output error"
except IOError, exc:
    print exc
```

В Python 3 запятые исключены и заменены круглыми скобками и ключевым словом `as`:

```
try:
    raise IOError("input/output error")
except IOError as exc:
    print(exc)
```

Для обеспечения обратной совместимости синтаксис Python 3 доступен и в Python 2.6 и далее, поэтому его следует использовать для всего нового кода, который не должен быть совместим с предыдущими версиями.

Версия Python 3.x ≥ 3.0:

В Python 3 также добавлена цепочка исключений, с помощью которой можно сигнализировать о том, что причиной данного исключения стало другое исключение. Например:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from e
```

Исключение, вызванное в операторе `except`, имеет тип `DatabaseError`, но исходное исключение помечено как атрибут причины `__cause__` этого исключения. При выводе трассировки исходное исключение также будет отображено:

```
Traceback (most recent call last):
  File "", line 2, in
FileNotFoundError
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "", line 4, in
DatabaseError('Cannot open database.db')
```

Если вы добавите блок исключений без явной цепочки:

```
try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}')
```

Трассировка:

```
Traceback (most recent call last):
  File "", line 2, in
```

```

FileNotFoundError
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
File "", line 4, in
DatabaseError('Cannot open database.db')

```

Версия Python 2.x ≥ 2.0

Ни тот, ни другой вариант не поддерживается в Python 2.x; исходное исключение и его трассировка будут потеряны, если в блоке `except` будет вызвано другое исключение. Для совместимости можно использовать следующий код:

```

import sys
import traceback

try:
    funcWithError()
except:
    sys_ver = getattr(sys, 'version_info', (0,))
    if sys_ver < (3, 0):
        traceback.print_exc()
    raise Exception("new exception")

```

Версия Python 3.x ≥ 3.3:

Чтобы “забыть” о ранее созданном исключении, используйте `raise from None`:

```

try:
    file = open('database.db')
except FileNotFoundError as e:
    raise DatabaseError('Cannot open {}') from None

```

Теперь трассировка будет выглядеть следующим образом:

```

Traceback (most recent call last):
File "", line 4, in
DatabaseError('Cannot open database.db')

```

Или, для совместимости с Python 2 и 3, можно использовать пакет `six` следующим образом:

```

import six
try:
    file = open('database.db')
except FileNotFoundError as e:
    six.raise_from(DatabaseError('Cannot open {}'), None)

```

149.7. Утечка переменных в генераторе списка

Версия Python 2.x ≥ 2.3:

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']

print(vowels)    # Результат: ['A', 'E', 'I', 'O', 'U']
print(x)         # Результат: 'U'

```

Python 3.x Version ≥ 3.0

```

x = 'hello world!'
vowels = [x for x in 'AEIOU']
print(vowels)    # Результат: ['A', 'E', 'I', 'O', 'U']
print(x)         # Результат: 'hello world!'

```

Как видно из примера, в Python 2 произошла утечка значения `x`: оно вывело `U`, так как это было последнее значение `x` на момент завершения цикла. Однако в Python 3.x выводится

первоначально заданное `hello world!`, поскольку локальная переменная из генератора списка не маскирует переменные из окружающей области видимости.

Кроме того, ни генераторные выражения (доступные в Python начиная с версии 2.5), ни генераторы наборов и словарей (которые были перенесены в Python 2.7 из Python 3) не вызывают утечки переменных в Python 2.

Обратите внимание, что как в Python 2, так и в Python 3 при использовании цикла `for` возникает утечка переменных в окружающую область видимости:

```
x = 'hello world!'
vowels = []
for x in 'AEIOU':
    vowels.append(x)
print(x)
# Вывод: 'U'
```

149.8. True, False и None

В Python 2 `True`, `False` и `None` являются встроенными константами. Это означает, что их можно переопределить.

Версия Python 2.x ≥ 2.0:

```
True, False = False, True
True      # False
False     # True
```

Начиная с версии Python 2.4 вы не можете переопределять `None`.

Версия Python 2.x ≥ 2.4:

```
None = None # SyntaxError: cannot assign to None
```

В Python 3 `True`, `False` и `None` стали ключевыми словами.

Версия Python 3.x ≥ 3.0:

```
True, False = False, True # SyntaxError: can't assign to keyword (невозможно присвоить ключевое слово)
None = None # SyntaxError: can't assign to keyword (невозможно присвоить ключевое слово)
```

149.9. Ввод данных пользователем

В Python 2 пользовательский ввод принимается с помощью функции `raw_input`.

Версия Python 2.x ≥ 2.3:

```
user_input = raw_input()
```

В то время как в Python 3 пользовательский ввод принимается с помощью функции `input`.

Версия Python 3.x ≥ 3.0:

```
user_input = input()
```

В Python 2 функция `input` принимает входные данные и *интерпретирует* их. Хотя это может быть полезно, но вызывает сомнения в безопасности и было убрано в Python 3. Для доступа к той же функциональности можно использовать `eval(input())`. Чтобы сохранить переносимость кода в двух версиях Python, можно поместить приведенный ниже код в начало скрипта:

```
try:
    input = raw_input
except NameError:
    pass
```

149.10. Сравнение разных типов

Версия Python 2.x ≥ 2.3:

Можно сравнивать объекты разных типов. Результаты получаются произвольными, но непротиворечивыми. Они упорядочены таким образом, что None меньше, чем все остальное, числовые типы меньше, чем нечисловые, а все остальное упорядочено лексикографически по типу. Таким образом, int меньше, чем str, а кортеж больше, чем список:

```
[1, 2] > 'foo'           # Результат: False
(1, 2) > 'foo'           # Результат: True
[1, 2] > (1, 2)          # Результат: False
100 < [1, 'x'] < 'xyz' < (1, 'x') # Результат: True
```

Изначально это было сделано для того, чтобы можно было отсортировать список смешанных типов и сгруппировать объекты по типам:

```
l = [7, 'x', (1, 2), [5, 6], 5, 8.0, 'y', 1.2, [7, 8], 'z']
sorted(l)
# Результат: [1.2, 5, 7, 8.0, [5, 6], [7, 8], 'x', 'y', 'z', (1, 2)]
```

Версия Python 3.x ≥ 3.0:

При сравнении различных (нечисловых) типов возникает исключение:

```
1 < 1.5 # Результат: True
[1, 2] > 'foo'
# TypeError: unorderable types: list() > str() (неупорядоченные типы: list() > str())
(1, 2) > 'foo'
# TypeError: unorderable types: tuple() > str() (неупорядоченные типы: tuple() > str())
[1, 2] > (1, 2)
# TypeError: unorderable types: list() > tuple() (неупорядоченные типы: list() > tuple())
```

Для сортировки смешанных списков в Python 3 по типам и для достижения совместимости между версиями необходимо предоставить ключ функции sorted:

```
>>> list = [1, 'hello', [3, 4], {'python': 2}, 'stackoverflow', 8, {'python': 3}, [5, 6]]
>>> sorted(list, key=str)
# Результат: [1, 8, [3, 4], [5, 6], 'hello', 'stackoverflow', {'python': 2}, {'python': 3}]
```

Использование функции str в качестве ключевой (key) временно преобразует каждый элемент в строку только для целей сравнения. Затем она видит строковое представление, начинающееся с [, { или 0-9, и может отсортировать их (и все последующие символы).

149.11. Переименование метода .next() для итераторов

В Python 2 для обхода итератора можно использовать метод next на самом итераторе:

Версия Python 2.x ≥ 2.3:

```
g = (i for i in range(0, 3))
g.next() # Выдает 0
g.next() # Выдает 1
g.next() # Выдает 2
```

В Python 3 метод .next был переименован в __next__, что свидетельствует о его “магической” роли, поэтому при вызове .next будет возникать ошибка AttributeError. Правильным способом доступа к этой функциональности как в Python 2, так и в Python 3 является вызов функции next с итератором в качестве аргумента.

Версия Python 3.x ≥ 3.0:

```
g = (i for i in range(0, 3))
next(g) # Выдает 0
next(g) # Выдает 1
next(g) # Выдает 2
```

Этот код переносится на все версии, начиная с 2.6 и заканчивая текущими релизами.

149.12. Функции `filter()`, `map()` и `zip()` возвращают итераторы вместо последовательностей

Версия Python 2.x ≤ 2.7:

В Python 2 встроенные функции `filter`, `map` и `zip` возвращают последовательность: `map` и `zip` всегда возвращают список, в то время как у `filter` тип возвращаемого параметра зависит от типа заданного параметра:

```
>>> s = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> s
'abc'
>>> s = map(lambda x: x * x, [0, 1, 2])
>>> s
[0, 1, 4]
>>> s = zip([0, 1, 2], [3, 4, 5])
>>> s
[(0, 3), (1, 4), (2, 5)]
```

Версия Python 3.x ≥ 3.0:

В Python 3 функции `filter`, `map` и `zip` вместо этого возвращают итератор:

```
>>> it = filter(lambda x: x.isalpha(), 'a1b2c3')
>>> it
<filter object at 0x00000098A55C2518>
>>> ".join(it)
'abc'
>>> it = map(lambda x: x * x, [0, 1, 2])
>>> it
<map object at 0x000000E0763C2D30>
>>> list(it)
[0, 1, 4]
>>> it = zip([0, 1, 2], [3, 4, 5])
>>> it
<zip object at 0x000000E0763C52C8>
>>> list(it)
[(0, 3), (1, 4), (2, 5)]
```

Поскольку `itertools.izip` из Python 2 эквивалентен `zip` в Python 3, `izip` был убран из Python 3.

149.13. Переименованные модули

Несколько модулей в стандартной библиотеке были переименованы:

Старое имя	Новое имя
<code>_winreg</code>	<code>winreg</code>
<code>ConfigParser</code>	<code>configparser</code>
<code>copy_reg</code>	<code>copyreg</code>
<code>Queue</code>	<code>queue</code>
<code>SocketServer</code>	<code>socketserver</code>
<code>_markupbase</code>	<code>markupbase</code>
<code>repr</code> <code>reprlib</code>	
<code>test.test_support</code>	<code>test.support</code>
<code>Tkinter</code>	<code>tkinter</code>
<code>tkFileDialog</code>	<code>tkinter.filedialog</code>
<code>urllib</code> / <code>urllib2</code>	<code>urllib</code> , <code>urllib.parse</code> , <code>urllib.error</code> , <code>urllib.response</code> , <code>urllib.request</code> , <code>urllib.robotparser</code>

Некоторые модули даже были преобразованы в библиотеки. В качестве примера можно привести `tkinter` и `urllib`.

Совместимость

При поддержании совместимости между версиями Python 2.x и 3.x можно использовать внешний пакет `future`, позволяющий импортировать пакеты стандартной библиотеки верхнего уровня с именами Python 3.x в версии Python 2.x.

149.14. Удалены операторы <> и ``, синонимичные операторам != и repr()

В Python 2 <> является синонимом для !=; аналогично, `foo` является синонимом для repr(foo).

Версия Python 2.x ≤ 2.7:

```
>>> 1 <> 2
True
>>> 1 <> 1
False
>>> foo = 'hello world'
>>> repr(foo)
"hello world"
>>> `foo`
"hello world"
```

Версия Python 3.x ≥ 3.0:

```
>>> 1 <> 2
File "<stdin>", line 1
  1 <> 2
    ^
SyntaxError: invalid syntax
>>> `foo`
File "<stdin>", line 1
  `foo`
    ^
SyntaxError: invalid syntax
```

149.15. Типы данных long и int

В Python 2 любое целое число, большее, чем `ssize_t` в языке C, преобразуется в тип данных `long`, что обозначается суффиксом `L` на литерале. Например, в 32-битной сборке Python:

Версия Python 2.x ≤ 2.7:

```
>>> 2**31
2147483648L
>>> type(2**31)
<type 'long'>
>>> 2**30
1073741824
>>> type(2**30)
<type 'int'>
>>> 2**31 - 1 # 2**31 is long and long - int is long
2147483647L
```

Однако в Python 3 тип данных `long` был удален; независимо от размера целого числа, оно будет иметь тип `int`.

Версия Python 3.x ≥ 3.0:

```
2**1024
```

```
# Вывод: 1797693134862315907729305190789024733617976978942306572734300811577326758
055009631327084773224075360211201138798713933576587897688144166224928474306394741
243777678934248654852763022196012460941194530829520850057688381506823424628814739
13110540827237163350510684586298239947245938479716304835356329624224137216
type(2**1024)
# Вывод: <class 'int'>
```

149.16. Все классы в Python 3 являются “классами нового стиля”

В Python 3.x все классы являются классами нового стиля; при создании нового класса Python неявно заставляет его наследоваться от `object`. Таким образом, указание объекта в определении класса является совершенно необязательным:

Версия Python 3.x ≥ 3.0:

```
class X: pass
class Y(object): pass
```

Оба эти класса теперь содержат `object` в своем `mro` (method resolution order, порядок разрешения метода):

Python 3.x Версия ≥ 3.0:

```
>>> X.__mro__
(__main__.X, object)
>>> Y.__mro__
(__main__.Y, object)
```

В Python 2.x классы по умолчанию являются классами старого типа, они не наследуются неявно от `object`. Это приводит к тому, что семантика классов меняется в зависимости от того, добавляем ли мы явно `object` в качестве базового класса:

Версия Python 2.x ≥ 2.3:

```
class X: pass
class Y(object): pass
```

В этом случае, если попытаться вывести `mro` из `Y`, то появится вывод, аналогичный тому, что был в случае Python 3.x:

Версия Python 2.x ≥ 2.3:

```
>>> Y.__mro__
(<class '__main__.Y'>, <type 'object'>)
```

Это происходит потому, что при создании класса `Y` мы явно указали, что он наследуется от объекта: `class Y(object): pass`. Для класса `X`, который *не наследуется* от объекта, атрибут `__mro__` не существует, попытка доступа к нему приводит к ошибке `AttributeError`.

Для **обеспечения совместимости** между обеими версиями Python классы могут быть определены с использованием `object` в качестве базового класса:

```
class mycls(object):
    """Я полностью совместим с Python 2/3"""
```

В качестве альтернативы, если переменная `__metaclass__` установлена в значение `type` в глобальной области видимости, то все последующие определенные классы в данном модуле неявно становятся классами нового стиля без необходимости явного наследования от `object`:

```
__metaclass__ = type
```

```
class mycls:
    """Я также полностью совместим с Python 2/3"""
```

149.17. Функция reduce больше не является встроенной

В Python 2 функция reduce доступна либо как встроенная функция, либо из пакета `functools` (начиная с версии 2.6), в то время как в Python 3 reduce доступна только из `functools`. Однако синтаксис reduce в Python 2 и Python 3 одинаков и представляет собой `reduce(function_to_reduce, list_to_reduce)`.

В качестве примера рассмотрим сокращение списка до одного значения путем деления каждого из соседних чисел. Здесь мы используем функцию `truediv` из библиотеки `operator`.

В Python 2.x это делается просто:

Версия Python 2.x ≥ 2.3:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator
>>> reduce(operator.truediv, my_list)
0.008333333333333333
```

В Python 3.x пример несколько усложняется:

Версия Python 3.x ≥ 3.0:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> import operator, functools
>>> functools.reduce(operator.truediv, my_list)
0.008333333333333333
```

Мы также можем использовать `from functools import reduce`, чтобы избежать вызова reduce с именем пространства имен.

149.18. Абсолютный и относительный импорт

В Python 3 PEP 404 изменяет принцип работы импорта по сравнению с Python 2. Неявный относительный импорт больше не допускается в пакетах, а импорт `from ... import *` разрешен только в коде уровня модуля.

Чтобы добиться поведения Python 3 в Python 2:

- Функция абсолютного импорта может быть включена при помощи `from __future__ import absolute_import`
- *явный относительный импорт* поощряется вместо неявного относительного импорта

Например, в Python 2 модуль может импортировать содержимое другого модуля, расположенного в том же каталоге, следующим образом:

```
import foo
```

Обратите внимание, что местоположение `foo` только из выражения импорта неясно. Поэтому такой тип неявного относительного импорта не рекомендуется использовать в пользу явного относительного импорта, который выглядит следующим образом:

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
from ...package import bar
from ...sys import path
```

Точка `.` позволяет явно объявить местоположение модуля в дереве каталогов.

Подробнее об относительном импорте

Рассмотрим некоторый пользовательский пакет под названием `shapes`. Структура каталогов выглядит следующим образом:

```
shapes
├── __init__.py
├── circle.py
├── square.py
└── triangle.py
```

`circle.py`, `square.py` и `triangle.py` импортируют `util.py` в качестве модуля. Как они будут ссылаться на модуль на одном уровне?

```
from . import util # использовать util.PI, util.sq(x) и т. д.
```

ИЛИ

```
from .util import * #использовать PI, sq(x) и т. д. для вызова функций
```

Символ точки используется для относительного импорта того же уровня.

Теперь рассмотрим альтернативный вариант расположения модуля `форм`:

```
shapes
├── __init__.py
├── circle
│   ├── __init__.py
│   └── circle.py
├── square
│   ├── __init__.py
│   └── square.py
├── triangle
│   ├── __init__.py
│   └── triangle.py
└── util.py
```

Как эти 3 класса будут ссылаться на `util.py`?

```
from . import util # использовать util.PI, util.sq(x), и т. д.
```

ИЛИ

```
from ..util import * #использовать PI, sq(x) и т. д. для вызова функций
```

Символ `..` (двойные точки) используется для относительного импорта родительского уровня. Добавьте дополнительные точки с увеличением количества уровней между родительским и дочерним.

149.19. Функция `map()`

`map()` – это встроенная функция, полезная для применения функции к элементам итерируемого множества. В Python 2 функция `map` возвращает список. В Python 3 `map` возвращает объект `map`, который является генератором.

```
# Python 2.X
>>> map(str, [1, 2, 3, 4, 5])
['1', '2', '3', '4', '5']
>>> type(_)
>>> <class 'list'>
```

```
# Python 3.X
>>> map(str, [1, 2, 3, 4, 5])
<map object at 0x*>
>>> type(_)
<class 'map'>
```

Нам нужно снова применить map, так как мы “поглотили” предыдущую map....

```
>>> map(str, [1, 2, 3, 4, 5])
>>> list(_)
['1', '2', '3', '4', '5']
```

В Python 2 можно передавать None в качестве функции идентификации. В Python 3 это больше не работает.

Версия Python 2.x ≥ 2.3:

```
>>> map(None, [0, 1, 2, 3, 0, 4])
[0, 1, 2, 3, 0, 4]
```

Версия Python 3.x ≥ 3.0:

```
>>> list(map(None, [0, 1, 2, 3, 0, 5]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NoneType' object is not callable
```

Более того, в Python 2 при передаче в качестве аргумента более одного итерируемого множества map заполняет более короткие итерируемые множества при помощи None (аналогично itertools.zip_longest). В Python 3 итерация останавливается после самого короткого итерируемого множества.

В Python 2:

Версия Python 2.x ≥ 2.3:

```
>>> map(None, [1, 2, 3], [1, 2], [1, 2, 3, 4, 5])
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

В Python 3:

Версия Python 3.x ≥ 3.0:

```
>>> list(map(lambda x, y, z: (x, y, z), [1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2)]
```

для получения такого же дозаполнения (padding), как в Python 2, используйте zip_longest из itertools

```
>>> import itertools
>>> list(itertools.zip_longest([1, 2, 3], [1, 2], [1, 2, 3, 4, 5]))
[(1, 1, 1), (2, 2, 2), (3, None, 3), (None, None, 4), (None, None, 5)]
```

Примечание: вместо map можно использовать генераторы списков, которые совместимы с Python 2/3. Замена map(str, [1, 2, 3, 4, 5]):

```
>>> [str(i) for i in [1, 2, 3, 4, 5]]
['1', '2', '3', '4', '5']
```

149.20. Функция round() – “тай-брейкинг” и возвращаемый тип

“Тай-брейкинг” (“прекращение ничьей”) в функции round()

В Python 2 использование функции round() для числа, одинаково близкого к двум целым числам, вернет наиболее удаленное от 0. Например:

Версия Python 2.x ≤ 2.7:

```
round(1.5)      # Результат: 2.0
round(0.5)      # Результат: 1.0
```

```
round(-0.5)    # Результат: -1.0
round(-1.5)    # Результат: -2.0
```

Однако в Python 3 функция `round()` будет возвращать четное целое число (так называемое банковское округление). Например:

Версия Python 3.x ≥ 3.0:

```
round(1.5)     # Результат: 2
round(0.5)     # Результат: 0
round(-0.5)    # Результат: 0
round(-1.5)    # Результат: -2
```

Функция `round()` следует стратегии округления от половины к четным числам, в результате чего половинные числа округляются до ближайшего четного целого (например, `round(2.5)` теперь возвращает 2, а не 3.0). Согласно Википедии, эта стратегия также известна как несмещенное округление, сходящееся округление, статистическое округление, голландское округление, гауссово округление или четное округление. Округление от половины до четного является частью стандарта IEEE 754, а также режимом округления по умолчанию в Microsoft .NET. Такая стратегия округления, как правило, уменьшает суммарную ошибку округления. Поскольку в среднем количество чисел, округленных в большую сторону, равно количеству чисел, округленных в меньшую сторону, ошибки округления нивелируются. Другие методы округления, наоборот, имеют тенденцию к смещению средней ошибки в большую или меньшую сторону.

round() return type

Функция `round()` возвращает тип `float` в Python 2.7.

Версия Python 2.x ≤ 2.7:

```
round(4.8) # 5.0
```

Начиная с версии Python 3.0, если второй аргумент (количество цифр) опущен, то возвращается `int`.

Версия Python 3.x ≥ 3.0:

```
round(4.8) # 5
```

149.21. Файловый ввод/вывод

В версиях 3.x `file` больше не является встроенным именем (но `open` по-прежнему работает). Внутренние детали ввода/вывода файлов были перенесены в модуль `io` стандартной библиотеки, в которой теперь также размещен модуль `StringIO`:

```
import io
assert io.open is open # встроенный модуль является псевдонимом (alias)
buffer = io.StringIO()
buffer.write('hello, ') # возвращает количество записанных символов
buffer.write('world!\n')

buffer.getvalue() # 'hello, world!\n'
```

Режим файла (текстовый или двоичный) теперь определяет тип данных, получаемых при чтении файла (и тип, требуемый для записи):

```
with open('data.txt') as f:
    first_line = next(f)
    assert type(first_line) is str
with open('data.bin', 'rb') as f:
    first_kb = f.read(1024)
    assert type(first_kb) is bytes
```

Кодировка, возвращаемая `locale.getpreferredencoding(False)`, используется по умолчанию для текстовых файлов. Для явного указания кодировки используйте ключевой параметр `encoding`:

149.22. Функция `cmp` отсутствует в Python 3

В Python 3 встроенная функция `cmp` была убрана вместе со специальным методом `__cmp__`.

Из документации: Функцию `cmp()` следует считать исчезнувшей, а специальный метод `__cmp__()` больше не поддерживается. Используйте `__lt__()` для сортировки, `__eq__()` с `__hash__()` и другие “богатые сравнения” (rich comparisons) при необходимости. (Если вам действительно нужна функциональность `cmp()`, вы можете использовать выражение `(a > b) - (a < b)` в качестве эквивалента `cmp(a, b)`).

Кроме того, все встроенные функции, принимавшие параметр `cmp`, теперь принимают только параметр для ключевых слов `key`. В модуле `functools` также имеется полезная функция `cmp_to_key(func)`, которая позволяет преобразовать функцию в стиле `cmp` в функцию в стиле `key`:

Преобразовывайте функцию сравнения старого типа в ключевую функцию. Она используется с инструментами, принимающими ключевые функции (такими как `sorted()`, `min()`, `max()`, `heapq.nlargest()`, `heapq.nsmallest()`, `itertools.groupby()`). Эта функция используется в основном как переходное средство для программ, конвертируемых из Python 2, который поддерживал использование функций сравнения.

149.23. Восьмеричные константы

В Python 2 восьмеричный литерал может быть определен как:

```
>>> 0755      # только Python 2
```

Для обеспечения кросс-совместимости используйте:

```
0o755 # как в Python 2, так и в Python 3
```

149.24. Возвращаемое значение при записи в файловый объект

В Python 2 запись непосредственно в дескриптор возвращает `None`:

Версия Python 2.x ≥ 2.3:

```
hi = sys.stdout.write('hello world\\n')
# Результат: hello world
type(hi)
# Результат: <type 'NoneType'>
```

В Python 3 запись в дескриптор возвращает количество записанных символов при записи текста и количество записанных байтов при записи байтов:

Версия Python 3.x ≥ 3.0:

```
import sys

char_count = sys.stdout.write('hello world ?\\n')
# Результат: hello world ?
char_count
# Результат: 14

byte_count = sys.stdout.buffer.write(b'hello world \\xf0\\x9f\\x90\\x8d\\n')
# Результат: hello world ?
byte_count
# Результат: 17
```

149.25. В Python 3 `exec` является функцией

В Python 2 `exec` – это оператор, имеющий специальный синтаксис: `exec code [in globals[, locals]]`. В Python 3 `exec` теперь представляет собой функцию: `exec(code, [globals[, locals]])`, а синтаксис Python 2 вызовет ошибку `SyntaxError`.

Однако никакого импорта `from __future__ import exec_function` не добавлено, поскольку в нем нет необходимости: оператор `exec` в Python 2 можно использовать и с синтаксисом, который выглядит точно так же, как вызов функции `exec` в Python 3. Таким образом, можно изменить утверждения:

Версия Python 2.x ≥ 2.3:

```
exec 'code'
exec 'code' in global_vars
exec 'code' in global_vars, local_vars

к формам
```

Версия Python 3.x ≥ 3.0:

```
exec('code')
exec('code', global_vars)
exec('code', global_vars, local_vars)
```

причем последнее гарантированно работает одинаково как в Python 2, так и в Python 3.

149.26. Кодирование в шестнадцатеричный формат и декодирование из него больше не доступно

Версия Python 2.x ≤ 2.7:

```
"1deadbeef3".decode('hex')
# Результат: '\x1d\xea\xdb\xee\xfb'
'\x1d\xea\xdb\xee\xfb'.encode('hex')
# Результат: 1deadbeef3
```

Версия Python 3.x ≥ 3.0:

```
"1deadbeef3".decode('hex')
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# AttributeError: 'str' object has no attribute 'decode' (у объекта 'str' нет атрибута 'decode')
```

```
b"1deadbeef3".decode('hex')
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.decode() to handle arbitrary codecs
# ('hex' не является текстовой кодировкой; используйте codecs.decode() для работы
# с произвольными кодами)
```

```
'\x1d\xea\xdb\xee\xfb'.encode('hex')
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# LookupError: 'hex' is not a text encoding; use codecs.encode() to handle arbitrary codecs
# ('hex' не является текстовой кодировкой; используйте codecs.encode() для работы
# с произвольными кодами)
```

```
b'\x1d\xea\xdb\xee\xfb'.encode('hex')
# Traceback (most recent call last):
# File "<stdin>", line 1, in <module>
# AttributeError: 'bytes' object has no attribute 'encode' (у объекта 'bytes' нет атрибута 'encode')
```

Однако, как следует из сообщения об ошибке, для достижения того же результата можно использовать модуль `codecs`:

```
import codecs
codecs.decode('1deadbeef4', 'hex') # Результат: b'\x1d\xea\xdb\xee\xfb'
codecs.encode(b'\x1d\xea\xdb\xee\xfb', 'hex') # Результат: b'1deadbeef4'
```

Обратите внимание, что `codecs.encode` возвращает объект с типом `bytes`. Для получения объекта `str` достаточно выполнить декодирование в ASCII:

```
codecs.encode(b'\x1d\xea\xdb\xee\xff', 'hex').decode('ascii') # Out: '1deadbeeff'
```

149.27. Изменения в методах словарей

В Python 3 многие методы словарей значительно отличаются по поведению от Python 2, а многие и вовсе были убраны: исчезли `has_key`, `iter*` и `view*`. Вместо давно устаревшей функции `d.has_key(key)` теперь нужно использовать `key in d`. В Python 2 методы словаря `keys`, `values` и `items` возвращают списки. В Python 3 они возвращают *объекты представления*; эти объекты не являются итераторами и отличаются от них двумя способами, а именно:

- они имеют размер (для них можно использовать функцию `len`)
- их можно многократно итерировать.

Кроме того, как и в случае с итераторами, изменения в словаре отражаются на объектах представления.

В Python 2.7 эти методы были перенесены из Python 3; они доступны в виде `viewkeys`, `viewvalues` и `viewitems`. Для преобразования кода Python 2 в код Python 3 используются следующие подходы:

- `d.keys()`, `d.values()` и `d.items()` из Python 2 должны быть заменены на `list(d.keys())`, `list(d.values())` и `list(d.items())`
- `d.iterkeys()`, `d.itervalues()` и `d.iteritems()` следует заменить на `iter(d.keys())`, а еще лучше на `iter(d)`; `iter(d.values())` и `iter(d.items())` соответственно
- вызовы методов Python 2.7 `d.viewkeys()`, `d.viewvalues()` и `d.viewitems()` могут быть заменены на `d.keys()`, `d.values()` и `d.items()`.

Перенос на Python 2 кода, *выполняющего итерации* по ключам, значениям или элементам словаря с одновременным его изменением, иногда оказывается непростой задачей. Рассмотрим:

```
d = {'a': 0, 'b': 1, 'c': 2, 't': 3}
for key in d.keys():
    if key.isalpha():
        del d[key]
```

Код выглядит так, как будто он будет аналогично работать и в Python 3, но там метод `keys` возвращает объект представления, а не список, и если в процессе итерации словарь изменит размер, то код Python 3 завершится с ошибкой `RuntimeError: dictionary changed size during iteration`. Решением, конечно, является правильная запись `for key in list(d)`. Аналогично объекты представления ведут себя иначе, чем итераторы: для них нельзя использовать функцию `next()`, нельзя *возобновить* итерацию – она будет перезапущена; если в коде Python 2 передается возвращаемое значение `d.iterkeys()`, `d.itervalues()` или `d.iteritems()` в метод, который ожидает итератор, а не итерируемый объект, то в Python 3 это должно быть `iter(d)`, `iter(d.values())` или `iter(d.items())`.

149.28. Булево значение класса

Версия Python 2.x ≤ 2.7:

В Python 2, если вы хотите самостоятельно определить булево значение класса, вам необходимо реализовать метод `__nonzero__` в вашем классе. По умолчанию его значение равно `True`.

```
class MyClass:
    def __nonzero__(self):
        return False

my_instance = MyClass()
print bool(MyClass) # True
print bool(my_instance) # False
```

Версия Python 3.x ≥ 3.0:

В Python 3 вместо `__nonzero__` используется `__bool__`:

149.29. Ошибка функции `hasattr` в Python 2

В Python 2, если свойство вызывает ошибку, `hasattr` будет игнорировать это свойство, возвращая `False`.

```
class A(object):
    @property
    def get(self):
        raise IOError

class B(object):
    @property
    def get(self):
        return 'get in b'

a = A()
b = B()

print 'a hasattr get: ', hasattr(a, 'get')
# выводит False в Python 2 (исправлено, True в Python 3)
print 'b hasattr get', hasattr(b, 'get')
# выводит True в Python 2 и Python 3
```

Эта ошибка исправлена в Python3. Поэтому, если вы работаете в Python 2, используйте:

```
try:
    a.get
except AttributeError:
    print("нет свойства get!")

или используйте вместо этого getattr:

p = getattr(a, "get", None)
if p is not None:
    print(p)
else:
    print("нет свойства get!")
```

Глава 150. Инструмент 2to3

Параметр	Описание
<code>filename / directory_name</code>	В качестве аргумента <code>2to3</code> принимает список файлов или каталогов, которые необходимо преобразовать. Каталоги рекурсивно просматриваются в поисках источников Python.
Опция	Описание опции
<code>-f FIX, --fix=FIX</code>	Укажите, какие преобразования должны быть применены; по умолчанию: <code>all</code> . Список доступных преобразований доступен с помощью команды <code>--list-fixes</code>
<code>-j PROCESSES, --processes=PROCESSES</code>	Одновременный запуск <code>2to3</code>
<code>-x NOFIX, --nofix=NOFIX</code>	Исключить преобразование
<code>-l, --list-fixes</code>	Список доступных преобразований
<code>-p, --print-function</code>	Изменение грамматики таким образом, чтобы <code>print()</code> считалась функцией

<code>-v, --verbose</code>	Более подробный вывод
<code>--no-diffs</code>	Не выводить различия рефакторинга
<code>-w</code>	Записать обратно измененные файлы
<code>-n, --nobackups</code>	Не создавать резервные копии модифицированных файлов
<code>-o OUTPUT_DIR,</code> <code>--output-dir=OUTPUT_DIR</code>	Поместить выходные файлы в этот каталог вместо перезаписи входных файлов. Требуется флаг <code>-n</code> flag, так как резервные файлы не нужны, если входные файлы не модифицируются.
<code>W, --write-unchanged-files</code>	Записать выходные файлы, даже если никаких изменений не требовалось. Используется вместе с <code>-o</code> , чтобы перевести и скопировать полное дерево источников. Подразумевает <code>-w</code> .
<code>--add-suffix=ADD_SUFFIX</code>	Укажите строку, которая будет добавляться ко всем выводимым именам файлов. Требуется <code>-n</code> , если она непустая. Пример: <code>--add-suffix='3'</code> будет генерировать файлы .py3.

150.1. Базовое использование

Рассмотрим следующий код Python 2.x. Сохраните файл под именем `example.py`.

Версия Python 2.x ≥ 2.0:

```
def greet(name):
    print "Hello, {0}!".format(name)
print "What's your name?"
name = raw_input()
greet(name)
```

В приведенном выше примере есть несколько несовместимых строк. В Python 3.x метод `raw_input()` был заменен на `input()`, а `print` теперь не оператор, а функция. Этот код может быть преобразован в код Python 3.x с помощью инструмента `2to3`.

Unix

\$ `2to3 example.py`

Windows

> `path/to/2to3.py example.py`

В результате выполнения приведенного выше кода будут выведены различия по сравнению с исходным текстом, как показано ниже.

```
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored example.py
— example.py (original)
+++ example.py (refactored)
@@ -1,5 +1,5 @@
def greet(name):
- print "Hello, {0}!".format(name)
-print "What's your name?"
-name = raw_input()
+ print("Hello, {0}!".format(name))
+ print("What's your name?")
+ name = input()
greet(name)
RefactoringTool: Files that need to be modified:
RefactoringTool: example.py
```

Модификации могут быть записаны обратно в исходный файл с помощью флага `-w`. Резервная копия исходного файла под названием `example.py.bak` будет создана, если флаг `-n` не указан.

Unix

```
$ 2to3 -w example.py
```

Windows

```
> path/to/2to3.py -w example.py
```

Теперь файл `example.py` преобразован из кода Python 2.x в код Python 3.x. После завершения преобразования `example.py` будет содержать следующий корректный код Python3.x:

Версия Python 3.x ≥ 3.0:

```
Python 3.x Version ≥ 3.0
def greet(name):
    print("Hello, {0}!".format(name))
    print("What's your name?")
    name = input()
    greet(name)
```

Глава 151. Неофициальные реализации Python

151.1. IronPython

Это реализация с открытым исходным кодом для .NET и Mono, написанная на языке C# и лицензируемая по лицензии Apache License 2.0. Она опирается на DLR (Dynamic Language Runtime). Поддерживается только версия 2.7, версия 3 находится в стадии разработки.

Отличия от CPython:

- Тесная интеграция с .NET Framework.
- Строки по умолчанию имеют формат Unicode.
- Не поддерживает расширения для CPython, написанные на языке C.
- Не защищает от глобальной блокировки интерпретатора.
- Производительность обычно ниже, хотя это зависит от используемых тестов.

Hello World

```
print "Hello World!"
```

Можно также использовать функции .NET:

```
import clr
from System import Console
Console.WriteLine("Hello World!")
```

Внешние ссылки

- Официальный сайт: <http://ironpython.net/>
- Репозиторий GitHub: <https://github.com/IronLanguages/main>

151.2. Jython

Это реализация с открытым исходным кодом для JVM (Java Virtual Machine), написанная на языке Java и лицензированная по лицензии Python Software Foundation License. Поддерживается только версия 2.7, версия 3 находится в стадии разработки.

Отличия от CPython:

- Тесная интеграция с JVM.
- Строки имеют формат Unicode.
- Не поддерживает расширения для CPython, написанные на языке C.
- Не защищает от глобальной блокировки интерпретатора.
- Производительность обычно ниже, хотя это зависит от тестов.

Hello World

```
print "Hello World!"
```

В Jython можно использовать функции Java:

```
from java.lang import System
System.out.println("Hello World!")
```

Внешние ссылки

- Официальный сайт: <http://www.jython.org/>
- Репозиторий Mercurial: <https://hg.python.org/jython>

151.3. Transcrypt

Transcrypt – это инструмент для предварительной компиляции достаточно обширного подмножества языка Python в компактный, легко читаемый Javascript. Он обладает следующими характеристиками:

- Позволяет осуществлять классическое объектно-ориентированное программирование с множественным наследованием с использованием чистого синтаксиса Python, разбираемого собственным синтаксическим анализатором CPython
- Интеграция с множеством высококачественных веб-ориентированных JavaScript-библиотек, а не десктоп-ориентированных Python-библиотек
- Иерархическая система модулей на основе URL, позволяющая распространять модули через PyPi
- Простая связь между исходным кодом Python и сгенерированным JavaScript-кодом для удобства отладки
- Многоуровневые карты исходного кода и дополнительная аннотация целевого кода со ссылками на исходный текст
- Компактные загрузки (килобайты, а не мегабайты)
- Оптимизированный JavaScript-код, использующий мемоизацию (кэширование вызовов) для опционального обхода цепочки поиска прототипов
- Перегрузка операторов может быть включена и выключена локально для облегчения чтения числовых вычислений

Размер и скорость кода

Опыт показывает, что 650 кБ исходного кода Python примерно переводится в такой же объем исходного кода JavaScript. Скорость соответствует скорости рукописного JavaScript и может превосходить ее, если включена мемоизация вызовов.

Интеграция с HTML

```
<script src="__javascript__/_hello.js"></script>
<h2>Hello demo</h2>
```

```

<p>
<div id = "greet">...</div>
<button onclick="hello.solarSystem.greet ()">Нажмите на меня еще раз!</button>

<p>
<div id = "explain">...</div>
<button onclick="hello.solarSystem.explain ()">И меня тоже нажмите несколько раз!</button>

```

Интеграция с JavaScript и DOM

```
from itertools import chain
```

```

class SolarSystem:
    planets = [list (chain (planet, (index + 1,))) for index, planet in enumerate ((
        ('Mercury', 'hot', 2240),
        ('Venus', 'sulphurous', 6052),
        ('Earth', 'fertile', 6378),
        ('Mars', 'reddish', 3397),
        ('Jupiter', 'stormy', 71492),
        ('Saturn', 'ringed', 60268),
        ('Uranus', 'cold', 25559),
        ('Neptune', 'very cold', 24766)
    ))]

    lines = (
        '{} is a {} planet',
        'The radius of {} is {} km',
        '{} is planet nr. {} counting from the sun'
    )

    def __init__ (self):
        self.lineIndex = 0

    def greet (self):
        self.planet = self.planets [int (Math.random () * len (self.planets))]
        document.getElementById ('greet') .innerHTML = 'Hello {}'.format (self.planet [0])
        self.explain ()

    def explain (self):
        document.getElementById ('explain') .innerHTML = (
            self.lines [self.lineIndex] .format (self.planet [0], self.planet [self.lineIndex + 1])
        )
        self.lineIndex = (self.lineIndex + 1) % 3
        solarSystem = SolarSystem ()

```

Интеграция с другими библиотеками JavaScript

Transcript может быть использован в сочетании с любой библиотекой JavaScript без специальных мер и синтаксиса. В документации приведены примеры для react.js, riot.js, fabric.js и node.js.

Взаимосвязь между Python и JavaScript кодом Python

```

class A:
    def __init__ (self, x):
        self.x = x

    def show (self, label):
        print ('A.show', label, self.x)

```

```

class B:
    def __init__(self, y):
        alert('In B constructor')
        self.y = y

    def show(self, label):
        print('B.show', label, self.y)

class C(A, B):
    def __init__(self, x, y):
        alert('In C constructor')
        A.__init__(self, x)
        B.__init__(self, y)
        self.show('constructor')

    def show(self, label):
        B.show(self, label)
        print('C.show', label, self.x, self.y)

a = A(1001)
a.show('america')

b = B(2002)
b.show('russia')

c = C(3003, 4004)
c.show('netherlands')

show2 = c.show
show2('copy')

```

JavaScript

```

var A = __class__('A', [object], {
    get __init__() {return __get__(this, function(self, x) {
        self.x = x;
    }}),
    get show() {return __get__(this, function(self, label) {
        print('A.show', label, self.x);
    }});
});

var B = __class__('B', [object], {
    get __init__() {return __get__(this, function(self, y) {
        alert('In B constructor');
        self.y = y;
    }}),
    get show() {return __get__(this, function(self, label) {
        print('B.show', label, self.y);
    }});
});

var C = __class__('C', [A, B], {
    get __init__() {return __get__(this, function(self, x, y) {
        alert('In C constructor');
        A.__init__(self, x);
        B.__init__(self, y);
        self.show('constructor');
    }}),
    get show() {return __get__(this, function(self, label) {
        B.show(self, label);
        print('C.show', label, self.x, self.y);
    }});
});

```



```
var a = A (1001);
a.show ('america');
var b = B (2002);
b.show ('russia');
var c = C (3003, 4004);
c.show ('netherlands');
var show2 = c.show;
show2 ('copy');
```

Внешние ссылки

- Официальный сайт: <http://www.transcript.org/>
- Репозиторий GitHub: <https://github.com/JdeH/Transcript>

Глава 152. Абстрактное синтаксическое дерево

152.1. Анализ функций в скрипте Python

Приведенный ниже код анализирует скрипт Python и для каждой удаленной функции сообщает номер строки, где начинается функция, где заканчивается сигнатура, где заканчивается docstring и где заканчивается описание функции.

