ВЫСШЕЕ ОБРАЗОВАНИЕ

С. А. Чернышев

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА PYTHON

УМО ВО РЕКОМЕНДУЕТ

Юрайт издательство

С. А. Чернышев

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА PYTHON

УЧЕБНОЕ ПОСОБИЕ ДЛЯ ВУЗОВ

Рекомендовано Учебно-методическим отделом высшего образования в качестве учебного пособия для студентов высших учебных заведений, обучающихся по ИТ направлениям

Книга доступна на образовательной платформе «Юрайт» urait.ru, а также в мобильном приложении «Юрайт.Библиотека»

Москва • Юрайт • 2022

УДК 004.432(075.8)
ББК 32.973.2я73
Ч-49

Автор:

Чернышев Станислав Андреевич — кандидат технических наук, доцент кафедры интеллектуальных систем и защиты информации Института информационных технологий и автоматизации Санкт-Петербургского государственного университета промышленных технологий и дизайна, доцент кафедры информатики факультета информатики и прикладной математики Санкт-Петербургского государственного экономического университета.

Рецензенты:

Гордеев А. В. — профессор, доктор технических наук, профессор кафедры вычислительных систем и сетей Института вычислительных систем и программирования Санкт-Петербургского государственного университета аэрокосмического приборостроения;

Штеренберг С. И. — кандидат технических наук, доцент кафедры защищенных систем связи факультета инфокоммуникационных сетей и систем Санкт-Петербургского государственного университета телекоммуникаций имени профессора М. А. Бонч-Бруевича.

Чернышев, С. А.

Ч-49 Основы программирования на Python : учебное пособие для вузов / С. А. Чернышев. — Москва : Издательство Юрайт, 2022. — 286 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-14350-8

В курсе подробно описывается не только большое количество базовых понятий и операторов языка программирования Python, но и ряд нюансов, с которыми так или иначе предстоит встретиться при его использовании в процессе написания программных продуктов. Материал подается по принципу «от простого к сложному» и сопровождается большим количеством примеров и упражнений, что позволяет сформировать у студентов практические навыки программирования и тестирования разрабатываемых приложений. Все исходные коды рассматриваемых примеров можно скачать с репозитория автора на GitHub.

Соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования.

Курс предназначен для студентов высших учебных заведений, которые обучаются по инженерно-техническим направлениям.

УДК 004.432(075.8)
ББК 32.973.2я73

ISBN 978-5-534-14350-8

Оглавление

Введение

Данная книга познакомит вас с основами языка программирования Python. Конечно, в одной книге невозможно рассмотреть все аспекты и нюансы этого языка программирования, но автор проделал значительную работу, стараясь максимально осветить его базовые возможности.

Все исходные коды приведенных в книге примеров можно скачать из репозитория автора: https://github.com/MADTeacher/python_basics. Для их запуска вам понадобится open-source экосистема Anaconda с версией Python 3.8 или старше[1], которая включает в себя такой инструмент, как Jupyter Notebook, а также интегрированная среда разработки PyCharm[2]. Их установка, настройка и использование не представляют сложности, поэтому и не рассматриваются в данной книге, но если у вас возникнут какие-то трудности, то в сети интернет достаточно материала для их решения.

Это сделано специально, так как в процессе работы в любой организации (да даже над упражнениями данной книги) вам все равно, так или иначе, придется осуществлять поиск нужной информации в интернете. Неважно, какую позицию в дальнейшем придется занимать, от умения быстро найти ту информацию, которая позволит решить возникающие проблемы, будет зависеть очень многое. И именно в этом месте кроется первая ловушка, в которую попадают начинающие разработчики — они бездумно копируют найденный код и переносят его в рабочий проект. Пусть на начальных стадиях это и поможет двигаться быстрыми шагами, но вы даже не представляете, какую бомбу замедленного действия подкладываете всем участникам проекта. Конечно, есть code review, одной из задач которого является поиск таких моментов. Но все мы люди и имеем свойство ошибаться. Особенно, когда трещат сроки проекта и приходится постоянно задерживаться на работе, чтобы успеть в обозначенный договором срок. К тому же в такие моменты зачастую не до code review. И мало того, если сборка проекта будет падать с указанием ошибок на ваш кусок кода, вы не сможете ответить на вопрос: «Что этот код в действительности делает?» (а не ка-

---

[1] Она доступна для скачивания по ссылке: https://www.anaconda.com/products/individual.