```
#!/usr/local/bin/python3
```

```
import ast
import sys
```

```
""" Данные, которые мы собираем. Каждый ключ - это имя функции; каждое значение - это
словарь с ключами: firstline, sigend, docend и lastline и значениями номеров строк, где это
происходит. """
functions = {}
```

```
def process(functions):
    """ Работа с данными функций, хранящимися в функциях. """
    for funcname,data in functions.items():
        print("function:",funcname)
        print("\tstarts at line:",data['firstline'])
        print("\tsignature ends at line:",data['sigend'])
        if ( data['sigend'] < data['docend'] ):
            print("\tdocstring ends at line:",data['docend'])
        else:
            print("\tno docstring")
        print("\tfunction ends at line:",data['lastline'])
        print()
```

```
class FuncLister(ast.NodeVisitor):
    def visit_FunctionDef(self, node):
        """ Рекурсивно посетить все функции, определяя, где начинается каждая функция, где
        заканчивается ее сигнатура, где заканчивается docstring и где заканчивается
        сама функция. """
        functions[node.name] = {'firstline':node.lineno}
        sigend = max(node.lineno,lastline(node.args))
        functions[node.name]['sigend'] = sigend
        docstring = ast.get_docstring(node)
```

```

docstringlength = len(docstring.split("\n")) if docstring else -1
functions[node.name]['docend'] = sigend+docstringlength
functions[node.name]['lastline'] = lastline(node)
self.generic_visit(node)

def lastline(node):
    """ Рекурсивно найти последнюю строку узла """
    return max( [ node.lineno if hasattr(node,'lineno') else -1 , ]
               +[lastline(child) for child in ast.iter_child_nodes(node)] )

def readin(pythonfilename):
    """ Считать имя файла и сохранить данные функции в функции """
    with open(pythonfilename) as f:
        code = f.read()
    FuncLister().visit(ast.parse(code))

def analyze(file,process):
    """ Чтение файла и обработка данных функции """
    readin(file)
    process(functions)

if __name__ == '__main__':
    if len(sys.argv)>1:
        for file in sys.argv[1:]:
            analyze(file,process)
    else:
        analyze(sys.argv[0],process)

```

Глава 153. Unicode и bytes

Параметр	Подробности
encoding	Используемая кодировка, например, 'ascii', 'utf8' и т. д.
errors	Режим ошибок, например, 'replace' для замены "плохих" символов на знаки вопроса, 'ignore' для игнорирования "плохих" символов и т. д.

153.1. Обработка ошибок кодирования и декодирования

И .encode и .decode имеют режимы ошибок.

По умолчанию используется режим 'strict', который вызывает исключения при ошибках. Другие режимы более щадящие.

Кодирование

```

>>> "£13.55".encode('ascii', errors='replace')
b'?13.55'
>>> "£13.55".encode('ascii', errors='ignore')
b'13.55'
>>> "£13.55".encode('ascii', errors='namereplace')
b'\N{POUND SIGN}13.55'
>>> "£13.55".encode('ascii', errors='xmlcharrefreplace')
b'&#163;13.55'
>>> "£13.55".encode('ascii', errors='backslashreplace')
b'\\xa313.55'

```

Декодирование

```
>>> b = "£13.55".encode('utf8')
>>> b.decode('ascii', errors='replace')
'??13.55'
>>> b.decode('ascii', errors='ignore')
'13.55'
>>> b.decode('ascii', errors='backslashreplace')
'\xc2\xa313.55'
```

Вывод

Из вышесказанного ясно, что при работе с Unicode и bytes крайне важно соблюдать правильность кодировок.

153.2. Файловый ввод/вывод

Файлы, открытые в недвоичном режиме (например, 'r' или 'w'), работают со строками. По умолчанию используется кодировка 'utf8'.

```
open(fn, mode='r') # открывает файл для чтения в формате utf8
open(fn, mode='r', encoding='utf16') # открывает файл для чтения в формате utf16
# ERROR: невозможно записать байты, когда ожидается строка
open("foo.txt", "w").write(b"foo")
```

Файлы, открытые в двоичном режиме (например, 'rb' или 'wb'), работают с байтами. Аргумент кодировки не может быть задан, так как кодировки не существует.

```
open(fn, mode='wb') # открыть файл для записи байтов
# ERROR: невозможно записать строку, когда ожидаются байты
open(fn, mode='wb').write("hi")
```

153.3. Основы

В Python 3 str – это тип для строк с поддержкой Unicode, а bytes – тип для последовательностей “сырых”, необработанных байтов (raw bytes).

```
type("f") == type(u"f") # True, <class 'str'>
type(b"f") # <class 'bytes'>
```

В Python 2 обычная строка по умолчанию представляла собой последовательность необработанных байтов, а строкой Unicode была любая строка с префиксом "u".

```
type("f") == type(b"f") # True, <type 'str'>
type(u"f") # <type 'unicode'>
```

Перевод Unicode в байты

Строки Unicode могут быть преобразованы в байты с помощью функции .encode(encoding).

Python 3

```
>>> "£13.55".encode('utf8')
b'\xc2\xa313.55'
>>> "£13.55".encode('utf16')
b'\xff\xfe\xa3\x001\x003\x00.\x005\x005\x00'
```

Python 2

Кодировка консоли по умолчанию равна sys.getdefaultencoding() == 'ascii', а не utf-8, как в Python 3, поэтому ее вывод напрямую, как в предыдущем примере, невозможен.

```
>>> print type(u"£13.55".encode('utf8'))
<type 'str'>
>>> print u"£13.55".encode('utf8')
SyntaxError: Non-ASCII character '\xc2' in...
```

```
# с кодировкой, установленной внутри файла
# -*- coding: utf-8 -*-
>>> print u"£13.55".encode('utf8')
?ú13.55
```

Если кодировка не может обработать строку, то выдается ошибка `UnicodeEncodeError`:

Перевод байтов в Unicode

Байты могут быть преобразованы в строки Unicode с помощью функции `.decode(encoding)`. Последовательность байтов может быть преобразована в строку юникода только с помощью соответствующей кодировки!

```
>>> b'\xc2\xa313.55'.decode('utf8')
'£13.55'
```

Если кодировка не может обработать строку, то выдается ошибка `UnicodeDecodeError`:

```
>>> b'\xc2\xa313.55'.decode('utf16')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/csaftoiu/csaftoiu-github/yahoo-groupsbackup/
virtualenv/bin/../lib/python3.5/encodings/utf_16.py", line 16, in decode
    return codecs.utf_16_decode(input, errors, True)
UnicodeDecodeError: 'utf-16-le' codec can't decode byte 0x35 in position 6: truncated data (кодк
'utf-16-le' не может декодировать байт 0x35 в позиции 6: усеченные данные)
```

Глава 154. Последовательное соединение в Python (pyserial)

Параметр	Подробности
port	Имя устройства, например, <code>/dev/ttyUSB0</code> в GNU/Linux или COM3 в Windows
baudrate	Тип скорости передачи: int по умолчанию: 9600; стандартные значения: 50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 115200

154.1. Инициализация последовательного устройства

```
import serial
#Serial принимает эти два параметра: последовательное устройство и скорость передачи данных
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

154.2. Чтение из последовательного порта

Инициализация последовательного устройства:

```
import serial
#Serial принимает два параметра: последовательное устройство и скорость передачи данных
ser = serial.Serial('/dev/ttyUSB0', 9600)
```

считывание одного байта с последовательного устройства

```
data = ser.read()
```

считывание заданного количества байт с последовательного устройства

```
data = ser.read(size=5)
```

для чтения одной строки из последовательного устройства.

```
data = ser.readline()
```

считывать данные с последовательного устройства в то время, когда на него что-то записывается.

```
data = ser.read(ser.inWaiting())    #для Python 2.7
ser.read(ser.inWaiting())          #для Python 3
```

154.3. Проверка доступности последовательных портов

Для получения списка доступных последовательных портов используйте

```
python -m serial.tools.list_ports
```

в командной строке или

```
from serial.tools import list_ports
```

```
list_ports.comports() # Выводит список доступных последовательных портов
```

из оболочки Python.

Глава 155. Работа с Neo4j и Cypher с использованием библиотеки Py2Neo

155.1. Добавление узлов в граф в графовой СУБД Neo4j

```
results = News.objects.today_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    article.properties["title"] = results[r]['news_title']
    article.properties["timestamp"] = results[r]['news_timestamp']
    article.push()
[...]
```

Добавление узлов в граф является довольно простой задачей, важен `graph.merge_one`, так как он предотвращает дублирование. (Если запустить скрипт дважды, то во второй раз он обновит заголовок и не будет создавать новые узлы для одних и тех же артикулов (articles)). `timestamp` должен быть целым числом, а не строкой даты, так как в Neo4j нет типа данных `date`. Это приводит к проблемам сортировки при хранении даты в виде `'05-06-1989'`. `article.push()` – это вызов, который фактически фиксирует операцию в Neo4j. Не забывайте об этом шаге.

155.2. Импорт и аутентификация

```
from py2neo import authenticate, Graph, Node, Relationship
authenticate("localhost:7474", "neo4j", "<pass>")
graph = Graph()
```

Необходимо убедиться, что база данных Neo4j существует по адресу `localhost:7474` с соответствующими учетными данными. Объект графа является вашим интерфейсом к экземпляру Neo4j в остальном коде на языке Python. Вместо того чтобы делать его глобальной переменной, лучше хранить его в методе `__init__` класса.

155.3. Добавление отношений в граф Neo4j

```
results = News.objects.today_news()
for r in results:
    article = graph.merge_one("NewsArticle", "news_id", r)
    if 'LOCATION' in results[r].keys():
        for loc in results[r]['LOCATION']:
```

```

loc = graph.merge_one("Location", "name", loc)
try:
    rel = graph.create_unique(Relationship(article, "about_place", loc))
except Exception, e:
    print e

```

Для исключения дубликатов важно использовать `create_unique`; в целом операция довольно проста. Имя отношения также важно, поскольку его можно использовать в расширенных случаях.

155.4. Запрос 1: автозаполнение заголовков новостей

```

def get_autocomplete(text):
    query = """
    start n = node(*) where n.name =~ '(?i)%s.*' return n.name,labels(n) limit 10;
    """

    query = query % (text)
    obj = []
    for res in graph.cypher.execute(query):
        # print res[0],res[1]
        obj.append({'name':res[0],'entity_type':res[1]})
    return res

```

Это пример Cypher-запроса для получения всех узлов со свойством `name`, которое начинается с аргумента `text`.

155.5. Запрос 2: получение новостных статей по местоположению на определенную дату

```

def search_news_by_entity(location,timestamp):
    query = """
    MATCH (n)-[]->(l)
    where l.name='%s' and n.timestamp='%s'
    RETURN n.news_id limit 10
    """

    query = query % (location,timestamp)

    news_ids = []
    for res in graph.cypher.execute(query):
        news_ids.append(str(res[0]))

    return news_ids

```

С помощью этого запроса можно найти все новостные статьи (`n`), связанные отношениями с местом (`l`).

155.6. Образцы запросов языка Cypher

Подсчет статей, связанных с конкретным человеком, с течением времени

```

MATCH (n)-[]->(l)
where l.name='Donald Trump'
RETURN n.date,count(*) order by n.date

```

Поиск других людей / местностей, связанных с теми же новостными статьями, что и Трамп, с не менее чем 5 общими узлами взаимосвязи.

```

MATCH (n:NewsArticle)-[]->(l)
where l.name='Donald Trump'
MATCH (n:NewsArticle)-[]->(m)
with m,count(n) as num where num>5
return labels(m)[0],(m.name), num order by num desc limit 10

```

Глава 156. Основы работы с библиотекой Curses в Python

156.1. Вспомогательная функция wrapper()

Библиотека Curses предоставляет вспомогательную функцию `wrapper(func, ...)`. В приведенном примере `wrapper` инициализирует `curses`, создает `stdscr`, `WindowObject` и передает в `func` как `stdscr`, так и все дополнительные аргументы. Когда `func` вернется, `wrapper` восстановит терминал перед выходом из программы.

```
main(scr, *args):
    # -- Выполнить действие с экраном --
    scr.border(0)
    scr.addstr(5, 5, 'Привет от Curses!', curses.A_BOLD)
    scr.addstr(6, 5, 'Нажмите q, чтобы закрыть этот экран', curses.A_NORMAL)

    while True:
        # оставаться в этом цикле до тех пор, пока пользователь не нажмет 'q'
        ch = scr.getch()
        if ch == ord('q'):

curses.wrapper(main)
```

156.2. Пример базового обращения

```
import curses
import traceback

try:
    # -- Инициализация... --
    stdscr = curses.initscr() # инициализировать экран curses
    curses.noecho()           # отключить вывод на экран эхо-сигнала о нажатии клавиш
    curses.cbreak()           # вход в режим прерывания при нажатии клавиши Enter
                                # после нажатия клавиши не требуется для регистрации
    stdscr.keypad(1)          # включить специальные значения клавиш, как curses.KEY_LEFT и т. д.

    # -- Выполнить действие с экраном --
    stdscr.border(0)
    stdscr.addstr(5, 5, 'Привет от Curses!', curses.A_BOLD)
    stdscr.addstr(6, 5, 'Нажмите q, чтобы закрыть этот экран', curses.A_NORMAL)

    while True:
        # оставаться в этом цикле до тех пор, пока пользователь не нажмет 'q'
        ch = stdscr.getch()
        if ch == ord('q'):
            break

    # -- End of user code --

except:
    traceback.print_exc() # вывести журнал трассировки ошибки

finally:
    # -- Очистка при выходе ---
    stdscr.keypad(0)
    curses.echo()
    curses.nocbreak()
    curses.endwin()
```

Глава 157. Шаблоны в Python

157.1. Простая программа вывода данных с использованием шаблона

```
from string import Template
```

```
data = dict(item = "конфеты", price = 8, qty = 2)
```

```
# определение шаблона (template)
```

```
t = Template("Саймон купил $qty $item за $price долларов")
print(t.substitute(data))
```

Результат:

Саймон купил 2 конфеты за 8 долларов

Шаблоны поддерживают подстановки на основе \$ вместо подстановок на основе %. Substitute (mapping, keywords) выполняет подстановку шаблона, возвращая новую строку.

Отображение (mapping) – это любой словареподобный объект с ключами, совпадающими с заполнителями шаблонов. В данном примере в качестве заполнителей (местодержатели, placeholders) выступают price и qty. Аргументы ключевых слов также могут использоваться в качестве заполнителей. Если оба аргумента присутствуют, то приоритет имеют заполнители из ключевых слов.

157.2. Изменение разделителя

Разделитель "\$" можно заменить на любой другой. Например:

```
from string import Template
```

```
class MyOtherTemplate(Template):
    delimiter = "#"
```

```
data = dict(id = 1, name = "Ricardo")
```

```
t = MyOtherTemplate("My name is #name and I have the id: #id")
print(t.substitute(data))
```

Глава 158. Библиотека Pillow

158.1. Чтение файла изображения

```
from PIL import Image
```

```
im = Image.open("Image.bmp")
```

158.2. Преобразование файла в формат JPEG

```
from __future__ import print_function
```

```
import os, sys
```

```
from PIL import Image
```

```
for infile in sys.argv[1:]:
```

```
    f, e = os.path.splitext(infile)
```

```
    outfile = f + ".jpg"
```

```
    if infile != outfile:
```

```
        try:
```

```
            Image.open(infile).save(outfile)
```

```
        except IOError:
```

```
            print("cannot convert", infile)
```


Глава 159. Оператор pass

159.1. Игнорирование исключения

```
try:
    metadata = metadata['properties']
except KeyError:
    pass
```

159.2. Создание нового исключения, которое может быть перехвачено

```
class CompileError(Exception):
    pass
```

Глава 160. Подкоманды CLI (Command Line Interface) для аккуратного вывода справочной информации

Тема разбора аргументов командной строки относится к более широкой теме разбора аргументов.

160.1. Способ без использования библиотек

```
usage: sub <command>
```

```
commands:
```

```
status - show status
list   - print list
```

```
import sys
```

```
def check():
    print("status")
    return 0
```

```
if sys.argv[1:] == ['status']:
    sys.exit(check())
elif sys.argv[1:] == ['list']:
    print("list")
else:
    print(__doc__.strip())
```

Вывод без аргументов:

```
usage: sub
```

```
commands:
```

```
status - show status
list - print list
```

Плюсы:

- без библиотек
- каждый может прочитать
- полный контроль над форматированием справки

160.2. Использование argparse (средство форматирования справки по умолчанию)

```
import argparse
import sys
```

```
def check():
    print("status")
    return 0
```

```
parser = argparse.ArgumentParser(prog="sub", add_help=False)
subparser = parser.add_subparsers(dest="cmd")
```

```
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')
```

```
# хак для отображения справки при отсутствии аргументов
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)
```

```
args = parser.parse_args()
```

```
if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Вывод без аргументов:

```
usage: sub {status,list} ...
```

positional arguments:

```
{status,list}
status          show status
list            print list
```

Плюсы:

- поставляется с Python
- включен парсинг опций

160.3. Использование argparse (создание пользовательского средства форматирования справки)

```
import argparse
import sys
```

```
class CustomHelpFormatter(argparse.HelpFormatter):
    def _format_action(self, action):
        if type(action) == argparse._SubParsersAction:
```

```
# ввести новую переменную класса для форматирования подкоманд
subactions = action._get_subactions()
invocations = [self._format_action_invocation(a) for a in subactions]
self._subcommand_max_length = max(len(i) for i in invocations)
```

```
if type(action) == argparse._SubParsersAction._ChoicesPseudoAction:
    # форматирование строки справки по подкомандам
    subcommand = self._format_action_invocation(action) # type: str
    width = self._subcommand_max_length
    help_text = ""
    if action.help:
        help_text = self._expand_help(action)
    return " {:width}} - {}{}\n".format(subcommand, help_text, width=width)
```

```
elif type(action) == argparse._SubParsersAction:
    # раздел справки по подкомандам процесса
    msg = '\n'
    for subaction in action._get_subactions():
        msg += self._format_action(subaction)
    return msg
else:
```

```
    return super(CustomHelpFormatter, self)._format_action(action)
```

```
def check():
    print("status")
    return 0
```

```
parser = argparse.ArgumentParser(usage="sub <command>", add_help=False,
    formatter_class=CustomHelpFormatter)
```

```
subparser = parser.add_subparsers(dest="cmd")
subparser.add_parser('status', help='show status')
subparser.add_parser('list', help='print list')
```

```
# пользовательское сообщение о помощи
parser._positionals.title = "commands"
```

```
# хак для отображения справки при отсутствии аргументов
```

```
if len(sys.argv) == 1:
    parser.print_help()
    sys.exit(0)
```

```
args = parser.parse_args()
```

```
if args.cmd == 'list':
    print('list')
elif args.cmd == 'status':
    sys.exit(check())
```

Вывод без аргументов:

```
usage: sub <command>
```

```
commands:
```

```
status - show status
list   - print list
```

Глава 161. Доступ к базам данных

161.1. SQLite

SQLite – это легкая дисковая база данных. Поскольку она не требует отдельного сервера баз данных, ее часто используют для создания прототипов или для небольших приложений, которые часто используются одним пользователем или одним пользователем в определенный момент времени.

```
import sqlite3

conn = sqlite3.connect("users.db")
c = conn.cursor()

c.execute("CREATE TABLE user (name text, age integer)")

c.execute("INSERT INTO user VALUES ('User A', 42)")
c.execute("INSERT INTO user VALUES ('User B', 43)")

conn.commit()

c.execute("SELECT * FROM user")
print(c.fetchall())

conn.close()
```

Приведенный выше код подключается к базе данных, хранящейся в файле с именем users.db, создавая файл, если он еще не существует. Взаимодействовать с базой данных можно с помощью операторов SQL. Результат этого примера:

```
[(u'User A', 42), (u'User B', 43)]
```

Синтаксис SQLite: Углубленный анализ

Начало работы

1. Импортируйте модуль sqlite с помощью

```
>>> import sqlite3
```

2. Для использования модуля необходимо сначала создать объект Connection, представляющий базу данных. Здесь данные будут храниться в файле example.db:

```
>>> conn = sqlite3.connect('users.db')
```

Кроме того, для создания временной базы данных в оперативной памяти можно задать специальное имя :memory:, как показано ниже:

```
>>> conn = sqlite3.connect(':memory:')
```

3. Теперь можно создать объект Cursor и вызвать его метод execute() для выполнения SQL-команд:

```
c = conn.cursor()
```

```
# Создать таблицу
```

```
c.execute("""CREATE TABLE stocks
            (date text, trans text, symbol text, qty real, price real)""")
```

```
# Вставить строку данных
```

```
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
```

```
# Сохранить (зафиксировать) изменения
```

```
conn.commit()
```

Мы также можем закрыть соединение, если закончили с ним работать.

Убедитесь, что все изменения были зафиксированы, иначе они будут потеряны `conn.close()`

Важные атрибуты и функции соединения (Connection)

1. `isolation_level` – это атрибут, используемый для получения или установки текущего уровня изоляции. Нет для режима автокоммита или один из `DEFERRED`, `IMMEDIATE` или `EXCLUSIVE`.

2. `cursor` – этот объект используется для выполнения команд и запросов SQL.

3. `commit()` – выполняет фиксацию текущей транзакции.

4. `rollback()` – откатывает все изменения, сделанные с момента предыдущего вызова `commit()`.

5. `close()` – закрывает соединение с базой данных. При этом не происходит автоматического вызова `commit()`. Если вызвать `close()` без предварительного вызова `commit()` (при условии, что вы не находитесь в режиме `autocommit`), то все сделанные изменения будут потеряны.

6. `total_changes` – атрибут, регистрирующий общее количество строк, модифицированных, удаленных или вставленных с момента открытия базы данных.

7. `execute`, `executemany` и `executescript` – эти функции выполняются так же, как и функции объекта `cursor`. Это сокращение, поскольку вызов этих функций через объект соединения приводит к созданию промежуточного объекта курсора и вызову соответствующего метода объекта курсора.

8. `row_factory` – этот атрибут можно заменить на вызываемый элемент, который будет принимать курсор и исходную строку в виде кортежа и возвращать реальную строку результата.

Важные функции курсора

1. `execute(sql[, parameters])` – выполняет один SQL-оператор. SQL-оператор может быть параметризован (т.е. вместо SQL-литералов могут использоваться заполнители). Модуль `sqlite3` поддерживает два вида заполнителей: вопросительные знаки `?` (“стиль `qmark`”) и именованные заполнители `:name` (“именованный стиль”).

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.execute("create table people (name, age)")
```

```
who = "Sophia"
age = 37
# стиль qmark:
cur.execute("insert into people values (?, ?)",
            (who, age))
```

```
# именованный стиль:
cur.execute("select * from people where name=:who and age=:age",
            {"who": who, "age": age}) # the KEYS correspond TO the placeholders IN SQL
```

```
print(cur.fetchone())
```

Осторожно: не используйте `%` для вставки строк в SQL-команды, так как это может сделать вашу программу уязвимой для кибератаки внедрения SQL-кода (SQL injection).

2. `executemany(sql, seq_of_parameters)` – выполняет команду SQL по всем последовательностям параметров или сопоставлениям, найденным в последовательности `sql`. Модуль `sqlite3` также позволяет использовать вместо последовательности итератор, выдающий параметры.

```
L = [(1, 'abcd', 'dfj', 300), # Список кортежей, которые необходимо вставить в базу данных
     (2, 'cfgd', 'dyfj', 400),
     (3, 'sdd', 'dfjh', 300.50)]
```

```
conn = sqlite3.connect("test1.db")
conn.execute("create table if not exists book (id int, name text, author text, price real)")
conn.executemany("insert into book values (?, ?, ?, ?)", L)
```

```
for row in conn.execute("select * from book"):
    print(row)
```

В качестве параметра функции `executemany` можно передавать объекты-итераторы, и функция будет выполнять итерацию по каждому кортежу значений, который возвращает итератор. Итератор должен возвращать кортеж значений.

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')

    def __iter__(self):
        return self

    def __next__(self):      # (use next(self) for Python 2)
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),)

conn = sqlite3.connect("abc.db")
cur = conn.cursor()
cur.execute("create table characters(c)")

thelster = IterChars()
cur.executemany("insert into characters(c) values (?)", thelster)

rows = cur.execute("select c from characters")
for row in rows:
    print(row[0]),
```

3. `executescript(sql_script)` – это нестандартный удобный метод для одновременного выполнения нескольких SQL-операторов. Сначала он выдает оператор `COMMIT`, а затем выполняет сценарий SQL, который получает в качестве параметра. `sql_script` может быть экземпляром типов `str` или `bytes`.

```
import sqlite3
conn = sqlite3.connect(":memory:")
cur = conn.cursor()
cur.executescript("""
    create table person(
        firstname,
        lastname,
        age
    );

    create table book(
        title,
        author,
        published
    );

    insert into book(title, author, published)
    values (
        'Dirk Gently's Holistic Detective Agency',
        'Douglas Adams',
        1987
    );
""")
```

Следующий набор функций используется совместно с операторами `SELECT` в SQL. Для получения данных после выполнения оператора `SELECT` можно либо рассматривать кур-

сор как итератор, либо вызвать метод `fetchone()` курсора для получения одной совпадающей строки, либо вызвать `fetchall()` для получения списка совпадающих строк.

Пример с итератором:

```
import sqlite3
stocks = [('2006-01-05', 'BUY', 'RHAT', 100, 35.14),
          ('2006-03-28', 'BUY', 'IBM', 1000, 45.0),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.0),
          ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
conn = sqlite3.connect(":memory:")
conn.execute("create table stocks (date text, buysell text, symb text, amount int, price real)")
conn.executemany("insert into stocks values (?, ?, ?, ?, ?)", stocks)
cur = conn.cursor()

for row in cur.execute("SELECT * FROM stocks ORDER BY price"):
    print(row)
```

```
# Выходные данные:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

4. `fetchone()` – получает следующую строку из набора результатов запроса, возвращая единственную последовательность, или `None`, если больше нет данных.

```
cur.execute("SELECT * FROM stocks ORDER BY price")
i = cur.fetchone()
while(i):
    print(i)
    i = cur.fetchone()
```

```
# Выходные данные:
# ('2006-01-05', 'BUY', 'RHAT', 100, 35.14)
# ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)
# ('2006-04-06', 'SELL', 'IBM', 500, 53.0)
# ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)
```

5. `fetchmany(size=cursor.arraysize)` – отбирает очередной набор строк результата запроса (заданный размером), возвращая список. Если размер не указан, то `fetchmany` возвращает один ряд. Пустой список возвращается, если больше нет доступных строк.

```
cur.execute("SELECT * FROM stocks ORDER BY price")
print(cur.fetchmany(2))
```

```
# Выходные данные:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0)]
```

6. `fetchall()` – выбирает все (оставшиеся) строки результата запроса, возвращая список.

```
cur.execute("SELECT * FROM stocks ORDER BY price")
print(cur.fetchall())
```

```
# Выходные данные:
# [('2006-01-05', 'BUY', 'RHAT', 100, 35.14), ('2006-03-28', 'BUY', 'IBM', 1000, 45.0), ('2006-04-06', 'SELL',
'IBM', 500, 53.0), ('2006-04-05', 'BUY', 'MSFT', 1000, 72.0)]
```

Типы данных SQLite и Python

SQLite нативно поддерживает следующие типы: `NULL`, `INTEGER`, `REAL`, `TEXT`, `BLOB`.

Преобразование типов данных при переходе от SQL к Python или наоборот производится следующим образом:

None	<->	NULL
int	<->	INTEGER/INT
float	<->	REAL/FLOAT
str	<->	TEXT/VARCHAR(n)
bytes	<->	BLOB

161.2. Доступ к базе данных MySQL с помощью MySQLdb

Прежде всего необходимо создать соединение с базой данных с помощью метода `connect`. После этого необходимо создать курсор, который будет работать с этим соединением. Для взаимодействия с базой данных используйте метод `execute` курсора, время от времени фиксируйте изменения с помощью метода `commit` объекта соединения.

Когда все будет сделано, не забудьте закрыть курсор и соединение. Ниже приведен класс `Dbconnect`, содержащий все необходимое.

```
import MySQLdb

class Dbconnect(object):

    def __init__(self):

        self.dbconnection = MySQLdb.connect(host='host_example',
                                             port=int('port_example'),
                                             user='user_example',
                                             passwd='pass_example',
                                             db='schema_example')
        self.dbcursor = self.dbconnection.cursor()

    def commit_db(self):
        self.dbconnection.commit()

    def close_db(self):
        self.dbcursor.close()
        self.dbconnection.close()
```

Взаимодействие с базой данных является очень простым. После создания объекта достаточно воспользоваться методом `execute`.

```
db = Dbconnect()
db.dbcursor.execute('SELECT * FROM %s' % 'table_example')
```

Если вы хотите вызвать хранимую процедуру, используйте следующий синтаксис. Обратите внимание, что список параметров является необязательным.

```
db = Dbconnect()
db.callproc('stored_procedure_name', [parameters])
```

После выполнения запроса можно получить доступ к результатам различными способами. Объект курсора представляет собой генератор, который может получить все результаты или быть заиклен.

```
results = db.dbcursor.fetchall()
for individual_row in results:
    first_field = individual_row[0]
```

Если вы хотите получить цикл, использующий непосредственно генератор:

```
for individual_row in db.dbcursor:
    first_field = individual_row[0]
```

Если вы хотите зафиксировать изменения в базе данных:

```
db.commit_db()
```

Если необходимо закрыть курсор и соединение:

```
db.close_db()
```


161.3. Соединение

Создание соединения

Согласно PEP 249, соединение с базой данных должно устанавливаться с помощью конструктора `connect()`, который возвращает объект `Connection`. Аргументы этого конструктора зависят от базы данных. Соответствующие аргументы следует искать в спецификациях баз данных.

```
import MyDBAPI
con = MyDBAPI.connect(*database_dependent_args)
```

Объект соединения имеет четыре метода:

1: close

```
con.close()
```

Мгновенно закрывает соединение. Обратите внимание, что соединение автоматически закрывается, если вызывается метод `Connection.__del__`. Произойдет неявный откат всех незавершенных транзакций.

2: commit

```
con.commit()
```

Выполняет фиксацию всех незавершенных транзакций в базе данных.

3: rollback

```
con.rollback()
```

Откат к началу любой ожидающей транзакции. Другими словами, отменяется любая незафиксированная транзакция в базе данных.

4: cursor

Возвращает объект `Cursor`. Используется для выполнения транзакций над базой данных.

161.4. Доступ к базе данных PostgreSQL с помощью psycopg2

`psycopg2` – самый популярный адаптер баз данных PostgreSQL, который одновременно является легким и удобным. Это текущая реализация адаптера PostgreSQL.

Его основными особенностями являются полная реализация спецификации Python DB API 2.0 и потокобезопасность (несколько потоков могут использовать одно и то же соединение).

Установление соединения с базой данных и создание таблицы

```
import psycopg2
```

```
# Установить соединение с базой данных.
```

```
# Замените значения параметров на учетные данные базы данных.
```

```
conn = psycopg2.connect(database="testpython",
                        user="postgres",
                        host="localhost",
                        password="abc123",
                        port="5432")
```

```
# Создать курсор. Курсор позволяет выполнять запросы к базе данных.
```

```
cur = conn.cursor()
```

```
# Создать таблицу. Инициализируйте имя таблицы, имена столбцов и тип данных.
```

```
cur.execute("""CREATE TABLE FRUITS (
            id      INT,
            fruit_name TEXT,
```

```

        color    TEXT,
        price    REAL
    )"""
conn.commit()
conn.close()

```

Вставка данных в таблицу:

После создания таблицы, как показано выше, заполните ее значениями.

```
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
VALUES (1, 'Apples', 'green', 1.00)""")
```

```
cur.execute("""INSERT INTO FRUITS (id, fruit_name, color, price)
VALUES (1, 'Bananas', 'yellow', 0.80)""")
```

Получение данных таблицы:

Создать запрос и выполнить его

```
cur.execute("""SELECT id, fruit_name, color, price
FROM fruits""")
```

Получить данные

```
rows = cur.fetchall()
```

Выполнить действия с данными

```
for row in rows:
    print "ID = {}".format(row[0])
    print "FRUIT NAME = {}".format(row[1])
    print("COLOR = {}".format(row[2]))
    print("PRICE = {}".format(row[3]))
```

Выходными данными будут:

```
ID = 1
NAME = Apples
COLOR = green
PRICE = 1.0
```

```
ID = 2
NAME = Bananas
COLOR = yellow
PRICE = 0.8
```

161.5. Работа с базами данных Oracle

Предварительные требования:

- Пакет cx_Oracle – смотрите все версии по адресу https://pypi.python.org/pypi/cx_Oracle/5.2.1
- Мгновенный клиент Oracle instant client – для Windows x64 (<http://www.oracle.com/technetwork/topics/winx64soft-089540.html>), Linux x64 (<http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>)

Установка:

- Установите пакет cx_Oracle при помощи:

```
sudo rpm -i <YOUR_PACKAGE_FILENAME>
```

- Извлеките клиент Oracle instant client и установите переменные окружения:

```
ORACLE_HOME=<PATH_TO_INSTANTCLIENT>
PATH=$ORACLE_HOME:$PATH
LD_LIBRARY_PATH=<PATH_TO_INSTANTCLIENT>:$LD_LIBRARY_PATH
```

Создание соединения:

```
import cx_Oracle

class OraExec(object):
    _db_connection = None
    _db_cur = None

    def __init__(self):
        self._db_connection =
            cx_Oracle.connect('<USERNAME>/<PASSWORD>@<HOSTNAME>:<PORT>/<SERVICE_NAME>')
        self._db_cur = self._db_connection.cursor()
```

Получение версии базы данных:

```
ver = con.version.split(".")
print ver
```

Пример вывода: ['12', '1', '0', '2', '0']

Выполнить запрос: SELECT

```
_db_cur.execute("select * from employees order by emp_id")
for result in _db_cur:
    print result
```

Выходные данные будут представлены в виде кортежей Python:

```
(10, 'SYSADMIN', 'IT-INFRA', 7)
(23, 'HR ASSOCIATE', 'HUMAN RESOURCES', 6)
```

Выполнить запрос: INSERT

```
_db_cur.execute("insert into employees(emp_id, title, dept, grade)
                values (31, 'MTS', 'ENGINEERING', 7)
_db_connection.commit()
```

Когда вы выполняете операции вставки, обновления или удаления в Oracle Database, изменения доступны только в пределах вашей сессии до тех пор, пока не будет выполнена фиксация (commit). Когда обновленные данные фиксируются в базе данных, они становятся доступными для других пользователей и сеансов.

Выполнить запрос: INSERT с использованием связанных переменных Bind variables

Привязка переменных позволяет повторно выполнять операторы с новыми значениями, не прибегая к повторному разбору оператора. Это повышает удобство повторного использования кода и снижает риск атак типа SQL Injection.

```
rows = [ (1, "First" ),
          (2, "Second" ),
          (3, "Third" ) ]
_db_cur.bindarraysize = 3
_db_cur.setinputsizes(int, 10)
_db_cur.executemany("insert into mytab(id, data) values (:1, :2)", rows)
_db_connection.commit()
```

Закрытие соединения

```
_db_connection.close()
```

Метод close() закрывает соединение. Все соединения, не закрытые явным образом, будут автоматически освобождены при завершении работы скрипта.