[2] https://www.jetbrains.com/ru-ru/pycharm/download/

кую вашу проблему он решил). Поэтому, если хотите перенести найденный код в проект, не копируйте его, а перепечатывайте вручную. Да, это медленнее, но это позволит вам иметь представление, что же на самом деле происходит (или найти ошибки), а не верить на слово тому человеку, который выложил этот код в открытый доступ.

Теперь коснемся упражнений к разделам. Упражнения к темам с 1 по 7 предназначены для выполнения одним человеком. Вы и далее, конечно, можете продолжать выполнять упражнения в одиночку, но лучше скооперироваться с кем-нибудь и выполнять их, как минимум, в паре. Этот опыт вам пригодится в дальнейшем, поскольку разработка современных информационных систем ведется командой, а не одиночками. Таким образом, работая в паре, вы научитесь:

1) декомпозировать задачи, чтобы их можно было выполнять параллельно, а после интегрировать в одно приложение;

2) работать в команде, что подразумевает постоянное обсуждение возникающих трудностей и нахождение компромисса относительно их решения. Так, например, для начала попробуйте договориться о стиле кодирования, которого будете придерживаться (или возьмите за основу PEP 8);

3) тайм-менеджменту, а именно предварительной оценке необходимого времени для решения задач и упражнения в целом. Не переживайте, если поначалу не будет получаться. Все дело в опыте (и ваша задача — его получить).

Само собой, это еще не весь список того, какой опыт вы получите при выполнении упражнений с кем-то, а не в одиночку, так как с собой договориться намного проще, чем с другим человеком.

В первой теме книги затрагиваются такие моменты, как: встроенные переменные, менеджмент памяти, глобальная блокировка интерпретатора, интернированные объекты и зачем они нужны, а также почему принято считать, что в Python нет переменных.

Вторая тема посвящена синтаксису Python. В ней рассматривается, какие операторы существуют и как используются, правила именования переменных, уровни отступов, что такое ветвление и циклы, для чего используется протокол итерации, как он работает и т. д.

В третьей теме речь пойдет о функциях и всем, что с ними связано: областях видимости, замыканиях, аннотациях, генераторах, декораторах и т. д.

Четвертая тема приоткрывает завесу тайн над модулями и пакетами, описывает, как они импортируются и что при этом происходит, а также способы их создания.

В пятой теме рассматривается, что такое объектно-ориентированное программирование и то, как его концепции (инкапсуляция,

наследование и полиморфизм) реализованы в Python. Какие существуют различия между: классом и экземпляром класса, переменной (методом) класса и экземпляром класса и т. д.

В шестой теме затрагивается такой аспект всех современных языков программирования, как исключения, которые являются высокоуровневым механизмом управления потоком выполнения и могут генерироваться интерпретатором или самим программистом «вручную».

В седьмой теме обсуждаются многопоточное, многопроцессорное и асинхронное программирование средствами языка программирования Python, их различия и ситуации, в которых их следует использовать.

В восьмой теме рассматривается работа с библиотекой PySide2, которая позволяет разрабатывать приложения с графическим пользовательским интерфейсом и является официальной реализацией кроссплатформенного фреймворка Qt для Python, развитие которой поддерживается The Qt Company.

Девятая тема — сетевое программирование средствами Python, базовые протоколы обмена данными с установлением и без установления соединения между клиентами и сервером.

В десятой теме рассматривается работа с встроенной реляционной базой данных — SQLite, средствами модуля *sqlite3*.

В одиннадцатой теме рассматриваются способы тестирования разрабатываемых приложений на языке программирования Python и некоторые инструменты для этого.

Хочу выразить огромную благодарность свой невесте, ученикам (Антонову Александру, Быкову Алексею, Дуку Герману, Нагорных Максиму и Твардовскому Георгию) за оказанную помощь и поддержку в процессе написания данной книги.

В результате изучения материалов данного учебного пособия обучающиеся должны:

знать

- стандартные модули языка программирования Python,
- основные типы, структуры данных и особенности работы с ними,
- синтаксис языка программирования,
- ключевые особенности интерпретатора CPython;

уметь

- применять стандартные структуры данных при разработке приложений,
- разрабатывать многопоточные, асинхронные и многопроцессорные приложения,
- разрабатывать приложения с графическим пользовательским интерфейсом,
- писать тесты для проверки правильности функционирования разрабатываемых приложений,

- разрабатывать приложения, устойчивые к возникающим ошибкам в процессе их работы,
- разрабатывать приложения, способные взаимодействовать между собой по сети;

владеть

- основными принципами объектно-ориентированного программирования,
- навыками разработки многопоточных, асинхронных и многопроцессорных приложений,
- навыками работы с исключениями,
- навыками разработки клиент-серверных приложений,
- основами тестирования приложений.