161.6. Использование библиотеки sqlalchemy

Использование sqlalchemy для базы данных:

```
from sqlalchemy import create_engine
from sqlalchemy.engine.url import URL
```

```
url = URL(drivename='mysql',
          username='user',
          password='passwd',
          host='host',
          database='db')
```

```
engine = create_engine(url) # движок sqlalchemy
```

Теперь этот движок можно использовать: например, вместе с Pandas для получения данных из mysql.

```
import pandas as pd
```

```
con = engine.connect()
dataframe = pd.read_sql(sql=query, con=con)
```

Глава 162. Подключение Python к SQL Server

162.1. Подключение к серверу, создание таблицы, запрос данных

Установите пакет:

```
import pymssql
```

```
SERVER = "servername"
USER = "username"
PASSWORD = "password"
DATABASE = "dbname"
```

```
connection = pymssql.connect(server=SERVER, user=USER,
                             password=PASSWORD, database=DATABASE)
```

```
cursor = connection.cursor() # для доступа к полю как к словарю используйте cursor(as_dict=True)
cursor.execute("SELECT TOP 1 * FROM TableName")
row = cursor.fetchone()
```

```
##### создать таблицу #####
```

```
cursor.execute("""
CREATE TABLE posts (
    post_id INT PRIMARY KEY NOT NULL,
    message TEXT,
    publish_date DATETIME
)
```

```
##### вставить данные в таблицу #####
cursor.execute("""
```

```

INSERT INTO posts VALUES(1, "Hey There", "11.23.2016")
""")
# commit your work to database
connection.commit()

##### итерация результатов #####
cursor.execute("SELECT TOP 10 * FROM posts ORDER BY publish_date DESC")
for row in cursor:
    print("Message: " + row[1] + " | " + "Date: " + row[2])
    # если передать курсору значение as_dict=True
    # print(row["message"])

connection.close()

```

Если ваша работа связана с SQL-выражениями, вы можете делать что угодно, просто передавайте эти выражения в метод `execute` (CRUD-операции). Для получения информации об операторе `with`, вызове хранимой процедуры, обработке ошибок и т. д. обратитесь к ресурсу <http://pymssql.org/>.

Глава 163. Реляционная база данных PostgreSQL

163.1. Начало работы

PostgreSQL – активно развивающаяся и зрелая база данных с открытым исходным кодом. Используя модуль `psycopg2`, можно выполнять запросы к этой базе данных.

Установка с помощью `pip`

```
pip install psycopg2
```

Базовое использование

Предположим, что у нас есть таблица `my_table` в базе данных `my_database`, заданная следующим образом.

id	first_name	last_name
1	John	Doe

Мы можем использовать модуль `psycopg2` для выполнения запросов к базе данных следующим образом.

```

import psycopg2

# Установить соединение с существующей базой данных 'my_database', используя
# пользователя 'my_user' с паролем 'my_password'
con = psycopg2.connect("host=localhost dbname=my_database user=my_user password=my_password")

# Создать курсор
cur = con.cursor()

# Вставка записи в 'my_table'
cur.execute("INSERT INTO my_table(id, first_name, last_name) VALUES (2, 'Jane', 'Doe');")

```

```
# Выполнить текущую транзакцию
con.commit()

# Получение всех записей из 'my_table' cur.execute("SELECT * FROM my_table;") results = cur.fetchall()

# Закрыть соединение с базой данных
con.close()

# Вывести результаты
print(results)

# Результат: [(1, 'John', 'Doe'), (2, 'Jane', 'Doe')]
```

Глава 164. Python и Excel

164.1. Чтение данных Excel с помощью модуля xlrd

Библиотека Python xlrd предназначена для извлечения данных из электронных таблиц Microsoft Excel.

Установка

```
pip install xlrd
```

Или можно использовать setup.py из репозитория:

<https://pypi.python.org/pypi/xlrd>.

Чтение таблицы Excel

Импортируйте модуль xlrd и откройте таблицу Excel с помощью метода open_workbook().

```
import xlrd
book=xlrd.open_workbook('sample.xlsx')
```

Проверка количества листов в Excel:

```
print book.nsheets
```

Печать названий листов:

```
print book.sheet_names()
```

Получение листа по индексу:

```
sheet=book.sheet_by_index(1)
```

Чтение содержимого ячейки:

```
cell = sheet.cell(row,col) #где row=номер строки и col=номер столбца
print cell.value #для вывода содержимого ячейки
```

Получение количества строк и количества столбцов в листе Excel:

```
num_rows=sheet.nrows
num_col=sheet.ncols
```

Получение листа Excel по имени:

```
sheets = book.sheet_names()
cur_sheet = book.sheet_by_name(sheets[0])
```

164.2. Форматирование файлов Excel с помощью модуля `xlsxwriter`

```
import xlsxwriter

# создать новый файл
workbook = xlsxwriter.Workbook('your_file.xlsx')

# добавить несколько новых форматов, которые будут использоваться рабочей книгой
percent_format = workbook.add_format({'num_format': '0%'})
percent_with_decimal = workbook.add_format({'num_format': '0.0%'})
bold = workbook.add_format({'bold': True})
red_font = workbook.add_format({'font_color': 'red'})
remove_format = workbook.add_format()

# добавить новый лист
worksheet = workbook.add_worksheet()

# установить ширину столбца A
worksheet.set_column('A:A', 30, )

# установить в столбце B значение 20 и включить формат процентов, который мы создали ранее
worksheet.set_column('B:B', 20, percent_format)

# удалить форматирование из первой строки (изменить на height=None)
worksheet.set_row('0:0', None, remove_format)

workbook.close()
```

164.3. Помещение списковых данных в файл Excel

```
import os, sys
from openpyxl import Workbook
from datetime import datetime

dt = datetime.now()
list_values = [['01/01/2016', "05:00:00", 3], \
               ["01/02/2016", "06:00:00", 4], \
               ["01/03/2016", "07:00:00", 5], \
               ["01/04/2016", "08:00:00", 6], \
               ["01/05/2016", "09:00:00", 7]]

# Создать рабочую книгу в Excel:
wb = Workbook()
sheet = wb.active
sheet.title = 'data'

# Вывести заголовки в рабочую книгу Excel:
row = 1
sheet['A'+str(row)] = 'Date'
sheet['B'+str(row)] = 'Hour'
sheet['C'+str(row)] = 'Value'

# Заполнить данными:
for item in list_values:
    row += 1
    sheet['A'+str(row)] = item[0]
    sheet['B'+str(row)] = item[1]
    sheet['C'+str(row)] = item[2]
```

```
# Сохранить файл по дате:
filename = 'data_' + dt.strftime("%Y%m%d_%I%M%S") + '.xlsx'
wb.save(filename)

# Открыть файл для пользователя:
os.chdir(sys.path[0])
os.system('start excel.exe "%s\\%s"' % (sys.path[0], filename, ))
```

164.4. Модуль OpenPyXL

OpenPyXL – это модуль для создания рабочих книг (workbooks) форматов xlsx/xlsm/xltx/xltn и работы с ними в памяти.

Работа с существующим файлом

```
import openpyxl as opx
#Для изменения существующей workbook мы находим ее, ссылаясь на ее путь
workbook = opx.load_workbook(workbook_path)
```

load_workbook() содержит параметр read_only, установка которого в значение True приводит к загрузке рабочей книги в режиме read_only, что удобно при чтении больших xlsx-файлов:

```
workbook = opx.load_workbook(workbook_path, read_only=True)
```

После загрузки рабочей книги в память можно получить доступ к отдельным листам с помощью workbook.sheets:

```
first_sheet = workbook.worksheets[0]
```

Если необходимо указать имя доступного листа, можно воспользоваться функцией workbook.get_sheet_names().

```
sheet = workbook.get_sheet_by_name('Sheet Name')
```

Наконец, доступ к строкам листа можно получить с помощью sheet.rows. Для итерации по строкам листа используйте:

```
for row in sheet.rows:
    print row[0].value
```

Поскольку каждая строка представляет собой список ячеек (Cells), для получения содержимого ячейки используйте Cell.value.

Создание нового файла в памяти

```
#Вызов функции Workbook() создает в памяти новую книгу
wb = opx.Workbook()
#Мы можем создать новый лист в книге
ws = wb.create_sheet('Sheet Name', 0) #0 обозначает индекс порядка листа в рабочей книге
```

Некоторые свойства могут быть изменены с помощью openpyxl, например, tabColor:

```
ws.sheet_properties.tabColor = 'FFC0CB'
```

Чтобы сохранить созданную нами рабочую книгу, выполним следующее:

```
wb.save('filename.xlsx')
```

164.5. Создание диаграмм Excel с помощью xlswriter

```
import xlswriter

# образец данных
chart_data = [
    {'name': 'Lorem', 'value': 23},
    {'name': 'Ipsum', 'value': 48},
```



```
{'name': 'Dolor', 'value': 15},
{'name': 'Sit', 'value': 8},
{'name': 'Amet', 'value': 32}
]

# путь к файлу excel
xls_file = 'chart.xlsx'

# рабочая книга
workbook = xlswriter.Workbook(xls_file)

# добавление листа в рабочую книгу
worksheet = workbook.add_worksheet()

row_ = 0
col_ = 0

# запись заголовков
worksheet.write(row_, col_, 'NAME')
col_ += 1
worksheet.write(row_, col_, 'VALUE')
row_ += 1

# запись данных
for item in chart_data:
    col_ = 0
    worksheet.write(row_, col_, item['name'])
    col_ += 1
    worksheet.write(row_, col_, item['value'])
    row_ += 1

# создание круговой диаграммы
pie_chart = workbook.add_chart({'type': 'pie'})

# добавление series на диаграмму
pie_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})

# вставка круговой диаграммы
worksheet.insert_chart('D2', pie_chart)

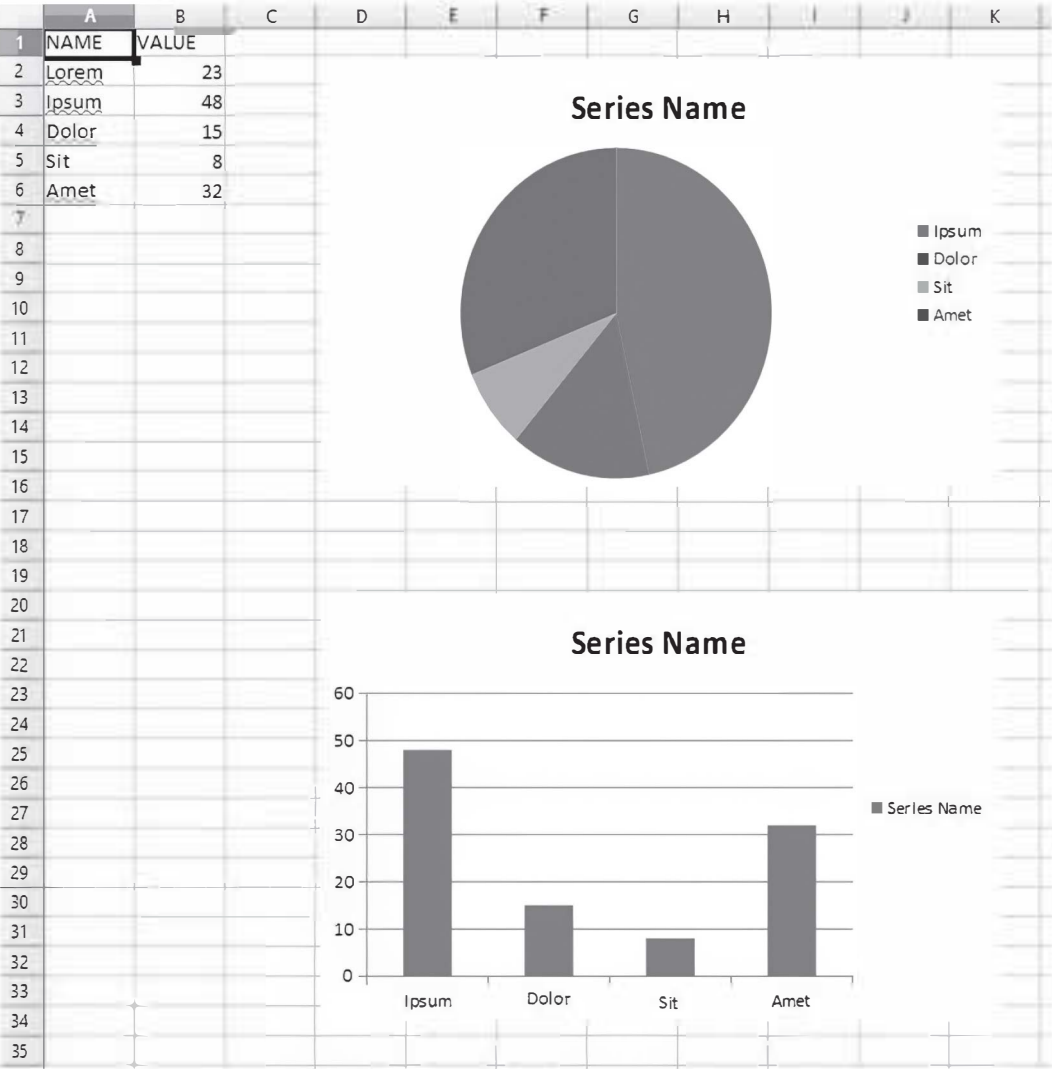
# создание столбцовой диаграммы
column_chart = workbook.add_chart({'type': 'column'})

# добавление series в столбцовую диаграмму
column_chart.add_series({
    'name': 'Series Name',
    'categories': '=Sheet1!$A$3:$A$%s' % row_,
    'values': '=Sheet1!$B$3:$B$%s' % row_,
    'marker': {'type': 'circle'}
})

# вставка столбцовой диаграммы
worksheet.insert_chart('D20', column_chart)

workbook.close()
```

Результат:



Глава 165. Turtle Graphics ("Черепашья графика")

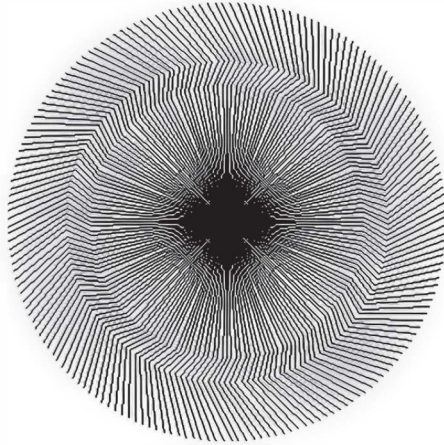
165.1. Создание Ninja Twist при помощи Turtle Graphics

```
import turtle
ninja = turtle.Turtle()
ninja.speed(10)
for i in range(180):
    ninja.forward(100)
    ninja.right(30)
    ninja.forward(20)
```

```
ninja.left(60)
ninja.forward(50)
ninja.right(30)

ninja.penup()
ninja.setposition(0, 0)
ninja.pendown()
ninja.right(2)

turtle.done()
```



Глава 166. Продолжающиеся действия (Persistence) в Python

Параметр	Описание
obj	Pickle-репрезентация объекта открытому файлу объекта файла
protocol	целое число, указывающее на использование заданного протокола, где 0 – ASCII, 1 – старый двоичный формат
file	Аргумент file должен иметь write() метод wb для метода <i>dump</i> и для загрузки read() метод rb

166.1. Продолжающиеся действия в Python

Такие объекты, как числа, списки, словари, вложенные структуры и объекты экземпляров классов, “живут” в памяти компьютера и теряются при завершении работы скрипта. Модуль pickle хранит данные продолжающимся образом в отдельном файле. Его представление объекта всегда и во всех случаях является байтовым объектом, поэтому для хранения данных необходимо открывать файлы, указывая wb, а для загрузки данных из pickle – rb.

Данные могут быть любого вида, например:

```
data={'a':'some_value',
      'b':[9,4,7],
      'c':['some_str','another_str','spam','ham'],
      'd':{'key':'nested_dictionary'},
}
```

Хранение данных:

```
import pickle
file=open('filename','wb') #файловый объект в режиме двоичной записи
pickle.dump(data,file)     #выгрузка данных в файловый объект
file.close()               #закрыть файл для записи в него данных
```

Загрузка данных:

```
import pickle
file=open('filename','rb') #объект файла в режиме бинарного чтения
data=pickle.load(file)     #загрузить данные обратно
file.close()
```

```
>>>data
{'b': [9, 4, 7], 'a': 'some_value', 'd': {'key': 'nested_dictionary'},
'c': ['some_str', 'another_str', 'spam', 'ham']}
```

Модуль pickle принимает:

1. None, True и False.
2. Целые числа, числа с плавающей точкой, комплексные числа.
3. Строки, байты, байтовые массивы.
4. Кортежи, списки, наборы и словари, содержащие только picklable-объекты.
5. Функции, определяемые на верхнем уровне модуля (с использованием def, а не lambda).
6. Встроенные функции, определяемые на верхнем уровне модуля.
7. Классы, определяемые на верхнем уровне модуля.
8. Экземпляры классов, чей словарь или результат вызова getstate() может быть обработан.

166.2. Функциональная утилита для сохранения и загрузки

Сохранение данных в файл и из него:

```
import pickle
def save(filename,object):
    file=open(filename,'wb')
    pickle.dump(object,file)
    file.close()

def load(filename):
    file=open(filename,'rb')
    object=pickle.load(file)
    file.close()
    return object

>>>list_object=[1,1,2,3,5,8,'a','e','i','o','u']
>>>save(list_file,list_object)
>>>new_list=load(list_file)
>>>new_list
[1, 1, 2, 3, 5, 8, 'a', 'e', 'i', 'o', 'u']
```

Глава 167. Шаблоны проектирования

Шаблон (паттерн) проектирования – это общее решение часто встречающейся проблемы при разработке программного обеспечения. В данной главе представлены примеры распространенных паттернов проектирования в Python.

167.1. Введение в паттерны проектирования и паттерн синглтон

Шаблоны проектирования обеспечивают решение часто встречающихся проблем при проектировании программного обеспечения. Впервые они были представлены группой GoF (Gang of Four), которые описали общие шаблоны как проблемы, которые возникают снова и снова, и пути решения этих проблем.

Шаблоны проектирования состоят из четырех основных элементов:

1. Имя (name) – это дескриптор, с помощью которого мы можем кратко описать проблему проектирования, ее решения и последствия.
2. Задача (problem) описывает, когда следует применять шаблон.
3. Решение (solution) описывает элементы, составляющие проект, их взаимосвязи, ответственность и взаимодействие.
4. Последствия (consequences) – это результаты применения паттерна и компромиссы при его использовании.

Преимущества шаблонов проектирования:

1. Они многократно используются в различных проектах.
2. Проблемы могут быть решены на проектном уровне.
3. Они проверены временем и опытом разработчиков и проектировщиков.
4. Они обладают надежностью и зависимостью.

Шаблоны проектирования можно разделить на три категории:

1. Creational Pattern (творческий шаблон, творческая модель).
2. Structural Pattern (структурный шаблон, структурная схема).
3. Behavioral Pattern (поведенческий шаблон).

Creational Pattern – относятся к тому, как может быть создан объект, изолируя детали создания объекта.

Structural Pattern – разрабатывают структуру классов и объектов таким образом, чтобы их можно было компоновать для достижения больших результатов.

Behavioral Pattern – концентрируются на взаимодействии между объектами и ответственности объектов.

Паттерн синглтона:

Синглтон (Singleton, “одиночка”) – это разновидность творческого шаблона, которая предоставляет механизм, позволяющий иметь единственный объект заданного типа и обеспечивающий глобальную точку доступа. Например, синглтон может использоваться в операциях с базами данных, когда мы хотим, чтобы объект базы данных поддерживал согласованность данных.

Реализация

Мы можем реализовать паттерн синглтона в Python, создав только один экземпляр класса Singleton и обслуживая тот же объект повторно.

```
class Singleton(object):
    def __new__(cls):
        # Метод hasattr проверяет, есть ли у объекта класса свойство экземпляра или нет
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance

s = Singleton()
print ("Object created", s)

s1 = Singleton()
print ("Object2 created", s1)
```

Результат:

```
('Object created', <__main__.Singleton object at 0x10a7cc310>)
('Object2 created', <__main__.Singleton object at 0x10a7cc310>)
```

Заметим, что в таких языках, как C++ или Java, этот паттерн реализуется за счет того, что конструктор делается приватным и создается статический метод, выполняющий инициализацию объекта. Таким образом, при первом вызове создается один объект, и в дальнейшем класс возвращает один и тот же объект. Но в Python мы не имеем возможности создавать такие конструкторы.

Фабричный шаблон (Factory Pattern)

Это тоже творческий шаблон. Термин “фабричный” означает, что класс отвечает за создание объектов других типов. В качестве фабрики выступает класс, который имеет объекты и методы, связанные с ним. Клиент создает объект, вызывая методы с определенными параметрами, а фабрика создает объект нужного типа и возвращает его клиенту.

```
from abc import ABCMeta, abstractmethod
```

```
class Music():
    __metaclass__ = ABCMeta
    @abstractmethod
    def do_play(self):
        pass
```

```
class Mp3(Music):
    def do_play(self):
        print ("Воспроизведение музыки в формате .mp3!")
```

```
class Ogg(Music):
    def do_play(self):
        print ("Воспроизведение музыки в формате .ogg!")
```

```
class MusicFactory(object):
    def play_sound(self, object_type):
        return eval(object_type)().do_play()
```

```
if __name__ == "__main__":
    mf = MusicFactory()
    music = input("Какую музыку вы хотите воспроизвести - Mp3 или Ogg?")
    mf.play_sound(music)
```

Выход:

```
Какую музыку вы хотите воспроизвести - Mp3 или Ogg?
Воспроизведение музыки в формате .ogg!
```

MusicFactory – это фабричный класс, который создает объект типа Mp3 или Ogg в зависимости от выбора пользователя.

167.2. Стратегический шаблон

Этот шаблон позволяет определить семейство алгоритмов, инкапсулировать каждый из них и сделать их взаимозаменяемыми. Стратегический шаблон позволяет алгоритму изменяться независимо от клиентов, которые его используют.

Например, животные могут “ходить” разными способами. Ходьба может рассматриваться как стратегия, реализуемая разными видами животных:

```
from types import MethodType
```

```
class Animal(object):
    def __init__(self, *args, **kwargs):
```

```
self.name = kwargs.pop('name', None) or 'Animal'
if kwargs.get('walk', None):
    self.walk = MethodType(kwargs.pop('walk'), self)
```

```
def walk(self):
```

```
    """
    Функциональность ходьбы является стратегией и предназначена
    для раздельной реализации разными видами животных.
```

```
    message = '{} should implement a walk method'.format(
        self.__class__.__name__)
    raise NotImplementedError(message)
```

Вот несколько различных алгоритмов ходьбы, которые могут быть использованы с Animal

```
def snake_walk(self):
    print('Я ползаю, потому что я {}'.format(self.name))
```

```
def four_legged_animal_walk(self):
    print('Я использую все четыре ноги, чтобы ходить, потому что я {}'.format(
        self.name))
```

```
def two_legged_animal_walk(self):
    print('Я встаю на две ноги, чтобы ходить, потому что я {}'.format(
        self.name))
```

В результате выполнения этого примера будет получен следующий результат:

```
generic_animal = Animal()
king_cobra = Animal(name='кобра', walk=snake_walk)
elephant = Animal(name='слон', walk=four_legged_animal_walk)
kangaroo = Animal(name='кенгуру', walk=two_legged_animal_walk)

kangaroo.walk()
elephant.walk()
king_cobra.walk()
# ошибка NotImplementedError, чтобы дать понять программисту
# что метод walk предназначен для использования в качестве стратегии
generic_animal.walk()
```

Результат:

Я встаю на две ноги, чтобы ходить, потому что я кенгуру.

Я использую все четыре ноги, чтобы ходить, потому что я слон.

Я ползаю, потому что я кобра.

Traceback (most recent call last):

File "./strategy.py", line 56, in <module>

generic_animal.walk()

File "./strategy.py", line 30, in walk

raise NotImplementedError(message)

NotImplementedError: Animal should implement a walk method (животное должно реализовать метод walk)

Заметим, что в таких языках, как C++ или Java, этот паттерн реализуется с помощью абстрактного класса или интерфейса для определения стратегии. В Python более целесообразно просто определить некоторые функции извне, которые можно динамически добавлять в класс с помощью `types.MethodType`.

167.3. Прокси-объект

Прокси-объект часто используется для обеспечения защищенного доступа к другому объекту, внутреннюю логику которого мы не хотим "загрязнять" требованиями безопасности.

Предположим, мы хотим гарантировать, что доступ к ресурсу может получить только пользователь с определенными правами.

Определим прокси: (будет гарантировать, что только те пользователи, которые могут видеть резервирование, смогут воспользоваться услугой reservation_service):

```
from datetime import date
from operator import attrgetter
```

```
class Proxy:
    def __init__(self, current_user, reservation_service):
        self.current_user = current_user
        self.reservation_service = reservation_service

    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        if self.current_user.can_see_reservations:
            return self.reservation_service.highest_total_price_reservations(
                date_from,
                date_to,
                reservations_count
            )
        else:
            return []
```

#Models and ReservationService:

```
class Reservation:
    def __init__(self, date, total_price):
        self.date = date
        self.total_price = total_price
```

```
class ReservationService:
    def highest_total_price_reservations(self, date_from, date_to, reservations_count):
        # обычно считывается из базы данных/внешнего сервиса
        reservations = [
            Reservation(date(2014, 5, 15), 100),
            Reservation(date(2017, 5, 15), 10),
            Reservation(date(2017, 1, 15), 50)
        ]

        filtered_reservations = [r for r in reservations if (date_from <= r.date <= date_to)]

        sorted_reservations = sorted(filtered_reservations, key=attrgetter('total_price'), reverse=True)

        return sorted_reservations[0:reservations_count]

class User:
    def __init__(self, can_see_reservations, name):
        self.can_see_reservations = can_see_reservations
        self.name = name
```

#Потребительский сервис:

```
class StatsService:
    def __init__(self, reservation_service):
        self.reservation_service = reservation_service

    def year_top_100_reservations_average_total_price(self, year):
        reservations = self.reservation_service.highest_total_price_reservations(
            date(year, 1, 1),
            date(year, 12, 31),
            100
        )

        if len(reservations) > 0:
            total = sum(r.total_price for r in reservations)
```



```

        return total / len(reservations)
    else:
        return 0

```

#Тест:

```

def test(user, year):
    reservations_service = Proxy(user, ReservationService())
    stats_service = StatsService(reservations_service)
    average_price = stats_service.year_top_100_reservations_average_total_price(year)
    print("{0} will see: {1}".format(user.name, average_price))

test(User(True, "John the Admin"), 2017)
test(User(False, "Guest"), 2017)

```

Выгоды

- Мы избегаем любых изменений в ReservationService при изменении ограничений доступа.
- Мы не смешиваем данные, связанные с бизнесом (date_from, date_to, reservations_count), с понятиями, не связанными с предметной областью (разрешениями для пользователей).
- Потребитель (StatsService) также свободен от логики, связанной с разрешениями.

Предостережения

- Интерфейс прокси всегда точно такой же, как и скрываемый им объект, поэтому пользователь, использующий сервис, “обернутый” прокси, даже не подозревает о его наличии.

Глава 168. Модуль hashlib

hashlib реализует общий интерфейс для многих различных алгоритмов безопасного хеширования и дайджеста сообщений. В него входят защищенные по стандарту FIPS хеш-алгоритмы SHA1, SHA224, SHA256, SHA384 и SHA512.

168.1. MD5-хеш строки

Данный модуль реализует общий интерфейс для многих различных безопасных алгоритмов хеширования и дайджеста сообщений. В модуль включены защищенные алгоритмы хеширования SHA1, SHA224, SHA256, SHA384 и SHA512 (определенные в стандарте FIPS 180-2), а также алгоритм MD5 компании RSA (описанный в документе Internet RFC 1321).

Для каждого типа хеша существует свой метод-конструктор. Все они возвращают хеш-объект с одним и тем же простым интерфейсом. Например: используйте sha1() для создания хеш-объекта SHA1.

```
hash.sha1()
```

Конструкторы для хеш-алгоритмов, которые всегда присутствуют в этом модуле, – это md5(), sha1(), sha224(), sha256(), sha384() и sha512(). Теперь можно передавать этому объекту произвольные строки с помощью метода update(). В любой момент можно запросить у него дайджест конкатенации строк, переданных ему на данный момент, используя методы digest() или hexdigest().

```
hash.update(arg)
```

Обновление хеш-объекта с помощью строкового аргумента. Повторные вызовы эквивалентны одному вызову с конкатенацией всех аргументов: m.update(a); m.update(b) эквивалентно m.update(a+b).

hash.digest()

Возвращает дайджест строк, переданных на данный момент в метод update(). Это строка размером digest_size байт, которая может содержать символы, отличные от ASCII, включая нулевые байты.

hash.hexdigest()

Аналогично digest(), только дайджест возвращается в виде строки двойной длины, содержащей только шестнадцатеричные цифры. Это может быть использовано для безопасного обмена значениями в электронной почте или других недвоичных средах.

Приведем пример:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update("Никто не проверяет")
>>> m.update("спам-повторение")
>>> m.digest()
'\xbbd\x9c\x83\xdd\x1e\xa5\xc9\xd9\xde\xc9\xa1\x8d\xf0\xff\xe9'
>>> m.hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
>>> m.digest_size
16
>>> m.block_size
64
```

или:

```
hashlib.md5("Никто не проверяет spam-повторение").hexdigest()
'bb649c83dd1ea5c9d9dec9a18df0ffe9'
```

168.2. Алгоритм, предоставляемый OpenSSL

Для доступа к перечисленным выше хешам, а также к любым другим алгоритмам, которые может использовать ваша библиотека OpenSSL, также существует общий конструктор new(), принимающий в качестве первого параметра строковое имя требуемого алгоритма. Именованные конструкторы работают гораздо быстрее, чем new(), и им следует отдавать предпочтение. Использование new() с алгоритмом, предоставляемым OpenSSL:

```
>>> h = hashlib.new('ripemd160')
>>> h.update("Никто не проверяет spam-повторение")
>>> h.hexdigest()
'cc4a5ce1b3df48aec5d22d1f16b894a0b894eccc'
```

Глава 169. Создание службы Windows с помощью Python

Процессы без пользовательского интерфейса (Headless processes) в Windows называются службами. Ими можно управлять (запускать, останавливать и т. д.) с помощью стандартных средств управления Windows, таких как командная консоль, PowerShell или вкладка Services в Task Manager. Хорошим примером может служить приложение, предоставляющее сетевые услуги, например, веб-приложение, или приложение для резервного копирования, выполняющее различные фоновые архивные задачи. Существует несколько способов создания и установки приложения Python в качестве службы в Windows.

169.1. Сценарий на языке Python, который может быть запущен как служба

```
import win32serviceutil
import win32service
import win32event
import servicemanager
import socket

class AppServerSvc (win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"

    def __init__(self,args):
        win32serviceutil.ServiceFramework.__init__(self,args)
        self.hWaitStop = win32event.CreateEvent(None,0,0,None)
        socket.setdefaulttimeout(60)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self):
        servicemanager.LogMsg(servicemanager.EVENTLOG_INFORMATION_TYPE,
                              servicemanager.PYS_SERVICE_STARTED,
                              (self._svc_name_,))
        self.main()

    def main(self):
        pass

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(AppServerSvc)
```

Это просто шаблон. Ваш прикладной код, возможно вызывающий отдельный скрипт, будет находиться в функции main(). Кроме того, необходимо установить его в качестве службы. Лучшим решением для этого на данный момент является использование Nonsucking Service Manager. Он позволяет установить службу и предоставляет графический интерфейс для настройки командной строки, которую выполняет служба. Для Python можно создать службу за один раз:

```
nssm install MyServiceName c:\python27\python.exe c:\temp\myscript.py
```

где my_script.py – это приведенный выше шаблонный скрипт, модифицированный для вызова сценария или кода вашего приложения в функции main(). Обратите внимание, что сервис не запускает Python-скрипт напрямую, а запускает интерпретатор Python и передает ему основной скрипт в командной строке.

Кроме того, для создания службы можно использовать инструменты, входящие в комплект Windows Server Resource Kit для вашей версии операционной системы.

169.2. Запуск веб-приложения Flask в качестве сервиса

Это вариация на тему общего примера. Достаточно импортировать сценарий приложения и вызвать его метод run() в функции main() сервиса. В данном случае мы также используем модуль многопроцессорной обработки из-за проблемы с доступом к WSGIRequestHandler.

```
import win32serviceutil
import win32service
import win32event
```

```

import servicemanager
from multiprocessing import Process

from app import app

class Service(win32serviceutil.ServiceFramework):
    _svc_name_ = "TestService"
    _svc_display_name_ = "Test Service"
    _svc_description_ = "Тестирует сервисный фреймворк Python, принимая и передавая эхо-
сообщения через именованный pipe"

    def __init__(self, *args):
        super().__init__(*args)

    def SvcStop(self):
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        self.process.terminate()
        self.ReportServiceStatus(win32service.SERVICE_STOPPED)

    def SvcDoRun(self):
        self.process = Process(target=self.main)
        self.process.start()
        self.process.run()

    def main(self):
        app.run()

if __name__ == '__main__':
    win32serviceutil.HandleCommandLine(Service)

```

Адаптированный пример взят с ресурса <http://stackoverflow.com/a/25130524/318488>.

Глава 170. Изменяемые, неизменяемые и хешируемые в Python

170.1. Изменяемые и неизменяемые типы

В Python существует два вида типов: неизменяемые и изменяемые.

Неизменяемые

Объект неизменяемого типа не может быть изменен. Любая попытка модифицировать объект приведет к созданию его копии. В эту категорию входят: целые числа, комплексные, строки, байты, кортежи, диапазоны и фрозенсеты.

Чтобы выделить это свойство, давайте используем встроенную функцию `id`. Эта функция возвращает уникальный идентификатор объекта, переданного в качестве параметра. Если `id` не изменился, то это тот же самый объект. Если он изменился, то это другой объект. (Некоторые утверждают, что на самом деле это адрес памяти объекта, но остерегайтесь их, они с темной стороны силы...)

```

>>> a = 1
>>> id(a)
140128142243264

```

```
>>> a += 2
>>> a
3
>>> id(a)
140128142243328
```

Так, 1 - это не 3... Интересные новости... Однако об этом поведении часто забывают, когда речь заходит о более сложных типах, особенно о строках.

```
>>> stack = "Overflow"
>>> stack
'Overflow'
>>> id(stack)
140128123955504
>>> stack += " rocks!"
>>> stack
'Overflow rocks!'
```

Видите? Мы можем это изменять!

```
>>> id(stack)
140128123911472
```

Нет. Хотя кажется, что мы можем изменить строку, названную переменной стека, на самом деле мы создаем новый объект, содержащий результат конкатенации. Нас обманывают, потому что в процессе старый объект уничтожается. В другой ситуации это было бы более очевидно:

```
>>> stack = "Stack"
>>> stackoverflow = stack + "Overflow"
>>> id(stack)
140128069348184
>>> id(stackoverflow)
140128123911480
```

В этом случае понятно, что если мы хотим сохранить первую строку, то нам нужна ее копия. Но так ли это очевидно для других типов?

Упражнение

Теперь, зная, как работают неизменяемые типы, что бы вы сказали о приведенном ниже фрагменте кода? Является ли он разумным?

```
s = ""
for i in range(1, 1000):
    s += str(i)
    s += ","
```

Изменяемые

Объект изменяемого типа может быть изменен, причем изменение происходит *на месте*. Никаких неявных копий не производится. К этой категории относятся: списки, словари, байтовые массивы и наборы. Вернемся к нашей функции `id`.

```
>>> b = bytearray(b'Stack')
>>> b
bytearray(b'Stack')
>>> b = bytearray(b'Stack')
>>> id(b)
140128030688288
>>> b += b'Overflow'
>>> b
bytearray(b'StackOverflow')
>>> id(b)
140128030688288
```

(Для наглядности используются байты, содержащие данные в формате ASCII, но помните, что байты не предназначены для хранения текстовых данных). Что мы имеем? Мы создаем байтовый массив, модифицируем его и, используя `id`, можем убедиться, что это тот же самый объект, модифицированный. А не его копия.

Конечно, если объект будет часто модифицироваться, то изменяемый тип справится с этой задачей гораздо лучше, чем неизменяемый. К сожалению, о реальности этого свойства часто забывают, когда это наиболее болезненно.

```
>>> c = b
>>> c += b' rocks!'
>>> c
bytearray(b'StackOverflow rocks!')
```

Хорошо...

```
>>> b
bytearray(b'StackOverflow rocks!')
```

Подождите секунду...

```
>>> id(c) == id(b)
True
```

Действительно, `c` не является копией `b`. `c` – это `b`.

Упражнение

Теперь, когда вы лучше понимаете, какой побочный эффект вызывается изменяемым типом, можете ли вы объяснить, что в этом примере происходит не так?

```
>>> ll = [ [] ]*4 # Создание списка из 4 списков, содержащего наши результаты
>>> ll
[[ ], [ ], [ ], [ ]]
>>> ll[0].append(23) # Добавить результат 23 в первый список
>>> ll
[[23], [23], [23], [23]]
>>> # Oops...
```

170.2. Изменяемые и неизменяемые типы как аргументы

Одним из основных случаев, когда разработчику необходимо учитывать изменяемость, является передача аргументов в функцию. Это очень важно, поскольку от этого зависит возможность функции изменять объекты, не входящие в ее область видимости, или, другими словами, наличие у функции побочных эффектов. Это также важно для понимания того, где должен быть доступен результат работы функции.

```
>>> def list_add3(lin):
    lin += [3]
    return lin

>>> a = [1, 2, 3]
>>> b = list_add3(a)
>>> b
[1, 2, 3, 3]
>>> a
[1, 2, 3, 3]
```

Здесь ошибка заключается в том, что `lin`, как параметр функции, может быть изменен локально. Вместо этого `lin` и `a` ссылаются на один и тот же объект. Поскольку этот объект изменяемый, модификация выполняется на месте, что означает, что объект, на который ссылаются и `lin`, и `a`, модифицируется. На самом деле возвращать `lin` не нужно, поскольку у нас уже есть ссылка на этот объект в виде `a`. `a` и `b` в итоге ссылаются на один и тот же объект.

С кортежами дело обстоит иначе.