Тема 1
КРАТКАЯ ИСТОРИЯ PYTHON, ЕГО ОСОБЕННОСТИ И ТИПЫ ДАННЫХ

В данной теме будут рассматриваться история создания языка программирования Python, актуальность на текущий момент времени и различные нюансы его реализации, с которыми лучше познакомиться до того момента, как начнете на нем программировать. К таким нюансам относятся: глобальная блокировка интерпретатора при написании многопоточных программ с разделяемыми ресурсами на CPU, способ хранения значений «переменных» и т. д.

Также рассматриваются базовые встроенные типы данных Python и способы работы с ними.

В результате изучения данной темы обучающиеся должны:

знать

- историю развития языка программирования Python,
- встроенные типы, структуры данных и особенности работы с ними,
- ключевые особенности интерпретатора CPython;

уметь

- применять различные операции к типам и структурам данных,
- работать с переменными изменяемого и неизменяемого типа данных;

владеть

- навыками работы с переменными изменяемого типа данных,
- основными методами работы с существующими типами данных.

1.1. Краткая история языка программирования Python

Python — интерпретируемый язык программирования с неявной динамической типизацией, первая версия которого вышла в 1991 г. Автор языка — нидерландский программист Гвидо ван Россум (рис. 1.1), в Python-сообществе более известный как «великодушный пожизненный диктатор» (*Benevolent Dictator For Life* (BDFL)), полномочия которого он сложил с себя в июле 2018 г. Термин BDFL применяется к основателю проекта, который оставляет за собой право на окончательное принятие решения (добавление функциональности, развитие архитектуры проекта и т. д.). Название Python

язык программирования получил в честь комик-группы из Великобритании — Монти Пайтон (*Monty Python*).

Рис. 1.1. **Гвидо ван Россум**

На текущий момент времени существует две версии языка программирования: python3 (версия 3.0 вышла в 2008 г., а на момент написания пособия актуальная версия — 3.9) и python2 (версия 2.0 вышла в 2000 г., а «актуальная» версия — 2.7). При этом python3 в ряде случаев обратно не совместим с предыдущей версией языка. Таким образом, начиная с 2008 года Python-сообщество было разделено на два лагеря, поскольку большинство компаний не имело возможности отказаться от legacy-кода и библиотек в тех проектах, которые были уже запущены (поддерживались) и приносили прибыль, в угоду полному переносу на python3. Начиная же с 2020 г. была официально прекращена поддержка python2 [1], в связи с чем сообщество перестало устранять критические ошибки и уязвимости в данной версии языка программирования и полностью переключилось на развитие python3.

Согласно индексу TIOBE [2], который рассчитывает популярность языка программирования на основе поисковых запросов типа: «+"<language> programming"», в 2019 г. Python вышел на 3-е место (и продолжает занимать его, см. рис. 1.2), уступая только Java и C [3]. А по данным аналитической компании RedMonk, которая ориентируется в своей работе на разработчиков программного обеспечения, в январе 2020 г. Python вышел на второе место по популярности среди языков программирования, уступая только JavaScript [4]. RedMonk же формирует свой индекс популярности на основе базы данных GitHub Archive с ее перекрестной проверкой по базам Stack Overflow.