```
>>> def tuple_add3(tin):
    tin += (3,)
    return tin

>>> a = (1, 2, 3)
>>> b = tuple_add3(a)
>>> b
(1, 2, 3, 3)
>>> a
(1, 2, 3)
```

В начале функции `tin` и `a` ссылаются на один и тот же объект. Но это неизменяемый объект. Поэтому, когда функция попытается его модифицировать, `tin` получит новый объект с модификацией, а `a` сохранит ссылку на исходный объект. В этом случае возврат `tin` обязателен, иначе новый объект будет потерян.

Упражнение

```
>>> def yoda(prologue, sentence):
    sentence.reverse()
    prologue += " ".join(sentence)
    return prologue

>>> focused = ["You must", "stay focused"]
>>> saying = "Yoda said: "
>>> yoda_sentence = yoda(saying, focused)
```

Примечание: реверс работает в режиме “на месте”.

Что вы думаете об этой функции? Есть ли у нее побочные эффекты? Необходим ли возврат? После вызова какое значение имеет высказывание? В том, чтобы сосредоточиться? Что произойдет, если функцию вызвать еще раз с теми же параметрами?

Глава 171. Модуль configparser

Этот модуль предоставляет класс `ConfigParser`, который реализует базовый язык конфигурации в INI-файлах. С его помощью можно писать программы на языке Python, которые могут быть легко настроены конечными пользователями.

171.1. Создание конфигурационного файла программным способом

Конфигурационный файл содержит секции, каждая из которых содержит ключи и значения. Модуль `configparser` может использоваться для чтения и записи `config`-файлов. Создадим такой файл:

```
import configparser
config = configparser.ConfigParser()
config['settings']={'resolution':'320x240',
                  'color':'blue'}
with open('example.ini', 'w') as configfile:
    config.write(configfile)
```

Выходной файл содержит следующую структуру:

```
[settings]
resolution = 320x240
color = blue
```

Если вы хотите изменить конкретное поле, получите это поле и присвойте ему значение:

```
settings=config['settings']
settings['color']='red'
```

171.2. Базовое использование

В файле config.ini:

```
[DEFAULT]
debug = True
name = Test
password = password
```

```
[FILES]
path = /path/to/file
```

В Python:

```
from ConfigParser import ConfigParser
config = ConfigParser()
```

```
#Загрузка файла конфигурации
config.read("config.ini")
```

```
# Доступ к ключу "debug" в секции "DEFAULT"
config.get("DEFAULT", "debug")
# Return 'True'
```

```
# Доступ к ключу "path" в разделе "FILES"
config.get("FILES", "path")
# Возвращает '/path/to/file'
```

Глава 172. Оптическое распознавание символов (OCR)

Оптическое распознавание символов – это преобразование изображений текста в реальный текст. В этих примерах рассмотрены способы использования OCR в Python.

172.1. PyTesseract

PyTesseract – это разрабатываемый пакет для OCR на языке Python. Использовать его довольно просто:

```
try:
    import Image
except ImportError:
    from PIL import Image
```

```
import pytesseract
```

```
#Basic OCR
print(pytesseract.image_to_string(Image.open('test.png')))
```

```
#На французском языке
print(pytesseract.image_to_string(Image.open('test-european.jpg'), lang='fra'))
```


172.2. PyOCR

Другим полезным модулем является PyOCR, исходный код которого находится здесь: <https://github.com/jflesch/pyocr>. Он также прост в использовании и имеет больше возможностей, чем PyTesseract.

Для инициализации:

```
from PIL import Image
import sys
```

```
import pyocr
import pyocr.builders
```

```
tools = pyocr.get_available_tools()
# Инструменты возвращаются в рекомендуемом порядке использования
tool = tools[0]
```

```
langs = tool.get_available_languages()
lang = langs[0]
# Обратите внимание, что языки никак не сортируются. Обратитесь к настройкам системы для
# определения языка для использования по умолчанию.
```

Некоторые примеры использования:

```
txt = tool.image_to_string(
    Image.open('test.png'),
    lang=lang,
    builder=pyocr.builders.TextBuilder()
)
# txt - строка Python
```

```
word_boxes = tool.image_to_string(
    Image.open('test.png'),
    lang="eng",
    builder=pyocr.builders.WordBoxBuilder()
)
# список блоковых (box) объектов. Для каждого блокового объекта box:
# box.content - слово, находящееся в блоке
# box.position - его положение на странице (в пикселях) #
box
# Следует иметь в виду, что некоторые инструменты OCR (например, Tesseract)
# могут возвращать пустые блоки
```

```
line_and_word_boxes = tool.image_to_string(
    Image.open('test.png'), lang="fra",
    builder=pyocr.builders.LineBoxBuilder()
)
# список объектов строчек. Для каждого такого объекта:
# line.word_boxes - список блоков слов (отдельных слов в строчке)
# line.content - весь текст строчки
# line.position - положение всей строчки на странице (в пикселях)
#
# Следует иметь в виду, что некоторые инструменты OCR (например, Tesseract)
# могут возвращать пустые блоки
```

```
# Цифры - только Tesseract (пока не 'libtesseract')
digits = tool.image_to_string(
    Image.open('test-digits.png'),
    lang=lang,
    builder=pyocr.tesseract.DigitBuilder()
)
# digits - строка в формате python
```

Глава 173. Виртуальные среды

Виртуальная среда (виртуальное окружение) – это инструмент для хранения зависимостей, необходимых для разных проектов в разных местах, путем создания для них виртуальных окружений Python. Это позволяет решить проблему “*проект X зависит от версии 1.x, но проекту Y требуется 4.x*”, а также изолировать окружения для разных проектов друг от друга и от системных библиотек.

173.1. Создание и использование виртуальной среды

`virtualenv` – это инструмент для создания изолированных сред Python. Эта программа создает папку, содержащую все необходимые исполняемые файлы для использования пакетов, необходимых проекту Python.

Установка инструмента `virtualenv`

Это требуется только один раз. Программа `virtualenv` может быть доступна в вашем дистрибутиве. В Debian-подобных дистрибутивах пакет называется `python-virtualenv` или `python3-virtualenv`. В качестве альтернативы можно установить `virtualenv` с помощью `pip`:

```
$ pip install virtualenv
```

Создание новой виртуальной среды

Это требуется только один раз для каждого проекта. При запуске проекта, для которого требуется изолировать зависимости, можно настроить новую виртуальную среду для этого проекта:

```
$ virtualenv foo
```

В результате будет создана папка `foo`, содержащая инструментальные скрипты и копию самого бинарного файла Python. Имя папки не имеет значения. После создания виртуальной среды она становится автономной и не требует дальнейших манипуляций с инструментом `virtualenv`. Теперь можно приступить к работе с виртуальной средой.

Активация существующей виртуальной среды

Для активации требуется некоторая магия оболочки, чтобы ваш Python оказался внутри `foo`. Для этого используется файл `activate`, который необходимо расположить в текущей оболочке:

```
$ source foo/bin/activate
```

Пользователям Windows следует набрать:

```
$ foo\Scripts\activate.bat
```

После активации виртуальной среды внутри `foo` оказываются исполняемые файлы `python` и `pip`, а также все скрипты, установленные с помощью сторонних модулей. В частности, все модули, установленные с помощью `pip`, будут развернуты в виртуальной среде, что позволяет создать замкнутую среду разработки. Активация виртуальной среды также должна добавить в приглашение команду-префикс, как показано в следующих командах.

```
# Устанавливает 'requests' только на foo, а не глобально  
(foo)$ pip install requests
```

Сохранение и восстановление зависимостей

Для сохранения модулей, установленных с помощью `pip`, можно составить список всех этих модулей (и соответствующих версий) в виде текстового файла с помощью команды `freeze`. Это позволит другим пользователям быстро установить необходимые для приложения модули Python с помощью команды `install`. Традиционное название такого файла – `requirements.txt`:

Обратите внимание, что в файле `freeze` перечислены все модули, включая транзитивные зависимости, требуемые модулями верхнего уровня, которые вы установили вручную. Поэтому вы можете предпочесть составить файл `requirements.txt` вручную, включив в него только те модули верхнего уровня, которые вам необходимы.

Выход из виртуальной среды

Если вы закончили работу в виртуальной среде, вы можете отключить ее, чтобы вернуться в обычную оболочку:

```
(foo)$ deactivate
```

Использование виртуальной среды на общем хосте

Иногда нет возможности использовать `$ source bin/activate`, например, если вы используете `mod_wsgi` на общем хосте или не имеете доступа к файловой системе, как в Amazon API Gateway или Google AppEngine. В этих случаях можно развернуть установленные библиотеки в локальной `virtualenv` и внести исправления в `sys.path`. К счастью, виртуальная среда поставляется со скриптом, который обновляет как `sys.path`, так и `sys.prefix`.

```
import os

mydir = os.path.dirname(os.path.realpath(__file__))
activate_this = mydir + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Эти строки следует добавить в самое начало файла, который будет выполняться вашим сервером.

Это приведет к тому, что `bin/activate_this.py`, созданный виртуальной средой, окажется в том же каталоге, где находится выполняемый вами файл, и добавит вашу `lib/python2.7/site-packages` в `sys.path`. Если вы хотите использовать скрипт `activate_this.py`, не забудьте выполнить развертывание, по крайней мере, каталогов `bin` и `lib/python2.7/site-packages` и их содержимого.

Версия Python 3.x ≥ 3.3

Встроенные виртуальные среды

Начиная с Python 3.3 модуль `venv` позволяет создавать виртуальные среды. Команда `pyvenv` не требует отдельной установки:

```
$ pyvenv foo
$ source foo/bin/activate
```

или

```
$ python3 -m venv foo
$ source foo/bin/activate
```

173.2. Указание версии Python для использования в скрипте на Unix/Linux

Для того чтобы указать, какую версию Python должен использовать командный интерпретатор Linux, первой строкой Python-скриптов может быть шебанг-строка, которая начинается с `#!`:

```
#!/usr/bin/python
```

Если вы находитесь в виртуальной среде, то `python myscript.py` будет использовать Python из вашей виртуальной среды, а `./myscript.py` будет использовать интерпретатор Python в `#!`-строке. Чтобы убедиться, что используется Python из виртуальной среды, измените первую строку:

```
#!/usr/bin/env python
```

После указания шебанг-строки не забудьте дать скрипту права на выполнение, выполнив команду:

```
chmod +x myscript.py
```

Это позволит выполнить скрипт, запустив `./myscript.py` (или указав абсолютный путь к скрипту) вместо `python myscript.py` или `python3 myscript.py`.

173.3. Создание виртуальной среды для различных версий Python

Если предположить, что установлены и Python 2, и Python 3, то можно создать виртуальную среду для Python 3, даже если эта версия не используется по умолчанию:

```
virtualenv -p python3 foo
```

или

```
virtualenv --python=python3 foo
```

или

```
python3 -m venv foo
```

или

```
pyvenv foo
```

На самом деле вы можете создать виртуальную среду на основе любой версии рабочего Python вашей системы. Вы можете проверить разные версии рабочего Python в папке `/usr/bin/` или `/usr/local/bin/` (в Linux) или в папке `/Library/Frameworks/Python.framework/Versions/X.X/bin/` (OSX), затем определить имя и использовать его в флаге `-python` или `-p` при создании виртуальной среды.

173.4. Создание виртуальных сред с помощью Anaconda

Мощной альтернативой `virtualenv` является `Anaconda` (<https://www.continuum.io/downloads>) – кроссплатформенный менеджер пакетов, похожий на `pip` и снабженный функциями для быстрого создания и удаления виртуальных окружений. Установив `Anaconda`, выполните несколько команд для начала работы:

Создание среды

```
conda create --name <envname> python=<version>
```

где `<envname>` – произвольное имя вашей виртуальной среды, а `<version>` – конкретная версия Python, которую вы хотите установить.

Активация и деактивация среды

```
# Linux, Mac
source activate <envname>
source deactivate
```

или

```
# Windows
activate <envname>
deactivate
```

Просмотр списка созданных сред

```
conda env list
```

Удалить среду

```
conda env remove -n <envname>
```

Более подробную информацию о командах и возможностях можно найти в официальной документации менеджера (<http://conda.pydata.org/docs/using/envs.html#create-an-environment>).

173.5. Управление несколькими виртуальными средами с помощью утилиты `virtualenvwrapper`

Утилита `virtualenvwrapper` (<https://virtualenvwrapper.readthedocs.io/>) упрощает работу с виртуальными средами и особенно полезна, если вы имеете дело с большим количеством виртуальных сред и проектов. Вместо того чтобы самому разбираться с каталогами виртуальных сред, `virtualenvwrapper` управляет ими за вас, храня все виртуальные среды в центральном каталоге (по умолчанию `~/.virtualenvs`).

Установка

Установите `virtualenvwrapper` с помощью менеджера пакетов вашей системы.

На базе Debian/Ubuntu:

```
apt-get install virtualenvwrapper
```

Fedora/CentOS/RHEL:

```
yum install python-virtualenvwrapper
```

Arch Linux:

```
pacman -S python-virtualenvwrapper
```

или установите его из PyPI с помощью `pip`:

```
pip install virtualenvwrapper
```

Под Windows можно использовать `virtualenvwrapper-win` или `virtualenvwrapper-powershell`.

Использование

Виртуальные среды создаются с помощью команды `mkvirtualenv`. При этом принимаются все аргументы исходной команды `virtualenv`.

```
mkvirtualenv my-project
```

или, например

```
mkvirtualenv --system-site-packages my-project
```

Новая виртуальная среда активизируется автоматически. В новых оболочках можно включить виртуальную среду с помощью команды `workon`:

```
workon my-project
```

Преимущество команды `workon` по сравнению с традиционной `. path/to/my-env/bin/activate` заключается в том, что команда `workon` будет работать в любом каталоге; вам не нужно помнить, в каком каталоге хранится конкретное виртуальное окружение вашего проекта.

Каталоги проектов

Указать каталог проекта можно даже во время создания виртуальной среды с помощью опции `-a` или позже с помощью команды `setvirtualenvproject`.

```
mkvirtualenv -a /path/to/my-project my-project
```

или

```
workon my-project
```

```
cd /path/to/my-project
```

```
setvirtualenvproject
```

Установка проекта приведет к тому, что команда `workon` автоматически переключится на этот проект и включит команду `cdproject`, позволяющую поменять каталог проекта.

Для просмотра списка всех виртуальных сред, управляемых `virtualenvwrapper`, используйте команду `lsvirtualenv`. Для удаления виртуальной среды используйте команду `rmvirtualenv`:

```
rmvirtualenv my-project
```

Каждая виртуальная среда, управляемая `virtualenvwrapper`, включает в себя 4 пустых `bash`-скрипта: `preactivate`, `postactivate`, `predeactivate` и `postdeactivate`. Они служат в качестве

“крючков” (hooks) для выполнения `bash`-команд в определенные моменты жизненного цикла `virtualenv`; например, любые команды в скрипте `postactivate` будут выполняться сразу после активации `virtualenv`. Это хорошее место для установки специальных переменных окружения, псевдонимов (aliases) или чего-либо еще. Все четыре сценария находятся в папке `.virtualenvs/<имя_виртуальной_среды>/bin/`.

Более подробную информацию можно найти в документации по `virtualenvwrapper` (<https://virtualenvwrapper.readthedocs.io/>).

173.6. Установка пакетов в виртуальной среде

После активации виртуальной среды все устанавливаемые пакеты теперь будут устанавливаться в `virtualenv`, а не глобально. Таким образом, новые пакеты могут не требовать `root`-привилегий.

Для проверки того, что пакеты устанавливаются в `virtualenv`, выполните следующую команду, чтобы проверить путь к используемому исполняемому файлу:

```
(<Virtualenv Name>) $ which python  
/<Virtualenv Directory>/bin/python
```

```
(Virtualenv Name) $ which pip  
/<Virtualenv Directory>/bin/pip
```

Любой пакет, установленный затем с помощью `pip`, будет установлен в `virtualenv` в следующий каталог:

```
/<Virtualenv Directory>/lib/python2.7/site-packages/
```

В качестве альтернативы можно создать файл, содержащий список необходимых пакетов.

requirements.txt:

```
requests==2.10.0
```

Исполнение:

```
# Установить пакеты из requirements.txt  
pip install -r requirements.txt
```

установит версию 2.10.0 запросов (requests) пакетов.

Также можно получить список пакетов и их версий, установленных в настоящее время в активной виртуальной среде:

```
# Получить список установленных пакетов  
pip freeze  
# Вывести список пакетов и версий в файл requirement.txt, чтобы можно было воссоздать  
# виртуальную среду  
pip freeze > requirements.txt
```

В качестве альтернативы можно не активировать виртуальную среду каждый раз, когда требуется установить пакет. Для установки пакетов можно напрямую использовать исполняемый файл `pip` в каталоге виртуальной среды.

```
$ /<Virtualenv Directory>/bin/pip install requests
```

Более подробную информацию об использовании `pip` можно найти в соответствующей главе.

Поскольку установка производится без `root`-прав в виртуальной среде, это *не* глобальная установка на всю систему – установленный пакет будет доступен только в текущей виртуальной среде.

173.7. Определение используемой виртуальной среды

Если вы используете стандартное `bash`-приглашение в Linux, то в начале приглашения вы должны увидеть имя виртуальной среды.

```
(my-project-env) user@hostname:~$ which python
/home/user/my-project-env/bin/python
```

173.8. Проверка на работу в виртуальной среде

Иногда в приглашении командной строки не отображается имя виртуальной среды, и вы хотите быть уверены, находитесь ли вы в виртуальной среде или нет.

Запустите интерпретатор Python и попробуйте:

```
import sys
sys.prefix
sys.real_prefix
```

- Вне виртуальной среды `sys.prefix` будет указывать на системную установку Python, а `sys.real_prefix` не определен.
- В виртуальной среде `sys.prefix` будет указывать на установку Python в виртуальной среде, `sys.real_prefix` будет указывать на системную установку Python.

Для виртуальных сред, созданных с помощью модуля `venv`, отсутствует `sys.real_prefix`. Вместо этого следует проверить, совпадает ли `sys.base_prefix` с `sys.prefix`.

173.9. Использование виртуальной среды с оболочкой Fish shell

Fish shell “дружелюбна” к пользователям, но при использовании с `virtualenv` или `virtualenvwrapper` могут возникнуть проблемы. Тут на помощь приходит менеджер `virtualfish`. Чтобы начать использовать оболочку Fish shell с `virtualenv`, выполните следующую последовательность действий.

Установить `virtualfish` в глобальное пространство:

```
sudo pip install virtualfish
```

Загрузка Python-модуля `virtualfish` при запуске оболочки Fish shell:

```
$ echo "eval (python -m virtualfish)" > ~/.config/fish/config.fish
```

Отредактируйте функцию `fish_prompt` с помощью `$ funced fish_prompt --editor vim` и добавьте следующие строки, после чего закройте редактор `vim`:

```
if set -q VIRTUAL_ENV
  echo -n -s (set_color -b blue white) "(" (basename "$VIRTUAL_ENV") ")" (set_color normal) " "
end
```

Примечание: если вы не используете `vim`, просто поставьте ваш любимый редактор, например, так: `$ funced fish_prompt --editor nano` или `$ funced fish_prompt --editor gedit`.

Сохранение изменений с помощью функции `funcsave`:

```
funcsave fish_prompt
```

Для создания новой виртуальной среды используйте команду `vf new`:

```
vf new my_new_env # Убедитесь, что $HOME/.virtualenv существует
```

Если вы хотите создать новое окружение Python 3, укажите его при помощи флага `-p`:

```
vf new -p python3 my_new_env
```

Для переключения между виртуальными средами используйте `vf deactivate` и `vf activate another_env`.

Ссылки:

- <https://github.com/adambrenecki/virtualfish>
- <http://virtualfish.readthedocs.io/en/latest/>

Глава 174. Виртуальная среда Python – virtualenv

Виртуальная среда (virtualenv) – это инструмент для создания изолированных окружений Python. Она позволяет хранить зависимости, необходимые для разных проектов, в разных местах, создавая для них виртуальные среды Python. Это решает дилемму наподобие “проект А зависит от версии 2.xxx, но проекту В нужна версия 2.xxx” и позволяет сохранить глобальный каталог site-packages чистым и управляемым.

virtualenv создает папку, содержащую все необходимые ресурсы (lib и bin) для использования пакетов, необходимых проекту на Python.

174.1. Установка

Установите virtualenv с помощью pip / (apt-get):

```
pip install virtualenv
```

или

```
apt-get install python-virtualenv
```

Примечание: если у вас возникают проблемы с правами доступа, используйте sudo.

174.2. Использование

```
$ cd test_proj
```

Создание виртуальной среды:

```
$ virtualenv test_proj
```

Для начала использования виртуальной среды ее необходимо активировать:

```
$ source test_project/bin/activate
```

Для выхода из virtualenv достаточно набрать “deactivate”:

```
$ deactivate
```

174.3. Установка пакета в виртуальной среде

Если вы посмотрите на каталог bin в вашей виртуальной среде, то увидите easy_install, который был модифицирован для размещения пакетов в каталоге site-packages виртуальной среды. Чтобы установить приложение в виртуальную среду, выполните следующие действия:

```
$ source test_project/bin/activate
```

```
$ pip install flask
```

При этом не нужно использовать sudo, поскольку все элементы будут установлены в локальный каталог виртуальной среды site-packages. Он был создан под вашей собственной учетной записью пользователя.

174.4. Другие полезные команды virtualenv

lsvirtualenv: вывести список всех окружений.

cdvirtualenv: переход в каталог активированной в данный момент виртуальной среды, что позволяет просматривать, например, ее site-packages.

cdsitepackages: аналогично описанному выше, но переходит непосредственно в каталог site-packages.

lssitepackages: показывает содержимое каталога site-packages.

Глава 175. Создание виртуальной среды с помощью надстройки virtualenvwrapper

Предположим, что вам необходимо работать над тремя разными проектами – проектом А, проектом В и проектом С. Для проектов А и В требуются Python 3 и некоторые необходимые библиотеки. Но для проекта С нужны Python 2.7 и зависимые библиотеки. Поэтому лучше всего разделить эти проектные среды.

Хотя существует несколько путей создания виртуальной среды, предпочтительнее использовать virtualenvwrapper, потому что он предоставляет больше возможностей.

```
$ pip install virtualenvwrapper
```

```
$ export WORKON_HOME=~/.Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ printf '\n%s\n%s\n%s' '# virtualenv' 'export WORKON_HOME=~/.virtualenvs' 'source
/home/salayhin/bin/virtualenvwrapper.sh' >> ~/.bashrc
$ source ~/.bashrc
```

```
$ mkvirtualenv python_3.5
Installing
setuptools.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/salayhin/Envs/python_3.5/bin/postactivate
New python executable in python_3.5/bin/python
(python_3.5)$ ls $WORKON_HOME
python_3.5 hook.log
```

Теперь мы можем установить в среду некоторое программное обеспечение.

```
(python_3.5)$ pip install django
Downloading/unpacking django
Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
Running setup.py egg_info for package django
Installing collected packages: django
Running setup.py install for django
changing mode of build/scripts-2.6/django-admin.py from 644 to 755
changing mode of /Users/salayhin/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

Мы можем увидеть новый пакет с помощью `ls sitepackages`:

```
(python_3.5)$ ls sitepackages
Django-1.1.1-py2.6.egg-info easy-install.pth
setuptools-0.6.10-py2.6.egg pip-0.6.3-py2.6.egg
django setuptools.pth
```

При желании мы можем создать несколько виртуальных сред.

Переключение между средами осуществляется с помощью команды `workon`:

```
(python_3.6)$ workon python_3.5
(python_3.5)$ echo $VIRTUAL_ENV
/Users/salayhin/Envs/env1
(python_3.5)$
```

Выход из виртуальной среды

```
$ deactivate
```

Глава 176. Создание виртуальной среды с помощью virtualenvwrapper в Windows

Предположим, что вам необходимо работать над тремя разными проектами – проектом А, проектом В и проектом С. Для проектов А и В требуются Python 3 и некоторые необходимые библиотеки. А для проекта С нужны Python 2.7 и зависимые библиотеки.

Поэтому наилучшей практикой является разделение этих окружений проекта. Для создания отдельной виртуальной среды Python необходимо выполнить следующие шаги:

Шаг 1. Установите pip с помощью следующей команды: `python -m pip install -U pip`

Шаг 2. Затем установите пакет `virtualenvwrapper-win` с помощью нижеприведенной команды (эта команда может быть исполнена в Windows PowerShell):

```
pip install virtualenvwrapper-win
```

Шаг 3. Создайте новое окружение виртуальной среды с помощью команды:

```
mkvirtualenv python_3.5
```

Шаг 4: Активируйте среду с помощью команды:

```
workon <environment name>
```

Основные команды `virtualenvwrapper`:

```
mkvirtualenv <имя>
```

Создает новое окружение `virtualenv` с именем `<имя>`. Среда будет создана в `WORKON_HOME`.

```
lsvirtualenv
```

Перечисляет все окружения, хранящиеся в `WORKON_HOME`.

```
rmvirtualenv <имя>
```

Удаляет окружение `<имя>`. Использует файл `folder_delete.bat`.

```
workon [<имя>]
```

Если указано `<имя>`, активизирует окружение с именем `<имя>` (измените рабочий `virtualenv` на `<имя>`). Если каталог проекта был определен, то мы перейдем в него. Если аргумент не указан, то будет выдан список доступных окружений. Можно передать дополнительную опцию `-c` после имени `virtualenv` для перехода в каталог `virtualenv`, если не задан каталог `projectdir`.

```
deactivate
```

Деактивирует рабочую `virtualenv` и переключает обратно на стандартный системный Python.

```
add2virtualenv <полный или относительный путь>
```

Если среда `virtualenv` активна, то `<path>` добавляется в `virtualenv_path_extensions.pth` внутри `site-packages` среды, что фактически добавляет `<path>` в `PYTHONPATH` среды. Если среда `virtualenv` не активна, то `<path>` добавляется в `virtualenv_path_extensions.pth` внутри `site-packages Python` по умолчанию. Если `<path>` не существует, то он будет создан.

Глава 177. Модуль sys

Модуль `sys` предоставляет доступ к функциям и значениям, относящимся к среде выполнения программы, таким как параметры командной строки в `sys.argv` или функция `sys.exit()` для завершения текущего процесса из любой точки программного потока.

Хотя в чистом виде `sys` выделен в модуль, на самом деле он является встроенным и как таковой всегда будет доступен в обычных условиях.

177.1. Аргументы командной строки

```
if len(sys.argv) != 4:      # Имя скрипта также должно быть учтено.
    raise RuntimeError("expected 3 command line arguments" (ожидалось 3 аргумента командной строки))

f = open(sys.argv[1], 'rb') # Использовать первый аргумент командной строки.
start_line = int(sys.argv[2]) # Все аргументы приходят в виде строк, поэтому должны быть
end_line = int(sys.argv[3]) # преобразованы явно, если требуются другие типы
```

Заметим, что в больших и более совершенных программах можно использовать для обработки аргументов командной строки такие модули, как `click` (<http://click.pocoo.org/>), а не делать это самостоятельно.

177.2. Имя скрипта

```
# Имя выполняемого скрипта находится в начале списка argv.
print('usage:', sys.argv[0], '<filename> <start> <end>')

# Можно использовать для генерации префикса пути выполняемой программы
# (в отличие от текущего модуля) для доступа к файлам относительно него,
# что было бы полезно, например, для ассетов игры
program_file = sys.argv[0]

import pathlib
program_path = pathlib.Path(program_file).resolve().parent
```

177.3. Ошибки и стандартный вывод

```
# Сообщения об ошибках по возможности не должны выводиться на стандартный вывод
print('ERROR: We have no cheese at all.', file=sys.stderr)

try:
    f = open('nonexistent-file.xyz', 'rb')
except OSError as e:
    print(e, file=sys.stderr)
```

177.4. Преждевременное завершение процесса и возврат кода выхода

```
def main():
    if len(sys.argv) != 4 or '--help' in sys.argv[1:]:
        print('usage: my_program <arg1> <arg2> <arg3>', file=sys.stderr)

    sys.exit(1) # использовать код выхода для сигнала о неудачном завершении программы

process_data()
```

Глава 178. Пакет ChemPy

ChemPy – это пакет, предназначенный в основном для решения задач физической, аналитической и неорганической химии. Это бесплатный инструмент с открытым исходным кодом на языке Python для применения в химии, химической инженерии и материаловедении.

178.1. Разбор формул

```
from chempy import Substance
ferricyanide = Substance.from_formula('Fe(CN)6-3')
ferricyanide.composition == {0: -3, 26: 1, 6: 6, 7: 6}
True
print(ferricyanide.unicode_name)
Fe(CN)63-
print(ferricyanide.latex_name + ", " + ferricyanide.html_name)
Fe(CN)63-, Fe(CN)63-
print("%.3f" % ferricyanide.mass)
211.955
```

В качестве ключей используются атомные числа (и 0 для заряда), а количество каждого вида становится соответствующим значением.

178.2. Стехиометрия химических реакций

```
from chempy import balance_stoichiometry # Основная реакция в ракетах-носителях NASA:
reac, prod = balance_stoichiometry({'NH4ClO4', 'Al'}, {'Al2O3', 'HCl', 'H2O', 'N2'})
from pprint import pprint
pprint(reac)
{'Al': 10, 'NH4ClO4': 6}
pprint(prod)
{'Al2O3': 5, 'H2O': 9, 'HCl': 6, 'N2': 3}
from chempy import mass_fractions
for fractions in map(mass_fractions, [reac, prod]):
    ... pprint({k: '{0:.3g} wt%'.format(v*100) for k, v in fractions.items()})

{'Al': '27.7 wt%', 'NH4ClO4': '72.3 wt%'}
{'Al2O3': '52.3 wt%', 'H2O': '16.6 wt%', 'HCl': '22.4 wt%', 'N2': '8.62 wt%'}
```

178.3. Уравнение химической реакции

```
from chempy import Equilibrium
from sympy import symbols
K1, K2, Kw = symbols('K1 K2 Kw')
e1 = Equilibrium({'MnO4-': 1, 'H+': 8, 'e-': 5}, {'Mn+2': 1, 'H2O': 4}, K1)
e2 = Equilibrium({'O2': 1, 'H2O': 2, 'e-': 4}, {'OH-': 4}, K2)
coeff = Equilibrium.eliminate([e1, e2], 'e-')
coeff
[4, -5]
redox = e1*coeff[0] + e2*coeff[1]
print(redox)
20 OH- + 32 H+ + 4 MnO4- = 26 H2O + 4 Mn+2 + 5 O2; K1**4/K2**5
autoprot = Equilibrium({'H2O': 1}, {'H+': 1, 'OH-': 1}, Kw)
n = redox.cancel(autoprot)
n
20
redox2 = redox + n*autoprot
print(redox2)
12 H+ + 4 MnO4- = 4 Mn+2 + 5 O2 + 6 H2O; K1**4*Kw**20/K2**5
```

178.4. Химическое равновесие

```
from chempy import Equilibrium
from chempy.chemistry import Species
water_autop = Equilibrium({'H2O'}, {'H+', 'OH-'}, 10**-14) # принята единица измерения "моль"
```

```

ammonia_prot = Equilibrium({'NH4+'}, {'NH3', 'H+'}, 10**-9.24) # здесь также
from chempy.equilibria import EqSystem
substances = map(Species.from_formula, 'H2O OH- H+ NH3 NH4+'.split())
eqsys = EqSystem([water_autop, ammonia_prot], substances)
print('\n'.join(map(str, eqsys.rxns))) # "rxns" здесь сокращение от "реакций"
H2O = H+ + OH-; 1e-14
NH4+ = H+ + NH3; 5.75e-10
from collections import defaultdict
init_conc = defaultdict(float, {'H2O': 1, 'NH3': 0.1})
x, sol, sane = eqsys.root(init_conc)
assert sol['success'] and sane
print(sorted(sol.keys())) # см. пакет "pyneqsys" для получения дополнительной информации
['fun', 'intermediate_info', 'internal_x_vecs', 'nfev', 'njev', 'success', 'x', 'x_vecs']
print(' '.join("%.2g" % v for v in x))
1, 0.0013, 7.6e-12, 0.099, 0.0013

```

178.5. Ионная сила

```

from chempy.electrolytes import ionic_strength
ionic_strength({'Fe+3': 0.050, 'ClO4-': 0.150}) == .3
True

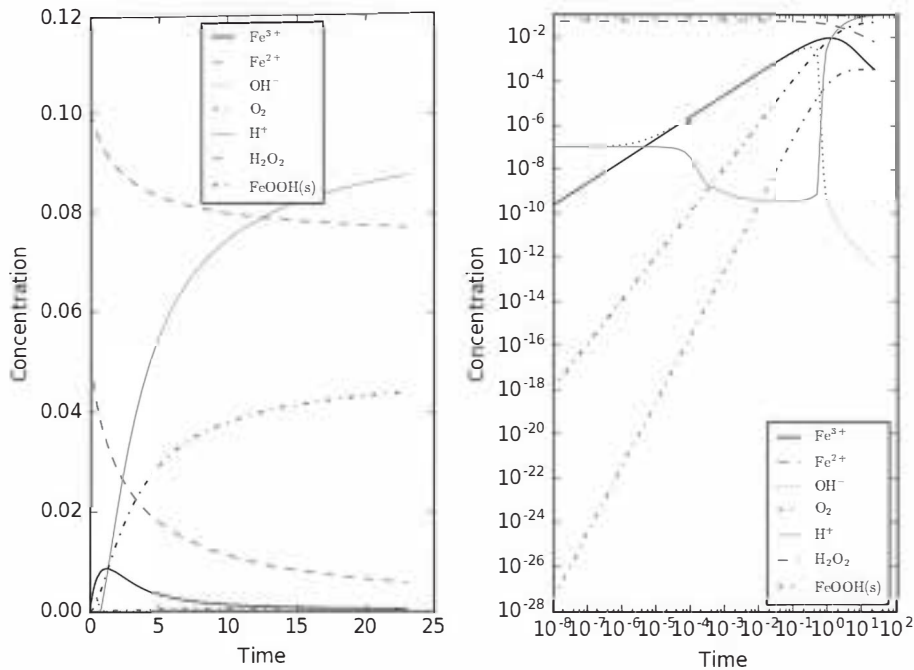
```

178.6. Химическая кинетика (система обыкновенных дифференциальных уравнений)

```

from chempy import ReactionSystem # Константы скорости ниже являются произвольными
rsys = ReactionSystem.from_string("""2 Fe+2 + H2O2 -> 2 Fe+3 + 2 OH-; 42
2 Fe+3 + H2O2 -> 2 Fe+2 + O2 + 2 H+; 17
H+ + OH- -> H2O; 1e10
H2O -> H+ + OH-; 1e-4
Fe+3 + 2 H2O -> FeOOH(s) + 3 H+; 1
FeOOH(s) + 3 H+ -> Fe+3 + 2 H2O; 2.5""")
from chempy.kinetics.ode import get_odesys
odesys, extra = get_odesys(rsys)
from collections import defaultdict
import numpy as np
tout = sorted(np.concatenate((np.linspace(0, 23), np.logspace(-8, 1))))
c0 = defaultdict(float, {'Fe+2': 0.05, 'H2O2': 0.1, 'H2O': 1.0, 'H+': 1e-7, 'OH-': 1e-7})
result = odesys.integrate(tout, c0, atol=1e-12, rtol=1e-14)
import matplotlib.pyplot as plt
_ = plt.subplot(1, 2, 1)
_ = result.plot(names=[k for k in rsys.substances if k != 'H2O'])
_ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ = plt.ylabel('Concentration')
_ = plt.subplot(1, 2, 2)
_ = result.plot(names=[k for k in rsys.substances if k != 'H2O'], xscale='log', yscale='log')
_ = plt.legend(loc='best', prop={'size': 9}); _ = plt.xlabel('Time'); _ = plt.ylabel('Concentration')
_ = plt.tight_layout()
plt.show()

```



Глава 179. Библиотека Pygame

Параметр	Подробности
count	Положительное целое число, представляющее собой нечто вроде количества каналов, которые необходимо зарезервировать.
force	Булево значение (False или True), определяющее, должен ли find_channel() возвращать канал (неактивный или нет) с True или нет (в случае если неактивных каналов нет) с False

Pygame – это библиотека для создания мультимедийных приложений, особенно игр, на языке Python. Официальный сайт: <http://www.pygame.org/>.

179.1. Модуль mixer в Pygame

Модуль pygame.mixer позволяет управлять музыкой, используемой в программах pygame. На данный момент существует 15 различных функций для модуля mixer.

Инициализация

Подобно тому, как необходимо инициализировать pygame с помощью pygame.init(), необходимо инициализировать и pygame.mixer.

Используя первую опцию, мы инициализируем модуль, используя значения по умолчанию. Однако эти значения можно переопределить. При использовании второй опции мы можем инициализировать модуль с помощью значений, которые мы сами вводим вручную. Стандартные значения:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096)
```

Чтобы проверить, инициализировали мы его или нет, можно использовать pygame.mixer.get_init(), которая возвращает True, если инициализировали, и False, если не инициализиро-

вали. Для выхода/отмены инициализации достаточно использовать команду `pygame.mixer.quit()`. Если вы хотите продолжить воспроизведение звуков с помощью модуля, то, возможно, придется инициализировать модуль заново.

Возможные действия

Во время воспроизведения звука его можно временно приостановить с помощью `pygame.mixer.pause()`. Чтобы возобновить воспроизведение звука, достаточно использовать `pygame.mixer.unpause()`. Также можно задать затухание звука по окончании с помощью `pygame.mixer.fadeout()`. Она принимает аргумент, который представляет собой количество миллисекунд, необходимое для затухания (fading out) музыки.

Каналы

Вы можете воспроизводить столько композиций, сколько необходимо, при условии, что открытых каналов достаточно для их поддержки. По умолчанию имеется 8 каналов. Чтобы изменить количество каналов, используйте `pygame.mixer.set_num_channels()`. Аргументом является целое неотрицательное число. При уменьшении числа каналов все звуки, воспроизводимые на удаленных каналах, немедленно прекращаются.

Чтобы узнать, сколько каналов используется в данный момент, вызовите `pygame.mixer.get_channels(count)`. В результате будет получено количество каналов, которые в данный момент не открыты. Также с помощью `pygame.mixer.set_reserved(count)` можно зарезервировать каналы для звуков, которые должны быть воспроизведены. Аргумент также является неотрицательным целым числом. Все звуки, воспроизводимые на зарезервированных каналах, не будут остановлены. Выяснить, какой канал не используется, можно с помощью `pygame.mixer.find_channel(force)`. Его аргументом является булево значение: либо `True`, либо `False`. Если не существует незадействованных каналов, а значение `force` равно `False`, то возвращается `None`. Если значение `force` равно `True`, то будет возвращен канал, который играл дольше всего.

179.2. Установка Pygame

При помощи `pip`:

```
pip install pygame
```

С использованием `conda`:

```
conda install -c tlatorre pygame=1.9.2
```

Прямая загрузка с сайта: <http://www.pygame.org/download.shtml>

Вы можете найти подходящие инсталляторы для Windows и других операционных систем. На сайте <http://www.pygame.org/> также можно найти примеры проектов.

Глава 180. Модуль Pyglet

Pyglet – это модуль Python, используемый для работы с визуальными и звуковыми эффектами. Он не имеет зависимостей от других модулей. Подробную информацию см. на сайте <http://pyglet.org>.