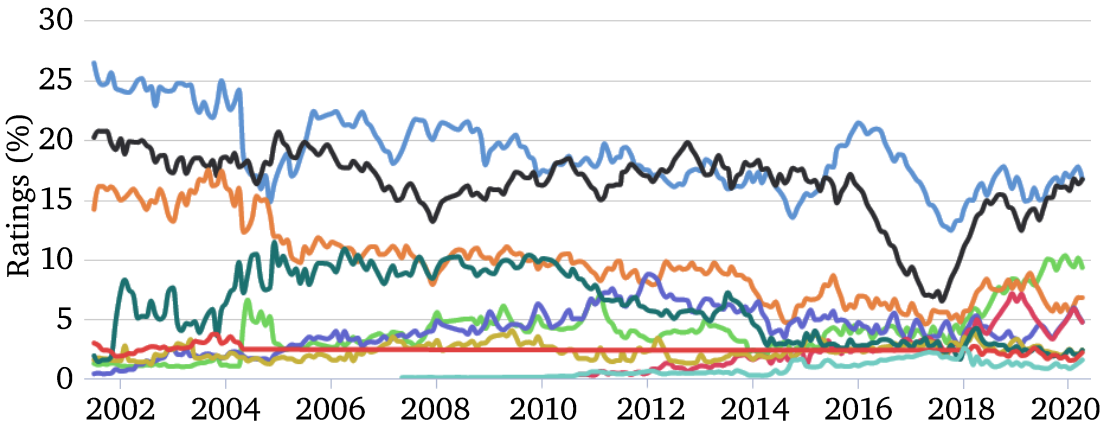

Рис. 1.2. **Популярность языков программирования с 2002 по 2020 г., согласно индексу TIOBE [3]**

Java; C; Python; C++; C#; Visual Basic; JavaScript; PHP; SQL; R

По данным (рис. 1.3), представленным в аналитической статье Stack Overflow (перевод статьи доступен в [5]), можно судить, что рост популярности Python пришелся на период 2012—2014 гг. и связан в первую очередь с интересом разработчиков к сферам аналитики данных, нейронных сетей, машинного обучения и компьютерного зрения. Так, например, библиотека для анализа и обработки данных Pandas довольно быстро набирала популярность: в 2011 г. на нее практически не приходилось никаких запросов из общего трафика сервиса, однако на момент публикации статьи количество запросов возросло до 1 %. Такая же динамика наблюдается по библиотекам numpy и matplotlib. Приведенная статистика свидетельствует о том, что развитие Python больше связано с областью Computer и Data Science, а не с веб-разработкой.

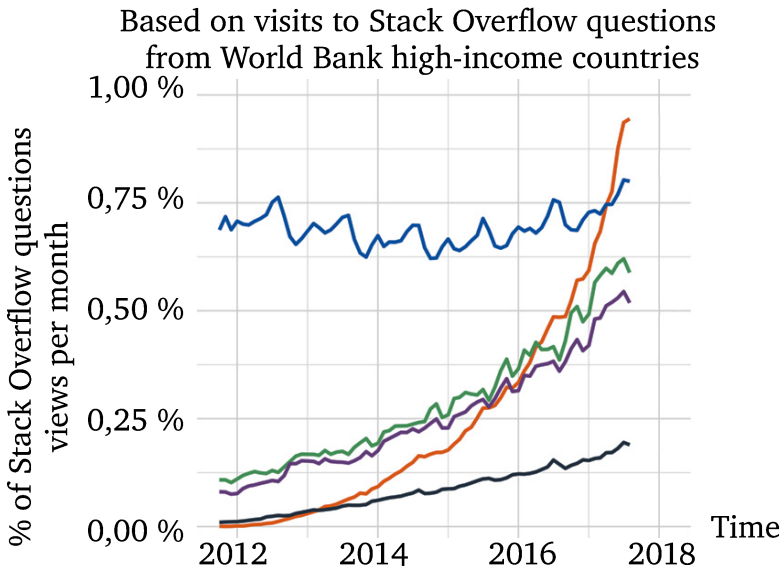

Рис. 1.3. **Процент от общего трафика запросов сервиса Stack Overflow по библиотекам Python в период с 2012 по 2018 год**

Pandas; Django; Numpy; Matlotlib; Flask

Такой рост популярности Python связан прежде всего с его простотой, большим количеством различных библиотек, удобными средствами развертывания и тестирования приложений. Это позволяет в более короткие сроки проверить возникающие у разработчика идеи или разработать MVP (*Minimum Viable Product*) — минимально жизнеспособный продукт, который можно представить потенциальным инвесторам или заказчикам стартапа.

Сама сборка проекта, написанного на Python, осуществляется следующим образом. В момент запуска исходного кода осуществляется его компиляция в «байт-код» (низкоуровневое и независимое от платформы представление исходного кода), после чего он передается на исполнение «виртуальной машине» [6]. Это позволяет разрабатываемой программе (написанному сценарию) выполняться быстрее, особенно при последующих запусках. При компиляции кода Python сохраняет его в промежуточные файлы с расширением «.pyc» (скомпилированный исходный файл .py). До версии 3.2 эти промежуточные файлы хранились в одном каталоге с проектом. Теперь же для них в каталоге проекта создается подкаталог «__pycache__». При следующем запуске программы Python загрузит файлы «.pyc» и пропустит шаг компиляции при условии, что исходный код проекта не изменялся после того, как байт-код был сохранен, и запускается той же версией Python, которая создавала байт-код.

Виртуальная машина Python (*Python Virtual Machine*, PVM) представляет собой большой закодированный цикл, проходящий по инструкциям поступающего в нее байт-кода. PVM по своей сути — исполняющий механизм, который всегда присутствует в виде части системы Python и представляет собой компонент, который по-настоящему выполняет подаваемые на вход сценарии. Именно PVM и является последним звеном того, что называется «интерпретатор Python».

На данный момент времени можно выделить следующие реализации языка Python.

1. CPython — первоначальная и стандартная реализация Python, поэтому букву «C», которая указывает на то, что CPython написан на переносимом языке ANSI C, зачастую не опускают.
2. Jython — альтернативная реализация языка Python. Jython компилирует исходный код Python в байт-код Java (бесшовная интеграция, позволяющая использовать сценарии Python в приложениях, написанных на Java), после чего подает его на вход виртуальной машине Java (*Java Virtual Machine*, JVM).
3. IronPython (Python для .NET) — реализация Python, схожая по своей концепции с Jython, только для платформы .NET.
4. Stackless — реализация стандартного языка CPython, ориентированная на параллелизм и предлагающая эффективные варианты многопроцессорной обработки выполняемого кода.

5. PyPy — реализация стандартной системы CPython, которая ориентирована на производительность. PyPy предлагает быструю реализацию Python с JIT-компилятором (*just-in-time*), способную выполнять ненадежный код в защищенной среде. Помимо этого, данная реализация Python по умолчанию включает поддержку Stackless Python.

В следующем разделе будут рассматриваться некоторые особенности реализации интерпретатора CPython (3.7). Часть из них можно встретить в других реализациях, а некоторые специфичны только для данной реализации языка программирования.

1.2. В Python все является объектом

Как уже обсуждалось ранее, в Python все является объектом! Давайте проверим это, используя следующий код:

```
i = 5
print(isinstance(i, object))
True
print(isinstance('Oo', object))
True
print(isinstance([2, 4, 't'], object))
True
print(isinstance({'a': 3, 'b': 5}, object))
True
print(isinstance((2, 4, 'w'), object))
True
def test_func():
    pass
print(isinstance(test_func, object))
True
```

Каждый объект Python хранит как минимум три вида данных:

— счетчик ссылок, который управляет временем жизни объекта в памяти;

— тип: изменяемый или неизменяемый;

— фактическое значение, к которому можно обратиться по имени объекта.

1.3. В Python нет переменных

Переменные в Python полностью отличаются от того, чем они являются в C++ или C. По факту их нет! **Вместо переменных используются имена.** Чаще всего, говоря о переменных, подразумеваются именно имена, поэтому важно понимать разницу [7, 8]. Для начала разберем, что такое переменные в рамках языка программирования C и как они работают.

При выполнении кода типа:

```
int val = 15;
```

программа отрабатывает следующие этапы:

1) выделение памяти;
2) запись в выделенную память значения;
3) отображение того, что переменная с именем *val* указывает на это значение.

В памяти это может выглядеть так, как показано на рис. 1.4.

val	
Адрес	0x5a1
Значение	15

Рис. 1.4. **Хранение значения переменной в памяти в C++**

Если в последующем переменной *val* присваивается новое значение:

```
val = 255;
```

то оно перепишет хранящееся по адресу *0x5a1* старое значение 15. Теперь область памяти будет выглядеть так, как показано на рис. 1.5.

val	
Адрес	0x5a1
Значение	255

Рис. 1.5. **Изменение значения переменной в C++**

Это означает, что переменная *val* — изменяемая, и фактически происходит работа с областью памяти, на которую она ссылается.