180.1. Установка Pyglet

Установите Python, войдите в командную строку и введите:

В Python 2:

```
pip install pyglet
```

В Python 3:

```
pip3 install pyglet
```

180.2. "Hello World" в Pyglet

```
import pyglet
window = pyglet.window.Window()
label = pyglet.text.Label('Hello, world',
    font_name='Times New Roman',
    font_size=36,
    x=window.width//2, y=window.height//2,
    anchor_x='center', anchor_y='center')
@window.event
def on_draw():
    window.clear()
    label.draw()
pyglet.app.run()
```

180.3. Воспроизведение звука в Pyglet

```
sound = pyglet.media.load(sound.wav)
sound.play()
```

180.4. Использование Pyglet для OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()

@win.event()
def on_draw():
    #Используйте OpenGL как обычно.

pyglet.app.run()
```

180.5. Рисование точек с помощью Pyglet и OpenGL

```
import pyglet
from pyglet.gl import *

win = pyglet.window.Window()
glClear(GL_COLOR_BUFFER_BIT)

@win.event
def on_draw():
    glBegin(GL_POINTS)
    glVertex2f(x, y) # x - желаемое расстояние от левой части окна,
    # y - желаемое расстояние от нижней части окна
    # создайте столько вершин, сколько хотите
    glEnd

    Чтобы соединить точки, замените GL_POINTS на GL_LINE_LOOP.
```

Глава 181. Работа со звуком

181.1. Работа с файлами в формате WAV

winsound

- Среда Windows

```
import winsound
winsound.PlaySound("path_to_wav_file.wav", winsound.SND_FILENAME)
```


wave

- Поддержка моно/стерео
- Не поддерживает компрессию/декомпрессию

```
import wave
with wave.open("path_to_wav_file.wav", "rb") as wav_file: # Открыть WAV-файл в режиме "только чтение".
    # Получение базовой информации.
    n_channels = wav_file.getnchannels()           # Количество каналов (1=Mono, 2=Stereo).
    sample_width = wav_file.getsampwidth()         # Ширина выборки в байтах.
    framerate = wav_file.getframerate()           # Частота.
    n_frames = wav_file.getnframes()              # Число фреймов.
    comp_type = wav_file.getcomptype()            # Тип сжатия (поддерживается только "NONE").
    comp_name = wav_file.getcompname()            # Имя сжатия.

    # Считывание аудиоданных.
    frames = wav_file.readframes(n_frames) # Считывание n_frames новых фреймов.
    assert len(frames) == sample_width * n_frames

# Дублирование в новый WAV-файл
with wave.open("path_to_new_wav_file.wav", "wb") as wav_file: # Открыть WAV-файл в режиме
# "только запись".
    # Запись аудиоданных
    params = (n_channels, sample_width, framerate, n_frames, comp_type, comp_name)
    wav_file.setparams(params)
    wav_file.writeframes(frames)
```

181.2. Преобразование любого звукового файла с помощью Python и ffmpeg

```
from subprocess import check_call
```

```
ok = check_call(['ffmpeg', '-i', 'input.mp3', 'output.wav'])
if ok:
    with open('output.wav', 'rb') as f:
        wav_file = f.read()
```

Рекомендованные ресурсы:

1. <http://superuser.com/questions/507386/why-would-i-choose-libav-over-ffmpeg-or-is-there-even-a-difference>
2. Каковы различия и сходства между ffmpeg, libav и avconv? <http://stackoverflow.com/questions/9477115/what-are-the-differences-and-similarities-between-ffmpeg-libav-and-avconv>

181.3. Воспроизведение звуковых сигналов Windows

Windows предоставляет интерфейс, через который модуль winsound позволяет воспроизводить необработанные звуковые сигналы с заданной частотой и длительностью.

```
import winsound
freq = 2500          # Установить частоту 2500 Гц
dur = 1000          # Установить длительность в 1000 мс == 1 секунда
winsound.Beep(freq, dur)
```

181.4. Воспроизведение звука с помощью Pyglet

```
import pyglet
audio = pyglet.media.load("audio.wav")
audio.play()
```

Дополнительную информацию см. по адресу https://pyglet.readthedocs.io/en/pyglet-1.2-maintenance/programming_guide/media.html

Глава 182. Pyaudio

Инструмент PyAudio обеспечивает связь Python с PortAudio, кроссплатформенной библиотекой ввода-вывода аудиоданных. С помощью PyAudio вы можете легко использовать Python для воспроизведения и записи звука на различных платформах. В основе PyAudio находятся:

1. pyPortAudio/fastaudio: связка Python для API PortAudio v18.
2. tkSnack: кроссплатформенный звуковой инструментарий для Tcl/Tk и Python.

182.1. Режим обратного вызова звукового ввода/вывода

"""Пример PyAudio: Воспроизведение .wav-файла (режим обратного вызова)"""

```
import pyaudio
import wave
import time
import sys

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# инстанцировать PyAudio (1)
p = pyaudio.PyAudio()

# определить обратный вызов (2)
def callback(in_data, frame_count, time_info, status):
    data = wf.readframes(frame_count)
    return (data, pyaudio.paContinue)

# открыть поток с помощью обратного вызова (3)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True,
                stream_callback=callback)

# запустить поток (4)
stream.start_stream()

# ожидание завершения потока (5)
while stream.is_active():
    time.sleep(0.1)

# остановить поток (6)
stream.stop_stream()
stream.close()
wf.close()

# закрыть PyAudio (7)
p.terminate()
```

В режиме обратного вызова PyAudio будет вызывать определенную функцию обратного вызова (2) всякий раз, когда ему потребуются новые аудиоданные (для воспроизведения) и/или когда будут доступны новые (записанные) аудиоданные. Обратите внимание, что PyAudio вызывает функцию обратного вызова в отдельном потоке. Функция имеет следующую

шую сигнатуру `callback(<input_data>, <frame_count>, <time_info>, <status_flag>)` и должна возвращать кортеж, содержащий `frame_count` фреймов аудиоданных и флаг, который обозначает, есть ли еще фреймы для воспроизведения/записи.

Начните обработку аудиопотока с помощью функции `pyaudio.Stream.start_stream()` (4), которая будет вызывать функцию обратного вызова несколько раз, пока эта функция не вернет значение `pyaudio.paComplete`.

Чтобы поток оставался активным, основной поток не должен завершаться, например из-за перехода в спящий режим (5).

182.2. Режим блокировки звукового ввода/вывода

""" Пример PyAudio: Воспроизведение .wav-файла. """

```
import pyaudio
import wave
import sys

CHUNK = 1024

if len(sys.argv) < 2:
    print("Plays a wave file.\n\nUsage: %s filename.wav" % sys.argv[0])
    sys.exit(-1)

wf = wave.open(sys.argv[1], 'rb')

# инстанцировать PyAudio (1)
p = pyaudio.PyAudio()

# запустить поток (2)
stream = p.open(format=p.get_format_from_width(wf.getsampwidth()),
                channels=wf.getnchannels(),
                rate=wf.getframerate(),
                output=True)

# считывание данных
data = wf.readframes(CHUNK)

# воспроизвести поток (3)
while len(data) > 0:
    stream.write(data)
    data = wf.readframes(CHUNK)

# остановить поток (4)
stream.stop_stream()
stream.close()

# закрыть PyAudio (5)
p.terminate()
```

Чтобы использовать PyAudio, для начала инстанцируйте его с помощью `pyaudio.PyAudio()` (1), который устанавливает систему `portaudio`.

Для записи или воспроизведения звука откройте на нужном устройстве поток с нужными аудиопараметрами с помощью `pyaudio.PyAudio.open()` (2). При этом создается поток `pyaudio.Stream` для воспроизведения или записи звука.

Воспроизведение звука осуществляется путем записи аудиоданных в поток с помощью функции `pyaudio.Stream.write()` или считывания аудиоданных из потока с помощью функции `pyaudio.Stream.read()`. (3)

Обратите внимание, что в "блокирующем режиме" каждый `pyaudio.Stream.write()` или `pyaudio.Stream.read()` блокируется до тех пор, пока не будут воспроизведены или записаны

все заданные/запрошенные кадры. В качестве альтернативы, чтобы генерировать аудиоданные на выходе или немедленно обрабатывать записанные аудиоданные, используйте “режим обратного вызова” (см. предыдущий пример).

Для приостановки воспроизведения/записи используйте `pyaudio.Stream.stop_stream()`, а для завершения потока – `pyaudio.Stream.close()`. (4)

Наконец, завершите сеанс `portaudio` с помощью `pyaudio.PyAudio.terminate()` (5)

Глава 183. Модуль Shelve

Shelve (“полка”) – это модуль Python, используемый для хранения объектов в файле. Этот модуль реализует постоянное хранилище для произвольных объектов Python, которые можно обрабатывать модулем `Pickle`, используя API, подобный словарю. Shelve может использоваться как простой вариант постоянного хранилища для объектов Python, когда реляционная база данных является излишеством. Доступ к “полке” осуществляется по ключам, как в словаре. Значения обрабатываются модулем `Pickle` и записываются в базу данных, созданную и управляемую модулем `anydbm`.

183.1. Использование Shelve

Самый простой способ использования Shelve – через класс `DbfilenameShelf`. Он использует модуль `anydbm` для хранения данных. Вы можете использовать класс напрямую или просто вызвать `shelve.open()`:

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float': 9.5, 'string': 'Sample data' }
finally:
    s.close()
```

Чтобы снова получить доступ к данным, откройте Shelve и используйте ее как словарь:

```
import shelve

s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

Если запустить оба примера скриптов, то можно увидеть:

```
$ python shelve_create.py
$ python shelve_existing.py
```

```
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Модуль `dbm` не поддерживает одновременную запись нескольких приложений в одну и ту же базу данных. Если вы знаете, что ваш клиент не будет модифицировать shelve, вы можете указать, что следует открыть базу данных только для чтения.

```
import shelve

s = shelve.open('test_shelf.db', flag='r')
try:
```

```

    existing = s['key1']
finally:
    s.close()

print existing

```

Если ваша программа пытается модифицировать базу данных, когда она открыта только для чтения, то генерируется исключение об ошибке доступа. Тип исключения зависит от модуля базы данных, выбранного модулем `anydbm` при создании базы.

183.2. Пример кода для Shelve

Чтобы поместить объект на Shelve, сначала импортируйте модуль, а затем присвойте значение объекта следующим образом:

```

import shelve
database = shelve.open(filename.suffix)
object = Object()
database['key'] = object

```

183.3. Обобщение информации о интерфейсе

Ключ – это строка, данные – произвольный объект:

```

import shelve

d = shelve.open(filename) # открыть -- файл может получить суффикс, добавленный
                           # низкоуровневой библиотекой

d[key] = data              # хранить данные по ключу (перезаписывает старые данные, если
                           # используется существующий ключ)
data = d[key]              # получить копию данных, хранящихся по ключу (ошибка KeyError
                           # если такого ключа нет)
del d[key]                 # удалить данные, хранящиеся по ключу (вызывает ошибку KeyError
                           # если такого ключа нет)

flag = key in d             # true если ключ существует
klist = list(d.keys())      # список всех существующих ключей (работает медленно!)

# поскольку d был открыт БЕЗ writeback=True, остерегайтесь:
d['xx'] = [0, 1, 2]         # это работает, как и ожидалось, но...
d['xx'].append(3)           # это нет – d['xx'] все еще [0, 1, 2]!

# открыв d без writeback=True, необходимо внимательно отнестись к коду:
temp = d['xx']              # извлекает копию
temp.append(5)              # изменяет копию
d['xx'] = temp              # сохраняет копию обратно, чтобы сохранить ее

# или, d=shelve.open(filename,writeback=True) позволит вам просто закодировать
# d['xx'].append(5) и заставить его работать как ожидалось, но это также
# будет занимать больше памяти и сделает операцию d.close() более медленной.

d.close()                  # закрыть

```

183.4. Обратная запись (writeback)

По умолчанию Shelve не отслеживает изменения непостоянных объектов. Это означает, что при изменении содержимого элемента, хранящегося в Shelve, необходимо явно обновить Shelve, заново сохранив этот элемент.

```
import shelve

s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'раньше этого здесь не было'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

В данном примере словарь по адресу 'key1' не сохраняется, поэтому при повторном открытии "полки" изменения не сохраняются.

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py
```

```
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

Для автоматического отслеживания состояния изменчивых объектов, хранящихся на полке, откройте "полку" с включенной функцией writeback. При включении функции writeback Shelve запоминает все объекты, извлеченные из базы данных, с помощью кэша в памяти. При закрытии каждый объект кэша также записывается обратно в базу данных.

```
import shelve

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'раньше этого здесь не было'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

Хотя это снижает вероятность ошибки программиста и делает сохранение объектов более прозрачным, использование такого режима может быть нежелательным в любой ситуации. Кэш потребляет дополнительную память, пока Shelve открыта, а пауза для записи каждого кэшированного объекта обратно в базу данных при ее закрытии может занять дополнительное время. Поскольку нет возможности определить, были ли модифицированы кэшированные объекты, все они записываются обратно. Если ваше приложение считывает данные чаще, чем записывает, то обратная запись будет занимать больше времени, чем хотелось бы.

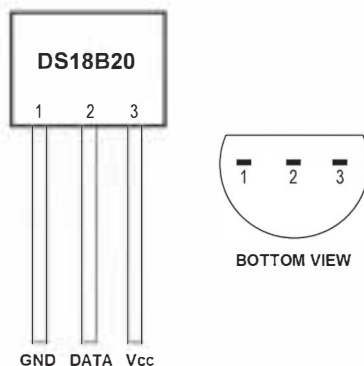
```
$ python shelve_create.py
$ python shelve_writeback.py
```

```
{'int': 10, 'float': 9.5, 'string': 'Образец данных'}
{'int': 10, 'new_value': 'раньше этого здесь не было', 'float': 9.5, 'string': 'Образец данных'}
{'int': 10, 'new_value': 'раньше этого здесь не было', 'float': 9.5, 'string': 'Образец данных'}
```

Глава 184. Программирование “интернета вещей” (IoT) с помощью Python и Raspberry Pi

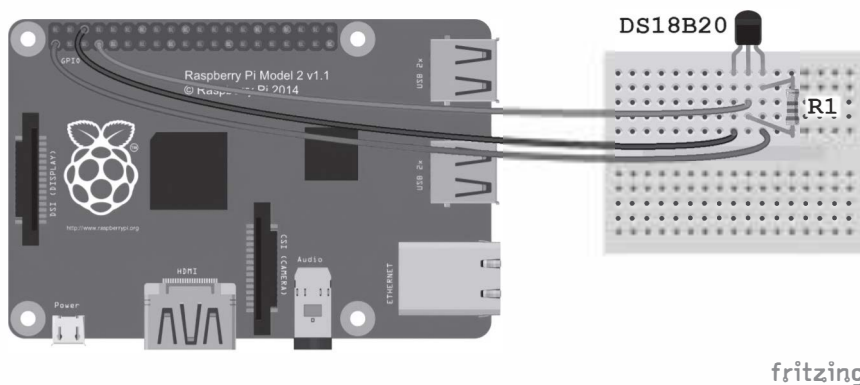
184.1. Пример – датчик температуры

Рассмотрим подключение датчика DS18B20 к мини-компьютеру Raspberry Pi



Вы можете видеть, что у датчика есть три клеммы:

1. Vcc
2. Gnd
3. Для передачи данных (One wire, однопроводной протокол)



R1 — сопротивление 4,7 кОм для повышения уровня напряжения

1. Vcc должен быть подключен к любому из выводов 5 В или 3,3 В Raspberry Pi (PIN : 01, 02, 04, 17).
2. Gnd должен быть подключен к любому из выводов Gnd на Raspberry Pi (PIN : 06, 09, 14, 20, 25).
3. DATA должен быть подключен к (PIN : 07)

Включение однопроводного интерфейса со стороны RPi

4. Войдите в систему Rasperry pi, используя putty или любой другой терминал linux/unix.
5. После входа в систему откройте файл /boot/config.txt file в любимом браузере.

```
nano /boot/config.txt
```

6. Добавьте строку `dtoverlay=w1-gpio` в конец файла.
7. Теперь перезагрузите Raspberry pi: `sudo reboot`.

8. Войдите в систему Raspberry pi и выполните команду `sudo modprobe g1-gpio`.
9. Затем выполните команду `sudo modprobe w1-therm`.
10. Теперь перейдите в каталог `/sys/bus/w1/devices` `cd /sys/bus/w1/devices`.
11. Теперь вы обнаружите, что для вашего датчика температуры создан виртуальный каталог, начинающийся с адреса `28-*****`.
12. Перейдите в этот каталог `cd 28-*****`.
13. Теперь найдите там файл с именем `w1-slave`, он содержит температуру и другую информацию, например CRC: `cat w1-slave`.

Теперь напомним модуль на языке Python для чтения температуры

```
import glob
import time

RATE = 30
sensor_dirs = glob.glob("/sys/bus/w1/devices/28*")

if len(sensor_dirs) != 0:
    while True:
        time.sleep(RATE)
        for directories in sensor_dirs:
            temperature_file = open(directories + "/w1_slave")
            # Чтение файлов
            text = temperature_file.read()
            temperature_file.close()
            # Разделите текст на новые строки (\n) и выделите вторую строку.
            second_line = text.split("\n")[1]
            # Разделите строку на слова и выделить 10-е слово
            temperature_data = second_line.split(" ")[9]
            # будем считывать после игнорирования первых двух символов
            temperature = float(temperature_data[2:])
            # Теперь нормализуем температуру путем деления на 1000.
            temperature = temperature / 1000
            print 'Address : ' + str(directories.split('/')[1]) + ', Temperature : ' + str(temperature)
```

Приведенный выше Python-модуль выводит зависимость температуры от значения Address в течение бесконечного времени. Параметр RATE предназначен для изменения или настройки частоты запросов температуры от датчика.

Схема входов-выходов GPIO: https://www.element14.com/community/servlet/JiveServlet/previewBody/73950-102-11-339300/pi3_gpio.png

Глава 185. Kivy – кроссплатформенный Python-фреймворк для разработки естественных пользовательских интерфейсов (NUI)

NUI, Естественный пользовательский интерфейс (Natural User Interface) – это система взаимодействия человека и компьютера, которой пользователь управляет с помощью интуитивных действий, связанных с естественным, повседневным поведением человека.

Kivy – это библиотека на языке Python для разработки мультисенсорных приложений, которые могут быть установлены на различных устройствах. Мультикас – это способность сенсорной поверхности (обычно это сенсорный экран или трекпад) обнаруживать или воспринимать ввод от двух или более точек контакта одновременно.

185.1. Первое приложение

Для создания приложения Kivy:

1. Подклассифицируйте класс **app**.
2. Реализуйте метод **build**, который будет возвращать виджет.
3. Инстанцируйте класс и вызовите команду **run**.

```
from kivy.app import App
from kivy.uix.label import Label
```

```
class Test(App):
    def build(self):
        return Label(text='Hello world')
```

```
if __name__ == '__main__':
    Test().run()
```

Пояснение

```
from kivy.app import App
```

Приведенное выше утверждение импортирует родительский класс **app**. Он будет присутствовать в каталоге установки `your_installation_directory/kivy/app.py`.

```
from kivy.uix.label import Label
```

Приведенное выше утверждение импортирует их-элемент **Label**. Все их-элементы присутствуют в каталоге установки `your_installation_directory/kivy/uix/`.

```
class Test(App):
```

Приведенное выше утверждение служит для создания приложения, а имя класса будет именем вашего приложения. Этот класс наследуется от родительского класса приложения.

```
    def build(self):
```

Приведенное выше утверждение переопределяет метод `build` класса `app`. В результате будет возвращен виджет, который должен быть показан при запуске приложения.

```
        return Label(text='Hello world')
```

Приведенное выше утверждение является телом метода `build`. Он возвращает `Label` с текстом "Hello world".

```
if __name__ == '__main__':
```

Приведенное выше утверждение является точкой входа, с которой интерпретатор Python начинает выполнение вашего приложения.

```
    Test().run()
```

Приведенный выше оператор инициализирует класс `Test`, создавая его экземпляр, и вызывает функцию класса приложения `run()`. Ваше приложение будет представлять собой окно с выведенным сообщением "Hello world".

Глава 186. Использование Pandas transform: предварительное выполнение операций с группами и конкатенация результатов

186.1. Простое преобразование

Сначала создадим фиктивный датафрейм.

Предположим, что клиент может иметь *n* заказов, заказ может содержать *m* товаров, а товары могут быть заказаны несколько раз:

```
orders_df = pd.DataFrame()
orders_df['id_клиента'] = [1,1,1,1,1,2,2,3,3,3,3]
orders_df['id_заказа'] = [1,1,1,2,2,3,3,4,5,6,6]
orders_df['item'] = ['яблоки', 'шоколад', 'шоколад', 'кофе', 'кофе', 'яблоки',
'бананы', 'кофе', 'молочный коктейль', 'шоколад', 'клубника', 'клубника']
```

Вот как выглядит датафрейм:

```
print(orders_df)
```

#	id_клиента	id_заказа	item
# 0	1	1	яблоки
# 1	1	1	шоколад
# 2	1	1	шоколад
# 3	1	2	кофе
# 4	1	2	кофе
# 5	2	3	яблоки
# 6	2	3	бананы
# 7	3	4	кофе
# 8	3	5	молочный коктейль
# 9	3	6	шоколад
# 10	3	6	клубника
# 11	3	6	клубника

Теперь мы воспользуемся функцией pandas transform для подсчета количества заказов на одного клиента:

```
# Сначала определим функцию, которая будет применяться для каждого id_клиента
count_number_of_orders = lambda x: len(x.unique())
```

И теперь мы можем преобразовать каждую группу, используя логику, определенную выше.

```
orders_df['число заказов на клиента'] = ( # Поместить результаты в новый столбец
'число заказов на клиента'
    orders_df # Взять исходный датафрейм
    .groupby(['id_клиента'])['id_заказа'] # Создаем отдельную группу для каждого
id_клиента и выбираем id_заказа
    .transform(count_number_of_orders)) # Применить функцию к каждой группе отдельно
```

Проверка результатов...

```
print(orders_df)
```

#	id_клиента	id_заказа	item	число заказов на клиента
# 0	1	1	яблоки	2
# 1	1	1	шоколад	2
# 2	1	1	шоколад	2
# 3	1	2	кофе	2
# 4	1	2	кофе	2

# 5	2	3	яблоки	1
# 6	2	3	бананы	1
# 7	3	4	кофе	3
# 8	3	5	молочный коктейль	3
# 9	3	6	шоколад	3
# 10	3	6	клубника	3
# 11	3	6	клубника	3

186.2. Несколько результатов для одной группы

Используем функции `transform`, возвращающие подвычисления для каждой группы

В предыдущем примере мы имели один результат для каждого клиента. Однако можно применять и функции, возвращающие разные значения для группы.

```
# Создадим фиктивный датафрейм
```

```
orders_df = pd.DataFrame()
```

```
orders_df['id_клиента'] = [1,1,1,1,1,2,2,3,3,3,3,3]
```

```
orders_df['id_заказа'] = [1,1,1,2,2,3,3,4,5,6,6,6]
```

```
orders_df['item'] = ['яблоки', 'шоколад', 'шоколад', 'кофе', 'кофе', 'яблоки', 'бананы', 'кофе',  
'молочный коктейль', 'шоколад', 'клубника', 'клубника']
```

```
# Попробуем проверить, были ли товары заказаны более одного раза в каждом заказе
```

```
# Сначала определим функцию, которая будет применяться к каждой группе
```

```
def multiple_items_per_order(_items):
```

```
    # Примените функцию .duplicated, которая будет возвращать True, если элемент встречается  
    # более одного раза
```

```
    multiple_item_bool = _items.duplicated(keep=False)
```

```
    return(multiple_item_bool)
```

```
# Затем преобразуем каждую группу в соответствии с заданной функцией
```

```
orders_df['дублирующийся товар в заказе'] = ( # Поместить результаты в новый столбец
```

```
    orders_df # Возьмем датафрейм по заказам
```

```
    .groupby(['id_заказа'])['item'] # Создаем отдельную группу для каждого id_заказа и
```

```
выбираем товар (item)
```

```
    .transform(multiple_items_per_order)) # Применить определенную функцию
```

```
к каждому элементу группы отдельно
```

```
# проверка результатов...
```

```
print(orders_df)
```

#	id_клиента	id_клиента	item	дублирующийся товар в заказе
# 0	1	1	яблоки	False
# 1	1	1	шоколад	True
# 2	1	1	шоколад	True
# 3	1	2	кофе	True
# 4	1	2	кофе	True
# 5	2	3	яблоки	False
# 6	2	3	бананы	False
# 7	3	4	кофе	False
# 8	3	5	молочный коктейль	False
# 9	3	6	шоколад	False
# 10	3	6	клубника	True
# 11	3	6	клубника	True

Глава 187. Сходство в синтаксисе, различия в значении: Python и JavaScript

Иногда случается, что два языка вкладывают в одно и то же или схожее по синтаксису выражение разные смыслы. Когда оба языка представляют интерес для программиста, прояснение этих точек бифуркации помогает лучше понять оба языка в их основах и тонкостях.

187.1. `in` со списками

2 in [2, 3]

В Python это значение равно True, а в JavaScript – False. Это связано с тем, что в Python `in` проверяет, содержится ли значение в списке, поэтому 2 находится в [2, 3] как его первый элемент. В JavaScript `in` используется с объектами и проверяет, содержит ли объект свойство с именем, выраженным значением. Таким образом, JavaScript рассматривает [2, 3] как объект или карту “ключ-значение”, наподобие:

{0: 2, 1: 3}

и проверяет, есть ли в нем свойство или ключ ‘2’. Целое число 2 преобразуется в строку ‘2’.

Глава 188. Вызов Python из C#

В документации приведен пример реализации межпроцессного взаимодействия между сценариями на языках программирования C# и Python.

188.1. Python-скрипт, который вызывается C#-приложением

```
import sys
import json

# загрузить входные аргументы из текстового файла
filename = sys.argv[ 1 ]
with open( filename ) as data_file:
    input_args = json.loads( data_file.read() )

# приведение строк к типу float
x, y = [ float(input_args.get( key )) for key in [ 'x', 'y' ] ]

print json.dumps( { 'sum' : x + y , 'subtract' : x - y } )
```

188.2. Код на C#, вызывающий Python-скрипт

```
using MongoDB.Bson;
using System;
using System.Diagnostics;
using System.IO;

namespace python_csharp
{
    class Program
    {
```

```

static void Main(string[] args)
{
    // полный путь к файлу .py
    string pyScriptPath = "...../sum.py";
    // преобразование входных аргументов в строку JSON
    BsonDocument argsBson = BsonDocument.Parse("{ 'x' : '1', 'y' : '2' }");

    bool saveInputFile = false;

    string argsFile = string.Format("{0} \\{1}.txt", Path.GetDirectoryName(pyScriptPath), Guid.NewGuid());

    string outputString = null;
    // создание нового процесса start info
    ProcessStartInfo prcStartInfo = new ProcessStartInfo
    {
        // полный путь к интерпретатору Python 'python.exe'
        FileName = "python.exe", // string.Format(@"{0}", "python.exe"),
        UseShellExecute = false,
        RedirectStandardOutput = true,
        CreateNoWindow = false
    };

    try
    {
        // запись входных аргументов в файл .txt
        using (StreamWriter sw = new StreamWriter(argsFile))
        {
            sw.WriteLine(argsBson);
            prcStartInfo.Arguments = string.Format("{0} {1}", string.Format(@"{0}", pyScriptPath),
string.Format(@"{0}", argsFile));
        }
        // запуск процесса
        using (Process process = Process.Start(prcStartInfo))
        {
            // чтение стандартной выходной JSON-строки
            using (StreamReader myStreamReader = process.StandardOutput)
            {
                outputString = myStreamReader.ReadLine();
                process.WaitForExit();
            }
        }
    }
    finally
    {
        // удаление/сохранение временного файла .txt
        if (!saveInputFile)
        {
            File.Delete(argsFile);
        }
    }
    Console.WriteLine(outputString);
}
}
}

```

Глава 189. Библиотека ctypes

ctypes – это встроенная библиотека Python, которая вызывает экспортируемые функции из собственных скомпилированных библиотек.

Примечание: поскольку эта библиотека работает с скомпилированным кодом, она относительно зависима от ОС.

189.1. Массивы ctypes

Как известно любому хорошему программисту на языке C – то, что действительно поможет продвинуться, так это массивы!

```
>>> c_int * 16
<class '__main__.c_long_Array_16'>
```

Это не настоящий массив, но очень близко к нему! Мы создали класс, обозначающий массив из 16 целых чисел. Теперь осталось только инициализировать его:

```
>>> arr = (c_int * 16)(*range(16))
>>> arr
<__main__.c_long_Array_16 object at 0xbaddcfae>
```

Теперь arr – это настоящий массив, содержащий числа от 0 до 15. Доступ к нему можно получить так же, как и к любому списку:

```
>>> arr[5]
5
>>> arr[5] = 20
>>> arr[5]
20
```

Как и любой другой объект ctypes, он также имеет размер и местоположение:

```
>>> sizeof(arr)
64 # размер(c_int) * 16
>>> hex(addressof(arr))
'0xc00010ff'
```

189.2. Обертывающие функции для ctypes

В некоторых случаях функция языка C принимает указатель функции. Как заядлые пользователи ctypes, мы хотели бы использовать эти функции и даже передавать в качестве аргументов функции Python.

Определим функцию:

```
>>> def max(x, y):
    return x if x >= y else y
```

Теперь эта функция принимает два аргумента и возвращает результат того же типа. Для примера предположим, что этот тип – int.

Как и в примере с массивом, мы можем определить объект, обозначающий этот прототип:

```
>>> CFUNCTYPE(c_int, c_int, c_int)
<CFunctionType object at 0xdeadbeef>
```

Этот прототип обозначает функцию, которая возвращает c_int (первый аргумент) и принимает два аргумента c_int (остальные аргументы).

Теперь обернем функцию:

```
>>> CFUNCTYPE(c_int, c_int, c_int)(max)
<CFunctionType object at 0xdeadbeef>
```

Прототипы функций имеют еще одно применение: Они могут обернуть функцию ctypes (например, libc.ntohl) и проверить, что при вызове функции используются правильные аргументы.

```
>>> libc.ntohl()
>>> CFUNCTYPE(c_int, c_int)(libc.ntohl)()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: this function takes at least 1 argument (0 given) (эта функция принимает как минимум
1 аргумент (задано 0))
```

189.3. Базовое использование

Допустим, мы хотим использовать функцию `ntohl` в `libc`. Сначала мы должны загрузить `libc.so`:

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('libc.so.6')
>>> libc
<CDLL 'libc.so.6', handle baadf00d at 0xdeadbeef>
```

Затем мы получаем объект функции:

```
>>> ntohl = libc.ntohl
>>> ntohl
<_FuncPtr object at 0xbaadf00d>
```

И теперь мы можем просто вызвать функцию:

```
>>> ntohl(0x6C)
1811939328
>>> hex(_)
'0x6c000000'
```

и получить именно то, чего мы ожидали.

189.4. Основные “подводные камни”

Невозможность загрузки файла

Первая возможная ошибка – это невозможность загрузить библиотеку. В этом случае обычно выдается ошибка `OSError`. Это происходит либо потому, что библиотека не существует (либо не может быть найдена ОС):

```
>>> cdll.LoadLibrary("foobar.so")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/ctypes/_init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
  File "/usr/lib/python3.5/ctypes/_init__.py", line 347, in _init__
    self._handle = _dlopen(self._name, mode)
OSError: foobar.so: cannot open shared object file: No such file or directory
```

Как видите, ошибка явная и довольно показательная.

Вторая причина заключается в том, что файл найден, но имеет неправильный формат.

```
>>> cdll.LoadLibrary("libc.so")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.5/ctypes/_init__.py", line 425, in LoadLibrary
    return self._dlltype(name)
  File "/usr/lib/python3.5/ctypes/_init__.py", line 347, in _init__
    self._handle = _dlopen(self._name, mode)
OSError: /usr/lib/i386-linux-gnu/libc.so: invalid ELF header
```

В этом случае файл является скриптовым файлом, а не `.so`-файлом. Подобное может произойти и при попытке открыть `.dll`-файл на Linux-машине или 64-разрядный файл на 32-разрядном интерпретаторе Python. Как видите, в этом случае ошибка более расплывчата и требует некоторого поиска.

Невозможность доступа к функции

Если мы успешно загрузили файл .so, то нам нужно обратиться к нашей функции, как это было сделано в первом примере. При использовании несуществующей функции возникает ошибка `AttributeError`:

```
>>> libc.foo
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python3.5/ctypes/__init__.py", line 360, in __getattr__
    func = self.__getitem__(name)
File "/usr/lib/python3.5/ctypes/__init__.py", line 365, in __getitem__
    func = self._FuncPtr((name_or_ordinal, self))
AttributeError: /lib/i386-linux-gnu/libc.so.6: undefined symbol: foo
```

189.5. Базовый объект ctypes

Самым простым объектом является целое число:

```
>>> obj = ctypes.c_int(12)
>>> obj
c_long(12)
```

Теперь `obj` ссылается на участок памяти, содержащий значение 12. К этому значению можно обращаться напрямую и даже модифицировать его:

```
>>> obj.value
12
>>> obj.value = 13
>>> obj
c_long(13)
```

Поскольку `obj` ссылается на участок памяти, мы также можем определить его размер и местоположение:

```
>>> sizeof(obj)
4
>>> hex(addressof(obj))
'0xdeadbeef'
```

189.6. Комплексное использование

Объединим все приведенные примеры в один комплексный сценарий: использование функции `libc.lfind`. Более подробно об этой функции можно прочитать на странице: <https://linux.die.net/man/3/lfind>.

Для начала определим соответствующие прототипы:

```
>>> compar_proto = CFUNCTYPE(c_int, POINTER(c_int), POINTER(c_int))
>>> lfind_proto = CFUNCTYPE(c_void_p, c_void_p, c_void_p, POINTER(c_uint), c_uint, compar_proto)
```

Затем создадим переменные:

```
>>> key = c_int(12)
>>> arr = (c_int * 16)(*range(16))
>>> nmemb = c_uint(16)
```

А теперь определим функцию сравнения:

```
>>> def compar(x, y):
    return x.contents.value - y.contents.value
```

Обратите внимание, что `x` и `y` являются `POINTER(c_int)`, поэтому для реального сравнения со значением, хранящимся в памяти, нам необходимо “разыменовать” их и взять их значения.

Теперь мы можем объединить все вместе:

```
>>> lfind = lfind_proto(libc.lfind)
>>> ptr = lfind(byref(key), byref(arr), byref(nmemb), sizeof(c_int), compar_proto(compar))
```


ptr – возвращаемый указатель void. Если бы ключ не был найден в arr, то значение было бы None, но в данном случае мы получили корректное значение.

Теперь мы можем преобразовать его и получить доступ к значению:

```
>>> cast(ptr, POINTER(c_int)).contents
c_long(12)
```

Также мы видим, что ptr указывает на правильное значение внутри arr:

```
>>> addressof(arr) + 12 * sizeof(c_int) == ptr
True
```

Глава 190. Написание расширений

190.1. “Hello World” расширение на C

Следующий файл на языке C (который мы для демонстрации будем называть hello.c) создает модуль расширения hello, содержащий единственную функцию greet():

```
#include <Python.h>
#include <stdio.h>

#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif

static PyObject *hello_greet(PyObject *self, PyObject *args)
{
    const char *input;
    if (!PyArg_ParseTuple(args, "s", &input)) {
        return NULL;
    }
    printf("%s", input);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] = {
    { "greet", hello_greet, METH_VARARGS, "Приветствовать пользователя" },
    { NULL, NULL, 0, NULL }
};

#ifdef IS_PY3K
static struct PyModuleDef hellomodule = {
    PyModuleDef_HEAD_INIT, "hello", NULL, -1, HelloMethods
};

PyMODINIT_FUNC PyInit_hello(void)
{
    return PyModule_Create(&hellomodule);
}
#else
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", HelloMethods);
}
#endif
```

Чтобы скомпилировать файл с помощью компилятора gcc, выполните в своем любимом терминале следующую команду:

```
gcc /path/to/your/file/hello.c -o /path/to/your/file/hello
```

Для выполнения функции `greet()`, которую мы написали ранее, создайте в том же каталоге файл и назовите его `hello.py`.

```
import hello # импортирует скомпилированную библиотеку
hello.greet("Hello!") # запуск функции greet() с аргументом "Hello!"
```

190.2. Расширение на языке C с помощью C++ и библиотеки Boost

Это базовый пример C-расширения на языке C++ с использованием библиотеки Boost.

Код на C++

Код на языке C++ помещен в файл `hello.cpp`:

```
#include <boost/python/module.hpp>
#include <boost/python/list.hpp>
#include <boost/python/class.hpp>
#include <boost/python/def.hpp>

// Возвращаем строку hello world.
std::string get_hello_function()
{
    return "Hello world!";
}

// класс hello, который может возвращать список строк count hello world
class hello_class
{
public:

    // Получение приветствия в конструкторе.
    hello_class(std::string message) : _message(message) {}

    // Возвращает счетчик количества сообщений в списке python.
    boost::python::list as_list(int count)
    {
        boost::python::list res;
        for (int i = 0; i < count; ++i) {
            res.append(_message);
        }
        return res;
    }

private:
    std::string _message;
};

// Определение python-модуля с присвоением ему имени "hello".
BOOST_PYTHON_MODULE(hello)
{
    // Здесь объявляется, какие функции и классы должны быть открыты в модуле.

    // Функция get_hello_function раскрывается в python как функция.
    boost::python::def("get_hello", get_hello_function);
```

```
// Класс hello_class раскрывается в python как класс.
boost::python::class_<hello_class>("Hello", boost::python::init<std::string>())
    .def("as_list", &hello_class::as_list)
};
```

Для компиляции этого модуля в модуль Python вам понадобятся заголовки Python и библиотека Boost. Данный пример был создан на Ubuntu 12.04 с использованием Python 3.4 и gcc. Boost поддерживается на многих платформах. В случае Ubuntu необходимые пакеты были установлены с помощью:

```
sudo apt-get install gcc libboost-dev libpython3.4-dev
```

Компиляция исходного текста в .so-файл, который впоследствии может быть импортирован как модуль, если он находится на пути к Python:

```
gcc -shared -o hello.so -fPIC -I/usr/include/python3.4 hello.cpp -lboost_python-py34 -lboost_system
-l:libpython3.4m.so
```

Код на языке Python в файле example.py:

```
import hello

print(hello.get_hello())

h = hello.Hello("World hello!")
print(h.as_list(3))
```

Исполнение python3 example.py выдаст следующий результат:

```
Hello world!
['World hello!', 'World hello!', 'World hello!']
```

190.3. Передача открытого файла в C-расширения

Рассмотрим передачу открытого объекта файла из языка Python в код расширения на языке C.

Преобразовать файл в целочисленный дескриптор файла можно с помощью функции PyObject_AsFileDescriptor:

```
PyObject *fobj;
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0){
    return NULL;
}
```

Для преобразования целочисленного дескриптора файла обратно в Python-объект используйте PyFile_FromFd.

```
int fd; /* Existing file descriptor */
PyObject *fobj = PyFile_FromFd(fd, "filename","r",-1,NULL,NULL,NULL,1);
```

Глава 191. Python Lex-Yacc

Python Lex-Yacc (PLY) – это Python-реализация популярных утилит Lex и Yacc.

191.1. Начало работы с Python Lex-Yacc

Чтобы установить PLY для Python2/3, выполните следующие действия:

1. Скачать исходный код можно отсюда: <http://www.dabeaz.com/ply/ply-3.10.tar.gz>.
2. Разархивируйте загруженный файл архива.
3. Перейдите в распакованную папку ply-3.10.
4. Выполните в терминале следующую команду: `python setup.py install`.

Если вы выполнили все вышеперечисленные действия, то теперь вы можете использовать модуль PLY. Вы можете протестировать его, открыв интерпретатор Python и набрав `import ply.lex`.

Примечание: не используйте `pip` для установки PLY, это приведет к установке нерабочего дистрибутива.

191.2. "Hello, World!" от PLY – простой калькулятор

Продemonстрируем возможности PLY на простом примере: эта программа принимает на вход строку с арифметическим выражением и пытается его решить.

Откройте свой любимый редактор и скопируйте следующий код:

```
from ply import lex
import ply.yacc as yacc

tokens = (
    'PLUS',
    'MINUS',
    'TIMES',
    'DIV',
    'LPAREN',
    'RPAREN',
    'NUMBER',
)

t_ignore = ' \t'

t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIV = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

def t_NUMBER(t):
    r'[0-9]+'
    t.value = int(t.value)
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_error(t):
    print("Invalid Token:", t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()

precedence = (
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIV'),
    ('nonassoc', 'UMINUS')
)

def p_add(p):
    'expr : expr PLUS expr'
    p[0] = p[1] + p[3]
```

```

def p_sub( p ) :
    'expr : expr MINUS expr'
    p[0] = p[1] - p[3]

def p_expr2uminus( p ) :
    'expr : MINUS expr %prec UMINUS'
    p[0] = - p[2]

def p_mult_div( p ) :
    """expr : expr TIMES expr
    | expr DIV expr"""

    if p[2] == '*' :
        p[0] = p[1] * p[3]
    else :
        if p[3] == 0 :
            print("Невозможно разделить на 0")
            raise ZeroDivisionError('целочисленное деление на 0')
        p[0] = p[1] / p[3]

def p_expr2NUM( p ) :
    'expr : NUMBER'
    p[0] = p[1]

def p_parens( p ) :
    'expr : LPAREN expr RPAREN'
    p[0] = p[2]

def p_error( p ) :
    print("Синтаксическая ошибка во входных данных!!")

parser = yacc.yacc()

res = parser.parse("-4*(3-5)") # входные данные
print(res)

```

Сохраните этот файл под именем calc.py и запустите его. Получим:

-8

Что является правильным ответом для выражения $-4 * (3 - 5)$.

191.3. Часть 1. Токенизация входных данных с помощью Lex

Код из первого примера выполнил два этапа: первый – это *токенизация* входных данных, то есть поиск символов, составляющих арифметическое выражение, и второй – *синтаксический разбор* (*парсинг*), который заключается в анализе извлеченных лексем и оценке результата.

В этом разделе приводится простой пример токенизации пользовательского ввода, а затем он разбирается построчно.

```

import ply.lex as lex

# Список имен токенов. Это всегда обязательно
tokens = [
    'NUMBER',
    'PLUS',
    'MINUS',
    'TIMES',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
]

```

```

# Правила регулярных выражений для простых лексем
t_PLUS = r\+'
t_MINUS = r\+'
t_TIMES = r\+'
t_DIVIDE = r\+'
t_LPAREN = r\+'
t_RPAREN = r\+'

# Правило регулярного выражения с некоторым кодом действия
def t_NUMBER(t):
    r\d+'
    t.value = int(t.value)
    return t

# Определите правило, чтобы мы могли отслеживать номера строк
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Строка, содержащая игнорируемые символы (пробелы и табуляции)
t_ignore = ' \t'

# Правило обработки ошибок
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

# Построим лексер
lexer = lex.lex()

# Дадим лексеру вводные данные
lexer.input(data)

# Токенизация
while True:
    tok = lexer.token()
    if not tok:
        break # Больше нет ввода
    print(tok)

```

Сохраните этот файл под именем `calclex.py`. Мы будем использовать его при создании нашего парсера Ясс.

Рассмотрим происшедшее поэтапно

1. Импорт модуля с помощью `import ply.lex`.
2. Все лексеры должны предоставлять список, называемый `tokens`, который определяет все возможные имена токенов, которые могут быть созданы лексером. Этот список всегда обязателен. `tokens` также может быть кортежем строк (а не строкой), где каждая строка, как и раньше, обозначает токен.
3. Правило `regex` для каждой строки может быть задано как в виде строки, так и в виде функции. В любом случае имя переменной должно быть дополнено символом `t_`, чтобы обозначить, что это правило для сопоставления лексем.

- Для простых лексем регулярное выражение может быть задано в виде строки:
`t_PLUS = r\+'`
- Если необходимо выполнить какое-либо действие, то токен-правило может быть задано в виде функции:

```

def t_NUMBER(t):
    r\d+'
    t.value = int(t.value)
    return t

```

Обратите внимание, что в функции правило задается в виде doc string. Функция принимает один аргумент, являющийся экземпляром LexToken, выполняет некоторое действие и возвращает аргумент.

Если вы хотите использовать внешнюю строку в качестве правила regex для функции вместо того, чтобы указывать doc string, рассмотрим следующий пример:

```
@TOKEN(identifier) # идентификатор - это строка, содержащая regex
def t_ID(t):
    ... # действия
```

- Экземпляр объекта LexToken (назовем этот объект t) имеет следующие атрибуты:

1. t.type – тип токена (в виде строки) (например, 'NUMBER', 'PLUS' и т. д.). По умолчанию t.type устанавливается в имя, следующее за t_ prefix.

2. t.value – лексема (собственно текст, с которым происходит совпадение).

3. t.lineno – номер текущей строки (он не обновляется автоматически, так как лексер ничего не знает о номерах строк). Обновление lineno производится с помощью функции t_newline.

```
def t_newline(t):
    r'\n+'
    t.lexpos += len(t.value)
```

4. t.lexpos, который представляет собой позицию лексемы относительно начала входного текста.

- Если из функции правила regex ничего не возвращается, то лексема отбрасывается. Если необходимо отбросить маркер, то можно добавить t_ignore_ prefix в переменную правила regex вместо того, чтобы создавать функцию для того же правила.

```
def t_COMMENT(t):
    r'\#.*'
    pass
    # Возвращаемое значение отсутствует. Токен отброшен
```

...То же, что:

```
t_ignore_COMMENT = r'\#.*'
```

Это, конечно, неверно, если при появлении комментария вы выполняете какое-либо действие. В этом случае следует использовать функцию для выделения правила regex. Если вы не определили маркер для некоторых символов, но все равно хотите их игнорировать, используйте t_ignore = "<символы для игнорирования>" (эти префиксы необходимы):

- При создании master regex, Lex добавит регексы, указанные в файле, следующим образом:

1. Токены, определяемые функциями, добавляются в том же порядке, в каком они встречаются в тексте.

2. Токены, определяемые строками, добавляются в порядке убывания длины строки, определяющей regex для этого токена.

Если вы сопоставляете == и = в одном и том же файле, воспользуйтесь этими правилами.

- Литералы – это токены, которые возвращаются в том виде, в котором они есть. И t.type, и t.value будут установлены в значение самого символа. Определите список литералов как таковых:

```
literals = [ '+', '-', '*', '/' ]
```

или

```
literals = "+-*/"
```

Можно написать функции, которые выполняют дополнительные действия при совпадении литералов. Однако для этого необходимо соответствующим образом задать тип токена. Например:

```
literals = [ '{', '}' ]

def t_lbrace(t):
    r'\{'
    t.type = '{' # Установить тип токена на ожидаемый литерал (обязательно, если
                  # это литерал)
    return t
```

- Обработка ошибок осуществляется с помощью функции `t_error`.
Правило обработки ошибок
def `t_error(t)`:
print("Illegal character '%s'" % t.value[0])
`t.lexer.skip(1)` # пропустить недопустимую лексему (не обрабатывать ее)

В общем случае `t.lexer.skip(n)` пропускает `n` символов во входной строке.

4. Заключительные приготовления:

Постройте лексер, используя `lexer = lex.lex()`.

Можно также поместить все в класс и вызывать экземпляр класса для определения лексера. Например:

```
import ply.lex as lex
class MyLexer(object):
    ... # все, что касается правил работы с токенами и обработки ошибок, как обычно,
    находится здесь

    # Построить лексер
    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    def test(self, data):
        self.lexer.input(data)
        for token in self.lexer.token():
            print(token)

    # Собрать лексер и опробовать его в работе

m = MyLexer()
m.build() # Построить лексер
m.test("3 + 4")
```

Предоставление входных данных с помощью функции `lexer.input(data)`, где `data` – строка.

Для получения лексем используется функция `lexer.token()`, которая возвращает найденные лексемы. Перебор лексем можно выполнять в цикле, как в примере:

```
for i in lexer:
    print(i)
```

191.4. Часть 2. Парсинг токенизированного ввода с помощью Yacc

В этом разделе объясняется, как обрабатывается токенизированный входной сигнал из части 1 – для этого используются контекстно-свободные грамматики (Context Free Grammars, CFG). Грамматика должна быть задана, и лексемы обрабатываются в соответствии с ней. В ходе обработки используется использует синтаксический анализатор LALR.

Пример на Yacc

```
import ply.yacc as yacc
```



```
# Получить карту токенов от лексера
from calclex import tokens

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

# Правило ошибок для синтаксических ошибок
def p_error(p):
    print("Синтаксическая ошибка во входных данных!")

# Построить синтаксический анализатор (парсер)
parser = yacc.yacc()

while True:
    try:
        s = raw_input('calc > ')
    except EOFError:
        break
    if not s: continue
    result = parser.parse(s)
    print(result)
```

Подробный разбор

1. Каждое правило грамматики задается функцией, в docstring которой содержится соответствующая спецификация контекстно-свободной грамматики. Операторы, составляющие тело функции, реализуют семантические действия правила. Каждая функция принимает один аргумент *p*, который представляет собой последовательность, содержащую значения каждого грамматического символа в соответствующем правиле. Значения *p[i]* сопоставлены с грамматическими символами, как показано здесь:

```
def p_expression_plus(p):
    'expression : expression PLUS term'
    # ^           ^           ^           ^
    # p[0]         p[1]         p[2]         p[3]

    p[0] = p[1] + p[3]
```

2. Для лексем (токенов) “значение” соответствующего $p[i]$ совпадает с атрибутом $p.value$, назначенным в модуле лексера. Таким образом, PLUS будет иметь значение +.

3. Для нетерминалов значение определяется тем, что помещается в $p[0]$. Если ничего не помещено, то значение равно None. Кроме того, $p[-1]$ – это не то же самое, что $p[3]$, поскольку p – это не простой список ($p[-1]$ может задавать встроенные действия (здесь не рассматривается)).

Заметим, что функция может иметь любое имя, если перед ним стоит p .

3. Правило $p_error(p)$ предназначено для перехвата синтаксических ошибок (аналогично $yerror$ в yacc/bison).

4. Несколько грамматических правил могут быть объединены в одну функцию, что является хорошей идеей, если предложения имеют схожую структуру.

```
def p_binary_operators(p):
    "expression : expression PLUS term
    | expression MINUS term
    term       : term TIMES factor
    | term DIVIDE factor"
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

5. Вместо лексем (токенов) могут использоваться символьные литералы.

```
def p_binary_operators(p):
    "expression : expression '+' term
    | expression '-' term
    term       : term '*' factor
    | term '/' factor"
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Разумеется, литералы должны быть заданы в модуле лексера.

6. Пустые постановки имеют вид “символ : ”. Чтобы явно задать символ начала, используйте $start = 'foo'$, где foo – некоторый нетерминал.

7. Задание старшинства и ассоциативности может быть выполнено с помощью переменной приоритета ($precedence$).

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'), # Неассоциативные операторы
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'), # Унарный оператор минус
```

Токены упорядочены от наименьшего к наибольшему приоритету. `nonassoc` означает, что эти токены не связаны между собой. Это означает, что что-то вроде $a < b < c$ является незаконным, в то время как $a < b$ по-прежнему законно.

8. `parser.out` – это отладочный файл, который создается при первом выполнении программы `Yacc`. При возникновении конфликта сдвига/редукции парсер всегда сдвигается.

Глава 192. Модульное тестирование

192.1. Использование методов `setUp` и `tearDown` при работе с `unittest.TestCase`

Иногда мы хотим подготовить контекст, в котором будет выполняться каждый тест. Метод `setUp` выполняется перед каждым тестом в классе. Метод `tearDown` выполняется в конце каждого теста. Эти методы являются необязательными. Следует помнить, что `TestCase` часто используются в совместном множественном наследовании, поэтому нужно быть внимательным и всегда вызывать `super` в этих методах, чтобы методы `setUp` и `tearDown` базового класса также вызывались. Базовая реализация `TestCase` предоставляет пустые методы `setUp` и `tearDown`, чтобы их можно было вызывать без возникновения исключений:

```
import unittest

class SomeTest(unittest.TestCase):
    def setUp(self):
        super(SomeTest, self).setUp()
        self.mock_data = [1,2,3,4,5]

    def test(self):
        self.assertEqual(len(self.mock_data), 5)

    def tearDown(self):
        super(SomeTest, self).tearDown()
        self.mock_data = []

if __name__ == '__main__':
    unittest.main()
```

Обратите внимание, что в Python2.7+ существует также метод `addCleanup`, который регистрирует функции, вызываемые после выполнения теста. В отличие от метода `tearDown` (который вызывается только в случае успешного завершения `setUp`), функции, зарегистрированные с помощью `addCleanup`, будут вызваны даже в случае необработанного исключения в `setUp`. В качестве конкретного примера можно привести удаление различных имитаторов (`mocks`), которые были зарегистрированы во время выполнения теста:

```
import unittest
import some_module

class SomeOtherTest(unittest.TestCase):
    def setUp(self):
        super(SomeOtherTest, self).setUp()

    # Заменить `some_module.method` на `mock.Mock`
    my_patch = mock.patch.object(some_module, 'method')
    my_patch.start()
```

```
# When the test finishes running, put the original method back.
self.addCleanup(my_patch.stop)
```

Еще одним преимуществом такой регистрации очисток является то, что она позволяет программисту поместить код очистки рядом с кодом установки и защитит вас в случае, если в подклассе не будет вызван `super` в `tearDown`.

192.2. Утверждения при исключениях

Проверить, что функция выбрасывает исключение, можно с помощью встроенного `unittest`, используя два разных метода.

Использование менеджера контекста

```
def division_function(dividend, divisor):
    return dividend / divisor

class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError):
            x = division_function(1, 0)
```

В результате код будет запущен внутри менеджера контекста, и если он выполнится успешно, то тест будет провален, поскольку исключение не было вызвано. Если же код вызовет исключение правильного типа, то тест будет продолжен.

Вы также можете получить содержимое полученного исключения, если хотите выполнить дополнительные утверждения относительно него.

```
class MyTestCase(unittest.TestCase):
    def test_using_context_manager(self):
        with self.assertRaises(ZeroDivisionError) as ex:
            x = division_function(1, 0)

        self.assertEqual(ex.message, 'integer division or modulo by zero')
```

Предоставление вызываемой функции

```
def division_function(dividend, divisor):
    """
    Деление двух чисел.

    :type делимого (dividend): int
    :type делителя (divisor): int

    :raises: ZeroDivisionError если делитель равен нулю (0).
    :rtype: int
    """
    return dividend / divisor

class MyTestCase(unittest.TestCase):
    def test_passing_function(self):
        self.assertRaises(ZeroDivisionError, division_function, 1, 0)
```

Проверяемое исключение должно быть первым параметром, а в качестве второго параметра должна быть передана вызываемая функция. Все остальные указанные параметры будут переданы непосредственно в вызываемую функцию, что позволяет указать параметры, вызывающие исключение.

192.3. Тестирование исключений

Программы выдают ошибки, например, при неправильном вводе данных. В связи с этим необходимо убедиться в том, что при фактическом неправильном вводе выдается ошибка. Для этого необходимо проверять исключения:

```
class WrongInputException(Exception):
    pass
```

Это исключение возникает при неправильном вводе, в следующем контексте, когда мы всегда ожидаем ввода числа в виде текста.

```
def convert2number(random_input):
    try:
        my_input = int(random_input)
    except ValueError:
        raise WrongInputException("Ожидается целое число!")
    return my_input
```

Чтобы проверить, произошло ли исключение, мы используем `assertRaises` для проверки этого исключения. `assertRaises` может использоваться двумя способами:

1. Использование обычного вызова функции. Первый аргумент принимает тип исключения, второй – вызываемый объект (обычно функцию), а остальные аргументы передаются этому вызываемому.

2. Использование предложения `with` позволяет передать функции только тип исключения. Преимущество этого способа состоит в том, что можно выполнить больше кода, однако его следует использовать с осторожностью, поскольку несколько функций могут использовать одно и то же исключение, что может вызвать проблемы. Пример: `with self.assertRaises(WrongInputException): convert2number("not a number")`.

Первый способ был реализован в следующем тестовом примере:

```
import unittest

class ExceptionTestCase(unittest.TestCase):

    def test_wrong_input_string(self):
        self.assertRaises(WrongInputException, convert2number, "не число")

    def test_correct_input(self):
        try:
            result = convert2number("56")
            self.assertIsInstance(result, int)
        except WrongInputException:
            self.fail()
```

Также может возникнуть необходимость проверить наличие исключения, которое не должно было быть выброшено. Однако при возникновении исключения тест автоматически завершится неудачно, и поэтому в этом может не быть необходимости. Чтобы показать возможные варианты, во втором методе тестирования показан пример того, как можно проверить, что исключение не должно быть выброшено. По сути, это перехват исключения и последующее завершение теста с помощью метода `fail`.

192.4. Выбор утверждений в рамках модуля Unittests

Хотя в Python есть оператор `assert`, фреймворк модульного тестирования Python имеет лучшие утверждения, специализированные для тестов: они более информативны при сбоях и не зависят от режима отладки в процессе выполнения. Пожалуй, самым простым утверждением является `assertTrue`, которое можно использовать следующим образом:

```
import unittest

class SimplisticTest(unittest.TestCase):
    def test_basic(self):
        self.assertTrue(1 + 1 == 2)
```

Это будет отлично работать, но заменить строку выше на строку:

```
self.assertTrue(1 + 1 == 3)
```

не удастся.

Утверждение `assertTrue`, скорее всего, является наиболее общим утверждением, поскольку все проверяемое можно привести к некоторому логическому булеву условию, но часто существуют лучшие альтернативы. При проверке на равенство, подобное описанному выше, лучше написать

```
self.assertEqual(1 + 1, 3)
```

Если первый вариант не работает, то появится сообщение

```
=====
FAIL: test (__main__.TruthTest)
```

```
Traceback (most recent call last):
```

```
File "stuff.py", line 6, in test
```

```
self.assertTrue(1 + 1 == 3)
```

```
AssertionError: False is not true
```

но когда последний не работает, сообщение выглядит следующим образом

```
=====
FAIL: test (__main__.TruthTest)
```

```
Traceback (most recent call last):
```

```
File "stuff.py", line 6, in test
```

```
self.assertEqual(1 + 1, 3)
```

```
AssertionError: 2 != 3
```

что является более информативным (действительно оценивается результат левой части).

Список утверждений можно найти в стандартной документации (<https://docs.python.org/2/library/unittest.html#assert-methods>). В общем случае следует выбирать утверждение, которое наиболее точно соответствует условию. Так, как показано выше, для утверждения $1 + 1 = 2$, лучше использовать `assertEqual`, а не `assertTrue`. Аналогично для утверждения, что `a is None`, лучше использовать `assertIsNone`, а не `assertEqual`. Отметим также, что утверждения имеют отрицательные формы. Так, `assertEqual` имеет свой отрицательный аналог `assertNotEqual`, а `assertIsNone` – свой отрицательный аналог `assertIsNotNone`. Опять же использование отрицательных аналогов, когда это уместно, приведет к более четким сообщениям об ошибках.

192.5. Модульные тесты с помощью pytest

Установка `pytest`:

```
pip install pytest
```

подготовка тестов:

```
mkdir tests
```

```
touch tests/test_docker.py
```

Функции для тестирования в файле `docker_something/helpers.py`:

```
from subprocess import Popen, PIPE
```

```
# над Popen проведен манкипатчинг при помощи `all_popens`
```

```
def copy_file_to_docker(src, dest):
```

```
try:
```

```
    result = Popen(['docker', 'cp', src, 'something_cont:{}'.format(dest)], stdout=PIPE, stderr=PIPE)
```

```
    err = result.stderr.read()
```

```
    if err:
```

```

        raise Exception(err)
    except Exception as e:
        print(e)
    return result

```

```

def docker_exec_something(something_file_string):
    fl = Popen(["docker", "exec", "-i", "something_cont", "something"], stdin=PIPE, stdout=PIPE, stderr=PIPE)
    fl.stdin.write(something_file_string)
    fl.stdin.close()
    err = fl.stderr.read()
    fl.stderr.close()
    if err:
        print(err)
        exit()
    result = fl.stdout.read()
    print(result)

```

Тест импортирует файл test_docker.py:

```

import os
from tempfile import NamedTemporaryFile
import pytest
from subprocess import Popen, PIPE

from docker_something import helpers
copy_file_to_docker = helpers.copy_file_to_docker
docker_exec_something = helpers.docker_exec_something

```

подражая файлоподобному объекту в test_docker.py:

```

class MockBytes():
    """Используется для сбора байтов"""
    all_read = []
    all_write = []
    all_close = []

    def read(self, *args, **kwargs):
        # print('read', args, kwargs, dir(self))
        self.all_read.append((self, args, kwargs))

    def write(self, *args, **kwargs):
        # print('wrote', args, kwargs)
        self.all_write.append((self, args, kwargs))

    def close(self, *args, **kwargs):
        # print('closed', self, args, kwargs)
        self.all_close.append((self, args, kwargs))

    def get_all_mock_bytes(self):
        return self.all_read, self.all_write, self.all_close

```

Манкипатчинг с pytest в test_docker.py:

```

@pytest.fixture
def all_popens(monkeypatch):
    """Это переопределяет / подражает встроенному Popen
    и заменяет stdin, stdout, stderr на объект MockBytes
    Примечание: monkeypatch импортируется волшебным образом
    """
    all_popens = []

    class MockPopen(object):
        def __init__(self, args, stdout=None, stdin=None, stderr=None):

```

```

    all_popen.append(self)
    self.args = args
    self.byte_collection = MockBytes()
    self.stdin = self.byte_collection
    self.stdout = self.byte_collection
    self.stderr = self.byte_collection
    pass
monkeypatch.setattr(helpers, 'Popen', MockPopen)

return all_popen

```

Примеры тестов в файле test_docker.py должны начинаться с префикса test_:

```

def test_docker_install():
    p = Popen(['which', 'docker'], stdout=PIPE, stderr=PIPE)
    result = p.stdout.read()
    assert 'bin/docker' in result

def test_copy_file_to_docker(all_popen):
    result = copy_file_to_docker('asdf', 'asdf')
    collected_popen = all_popen.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert mock_read
    assert result.args == ['docker', 'cp', 'asdf', 'something_cont:asdf']

def test_docker_exec_something(all_popen):
    docker_exec_something(something_file_string)

    collected_popen = all_popen.pop()
    mock_read, mock_write, mock_close = collected_popen.byte_collection.get_all_mock_bytes()
    assert len(mock_read) == 3
    something_template_stdin = mock_write[0][1][0]
    these = [os.environ['USER'], os.environ['password_prod'], 'table_name_here', 'test_vdm', 'col_a', 'col_b',
            '/tmp/test.tsv']
    assert all([x in something_template_stdin for x in these])

```

выполнение тестов по очереди:

```

py.test -k test_docker_install tests
py.test -k test_copy_file_to_docker tests
py.test -k test_docker_exec_something tests

```

запуск всех тестов в папке tests:

```
py.test -k test_ tests
```

192.6. Подражание функциям при помощи unittest.mock.create_autospec

Одним из способов подражания функции является использование функции create_autospec, которая подражает объекту в соответствии с его спецификациями. В случае с функциями это можно использовать для проверки правильности их вызова. Возьмем в качестве примера функцию multiply в файле custom_math.py:

```

def multiply(a, b):
    return a * b

```

И функцию multiples_of в файле process_math.py:

```
from custom_math import multiply
```

```
def multiples_of(integer, *args, num_multiples=0, **kwargs):
    —

```



```
:rtype: list
multiples = []

for x in range(1, num_multiples + 1):
    Передача здесь args и kwargs вызовет ошибку TypeError только в том случае, если
    значения были переданы в функцию multiples_of, в противном случае они игнорируются.
    Таким образом, мы можем проверить правильность использования multiples_of.
    Приведено для иллюстрации работы create_autospec. Не рекомендуется использовать
    в производственном коде.
    multiple = multiply(integer,x, *args, **kwargs)
    multiples.append(multiple)

return multiples
```

Мы можем протестировать `multiples_of` отдельно, выполнив имитацию `multiply`. В приведенном ниже примере используется стандартная библиотека Python `unittest`, но это может быть использовано и с другими фреймворками тестирования, такими как `pytest` или `nose`:

```
from unittest.mock import create_autospec
import unittest

# импортируем весь модуль, чтобы мы могли подражать multiply
import custom_math
custom_math.multiply = create_autospec(custom_math.multiply)
from process_math import multiples_of

class TestCustomMath(unittest.TestCase):
    def test_multiples_of(self):
        multiples = multiples_of(3, num_multiples=1)
        custom_math.multiply.assert_called_with(3, 1)

    def test_multiples_of_with_bad_inputs(self):
        with self.assertRaises(TypeError) as e:
            multiples_of(1, "extra arg", num_multiples=1) # это должно вызвать ошибку TypeError
```

Глава 193. Библиотека `py.test`

193.1. Настройка `py.test`

`Py.test` – это одна из сторонних библиотек тестирования, доступных для Python (см. их список на <http://docs.pytest.org/en/latest/>). Она может быть установлена с помощью `pip`:

```
pip install pytest
```

Код для тестирования

Допустим, мы тестируем функцию сложения в файле `projectroot/module/code.py`:

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

Код тестирования

Создаем тестовый файл: `projectroot/tests/test_code.py`. Для того чтобы файл был распознан как тестовый, он должен начинаться с `test_`.

```
# projectroot/tests/test_code.py
from module import code
```

```
def test_add():
    assert code.add(1, 2) == 3
```

Выполнение теста

Из projectroot просто запускаем py.test:

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
===== test session starts
=====
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items
tests/test_code.py .
===== 1 passed in 0.01 seconds
=====
```

193.2. Введение в тестовые фикстуры (Fixtures)

Более сложные тесты иногда требуют настройки перед запуском тестируемого кода. Это можно сделать в самой тестовой функции, но в этом случае получается, что большие тестовые функции делают так много, что трудно определить, где заканчивается настройка и начинается тест. Кроме того, можно получить много дублирующего кода настройки между различными тестовыми функциями.

Наш файл кода:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Наш тестовый файл:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff

def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000

def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Это довольно простые примеры, но если бы наш объект Stuff требовал гораздо больше настроек, то он стал бы громоздким. Мы видим, что между нашими тестовыми примерами есть дублирование кода, поэтому давайте сначала проведем его рефакторинг в отдельную функцию.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff
```

```
def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

```
def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000
```

```
def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Это выглядит лучше, но вызов `my_stuff = get_prepped_stuff()` по-прежнему загромождает наши тестовые функции.

Используем фикстуры `py.test`

Фикстуры представляют собой гораздо более мощные и гибкие версии функций тестовой установки. Они могут делать гораздо больше, чем мы используем здесь, но мы будем делать по одному шагу за раз. Сначала заменим `get_prepped_stuff` на фикстуру с именем `prepped_stuff`. Лучше называть свои фикстуры существительными, а не глаголами, поскольку впоследствии они будут использоваться в самих тестовых функциях. Оформление в виде `@pytest.fixture` указывает на то, что данная конкретная функция должна обрабатываться как фикстура, а не как обычная функция.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Теперь необходимо обновить тестовые функции так, чтобы они использовали эту фикстуру. Это делается путем добавления в их определение параметра, точно соответствующего имени фикстуры. При выполнении `py.test` перед запуском теста будет запущена фикстура, а затем через этот параметр в тестовую функцию будет передано возвращаемое значение `fixture`. (Обратите внимание, что фикстуры **не обязательно должны** возвращать значение; вместо этого они могут выполнять другие действия по настройке, например, вызывать внешний ресурс, организовывать элементы файловой системы, помещать значения в базу данных – все, что требуется тестам для настройки).

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000
```

```
def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Теперь понятно, почему мы называли фикстуру существительным. Но строка `my_stuff = prepped_stuff` практически бесполезна, поэтому вместо нее просто используем `prepped_stuff` напрямую.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000

def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Теперь мы используем фикстуры! Мы можем пойти дальше, изменив область видимости фикстуры (чтобы она запускалась только один раз на тестовый модуль или сеанс выполнения тестового набора, а не один раз на тестовую функцию), создавая фикстуры, которые используют другие фикстуры, параметризовать фикстуру (таким образом, фикстура и все тесты, использующие ее, запускаются несколько раз, по одному разу для каждого параметра, заданного фикстуре), фикстуры, которые считывают значения из вызывающего их модуля... Как уже говорилось, фикстуры обладают гораздо большей мощностью и гибкостью, чем обычная функция настройки.

Уборка после проведения исследований

Допустим, наш код разросся, и теперь наш объект `Stuff` нуждается в особой очистке.

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2

    def finish(self):
        self.foo = 0
        self.bar = 0
```

Мы могли бы добавить некоторый код для вызова очистки в нижней части каждой тестовой функции, но фикстуры предоставляет лучший способ сделать это. Если добавить функцию в фикстуру и зарегистрировать ее как **финализатор** (finalizer), то код в функции финализатора будет вызван после завершения теста, использующего фикстуру. Если область видимости фикстуры больше одной функции (например, модуль или сессия), то финализатор будет выполнен после завершения всех тестов в области видимости, то есть после завершения работы модуля или в конце всей сессии выполнения теста.

```
@pytest.fixture
def prepped_stuff(request): # нам нужно передать запрос для использования финализаторов
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin(): # функция-финализатор
        # выполняем всю очистку здесь
        my_stuff.finish()
    request.addfinalizer(fin) # зарегистрировать fin() в качестве финализатора
    # здесь можно сделать больше настроек
    return my_stuff
```

Использование функции-финализатора внутри функции может быть не совсем понятным на первый взгляд, особенно если у вас есть более сложные фикстуры. Вместо этого можно использовать выходные фикстуры (`yield fixtures`), чтобы сделать то же самое с более понятным для человека порядком выполнения. Единственное реальное отличие заключается в том, что вместо `return` мы используем `yield` в той части фикстуры, где выполнена настройка и управление должно перейти к тестовой функции, а затем добавляем весь очищающий код после `yield`. Мы также декорируем ее как `yield_fixture`, чтобы `py.test` знал, как с ней работать.

```
@pytest.yield_fixture
def prepped_stuff(): # теперь запрос не нужен!
    # выполнить настройку
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # настройка завершена, передайте управление тестовым функциям
    yield my_stuff
    # выполнить очистку
    my_stuff.finish()
```

Для получения дополнительной информации см. документацию фикстурам и выходным фикстурам на <http://doc.pytest.org/en/latest/fixture.html> и <http://doc.pytest.org/en/latest/fixture.html>.

193.3. Неудачные испытания

При неудачном тестировании можно получить полезную информацию о том, что именно было сделано не так:

```
# projectroot/tests/test_code.py
from module import code
```

```
def test_add__failing():
    assert code.add(10, 11) == 33
```

Результаты:

```
$ py.test
===== test session starts
=====
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items
tests/test_code.py F
===== FAILURES
=====
_____ test_add__failing _____

def test_add__failing():
> assert code.add(10, 11) == 33
E assert 21 == 33
E + where 21 = <function add at 0x105d4d6e0>(10, 11)
E + where <function add at 0x105d4d6e0> = code.add
tests/test_code.py:5: AssertionError
===== 1 failed in 0.01 seconds
=====
```

Глава 194. Профилирование

194.1. %%timeit и %timeit в IPython

Профилирование конкатенации строк:

```
In [1]: import string

In [2]: %%timeit s=""; long_list=list(string.ascii_letters)*50
.....: for substring in long_list:
.....:     s+=substring
.....:
1000 loops, best of 3: 570 us per loop
```

```
In [3]: %%timeit long_list=list(string.ascii_letters)*50
.....: s="".join(long_list)
```

```
100000 loops, best of 3: 16.1 us per loop
```

Профилирование циклов по итерируемым объектам и спискам:

```
In [4]: %%timeit for i in range(100000):pass
100 loops, best of 3: 2.82 ms per loop
```

```
In [5]: %%timeit for i in list(range(100000)):pass
100 loops, best of 3: 3.95 ms per loop
```

194.2. Использование cProfile (предпочтительный инструмент)

В Python входит модуль для профилирования cProfile. Его использование, как правило, предпочтительнее, чем использование timeit. Он разбивает весь скрипт на части и для каждого метода в нем сообщает:

- `ncalls`: Количество вызовов метода
- `tottime`: Общее время, затраченное на выполнение данной функции (без учета времени, затраченного на обращение к подфункциям)
- `percall`: Время, затраченное на один вызов. Или `tottime`, деленное на `ncalls`
- `sumtime`: Суммарное (кумулятивное) время, затраченное на выполнение данной функции и всех подфункций (от вызова до выхода). Расчет точен даже для рекурсивных функций.
- `percall`: представляет собой частное от деления `sumtime` на количество вызовов
- `filename:lineno(function)`: предоставляет соответствующие данные каждой функции

Модуль cProfiler может быть легко вызван в командной строке с помощью:

```
$ python -m cProfile main.py
```

Отсортировать возвращаемый список профилированных методов по времени, затраченному на выполнение метода:

```
$ python -m cProfile -s time main.py
```

194.3. Функция timeit()

Профилирование повторения элементов массива:

```
>>> import timeit
>>> timeit.timeit('list(itertools.repeat("a", 100))', 'import itertools', number = 10000000)
10.997665435877963
>>> timeit.timeit('["a"]*100', number = 10000000)
7.118789926862576
```

194.4. Командная строка timeit

Профилирование конкатенации чисел:

```
python -m timeit "''.join(str(n) for n in range(100))"
10000 loops, best of 3: 29.2 usec per loop
```

```
python -m timeit "''.join(map(str,range(100)))"
100000 loops, best of 3: 19.4 usec per loop
```

194.5. Использование модуля `line_profiler` и скрипта `kernprof` в командной строке

Исходный код с директивой `@profile` перед функцией, которую мы хотим профилировать:

```
import requests

@profile
def slow_func():
    s = requests.session()
    html=s.get("https://en.wikipedia.org/").text
    sum([pow(ord(x),3.1) for x in list(html)])

for i in range(50):
    slow_func()
```

Использование команды `kernprof` для построчного расчета профилирования:

```
$ kernprof -lv so6.py
Wrote profile results to so6.py.lprof
Timer unit: 4.27654e-07 s
Total time: 22.6427 s
File: so6.py
Function: slow_func at line 4
Line # Hits Time Per Hit % Time Line Contents
=====
4 @profile
5 def slow_func():
6 50 20729 414.6 0.0 s = requests.session()
7 50 47618627 952372.5 89.9 html=s.get("https://en.wikipedia.org/").text
8 50 5306958 106139.2 10.0 sum([pow(ord(x),3.1) for x in list(html)])
```

Глава 195. Скорость работы программы на языке Python

195.1. Deque-операции

Deque – это двусторонняя очередь.

```
class Deque:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def addFront(self, item):
        self.items.append(item)

    def addRear(self, item):
        self.items.insert(0,item)

    def removeFront(self):
        return self.items.pop()

    def removeRear(self):
        return self.items.pop(0)
```

```
def size(self):
    return len(self.items)
```

Операция: Средний случай (предполагается, что параметры генерируются случайным образом)

```
Append : O(1)
Appendleft : O(1)
Copy : O(n)
Extend : O(k)
Extendleft : O(k)
Pop : O(1)
Popleft : O(1)
Remove : O(n)
Rotate : O(k)
```

195.2. Алгоритмические нотации

Существуют определенные принципы, которые применимы к оптимизации в любом компьютерном языке, и Python не является исключением. **Не оптимизируйте на ходу:** пишите программу без учета возможных оптимизаций, концентрируясь на том, чтобы код был чистым, правильным и понятным. Если по завершении работы он окажется слишком большим или слишком медленным, то можно подумать о его оптимизации.

Помните правило 80/20: во многих областях можно получить 80 % результата, затратив 20 % усилий. Всякий раз, когда вы собираетесь оптимизировать код, используйте профилирование, чтобы выяснить, куда уходят эти 80 % времени выполнения, чтобы знать, на чем сосредоточить усилия.

Всегда проводите сравнительные тесты “до” и “после”: как еще можно убедиться, что оптимизация действительно дала результат? Если оптимизированный код окажется лишь немного быстрее или меньше исходной версии, отмените изменения и вернитесь к исходному, чистому коду.

Используйте правильные алгоритмы и структуры данных: Не используйте алгоритм пузырьковой сортировки $O(n^2)$ для сортировки тысячи элементов, когда можно использовать сортировку $O(n \log n)$. Аналогично не храните тысячу элементов в массиве, требующем $O(n)$ поиска, когда можно использовать $O(\log n)$ двоичное дерево или $O(1)$ хеш-таблицу Python.

Для представления временной сложности алгоритмов чаще всего используются следующие 3 асимптотических обозначения.

1. **Θ Нотация:** Тэта-нотация ограничивает функцию сверху и снизу, таким образом, она определяет точное асимптотическое поведение. Простой способ получить выражение в тэта-нотации состоит в том, чтобы отбросить члены низшего порядка и игнорировать ведущие константы. Например, рассмотрим следующее выражение. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$ Опускание членов низшего порядка всегда fine, так как всегда найдется n_0 , после которого $\Theta(n^3)$ имеет большее значение, чем $\Theta(n^2)$, независимо от участвующих констант. Для заданной функции $g(n)$ обозначим $\Theta(g(n))$ как следующее множество функций. $\Theta(g(n)) = \{f(n): \text{существуют положительные константы } c_1, c_2 \text{ и } n_0 \text{ такие, что } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ для всех } n \geq n_0\}$. Приведенное выше определение означает, что если $f(n)$ является тэтой $g(n)$, то значение $f(n)$ всегда находится между $c_1 g(n)$ и $c_2 g(n)$ для больших значений n ($n \geq n_0$). Из определения тэты также следует, что $f(n)$ должна быть неотрицательной при значениях n больше n_0 .

2. **Нотация Big O:** Нотация Big O обозначает верхнюю границу алгоритма, она ограничивает функцию только сверху. Например, рассмотрим случай сортировки вставками. В лучшем случае она занимает линейное время, а в худшем – квадратичное. Можно с уверенностью сказать, что временная сложность сортировки вставками равна $O(n^2)$. Заметим, что $O(n^2)$ также покрывает линейное время. Если использовать нотацию Θ для представления временной сложности сортировки вставками, то для наилучшего и наихудшего случаев придется использовать два утверждения:

1. Наихудшая временная сложность сортировки вставками составляет $\Theta(n^2)$.

2. В наилучшем случае временная сложность сортировки вставками составляет $\Theta(n)$.

Нотация Big O полезна, когда мы имеем только верхнюю границу на временную сложность алгоритма. Во многих случаях мы легко находим верхнюю границу, просто взглянув

на алгоритм. $O(g(n)) = \{f(n): \text{существуют положительные константы } c \text{ и } n_0 \text{ такие, что } 0 \leq f(n) \leq cg(n) \text{ для всех } n \geq n_0\}$

3. Нотация Ω : Так же как обозначение Big O дает асимптотическую верхнюю границу функции, обозначение Ω дает асимптотическую нижнюю границу. Нотация Ω может быть полезной, когда мы имеем нижнюю границу на временную сложность алгоритма. Как уже говорилось в предыдущей заметке, наилучшая производительность алгоритма, как правило, не является полезной, поэтому Omega-нотация является наименее используемой из всех трех нотаций. Для заданной функции $g(n)$ обозначим через $\Omega(g(n))$ множество функций. $\Omega(g(n)) = \{f(n): \text{существуют положительные константы } c \text{ и } n_0 \text{ такие, что } 0 \leq cg(n) \leq f(n) \text{ для всех } n \geq n_0\}$. Рассмотрим здесь тот же пример сортировки вставками. Временная сложность сортировки вставками может быть записана в виде $\Omega(n)$, но это не очень полезная информация о сортировке вставками, поскольку нас обычно интересует наихудший случай, а иногда и средний.

195.3. Нотация Big-O

Допустим, у вас есть функция:

```
def list_check(to_check, the_list):
    for item in the_list:
        if to_check == item:
            return True
    return False
```

Это простая функция для проверки наличия элемента в списке. Чтобы описать сложность этой функции, можно сказать $O(n)$. Это означает “Order of n ”, “порядок n ”, так как функция O известна как функция порядка (Order function).

$O(n)$ – в общем случае n – количество элементов в контейнере

$O(k)$ – в общем случае k – это значение параметра или количество элементов в параметре

195.4. Операции со списком

Операция: Средний случай (предполагается, что параметры генерируются случайным образом)

```
Append : O(1)
Copy : O(n)
Del slice : O(n)
Delete item : O(n)
Insert : O(n)
Get item : O(1)
Set item : O(1)
Iteration : O(n)
Get slice : O(k)
Set slice : O(n + k)
Extend : O(k)
Sort : O(n log n)
Multiply : O(nk)
x in s : O(n)
min(s), max(s) : O(n)
Get length : O(1)
```

195.5. Операции с множеством

Операция: Средний случай (параметры генерируются случайным образом): *Худший случай*

```
x in s : O(1)
Difference s - t : O(len(s))
Intersection s&t : O(min(len(s), len(t))) : O(len(s) * len(t))
Multiple intersection s1&s2&s3&...&sn : (n-1) * O(l) where l is max(len(s1), ..., len(sn))
s.difference_update(t) : O(len(t)) : O(len(t) * len(s))
s.symmetric_difference_update(t) : O(len(t))
Symetric difference s^t : O(len(s)) : O(len(s) * len(t))
Union s | t : O(len(s) + len(t))
```

Глава 196. Оптимизация производительности

196.1. Профилирование кода

Прежде всего вы должны уметь определить узкое место вашего скрипта, и учтите, что никакая оптимизация не сможет компенсировать неудачный выбор структуры данных или недостатки в проектировании алгоритма. Во-вторых, не пытайтесь оптимизировать слишком рано в процессе кодирования в ущерб читабельности, проектированию и качеству. Дональд Кнут сделал следующее заявление по поводу оптимизации:

“Мы должны забыть о мелких неприятностях, скажем, в 97% случаев: преждевременная оптимизация – корень всех зол. И все же мы не должны упускать свои возможности в этих критических 3%”.

Для профилирования кода есть несколько инструментов: cProfile (и более медленный profile) из стандартной библиотеки, line_profiler и timeit. Каждый из них служит разным целям.

cProfile является детерминированным средством профилирования: с его помощью отслеживаются события вызова функции, возврата функции и исключения, причем для интервалов между этими событиями (до 0,001 с) составляются точные тайминги. В документации к нему приведен простой пример использования:

```
import cProfile
def f(x):
    return "42!"
cProfile.run('f(12)')
```

Или, если вы предпочитаете оборачивать части существующего кода:

```
import cProfile, pstats, StringIO
pr = cProfile.Profile()
pr.enable()
# ... выполняйте здесь что-нибудь ...
pr.disable()
sortby = 'cumulative'
ps = pstats.Stats(pr, stream=s).sort_stats(sortby)
ps.print_stats()
print s.getvalue()
```

В результате будут получены результаты, похожие на таблицу, приведенную ниже, где можно быстро увидеть, на что программа тратит большую часть времени, и определить функции, которые необходимо оптимизировать.

```
3 function calls in 0.000 seconds
Ordered by: standard name
ncalls tottime pcall cumtime pcall filename:lineno(function)
1 0.000 0.000 0.000 0.000 :1(f)
1 0.000 0.000 0.000 0.000 :1()
1 0.000 0.000 0.000 0.000 {method 'disable' of '_Isprof.Profiler' objects}
```

Модуль line_profiler (https://github.com/rkern/line_profiler) полезен для построчного анализа вашего кода. Очевидно, что это не подходит для длинных скриптов, а предназначено для фрагментов. Более подробная информация приведена в документации. Самый простой способ начать работу – использовать скрипт kernprof, как описано на странице пакета; обратите внимание, что вам придется вручную указать функцию(и) для профилирования.

```
$ kernprof -l script_to_profile.py
```

kernprof создаст экземпляр LineProfiler и вставит его в пространство имен __builtins__ с именем профилера. Он был написан для использования в качестве декоратора, поэтому в своем

скрипте вы должны декорировать функции, которые вы хотите профилировать, символом `@profile`.

```
@profile
def slow_function(a, b, c):
```

По умолчанию `kernprof` помещает результаты в двоичный файл `script_to_profile.py.lprof`. С помощью опции `[-v/--view]` можно указать `kernprof` на немедленный просмотр отформатированных результатов на терминале. В противном случае можно просмотреть результаты позже следующим образом:

```
$ python -m line_profiler script_to_profile.py.lprof
```

Наконец, `timeit` предоставляет простой способ тестирования однострочников или небольших выражений как из командной строки, так и из оболочки Python. Этот модуль позволит ответить на такие вопросы, как, например, что быстрее – сделать генератор списка или использовать встроенную функцию `list()` при преобразовании набора в список. Для добавления кода настройки ищите ключевое слово `setup` или опцию `-s`.

```
>>> import timeit
>>> timeit.timeit("-".join(str(n) for n in range(100))', number=10000)
0.8187260627746582
```

с терминала:

```
$ python -m timeit "-".join(str(n) for n in range(100))'
10000 loops, best of 3: 40.3 usec per loop
```

Глава 197. Безопасность и криптография

Python, являясь одним из самых популярных языков в области компьютерной и сетевой безопасности, обладает большим потенциалом в области безопасности и криптографии. В данной теме рассматриваются криптографические возможности и реализации в Python, начиная с его использования в компьютерной и сетевой безопасности и заканчивая алгоритмами хеширования и шифрования/дешифрования.

197.1. Безопасное хеширование паролей

Для безопасного хеширования паролей можно использовать алгоритм PBKDF2, предоставляемый модулем `hashlib`. Хотя этот алгоритм не может предотвратить атаки методом перебора с целью восстановления исходного пароля по сохраненному хешу, он делает такие атаки очень ресурсоемкими.

```
import hashlib
import os
```

```
salt = os.urandom(16)
hash = hashlib.pbkdf2_hmac('sha256', b'password', salt, 100000)
```

PBKDF2 может работать с любым алгоритмом дайджеста, в приведенном примере используется SHA256, что обычно рекомендуется. Случайная “соль” (salt) должна храниться вместе с хешированным паролем, она понадобится для сравнения введенного пароля с сохраненным хешем. Важно, чтобы каждый пароль хешировался разной солью. Что касается количества раундов, то рекомендуется устанавливать его максимально возможным для вашего приложения. Если требуется получить результат в шестнадцатеричном виде, можно воспользоваться модулем `binascii`:

```
import binascii
hexhash = binascii.hexlify(hash)
```

Примечание: хотя PBKDF2 не плох, алгоритмы bcrypt и особенно scrypt считаются более сильными против атак методом перебора. В настоящее время ни один из них не входит в стандартную библиотеку Python.

197.2. Вычисление дайджеста сообщения

Модуль hashlib позволяет создавать генераторы дайджестов сообщений с помощью метода new. Эти генераторы превращают произвольную строку в дайджест фиксированной длины:

```
import hashlib
h = hashlib.new('sha256')
h.update(b'Nobody expects the Spanish Inquisition.')
h.digest()
# ==> b'\xda\xdaVR[\x12\x90\xff\x16\xfb\x17D\xcf\xb4\x82\xdd)\x14\xff\xbc\xb6ly\x0c\x0eX\
\x9eF--'
```

Обратите внимание, что перед вызовом digest можно вызывать update произвольное количество раз, что удобно при хешировании большого файла фрагментами. Также можно получить дайджест в шестнадцатеричном формате с помощью hexdigest:

```
h.hexdigest()
# ==> '2edfdada56525b1290ff16fb1744cfb482dd2914ffbc649790c0e589e462d3d'
```

197.3. Доступные алгоритмы хеширования

Функция hashlib.new требует указания имени алгоритма при его вызове для создания генератора. Чтобы узнать, какие алгоритмы доступны в текущем интерпретаторе Python, используйте hashlib.algorithms_available:

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA', 'SHA512', 'SHA224', 'dsaWithSHA', 'SHA', 'RIPEMD160', 'ecdsa-with-SHA1',
'sha1', 'SHA384', 'md5', 'SHA1', 'MD5', 'MD4', 'SHA256', 'sha384', 'md4', 'ripemd160', 'sha224',
'sha512', 'DSA', 'dsaEncryption', 'sha', 'whirlpool'}
```

Возвращаемый список зависит от платформы и интерпретатора; убедитесь, что ваш алгоритм доступен.

Существуют также некоторые алгоритмы, *гарантированно* доступные на всех платформах и интерпретаторах, которые доступны с помощью hashlib.algorithms_guaranteed:

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384', 'sha1', 'sha224', 'md5', 'sha512'}
```

197.4. Хеширование файлов

Хеш – это функция, преобразующая последовательность байтов переменной длины в последовательность фиксированной длины. Хеширование символов может быть полезным по многим причинам. С его помощью можно проверить, идентичны ли два файла, или убедиться, что содержимое файла не было повреждено или изменено. Для генерации хеша для файла можно использовать библиотеку hashlib:

```
import hashlib

hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    contents = f.read()
    hasher.update(contents)

print hasher.hexdigest()
```

При работе с более крупными элементами можно использовать буфер с фиксированной длиной:

```
import hashlib
SIZE = 65536
hasher = hashlib.new('sha256')
with open('myfile', 'r') as f:
    buffer = f.read(SIZE)
    while len(buffer) > 0:
        hasher.update(buffer)
        buffer = f.read(SIZE)
print(hasher.hexdigest())
```

197.5. Генерация RSA-подписей с помощью библиотеки Pycrypto

Криптографический алгоритм RSA может использоваться для создания подписи сообщения. Действительная подпись может быть создана только при наличии доступа к закрытому (private) ключу RSA, проверка же возможна только при наличии соответствующего открытого (public) ключа. Таким образом, если другая сторона знает ваш открытый ключ, она может проверить, что сообщение подписано вами и не изменилось – такой подход используется, например, для электронной почты. В настоящее время для этой функциональности требуется сторонний модуль, например Pycrypto.

```
import errno
```

```
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto.Signature import PKCS1_v1_5
```

```
message = b'Это сообщение от меня, я обещаю.'
```

```
try:
```

```
    with open('privkey.pem', 'r') as f:
        key = RSA.importKey(f.read())
```

```
except IOError as e:
```

```
    if e.errno != errno.ENOENT:
        raise
```

```
    # Нет закрытого ключа, сгенерируйте новый. Это может занять несколько секунд.
```

```
    key = RSA.generate(4096)
```

```
    with open('privkey.pem', 'wb') as f:
```

```
        f.write(key.exportKey('PEM'))
```

```
    with open('pubkey.pem', 'wb') as f:
```

```
        f.write(key.publickey().exportKey('PEM'))
```

```
hasher = SHA256.new(message)
```

```
signer = PKCS1_v1_5.new(key)
```

```
signature = signer.sign(hasher)
```

Проверка подписи работает аналогично, но при этом используется не закрытый, а открытый ключ:

```
with open('pubkey.pem', 'rb') as f:
```

```
    key = RSA.importKey(f.read())
```

```
hasher = SHA256.new(message)
```

```
verifier = PKCS1_v1_5.new(key)
```

```
if verifier.verify(hasher, signature):
```

```
    print('Отлично, подпись действительна!')
```

```
else:
```

```
    print('Нет, сообщение было подписано неверным закрытым ключом или изменено')
```

Примечание: в приведенных примерах используется очень распространенный алгоритм подписи PKCS#1 v1.5. Pycrypto также реализует более новый алгоритм PKCS#1 PSS, поэтому замена PKCS1_v1_5 на PKCS1_PSS в примерах должна работать, если вы хотите использовать именно его. Однако в настоящее время, по-видимому, нет особых причин для его использования (см. <http://crypto.stackexchange.com/questions/3850/is-rsassa-pkcs1-v1-5-a-good-signature-scheme-for-new-systems>).

197.6. Асимметричное RSA-шифрование с помощью Pycrypto

Преимущество асимметричного шифрования заключается в том, что сообщение может быть зашифровано без обмена секретным ключом с получателем сообщения. Отправителю достаточно знать открытый ключ получателя, что позволяет зашифровать сообщение таким образом, что расшифровать его сможет только назначенный получатель (имеющий соответствующий закрытый ключ). В настоящее время для реализации этой функциональности требуется сторонний модуль типа Pycrypto.

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA
```

```
message = b'Это очень секретное сообщение.'
```

```
with open('pubkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
encrypted = cipher.encrypt(message)
```

Получатель может расшифровать сообщение, если у него есть правильный закрытый ключ:

```
with open('privkey.pem', 'rb') as f:
    key = RSA.importKey(f.read())
cipher = PKCS1_OAEP.new(key)
decrypted = cipher.decrypt(encrypted)
```

Примечание: в приведенных примерах используется схема шифрования PKCS#1 OAEP. В pycrypto также реализована схема шифрования PKCS#1 v1.5, однако она не рекомендуется для новых протоколов из-за известных недостатков (см. <http://security.stackexchange.com/questions/32050/what-specific-padding-weakness-does-oaep-address-in-rsa>).

197.7. Симметричное шифрование с использованием Pycrypto

Встроенная в Python криптографическая функциональность в настоящее время ограничивается хешированием. Для шифрования требуется сторонний модуль, например Pycrypto. К примеру, он предоставляет алгоритм AES, который считается передовым для симметричного шифрования. Следующий код зашифрует заданное сообщение с помощью парольной фразы:

```
import hashlib
import math
import os
```

```
from Crypto.Cipher import AES
```

```
IV_SIZE = 16      # 128 бит, фиксированное значение для алгоритма AES
KEY_SIZE = 32     # 256 бит означает использование AES-256, также может быть 128 или 192 бита
SALT_SIZE = 16    # Этот размер является произвольным
```

```
cleartext = b'Lorem ipsum'
password = b'очень безопасный зашифрованный пароль'
salt = os.urandom(SALT_SIZE)
```

```

derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]

encrypted = salt + AES.new(key, AES.MODE_CFB, iv).encrypt(cleartext)

```

Алгоритм AES принимает три параметра: ключ шифрования, вектор инициализации (IV) и собственно шифруемое сообщение. Если у вас есть случайно сгенерированный ключ AES, то вы можете использовать его напрямую и просто сгенерировать случайный вектор инициализации. Однако парольная фраза не имеет подходящего размера, да и использовать ее напрямую не рекомендуется, поскольку она не является действительно случайной и, следовательно, обладает сравнительно небольшой энтропией. Вместо этого мы используем встроенную реализацию алгоритма PBKDF2 для генерации 128-битного вектора инициализации и 256-битного ключа шифрования из пароля.

Обратите внимание на случайную соль, которая важна для того, чтобы для каждого зашифрованного сообщения иметь разные вектор инициализации и ключ. Это, в частности, гарантирует, что два одинаковых сообщения не приведут к идентичному зашифрованному тексту, а также предотвращает повторное использование злоумышленниками работы по угадыванию одной ключевой фразы в сообщениях, зашифрованных другой ключевой фразой. Эта соль должна храниться вместе с зашифрованным сообщением, чтобы получить тот же вектор инициализации и ключ для расшифровки.

Следующий код снова расшифрует наше сообщение:

```

salt = encrypted[0:SALT_SIZE]
derived = hashlib.pbkdf2_hmac('sha256', password, salt, 100000,
                               dklen=IV_SIZE + KEY_SIZE)
iv = derived[0:IV_SIZE]
key = derived[IV_SIZE:]
cleartext = AES.new(key, AES.MODE_CFB, iv).decrypt(encrypted[SALT_SIZE:])

```

Глава 198. SSH-протокол в Python

Параметр	Использование
hostname	Этот параметр указывает хост, с которым необходимо установить соединение
username	имя пользователя, необходимое для доступа к хосту
port	порт хоста
password	пароль для учетной записи

198.1. SSH-соединение

```

from paramiko import client
ssh = client.SSHClient() # создание нового объекта SSHClient
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # автоматический прием неизвестных
ключей хоста
ssh.connect(hostname, username=username, port=port, password=password) # соединение с хостом
stdin, stdout, stderr = ssh.exec_command(command) # передача команды в ssh
print stdout.channel.recv_exit_status() #выдает статус 1 - задание не выполнено

```


Глава 199. Антипаттерны программирования в Python

199.1. Чрезмерное использование except

Исключения – это мощный инструмент, но важно не переусердствовать. Рассмотрим код:

```
try: res = get_result() res = res[0] log('got result: %r' % res) except: if not res: res = " print('got exception')
```

Данный пример демонстрирует сразу три признака антипаттерна (антишаблона):

1. В строке 5 except без типа исключения будут перехвачены все исключения, включая KeyboardInterrupt, которое позволяет в некоторых случаях предотвратить выход из программы.

2. Блок except не возвращает ошибку, а значит, мы не сможем определить, возникло ли исключение внутри get_result или потому, что res был пустым списком.

3. Хуже всего то, что если мы беспокоились о том, что результат окажется пустым, то мы привели к гораздо более неприятной ситуации. Если get_result не сработает, то res останется полностью незаполненным, а ссылка на res в блоке except вызовет NameError, полностью маскируя исходную ошибку.

Всегда думайте о типе исключения, которое вы пытаетесь обработать. Ознакомьтесь со страницей с информацией об исключениях (<https://docs.python.org/2/library/exceptions.html>) и получите представление о том, какие основные исключения существуют.

Вот исправленная версия приведенного выше примера:

```
import traceback try: res = get_result() except Exception: log_exception(traceback.format_exc()) raise try: res = res[0] except IndexError: res = " log('got result: %r' % res)
```

Здесь перехватываются более конкретные исключения, вызывая их вновь при необходимости. На несколько строк больше, но в целом более корректно.

199.2. Процессороемкие функции

Программа может легко потратить время, вызывая процессороемкую функцию несколько раз. Например, возьмем функцию, которая выглядит следующим образом: она возвращает целое число, он возвращает целое число, если входное значение может его создать, иначе None:

```
def intensive_f(value): # int -> Optional[int]
    # сложный и ресурсоемкий код
    if process_has_failed:
        return None
    return integer_output
```

И это может быть использовано следующим образом:

```
x = 5
if intensive_f(x) is not None:
    print(intensive_f(x) / 2)
else:
    print(x, "не удалось обработать")
print(x)
```

Хотя это и работает, но имеет проблему вызова intensive_f, что удваивает время выполнения кода. Лучшим решением было бы заранее получить возвращаемое значение функции.

```
x = 5
result = intensive_f(x)
if result is not None:
    print(result / 2)
else:
    print(x, "не удалось обработать")
```


Однако более понятным и, возможно, более “питоничным” способом является использование исключений, например:

```
x = 5
try:
    print(intensive_f(x) / 2)
except TypeError: # Исключение, возникающее при попытке использования None + 1
    print(x, “не удалось обработать”)
```

В данном случае временная переменная не нужна. Часто вместо этого бывает предпочтительнее использовать утверждение `assert` и перехватить `AssertionError`.

Ключи словаря

Частым примером того, как это можно сделать, является обращение к ключам словаря. Например, сравните код:

```
bird_speeds = get_very_long_dictionary()

if “европейская ласточка” in bird_speeds:
    speed = bird_speeds[“европейская ласточка”]
else:
    speed = input(“Какова скорость полета ласточки без груза?”)

print(speed)

с кодом:

bird_speeds = get_very_long_dictionary()

try:
    speed = bird_speeds[“европейская ласточка”]
except KeyError:
    speed = input(“Какова скорость полета ласточки без груза?”)

print(speed)
```

В первом примере приходится дважды просматривать словарь, а так как словарь длинный, то каждый раз это может занять много времени. Второй пример требует только одного поиска по словарю, что значительно экономит процессорное время.

Альтернативой этому может быть использование `dict.get(key, default)`, однако во многих случаях может потребоваться выполнение более сложных операций в случае отсутствия ключа.

Глава 200. Общие ошибки

Python – это язык программирования, который задумывался понятным и читаемым, без двусмысленностей и неожиданного поведения. К сожалению, эти цели достижимы не во всех случаях, и поэтому есть несколько ситуаций, когда он может сделать не то, что ожидалось.

В этой главе будут рассмотрены некоторые проблемы, с которыми можно столкнуться при написании кода на языке Python.

200.1. Умножение списков и общие ссылки

Рассмотрим случай создания структуры вложенного списка путем умножения:

```
li = [[]] * 3
print(li)
# Результат: [[], [], []]
```

На первый взгляд кажется, что мы имеем список, содержащий три разных вложенных списка. Попробуем добавить 1 к первому списку:

```
li[0].append(1)
print(li)
# Результат: [[1], [1], [1]]
```

1 добавлена ко всем спискам в списке li.

Причина в том, что выражение `[] * 3` не создает список из трех разных списков. Скорее, создается *список, содержащий три ссылки на один и тот же объект списка*. Поэтому при добавлении к `li[0]` изменятся все вложенные элементы списка li. Это эквивалентно записи:

```
li = []
element = []
li = element + element + element
print(li)
# Результат: [], [], []
element.append(1)
print(li)
# Результат: [[1], [1], [1]]
```

Это можно подтвердить, если вывести адреса памяти вложенного списка с помощью функции `id`:

```
li = [] * 3
print([id(inner_list) for inner_list in li])
# Результат: [6830760, 6830760, 6830760]
```

Решением является создание внутренних списков с помощью цикла:

```
li = [] for _ in range(3)
```

Вместо того чтобы создавать один список и затем делать на него три ссылки, мы теперь создаем три разных списка. Это, опять же, можно проверить с помощью функции `id`:

```
print([id(inner_list) for inner_list in li])
# Результат: [6331048, 6331528, 6331488]
```

Также можно поступить нижеприведенным образом. При этом в каждом вызове `append` будет создаваться новый пустой список:

```
>>> li = []
>>> li.append([])
>>> li.append([])
>>> li.append([])
>>> for k in li: print(id(k))
```

```
4315469256
4315564552
4315564808
```

Не используйте индекс для перебора последовательности

Не делайте так:

```
for i in range(len(tab)):
    print(tab[i])
```

А делайте так:

```
for elem in tab:
    print(elem)
```

for позволит автоматизировать большинство итерационных операций.

Используйте `enumerate`, если вам действительно нужны и индекс, и элемент

```
for i, elem in enumerate(tab):
    print((i, elem))
```

Будьте осторожны при использовании “==” для проверки на True или False

```
if (var == True):
    # это будет выполнено, если var равно True или 1, 1.0, 1L

if (var != True):
    # это будет выполнено, если var не является ни True, ни 1

if (var == False):
    # это будет выполнено, если var равно False или 0 (или 0.0, 0L, 0j)

if (var == None):
    # выполняется только в том случае, если var равен None

if var:
    # выполняется, если var - непустая строка/список/словарь/кортеж, не-0 и т. д.

if not var:
    # выполнить, если var имеет значения "", {}, [], (), 0, None и т. д.

if var is True:
    # выполняется только в том случае, если var - булево True, а не 1

if var is False:
    # выполняется только в том случае, если var - булево False, а не 0

if var is None:
    # то же самое, что var == None
```

Не проверяйте, можете ли вы это сделать, просто сделайте и обработайте ошибку

Python-программисты обычно говорят: “Легче попросить прощения, чем разрешения”.

Не делайте так:

```
if os.path.isfile(file_path):
    file = open(file_path)
else:
    # сделать что-нибудь
```

А делайте так:

```
try:
    file = open(file_path)
except OSError as e:
    # сделать что-нибудь
```

А еще лучше делать так (*версии Python 2.6+*):

```
with open(file_path) as file:
```

Этот способ гораздо лучше, поскольку является более универсальным. Можно применять try/except практически ко всему. Вам не нужно заботиться о том, что нужно сделать для предотвращения ошибки, достаточно просто обратить внимание на ошибку, которую вы рискуете допустить.

Не проверяйте соответствие типу

Python динамически типизирован, поэтому проверка типа приводит к потере гибкости. Вместо этого используйте “утиную типизацию” (duck typing), проверяя поведение. Если вы ожидаете в функции строку, то используйте str() для преобразования любого объекта в строку. Если вы ожидаете список, то используйте list() для преобразования любого итерируемого объекта в список.

Не делайте так:

```
def foo(name):
    if isinstance(name, str):
        print(name.lower())

def bar(listing):
    if isinstance(listing, list):
        listing.extend((1, 2, 3))
    return ", ".join(listing)
```

А делайте так:

```
def foo(name):
    print(str(name).lower())

def bar(listing):
    l = list(listing)
    l.extend((1, 2, 3))
    return ", ".join(l)
```

Если использовать последний способ, то `foo` будет принимать любой объект, а `bar` – строки, кортежи, множества, списки и многое другое. Простой способ использования принципа DRY (Don't repeat yourself, с англ. — “не повторяйся”).

Не смешивайте пробелы и табуляции

Используйте объект в качестве первого родителя

Это непросто, но по мере роста вашей программы это поможет вам. В Python 2.x есть старые и новые классы. Старые классы лишены некоторых возможностей и могут иметь неудобное поведение при наследовании. Чтобы быть пригодным для использования, любой из ваших классов должен быть выполнен в “новом стиле”. Для этого необходимо сделать так, чтобы класс наследовался от `object`.

Не делайте так:

```
class Father:
    pass

class Child(Father):
    pass
```

А делайте так:

```
class Father(object):
    pass

class Child(Father):
    pass
```

В Python 3.x все классы нового стиля, поэтому этого делать не нужно.

Не инициализируйте атрибуты класса вне метода `init`

Программистам, пришедшим из других языков, это кажется заманчивым, потому что именно так вы поступаете в Java или PHP. Вы пишете имя класса, затем перечисляете свои атрибуты и задаете им значение по умолчанию. Кажется, что это работает и в Python, однако работает это не так, как вы думаете. При этом будут заданы атрибуты класса (статические атрибуты), затем, когда вы попытаетесь получить атрибут объекта, он выдаст вам его значение, если только оно не пустое. В этом случае он вернет атрибуты класса. Это влечет за собой две большие опасности:

- Если атрибут класса изменяется, то изменяется и начальное значение.
- Если задать изменяемый объект в качестве значения по умолчанию, то получится один и тот же объект, разделяемый между экземплярами.

Не делайте так:

```
class Car(object):
    color = "red"
    wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

А делайте так:

```
class Car(object):
    def __init__(self):
        self.color = "red"
        self.wheels = [Wheel(), Wheel(), Wheel(), Wheel()]
```

200.2. Изменяемый аргумент по умолчанию

```
def foo(li=[]):
    li.append(1)
    print(li)
```

```
foo([2])
# Результат: [2, 1]
foo([3])
# Результат: [3, 1]
```

Этот код ведет себя так, как и ожидалось, но что если не передавать аргумент?

```
foo()
# Результат: [1] Как и ожидалось...
foo()
# Результат: [1, 1] Не так, как ожидалось...
```

Это связано с тем, что стандартные аргументы функций и методов оцениваются не во время выполнения, а во время определения. Поэтому у нас всегда есть только один экземпляр списка li. Способ обойти это – использовать для аргументов по умолчанию только неизменяемые типы:

```
def foo(li=None):
    if not li:
        li = []
    li.append(1)
    print(li)
```

```
foo()
# Результат: [1]
```

```
foo()
# Результат: [1]
```

Хотя это и улучшение, и if not li, и многие другие объекты, например, последовательности нулевой длины, корректно оцениваются в False, но приведенные ниже примеры аргументов могут привести к непредвиденным результатам:

```
x = []
foo(li=x)
# Результат: [1]
```

```
foo(li="")
# Результат: [1]
```

```
foo(li=0)
# Результат: [1]
```

Идиоматический подход заключается в непосредственной проверке аргумента на соответствие объекту None:

```
def foo(li=None):
    if li is None:
        li = []
    li.append(1)
    print(li)
    foo()
# Out: [1]
```

200.3. Изменение последовательности, над которой выполняется итерация

Цикл for выполняет итерацию по последовательности, поэтому **изменение последовательности внутри цикла может привести к неожиданным результатам** (особенно при добавлении или удалении элементов):

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
    alist.pop(index)
print(alist)
# Результат: [1]
```

Примечание: для удаления элементов из списка используется метод list.pop().

Второй элемент не был удален, так как итерация проходит по индексам по порядку. Приведенный выше цикл выполняет итерацию дважды, получая следующие результаты:

```
# 1-я итерация
index = 0
alist = [0, 1, 2]
alist.pop(0) # удаляет '0'

# 2-я итерация
index = 1
alist = [1, 2]
alist.pop(1) # удаляет '2'

# цикл завершается, но alist не пуст:
alist = [1]
```

Эта проблема возникает из-за того, что при итерации в направлении увеличения индекса происходит изменение индексов. Чтобы избежать этой проблемы, можно выполнить **итерацию в цикле в обратном направлении**:

```
alist = [1,2,3,4,5,6,7]
for index, item in reversed(list(enumerate(alist))):
    # удалить все четные элементы
    if item % 2 == 0:
        alist.pop(index)
print(alist)
# Результат: [1, 3, 5, 7]
```

Итерация в цикле, начиная с конца, по мере удаления (или добавления) элементов не изменяет индексы элементов, находящихся ранее в списке. Поэтому в данном примере будут корректно удалены все четные элементы из alist.

Аналогичная проблема возникает при **вставке или добавлении элементов в список, над которым выполняется итерация**, что может привести к возникновению бесконечного цикла:

```
alist = [0, 1, 2]
for index, value in enumerate(alist):
```

```
# break, чтобы избежать бесконечного цикла:
if index == 20:
    break
alist.insert(index, 'a')
print(alist)
# Результат (сокращенный): ['a', 'a', ..., 'a', 'a', 0, 1, 2]
```

Без условия останова (break) цикл будет вставлять 'a' до тех пор, пока на компьютере не закончится память и программа не сможет продолжить работу. В подобной ситуации обычно предпочтительнее создать новый список и добавлять элементы в него по мере выполнения цикла по исходному списку.

При использовании цикла for **нельзя изменять элементы списка с помощью переменной-заместителя**:

```
alist = [1,2,3,4]
for item in alist:
    if item % 2 == 0:
        item = 'even'
print(alist)
# Результат: [1,2,3,4]
```

В приведенном выше примере **изменение item фактически ничего не меняет в исходном списке**. Необходимо использовать индекс списка (alist[2]), и для этого хорошо подходит функция enumerate():

```
alist = [1,2,3,4]
for index, item in enumerate(alist):
    if item % 2 == 0:
        alist[index] = 'even'
print(alist)
# Результат: [1, 'even', 3, 'even']
```

В некоторых случаях лучше использовать цикл while:
Если вы собираетесь **удалить все элементы** в списке:

```
zlist = [0, 1, 2]
while zlist:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)
```

```
# Результат:    0
#              1
#              2
# После: zlist = []
```

Хотя простой сброс zlist приведет к тому же результату:

```
zlist = []
```

Приведенный выше пример можно также комбинировать с функцией len() для остановки после определенного момента или для удаления всех элементов списка, кроме x:

```
zlist = [0, 1, 2]
x = 1
while len(zlist) > x:
    print(zlist[0])
    zlist.pop(0)
print('After: zlist =', zlist)
# Результат:    0
#              1
# После: zlist = [2]
```

Или для **циклического просмотра списка с удалением элементов, удовлетворяющих определенному условию** (в данном случае удаление всех четных элементов):

```
zlist = [1,2,3,4,5]
i = 0
while i < len(zlist):
    if zlist[i] % 2 == 0:
        zlist.pop(i)
    else:
        i += 1
print(zlist)
# Результат: [1, 3, 5]
```

Обратите внимание, что после удаления элемента инкремент *i* не увеличивается. При удалении элемента по адресу *zlist[i]* индекс следующего элемента уменьшился на единицу, поэтому, проверяя *zlist[i]* с тем же значением *i* на следующей итерации, вы будете правильно проверять следующий элемент списка.

Противоположным способом удаления ненужных элементов из списка является **добавление требуемых элементов в новый список**. Следующий пример является альтернативой последнему примеру цикла *while*:

```
zlist = [1,2,3,4,5]

z_temp = []
for item in zlist:
    if item % 2 != 0:
        z_temp.append(item)
zlist = z_temp
print(zlist)
# Результат: [1, 3, 5]
```

Здесь мы перегоняем желаемые результаты в новый список. При желании мы можем переназначить временный список исходной переменной.

При таком подходе можно задействовать одну из самых элегантных и мощных возможностей Python – генераторы списков, которые позволяют отказаться от временных списков и расходиться с рассмотренной ранее идеологией изменения списков/индексов на месте.

```
zlist = [1,2,3,4,5]
[item for item in zlist if item % 2 != 0]
# Результат: [1, 3, 5]
```

200.4. Целочисленные и строковые тождества

Python использует внутреннее кэширование для диапазона целых чисел, чтобы уменьшить ненужные расходы на их повторное создание. Это может привести к запутанному поведению при сравнении целочисленных тождеств:

```
>>> -8 is (-7 - 1)
False
>>> -3 is (-2 - 1)
True
```

и другой пример:

```
>>> (255 + 1) is (255 + 1)
True
>>> (256 + 1) is (256 + 1)
False
```

Мы видим, что операция тождества выдает *True* для одних целых чисел (-3, 256), но не для других (-8, 257).

Если быть более точным, то целые числа в диапазоне `[-5, 256]` кэшируются при запуске интерпретатора и создаются только один раз. Поэтому они **идентичны** и сравнение их идентичности при помощи `is` дает `True`; целые числа за пределами этого диапазона обычно создаются “на лету”, и проверка их тождественности выдает результат `False`.

Это распространенный “подводный камень”, поскольку это обычный диапазон для тестирования, но достаточно часто код, прекрасно работавший в процессе разработки, без видимых причин отказывает на дальнейших стадиях. Решение состоит в том, чтобы **всегда сравнивать значения с помощью оператора равенства** (`==`), а не оператора тождества (`is`).

В Python также хранятся ссылки на часто используемые строки, что может привести к такому же запутанному поведению при сравнении идентичности (т.е. использовании оператора `is`) строк.

```
>>> 'python' is 'py' + 'thon'
True
```

Строка `'python'` является общеупотребительной, поэтому в Python есть один объект, который используется всеми ссылками на строку `'python'`. Для нестандартных строк сравнение идентичности не удастся, даже если строки те же.

```
>>> 'this is not a common string' is 'this is not' + ' a common string'
False
>>> 'this is not a common string' == 'this is not' + ' a common string'
True
```

Поэтому, как и в случае с целыми числами, **всегда сравнивайте строковые значения с помощью оператора равенства** (`==`), а не оператора тождества (`is`).

200.5. Словари неупорядочены

Можно было бы ожидать, что словарь Python будет сортироваться по ключам, как, например, `std::map` в языке C++, но это не так:

```
myDict = {'first': 1, 'second': 2, 'third': 3}
print(myDict)
# Результат: {'first': 1, 'second': 2, 'third': 3}
```

```
print([k for k in myDict])
# Результат: ['second', 'third', 'first']
```

В Python нет встроенного класса, который бы автоматически сортировал свои элементы по ключу. Однако если сортировка не является обязательным условием, и вы просто хотите, чтобы ваш словарь запоминал порядок вставки пар ключ/значение, вы можете использовать `collections.OrderedDict`:

```
from collections import OrderedDict

oDict = OrderedDict([('first', 1), ('second', 2), ('third', 3)])

print([k for k in oDict])
# Результат: ['first', 'second', 'third']
```

Следует помнить, что инициализация `OrderedDict` стандартным словарем никак не отсортирует словарь, а только *сохранит порядок вставки* ключей.

В Python 3.6 реализация словарей была изменена с целью улучшения потребления ими памяти. Побочным эффектом новой реализации является сохранение порядка аргументов ключевых слов, передаваемых в функцию:

Версия Python 3.x ≥ 3.6:

```
def func(**kw): print(kw.keys())
func(a=1, b=2, c=3, d=4, e=5)
dict_keys(['a', 'b', 'c', 'd', 'e']) # ожидаемый порядок
```

Предупреждение: следует иметь в виду, что “аспект сохранения порядка в этой новой реализации рассматривается как деталь реализации и на него не следует полагаться”, поскольку в будущем он может измениться (см. <https://docs.python.org/3.6/whatsnew/3.6.html#new-dict-implementation>).

200.6. Утечка переменных в генераторах списков и циклах for

Рассмотрим следующий генератор списков:

Версия Python 2.x ≤ 2.7:

```
i = 0
a = [i for i in range(3)]
print(i) # Выводит 2
```

Это происходит только в Python 2 из-за того, что в генераторе списка происходит “утечка” управляющей переменной цикла в окружающую область видимости (источник). Такое поведение может приводить к трудноотлавливаемым ошибкам, и в Python 3 оно было исправлено.

Версия Python 3.x ≥ 3.0:

```
i = 0
a = [i for i in range(3)]
print(i) # Выводит 0
```

Аналогично, циклы for не имеют частной области видимости для своей итерационной переменной.

```
i = 0
for i in range(3):
    pass
print(i) # Выводит 2
```

Подобное поведение встречается как в Python 2, так и в Python 3.

Чтобы избежать проблем с утечкой переменных, используйте новые переменные в генераторах списков и циклах for по мере необходимости.

200.7. Цепочка операторов “or”

При проверке любого из нескольких сравнений равенства:

```
if a == 3 or b == 3 or c == 3:
    возникает соблазн сократить его до
if a or b or c == 3: # Неправильно
```

Это неверно: оператор or имеет меньший приоритет, чем ==, поэтому выражение будет оценено как if (a) or (b) or (c == 3):. Правильным способом является явная проверка всех условий:

```
if a == 3 or b == 3 or c == 3: # Правильно
```

В качестве альтернативы вместо цепочки операторов or может быть использована встроенная функция any():

```
if any([a == 3, b == 3, c == 3]): # Правильно
```

Или, чтобы сделать его более эффективным:

```
if any(x == 3 for x in (a, b, c)): # Правильно
```

Или, для краткости:

```
if 3 in (a, b, c): # Правильно
```

Здесь мы используем оператор `in`, чтобы проверить, присутствует ли значение в кортеже, содержащем значения, с которыми мы хотим сравнить. Аналогично, неверна запись:

```
if a == 1 or 2 or 3:
```

что следует записать в виде:

```
if a in (1, 2, 3):
```

200.8. `sys.argv[0]` – имя исполняемого файла

Первым элементом `sys.argv[0]` является имя исполняемого Python-файла. Остальные элементы являются аргументами скрипта.

```
# script.py
import sys
```

```
print(sys.argv[0])
print(sys.argv)
```

```
$ python script.py
=> script.py
=> ['script.py']
```

```
$ python script.py fizz
=> script.py
=> ['script.py', 'fizz']
```

```
$ python script.py fizz buzz
=> script.py
=> ['script.py', 'fizz', 'buzz']
```

200.9. Доступ к атрибутам целочисленных литералов

Вы, наверное, слышали, что в Python все является объектом, даже литералы. Это означает, что, например, `7` тоже является объектом, а значит, у него есть атрибуты. Например, одним из таких атрибутов является `bit_length`. Он возвращает количество битов, необходимых для представления вызываемого значения.

```
x = 7
x.bit_length()
# Результат: 3
```

Видя, что приведенный выше код работает, можно интуитивно подумать, что и `7.bit_length()` тоже будет работать, но вместо этого возникает `SyntaxError`. Почему? Потому что интерпретатору необходимо провести различие между обращением к атрибуту и числом с плавающей точкой (например, `7.2` или `7.bit_length()`). Этого сделать ему не удастся, потому и возникает исключение. Существует несколько способов доступа к атрибутам целочисленного литерала:

```
# круглые скобки
(7).bit_length()
# пробел
7 .bit_length()
```

Использование двух точек (например, `7..bit_length()`) в данном случае не работает, так как создается `float`-литерал, а у чисел с плавающей точкой нет метода `bit_length()`. При обращении к атрибутам `float`-литералов этой проблемы не существует, поскольку интерпретатор достаточно “умен”, чтобы понять, что `float`-литерал не может содержать, например, две точки:

```
7.2.as_integer_ratio()
# Результат: (8106479329266893, 1125899906842624)
```

200.10. Глобальная блокировка интерпретатора (GIL) и блокировка потоков

О глобальной блокировке интерпретатора (GIL) в Python написано достаточно много. Иногда это может привести к путанице при работе с многопоточными (не путать с многопроцессорными) приложениями. Приведем пример:

```
import math
from threading import Thread

def calc_fact(num):
    math.factorial(num)

num = 600000
t = Thread(target=calc_fact, daemon=True, args=[num])
print("Приступаем к вычислению: {}".format(num))
t.start()
print("Вычисление...")
t.join()
print("Вычислено")
```

Можно было бы ожидать, что "Вычисление..." будет выведено сразу после начала потока, ведь мы хотели, чтобы вычисление происходило в новом потоке! Но на самом деле это сообщение выводится после завершения вычислений. Это связано с тем, что новый поток опирается на функцию языка C (`math.factorial`), которая блокирует GIL на время выполнения.

Существует несколько способов обойти эту проблему. Первый – реализовать функцию факториала на родном языке Python. Это позволит главному потоку перехватить управление, пока вы находитесь внутри цикла. Недостатком такого решения является то, что оно будет намного медленнее, поскольку мы больше не используем функцию на языке C.

```
def calc_fact(num):
    """ медленная версия факториала на "родном" Python """
    res = 1
    while num >= 1:
        res = res * num
        num -= 1
    return res
```

Перед началом выполнения можно также использовать функцию `sleep` в течение некоторого времени. Заметьте, что это не позволит вашей программе на самом деле прервать вычисления, происходящие внутри функции C, но позволит вашему основному потоку продолжить работу.

```
def calc_fact(num):
    sleep(0.001)
    math.factorial(num)
```

200.11. Многократное использование оператора return

Допустим, что функция `xyz` возвращает два значения – `a` и `b`:

```
def xyz():
    return a, b
```

Код, вызывающий `xyz`, сохраняет результат в одной переменной, предполагая, что функция возвращает только одно значение:

```
t = xyz()
```

Значение `t` на самом деле является кортежем (`a, b`), поэтому любое действие над `t`, предполагающее, что оно не является кортежем, может завершиться неудачей **в глубине** кода с неожиданной ошибкой о кортежах.

`TypeError: type tuple doesn't define ... method` (тип `tuple` не определяет ... метод)

В этом случае достаточно написать:

```
a, b = xyz()
```

Новичкам будет трудно понять причину этого сообщения, прочитав только сообщение об ошибке кортежа!

200.12. JSON-ключи в Python

```
my_var = 'bla';  
api_key = 'key';  
...тут много кода...  
params = {"language": "en", my_var: api_key}
```

Если вы привыкли к JavaScript, то оценка переменных в словарях Python будет не такой, как вы ожидаете. Это утверждение в JavaScript привело бы к тому, что объект `params` выглядел бы следующим образом:

```
{  
  "language": "en",  
  "my_var": "key"  
}
```

Однако в Python это привело бы к созданию следующего словаря:

```
{  
  "language": "en",  
  "bla": "key"  
}
```

Переменная `my_var` оценивается, и ее значение используется в качестве ключа.

Глава 201. Скрытые возможности

201.1. Перегрузка операторов

Все в Python является объектами. Каждый объект имеет некоторые специальные внутренние методы, которые он использует для взаимодействия с другими объектами. Как правило, эти методы соответствуют соглашению об именовании. В совокупности это называется моделью данных Python (см. <https://docs.python.org/2/reference/datamodel.html>).

Вы можете перегрузить *любой* из этих методов. Это широко используется в перегрузке операторов в Python. Ниже приведен пример перегрузки операторов с использованием модели данных Python. Класс `Vector` создает простой вектор из двух переменных. Мы добавим соответствующую поддержку математических операций над двумя векторами с помощью перегрузки операторов.

```
class Vector(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, v):  
        # Сложение с другим вектором.  
        return Vector(self.x + v.x, self.y + v.y)  
  
    def __sub__(self, v):  
        # Вычитание другого вектора  
        return Vector(self.x - v.x, self.y - v.y)
```

```
def __mul__(self, s):
    # Умножение на скаляр.
    return Vector(self.x * s, self.y * s)

def __div__(self, s):
    # Деление на скаляр
    float_s = float(s)
    return Vector(self.x / float_s, self.y / float_s)

def __floordiv__(self, s):
    # Деление на скаляр (с округлением в сторону уменьшения).
    return Vector(self.x // s, self.y // s)

def __repr__(self):
    # Вывести дружелюбное представление класса Vector. В противном случае оно будет
    # выглядеть так: <__main__.Vector instance at 0x01DDDDC8>.
    return '<Vector (%f, %f)>' % (self.x, self.y, )
```

```
a = Vector(3, 5)
b = Vector(2, 7)
```

```
print a + b # Результат: <Vector (5.000000, 12.000000)>
print b - a # Результат: <Vector (-1.000000, 2.000000)>
print b * 1.3 # Результат: <Vector (2.600000, 9.100000)>
print a // 17 # Результат: <Vector (0.000000, 0.000000)>
print a / 17 # Результат: <Vector (0.176471, 0.294118)>
```

Приведенный пример демонстрирует перегрузку основных числовых операторов. Более подробную информацию можно найти здесь: https://docs.python.org/2/reference/datamodel.html?object.__getattr__&emulating-numeric-types.

Благодарности

Большое спасибо всем участникам сообщества Stack Overflow Documentation, которые помогли предоставить читателю эти примеры. Присылайте материалы для публикации или обновления примеров на web@petercv.com.

Çağatay Uslu Глава 20
2Cubed Главы 39, 122 и 147
4444 Глава 21
A. Ciclet Глава 196
A. Raza Глава 1
Aaron Christiansen Главы 40 и 109
Aaron Critchley Глава 1
Aaron Hall Глава 38
Abhishek Kumar Глава 149
abukaj Глава 200
acdr Глава 21
Adam Brenecki Глава 85
Adam Matan Главы 84 и 104
Adam_92 Глава 40
adeora Глава 197
Aditya Главы 40, 53, 134, 195 и 200
Adrian Antunez Глава 88
Adriano Глава 33
afeique Глава 1
Aidan Глава 75
Ajean Глава 5
Akshat Mahajan Главы 33, 67, 120 и 146
aldanor Глава 40
Aldo Глава 103
Alec Глава 200
alesxe Главы 5, 40, 87, 92 и 93
alejosocorro Главы 1 и 75
Alex Gaynor Глава 55
Alex L Глава 16
Alex Logan Глава 1
AlexV Глава 33
Alfe Глава 88
alfonso.kim Глава 16
ALinuxLover Глава 1
Alireza Savi Глава 192
Alleo Главы 21 и 149
Alon Alexиер Глава 116
amblina Главы 83, 85 и 129
Ami Tavory Глава 192
amin Глава 9
Amir Rachum Главы 19 и 39
Amitay Stern Главы 41 и 91
Anaphory Глава 159
anatoly techtonik Глава 160
Irea Глава 1
игrew Глава 131
Игrew Schade Глава 84
Irii Abramov Глава 1
Irzej Pronobis Главы 8, 38 и 55
Иу Главы 1, 5, 7, 17, 33, 51, 82 и 88
Иу Hayden Главы 8, 33, 67, 73, 75, 77, 98 и 149
angussidney Глава 1
Ani Menon Главы 1, 4, 40, 149 и 154

Annonymous Главы 78, 87 и 199
Anthony Pham Главы 13, 15, 16, 19, 20, 30, 33, 43, 45, 55, 60, 70 и 179
Antoine Bolvy Главы 1 и 149
Antoine Pinsard Глава 39
Antti Naapala Главы 5, 16, 72, 87 и 106
Antwan Глава 149
APerson Главы 21, 69 и 72
Aquib Javed Khan Глава 1
Ares Главы 1, 15, 20 и 41
Arkady Глава 33
Arpit Solanki Главы 1, 82 и 125
Artem Kolontay Глава 173
ArtOfCode Главы 67, 173 и 197
Arun Главы 65 и 108
Aryaman Aroга Глава 179
ashes999 Глава 75
asmeurer Главы 45 и 47
atayenel Глава 124
Athari Глава 151
Avantol13 Глава 38
avb Глава 30
Ахе Главы 21 и 149
B8vrede Главы 1, 40, 75 и 103
Baaing Cow Главы 1 и 200
Bahrom Глава 8
Bakuriu Глава 149
balki Глава 53
Barry Глава 20
Bastian Глава 86
bbayles Глава 128
Beall619 Главы 74 и 101
bee Главы 161 и 164
Benedict Bunting Глава 99
Bharel Главы 30 и 149
Bhargav Глава 40
Bhargav Rao Главы 41, 149 и 200
bignose Глава 149
Billy Глава 200
Biswa_9937 Главы 178, 182 и 183
bitchaser Глава 149
bixel Глава 200
blubberdiblub Глава 177
blueberryfields Глава 181
blueenvelope Главы 9 и 20
Bluethon Глава 149
boboquack Главы 10, 11 и 18
bogdanciobanu Глава 111
Bonifacio2 Глава 64
BorpreH Глава 19
Bosoneио Глава 46
Božo Stojković Главы 1, 14, 21, 33 и 149
brachev Глава 53

- Brendan Abel Глава 84
 brennan Глава 173
 Brett Cannon Главы 11 и 43
 Brian С Глава 1
 Brien Глава 41
 Bryan P Глава 1
 BSL Главы 1 и 197
 Burhan Khalid Глава 19
 BusyAnt Главы 1, 20, 43, 60, 78 и 88
 Buzz Глава 18
 Cache Staheli Глава 41
 CamelBackNotation Глава 33
 Cameron Gagnon Глава 149
 Camsbury Главы 9 и 33
 caped114 Глава 41
 carrdelling Глава 77
 Cbeb24404 Глава 1
 ceruleus Глава 1
 cfi Главы 21, 26 и 69
 Chіan Purohit Главы 20 и 33
 ChaoticTwist Главы 21, 33, 41 и 107
 Charles Главы 10, 21, 30, 41, 149 и 200
 Charul Глава 167
 Chinmay Hegde Главы 50, 94, 132, 164 и 171
 ChongTang Глава 21
 Chris Hunt Глава 16
 Chris Larson Глава 33
 Chris Midgley Глава 1
 Chris Mueller Глава 19
 Christian Ternus Главы 1, 16, 43, 51 и 78
 Christofer Ohlsson Глава 45
 Christophe Roussy Глава 200
 Chromium Глава 134
 Cilyan Глава 170
 Cimbali Глава 8
 cizixs Главы 19, 20 и 128
 cjds Главы 110 и 134
 Clíodhna Глава 1
 Claudiu Главы 1, 31, 67, 75, 80, 83, 88, 111, 118, 153, 192 и 193
 Clayton Wahlstrom Глава 149
 cledoux Глава 108
 CodenameLambda Главы 1 и 67
 Cody Piersall Глава 8
 Colin Yang Глава 149
 Comrade SparklePony Главы 180 и 181
 Conrad.Dean Главы 1, 5, 21, 38 и 43
 crhodes Глава 30
 c-l-s- Главы 1, 149, 161 и 191
 Dair Главы 11, 120 и 173
 Daksh Gupta Главы 1 и 38
 Dania Глава 1
 danidee Глава 33
 Daniel Глава 43
 Daniil Ryzhkov Глава 173
 Darkade Глава 173
 DARTH Kotik Глава 16
 DARTH Shadow Главы 1, 40, 75, 97, 149 и 173
 Dartmouth Главы 1, 62, 149 и 150
 Dave J Главы 40 и 149
 David Главы 9, 33, 124 и 161
 David Cullen Главы 30 и 121
 David Neuman Главы 41 и 149
 davidism Глава 13
 DawnPaladin Глава 33
 Dee Глава 186
 deeeenes Главы 1, 20 и 67
 deepakkt Глава 14
 DeepSpace Главы 9, 16, 77, 100, 149 и 200
 Delgan Главы 1, 16, 20 и 40
 denvaar Глава 167
 depperm Главы 1, 3, 38, 41 и 99
 DevD Глава 1
 Devesh Saini Глава 115
 DhiaTN Глава 16
 dhimanta Главы 184 и 185
 Dilettant Глава 200
 Dima Tisnek Глава 21
 djaszczurowski Глава 167
 Дос Глава 143
 dodell Глава 1
 Doraemon Глава 29
 Doug Henderson Глава 41
 Douglas Starnes Глава 1
 Dov Глава 30
 dreftymac Глава 40
 driaх Главы 39, 75 и 88
 Duh Глава 149
 Dunatotatos Глава 171
 dwierson Глава 149
 eiersson Глава 127
 edwinksl Глава 173
 eenblam Глава 21
 Elazar Главы 1, 7, 8, 13, 15, 16, 20, 21, 28, 38, 41, 67, 87, 111, 144 и 149
 Eleftheria Глава 113
 elegant Глава 33
 Ellis Главы 20 и 45
 Elodin Главы 33 и 195
 Emma Главы 20 и 21
 Enamul Hassan Глава 16
 engineercoding Глава 192
 Enrico Maria De Angelis Глава 1
 enrico.bacis Главы 21, 56 и 149
 erewok Глава 149
 Eric Глава 107
 Eric Finn Глава 16
 Eric Zhang Глава 110
 Erica Глава 1
 ericdwang Глава 149
 ericmarkmartin Главы 67, 98, 110 и 149
 Erik Godard Глава 1
 EsmaeelE Глава 1
 Esteis Главы 13 и 21
 ettanany Главы 27 и 149
 Everyone_Else Глава 149
 evuez Главы 7, 8, 14, 20, 40, 113 и 149
 exhuma Глава 20
 Fábio Perez Глава 31
 Faiz Halde Главы 21, 114 и 119

- FazeL Главы 111 и 148
 Felix D. Глава 16
 Felk Глава 21
 Fermi парадокс Глава 21
 Fernio Глава 173
 Ffisegydd Главы 14, 38, 98, 108 и 193
 Filip Haglund Глава 1
 Firix Главы 1 и 150
 flamenco Глава 56
 Flickerlight Глава 20
 Florian Bender Глава 21
 FMc Глава 43
 Francisco Guimaraes Глава 94
 Franck Dernoncourt Глава 1
 FrankBr Глава 36
 frankyjuang Глава 181
 Fred Barclay Главы 1 и 157
 Freddy Глава 1
 fredley Глава 45
 freidrichen Глава 21
 Frustrated Глава 74
 Gal Dreiman Главы 16, 20, 21, 33, 40, 136 и 137
 ganesh gadila Главы 20 и 41
 Ganesh K Глава 148
 Gareth Latty Глава 19
 garg10may Главы 21 и 149
 Gavin Глава 149
 Geeklhem Главы 14, 110 и 124
 Generic Snake Глава 16
 geoffspear Глава 149
 Gerard Roche Глава 1
 gerrit Глава 40
 ghostarbeiter Главы 16, 21, 33, 41, 45, 88, 132 и 149
 Giannis Spiliopoulos Глава 40
 GiantsLoveDeathMetal Главы 40 и 164
 girish946 Главы 31 и 154
 giucal Глава 55
 GoatsWearHats Главы 1, 16, 29, 41 и 72
 goodmami Глава 75
 Greg Глава 22
 greut Глава 37
 Guy Главы 16, 19 и 156
 H. Pauwelyn Глава 1
 hackvan Глава 164
 Hannele Глава 21
 Hannes Karppila Главы 14 и 134
 Harrison Глава 40
 hashcode55 Глава 76
 ha_1694 Глава 173
 heyhey2k Глава 94
 hiro protagonist Главы 54 и 200
 HoverHell Глава 12
 Hridhhi Dey Глава 105
 Hurkyl Главы 21 и 33
 hxysayhi Главы 66 и 168
 Ian Глава 1
 IanAuld Главы 1 и 21
 iankit Главы 21 и 37
 iBelieve Глава 19
 idjaw Глава 41
 ifma Глава 108
 Igor Raush Главы 1, 19, 20, 38, 39, 45, 67 и 98
 Ilia Barahovski Главы 32, 41 и 88
 ilse2005 Глава 30
 Inbar Rose Глава 16
 Indradhanush Gupta Глава 49
 Infinity Главы 19 и 115
 InitializeSahib Главы 38, 39 и 82
 intboolstring Главы 10, 21 и 45
 iScrE4m Глава 149
 J F Главы 1, 9, 15, 20, 29, 33, 38, 88, 97, 111, 119, 125 и 149
 Jörn Hees Глава 137
 JOHN Главы 21 и 67
 j3485 Глава 20
 jackskis Главы 53 и 67
 Jacques de Hooge Главы 97 и 151
 JakeD Глава 22
 James Главы 8, 14, 19, 20, 33 и 100
 James Elderfield Главы 20, 40, 45 и 149
 James Taylor Глава 1
 JamesS Глава 21
 Jan Глава 75
 jani Глава 20
 japborst Глава 86
 Jean Главы 1 и 40
 jedwards Главы 1, 102 и 149
 Jeff Hutchins Глава 110
 Jeffrey Lin Главы 1, 16, 75, 132 и 200
 Jelmers Глава 102
 JGreenwell Главы 3, 9, 33 и 37
 JHS Глава 21
 Jim Fasarakis Hilliard Главы 1, 16, 33, 41, 55, 87, 110, 149 и 200
 jim opleyduIVEN Глава 1
 Jimmy Song Глава 149
 jimsug Главы 1 и 20
 jkdev Главы 20 и 38
 jkitchen Глава 33
 JL Peyret Главы 40 и 51
 jlsarsch Глава 38
 jmunsch Главы 1, 47, 92, 125, 181 и 192
 JNat Главы 10, 20 и 29
 joel3000 Главы 21, 103 и 192
 Johan Lundberg Глава 1
 John Slegers Главы 1 и 149
 John Zwinck Глава 11
 Jonatan Главы 40, 64 и 87
 jonrsharpe Главы 1, 75, 78 и 98
 Joseph True Глава 1
 Josh Глава 149
 Jossie Calderon Глава 86
 jrst Глава 16
 JRodDynamite Главы 1, 21 и 40
 jtbies Глава 70
 Juan T Главы 1, 67, 90 и 149
 JuanPablo Глава 110
 Julien Spronck Главы 53 и 75
 Julij Jegorov Глава 188
 Justin Главы 30, 33, 40, 74 и 149

- Justin Chadwell Глава 125
 Justin M. Ucar Глава 149
 j__ Глава 41
 Kabie Глава 149
 Kallz Глава 38
 Kamran Mackey Глава 1
 Karl Knechtel Главы 67, 69 и 149
 KartikKannapur Главы 20 и 38
 kdopen Главы 19, 21 и 110
 keiv.fly Глава 194
 Kevin Brown Главы 1, 5, 14, 30, 53, 75, 146, 149 и 192
 KeyWeeUsr Главы 80 и 153
 KIDJourney Глава 21
 Kinifwune Главы 43 и 193
 Kiran Vemuri Глава 1
 kisanme Глава 1
 knight Глава 40
 kollery Глава 156
 konpsych Глава 47
 krato Глава 83
 Kunal Marwaha Глава 149
 Kwarttz Глава 21
 L3viathan Главы 33 и 98
 Lafexlos Главы 1, 9 и 20
 LDP Глава 20
 Lee Netherton Главы 21, 33 и 114
 Leo Главы 49 и 97
 Leo Thumma Глава 20
 Leon Глава 1
 Leon Z. Глава 74
 Liteye Главы 21 и 38
 loading... Главы 83 и 114
 Locane Глава 21
 lorenzofeliz Глава 2
 LostAvatar Главы 1 и 161
 Luca Van Oort Глава 165
 Luke Taylor Глава 70
 lukewrites Глава 20
 Lyndsy Simon Глава 21
 machine yearning Главы 19, 53 и 67
 magu_ Глава 77
 Mahdi Главы 21, 82 и 120
 Mahmoud Hashemi Глава 199
 Majid Главы 19, 28, 77, 82, 98, 112 и 173
 Malt Глава 200
 manu Глава 1
 MANU Глава 1
 Marco Pashkov Главы 29, 39, 40, 83 и 173
 Mario Corchero Глава 192
 Mark Глава 125
 Mark Miller Глава 130
 Mark Omo Глава 168
 Markus Meskanen Глава 21
 MarkyPython Глава 41
 Marlon Abeykoon Глава 79
 Martijn Pieters Главы 1, 67, 77 и 97
 Martin Valgur Глава 104
 Math Глава 16
 Mathias711 Глава 1
 matsjoyce Главы 1 и 9
 Matt Dodge Главы 149 и 200
 Matt Giltaji Главы 33, 40, 173 и 193
 Matt Rowli Глава 149
 MattCorr Главы 4 и 121
 Matthew Whitt Главы 21, 37, 39, 110, 146, 173 и 192
 mattgathu Главы 19, 117, 124 и 190
 Matthew Глава 77
 max Глава 67
 Max Feng Глава 14
 mbrig Глава 21
 mbsingh Глава 161
 Md Sifatul Islam Главы 28 и 75
 mdegis Глава 1
 Mechanic Главы 1, 9, 19 и 28
 mertyildiran Глава 1
 MervS Глава 125
 metmirr Глава 162
 mezzode Глава 28
 mgilson Глава 192
 Michael Recachinas Глава 149
 Michel Touw Глава 161
 Mike Driscoll Главы 1 и 13
 Miljen Mikic Глава 1
 Mimouni Глава 1
 Mirec Miskuf Глава 21
 mnoronha Главы 1 и 149
 Mohammad Julfikar Глава 123
 moshemeirelles Глава 1
 MrP01 Глава 20
 mrtuovinen Глава 92
 MSD Глава 1
 MSeifert Главы 9, 16, 19, 21, 26, 33, 37, 41, 43, 45, 48, 55, 64, 67, 68, 69, 70, 71, 72, 73 и 200
 muddyfish Главы 1, 20, 21, 33, 37, 57, 111 и 149
 Mukunda Modell Глава 37
 Muntasir Alam Глава 1
 Murphy4 Глава 33
 MYGz Глава 40
 Naga2Raja Глава 150
 Nier Speerstra Главы 40, 75 и 116
 naren Главы 42 и 89
 Nathan Osman Глава 190
 Nathaniel Ford Главы 1, 29 и 41
 ncmathsadist Глава 200
 nd. Глава 33
 nehemiah Глава 173
 nemesifixx Глава 10
 Ni. Главы 1 и 92
 Nick Humrich Глава 139
 Nicolás Глава 29
 Nicole White Глава 5
 niemmi Глава 149
 niyasc Главы 1, 43, 45 и 149
 nlsdfnbch Главы 5, 19, 28, 30, 53, 67, 117, 120 и 121
 Nour Chawich Глава 40
 поццРР4zeцГлавы 1, 19, 21, 28, 88 и 149
 Nuhil Mehdy Глава 173
 numbermaniacs Главы 1 и 9
 obust Глава 54
 Ohad Eytan Глава 5

- ojas mohril Глава 38
 omgimanerd Глава 200
 Or East Главы 8, 21, 75, 112 и 189
 OrangeTux Глава 149
 Ortomala Lokni Глава 173
 orvi Главы 1, 25, 41, 125, 133 и 134
 Oz Bar Глава 20
 Ozair Kafray Глава 30
 Риа Глава 21
 Parousia Главы 23 и 69
 Pasha Главы 3, 20, 21, 33, 37, 38, 67, 70, 77, 83 и 149
 Patrick Naugh Главы 1 и 200
 Paul Глава 5
 Paul Weaver Глава 88
 paulmorris Глава 5
 Paulo Freitas Глава 149
 Paulo Scardine Глава 39
 Pavan Nath Главы 1, 20 и 179
 pcurry Главы 29, 110 и 149
 Peter Brittain Глава 77
 Peter Mølgaard Pallesen Глава 73
 Peter Shinnars Глава 109
 petrs Глава 77
 philngo Глава 16
 Pigman168 Глава 96
 pistache Глава 38
 pktangyue Глава 149
 poke Глава 20
 PolyGeo Глава 145
 porpie Глава 149
 ppperry Глава 55
 Prem Narain Глава 59
 Preston Главы 138 и 173
 proprefenetre Глава 147
 proprius Глава 5
 PSN Глава 1
 pylang Главы 1, 8, 21, 33, 38, 43, 53, 54, 73, 80, 128, 147, 173, 192 и 200
 PYPL Глава 79
 Pythonista Главы 32 и 149
 pzp Главы 1, 29, 30, 33, 41, 49, 64 и 67
 Quill Глава 1
 qwertyuip9 Главы 161, 173 и 181
 R Colmenares Глава 10
 R Nar Глава 21
 Rápfi Irás Глава 82
 Régis B. Глава 173
 Raghav Глава 125
 Rahul Nair Главы 1, 21, 43, 88 и 143
 RahulNP Главы 5, 8, 45, 53, 64 и 112
 rajah9 Главы 14, 16 и 45
 Ram Grihi Глава 1
 РиомHash Глава 95
 rassar Глава 172
 ravigadila Главы 20, 60, 85, 91 и 104
 Razik Глава 158
 Rednivrug Главы 2, 35 и 63
 regnarg Глава 75
 Reut Sharabani Главы 64 и 200
 rfkortekaas Главы 1 и 115
 Ricardo Глава 157
 Riccardo Petraglia Главы 21, 84 и 117
 Richard Fitzhugh Глава 38
 rlee827 Глава 62
 rll Глава 21
 Rob H Глава 121
 Rob Murray Глава 94
 Ronen Ness Глава 30
 ronrest Главы 19 и 41
 Roy Iacob Глава 19
 rrao Глава 1
 rrawat Глава 161
 Ryan Smith Главы 21 и 120
 ryanyuyu Глава 21
 Ry Глава 149
 Sachin Kalkur Глава 125
 sagism Глава 5
 Saiful Azad Глава 43
 Sam Krygsheld Глава 1
 Sam Whited Глава 111
 Sangeeth Sudheer Глава 1
 Saqib Shamsi Глава 92
 Sardathrion Глава 103
 Sardorbek Imomaliyev Глава 103
 sarvajeetsuman Главы 9 и 16
 SashaZd Главы 12, 14, 29, 64, 86, 134, 143, 144, 146 и 194
 satsumas Глава 67
 sayan Главы 1 и 96
 Scott Mermelstein Главы 110, 135 и 149
 Sebastian Schrader Глава 173
 Selcuk Главы 1, 28, 149 и 201
 Sempoo Глава 38
 Serenity Главы 20, 30, 40, 43, 149 и 173
 SerialDev Глава 82
 serv Глава 40
 Seth M. Larson Главы 54 и 87
 Seven Главы 1, 11, 20 и 33
 sevenforce Главы 16 и 67
 ShadowRanger Глава 149
 Shantanu Alshi Глава 173
 Shawn Mehan Главы 10, 15, 19, 20 и 47
 Shihab Shahrar Глава 200
 Shijo Глава 198
 Shoe Глава 21
 Shrey Gupta Главы 41, 56 и 173
 Shreyash S Sarnayak Глава 12
 Shuo Глава 77
 SiggyF Главы 16 и 111
 Simon Fraser Глава 173
 Simon Hibbs Глава 169
 Simplans Главы 1, 9, 10, 14, 16, 19, 21, 28, 33, 37, 38, 41, 43, 44, 45, 50, 69, 75, 77, 82, 88, 99, 147, 149 и 173
 Sirajus Salayhin Главы 5, 175 и 176
 sisanared Глава 39
 skrrgwasme Главы 16 и 99
 SN Ravichiran KR Глава 75
 solarc Главы 20 и 149
 Soumendra Kumar Sahoo Главы 14 и 38
 SouvikMaji Глава 1
 sricharan Глава 149

StardustGogeta Глава 43
 stark Глава 1
 Stephen Nyamweya Глава 161
 Steve Barnes Главы 33 и 82
 Steven Maude Главы 11, 33, 47 и 92
 sth Главы 91, 92 и 141
 strpeter Главы 85 и 192
 StuxCrystal Главы 21, 37, 43, 56, 76, 82, 121 и 142
 Sudip Bhiari Глава 88
 Sun Qingyao Глава 101
 Sunny Patel Глава 21
 SuperBiasedMan Главы 1, 4, 15, 16, 20, 21, 26, 29, 30, 33, 41, 43, 64, 69, 77, 80, 88, 132, 149 и 200
 supersam654 Глава 70
 surfthecity Глава 6
 Symmitchry Главы 47 и 53
 sytech Глава 92
 Снадошфа Главы 1 и 134
 Tadhg McDonald Глава 149
 talhasch Главы 92, 93 и 164
 Tasdik Rahman Глава 30
 taylor swift Глава 1
 techydesigner Главы 1, 38, 43 и 190
 Teepeeemm Глава 152
 Tejas Jadhav Глава 201
 Tejus Prasad Глава 1
 TemporalWolf Глава 90
 textshell Главы 16, 28, 33 и 121
 TheGenie OfTruth Глава 1
 theheadofabroom Главы 41, 49 и 64
 the_cat_lady Глава 43
 The_Curry_Man Глава 16
 Thomas Ahle Главы 60 и 134
 Thomas Crowley Глава 105
 Thomas Gerot Главы 30, 43, 58, 61, 126 и 181
 Thomas Moreau Глава 119
 Thtu Глава 80
 Tim Глава 149
 Tim D Глава 200
 tjohnson Глава 44
 tlo Глава 82
 tobias_k Главы 28 и 40
 Tom Глава 16
 Tom Barron Главы 1 и 21
 Tom de Geus Глава 1
 Tony Meyer Глава 43
 Tony Suffolk 66 Главы 9, 10, 38 и 40
 tox123 Глава 38
 TuringTux Глава 4
 Tyler Crompton Глава 86
 Tyler Gubala Главы 119 и 122
 Udi Глава 54
 UltraBob Глава 38
 Umibozu Глава 30
 Underyx Глава 49
 Undo Глава 9
 unutbu Глава 116
 user2027202827 Глава 163
 user2314737 Главы 8, 9, 13, 16, 20, 28, 30, 33, 38, 40, 41, 57, 60, 64, 69, 75, 88, 110, 134, 142, 149, 154, 161, 196 и 200
 user2683246 Главы 43 и 187
 user2728397 Глава 166
 user2853437 Глава 1
 user312016 Главы 1, 40 и 77
 user3333708 Глава 33
 user405 Глава 33
 user6457549 Глава 20
 Utsav T Глава 20
 vaichidrewar Глава 1
 valeas Глава 161
 Valentin Lorentz Главы 20, 21, 43, 110 и 149
 Valor Naram Глава 43
 vaultah Главы 20, 33, 43 и 77
 Veedrac Главы 21, 33, 41 и 110
 Vikash Kumar Jain Глава 174
 Vin Главы 1 и 17
 Vinayak Глава 149
 vinzee Главы 99, 116 и 120
 viveksyngh Главы 10 и 19
 VJ Magar Глава 31
 Vlad Bezden Глава 103
 weewooquestionnaire Глава 1
 WeizhongTu Главы 41 и 149
 Wickramaranga Глава 53
 Will Главы 5, 15, 16, 21, 30, 33, 91, 116, 117, 121 и 164
 Wingston Sharon Глава 155
 Wladimir Palant Главы 21, 37 и 197
 wnnmaw Главы 41, 43 и 53
 Wolf Глава 149
 WombatPM Глава 30
 Wombatz Глава 200
 wtwtwt Глава 173
 wwii Глава 14
 wythagoras Глава 9
 Xavier Combelle Главы 15, 19, 111 и 120
 XCoder Real Глава 47
 xgord Главы 14 и 30
 XonAether Глава 52
 xtreak Главы 67 и 149
 Y0da Глава 85
 ygram Глава 190
 Yogendra Sharma Главы 1 и 117
 yurib Глава 64
 Zach Janicki Глава 1
 Zags Глава 1
 Zaid Ajaj Глава 67
 zarak Глава 149
 Zaz Глава 21
 zenlc2000 Глава 34
 Zhanping Shi Глава 145
 zmo Главы 83, 140 и 141
 zondo Главы 75 и 83
 zopieux Глава 173
 zvone Главы 13, 37, 39 и 112
 zxxz Глава 33
 Zydnar Глава 81
 KrİSTOF Глава 117
 лuser Главы 67 и 77
 Некто Главы 24, 37 и 128



Серия «Программирование от экспертов»

Справочное издание
Анықтамалық басылым

САМОЕ ПОЛНОЕ РУКОВОДСТВО ПО РАЗРАБОТКЕ НА PYTHON В ПРИМЕРАХ ОТ СООБЩЕСТВА STACK OVERFLOW

Оформление обложки С. А. Арутюнян
Ответственный за выпуск И. В. Резько

Подписано в печать 12.12.2023.

Изготовлено в 2024 г.

Формат 70x100¹/₁₆. Бумага офсетная. Печать офсетная. Гарнитура Noto Serif SemiCondensed.

Усл. печ. л. 51,6. Тираж экз. Заказ №

Общероссийский классификатор продукции ОК-034-2014 (КПЕС 2008);
58.11.1 — книги, брошюры печатные

Произведено в Российской Федерации

Изготовитель: ООО «Издательство АСТ»

129085, Российская Федерация, г. Москва, Звездный бульвар, дом 21,
строение 1, комната 705, пом. I, этаж 7.

Адрес места осуществления деятельности по изготовлению продукции:

123112, Российская Федерация, г. Москва, Пресненская набережная, д. 6, стр. 2,
Деловой комплекс «Импери́я», 14, 15 этаж.

Наш электронный адрес: ask@ast.ru

Наш сайт: www.ast.ru Интернет-магазин: www.book24.ru

Өндіруші: «Издательство АСТ» ЖШҚ

129085, Ресей Федерациясы, Звездный бульвары, 21-үй, 1-құрылыс, 705-бөлме, I үй-жай, 7-қабат.

Өнім өндіру қызметін жүзеге асыру мекенжайы:

123112, Ресей Федерациясы, Мәскеу, Пресненская жағ., 6-үй, 2-құр.,
«Импери́я» іскерлік кешені, 14, 15-қабат

Біздің электрондық мекенжайымыз: www.ast.ru E-mail: ask@ast.ru

Интернет-магазин: www.book24.kz

Интернет-дукен: www.book24.kz

Импортер в Республику Казахстан и Представитель по приему претензий
в Республике Казахстан — ТОО РДЦ Алматы, г. Алматы.

Қазақстан Республикасына импорттаушы және Қазақстан Республикасында
наразылықтарды қабылдау бойынша өкіл — «РДЦ-Алматы» ЖШС,

Алматы қ., Домбровский көш., 3«а», Б литері офис 1.

Тел.: 8 (727) 2 51 59 90,91, факс: 8 (727) 251 59 92 ішкі 107;

E-mail: RDC-Almaty@eksmo.kz, www.book24.kz

Тауар белгісі: «АСТ» Өндірілген жылы: 2024

Өнімнің жарамдылық мерзімі шектелмеген

Ресей Федерациясында өндірілген

Многие специалисты едины во мнении о том, что один из самых мощных и «дружелюбных» языков программирования — Python — язык будущего.

В этой книге вы найдете множество практических примеров кодов на этом языке, написанных ведущими разработчиками программного обеспечения со всего мира. Все программы сопровождаются подробными комментариями технических специалистов сообщества Stack Overflow, изящно и профессионально решающих самые сложные задачи при создании ПО.

Здесь представлен мощный и универсальный инструментарий для профессиональной работы на Python в самых разных областях применения: при сетевом программировании, визуализации данных, многопоточности и многопроцессорности, программировании «интернета вещей» и многих других. Отдельное внимание уделено повышению скорости работы Python-кода и оптимизированию его производительности.



рекомендовано

Библиотекой программиста

книги для любого настроения здесь



ИЗДАТЕЛЬСКАЯ ГРУППА АСТ

www.ast.ru | www.book24.ru

vk.com/izdatelstvoast

ok.ru/izdatelstvoast

ISBN 978-5-17-160252-9



9 785171 602529