Если объявить новую переменную и инициализировать ее значением, хранящимся в переменной *val*, то будет выделено место под новое число и по адресу выделенной памяти запишется значение, хранящееся в *val* (рис. 1.6):

```
int new_val = val;
```

val	
Адрес	0x5a1
Значение	255

new_val	
Адрес	0x5a5
Значение	255

Рис. 1.6. **Объявление новой переменной в C++**

Обратите внимание на тот факт, что произошло выделение памяти под новую переменную, и она не ссылается на значение, хра-

нящееся по адресу переменной *val*, а сама хранит его. И перезапись значения переменной *new_val* (рис. 1.7) можно производить без опасения того, что изменится значение, хранящееся в переменной *val*:

```
new_val = 47;
```

val	
Адрес	0x5a1
Значение	255

new_val	
Адрес	0x5a5
Значение	47

Рис. 1.7. **Изменение значения новой переменной в C++**

На текущем этапе запомните, что при изменении значения переменной адрес, на который она ссылается, не поменялся. Это важно для понимания того, как дела обстоят в Python (CPython).

Теперь на тех же самых примерах рассмотрим, что произойдет при написании аналогичного кода на Python. При выполнении кода:

```
val = 15
```

интерпретатор CPython отработает следующие этапы:

1) создаст PyObject;
2) присвоит PyObject-у typecode;
3) в PyObject записывается значение 15;
4) создается имя *val*;
5) *val* указывает на новый PyObject;
6) счетчик ссылок PyObject-а увеличивается на 1.

PyObject характерен только для CPython и является базовой структурой для всех объектов:

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

Часть объектов расширяют данную структуру, добавляя необходимые поля, но *ob_refcnt* (счетчик ссылок) и **ob_type* (тип) должны обязательно присутствовать. Тип объекта (и только он) определяет, что можно делать с самим объектом, и не меняется в течение его жизни (может поменяться только в крайне редких обстоятельствах). Принцип работы счетчика ссылок и его место в Python будут рассмотрены немного позже.

На рис. 1.8 и далее приводится общий вид, а не подробное описание расположения объектов и их атрибутов в области памяти.

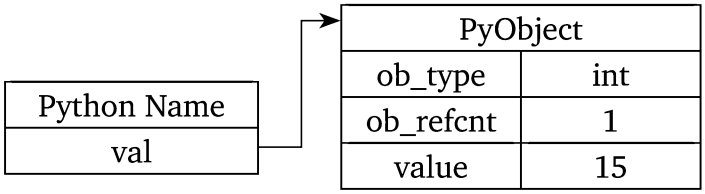

Рис. 1.8. **Хранение значения «переменной» в памяти в Python**

Как можно заметить, имя переменной ссылается не на участок памяти, в котором хранится значение, а на создаваемый объект Python, который уже хранит присваиваемое ей в коде значение.

Если *val* присвоить новое значение:

```
val = 255
```

интерпретатор CPython отработает следующие этапы:

1) создаст PyObject;
2) присвоит PyObject-у typecode;
3) в PyObject записывается значение 255;
4) *val* указывает на новый PyObject;
5) счетчик ссылок нового PyObject-а увеличивается на 1;
6) счетчик ссылок старого PyObject-а уменьшается на 1.

Область памяти после этого будет выглядеть так, как показано на рис. 1.9.

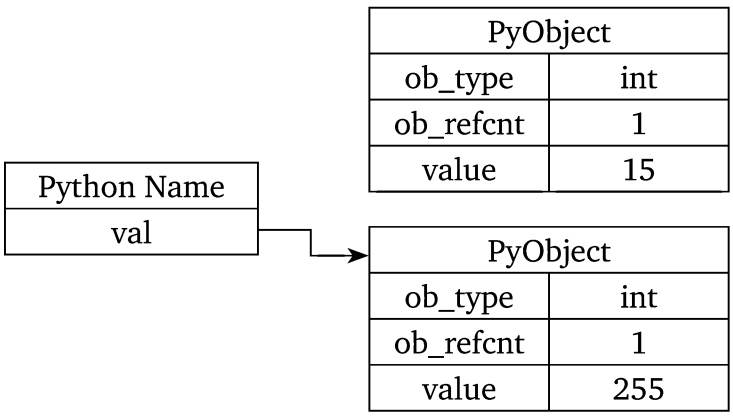

Рис. 1.9. **Изменение значения переменной в Python**

Теперь *val* указывает на новый PyObject, который хранит значение 255. Это явно указывает, что «=» является не присваиванием, а «привязкой» имени *val* к ссылке. Поскольку счетчик ссылок предыдущего объекта теперь равен нулю, он будет удален сборщиком мусора.

Теперь посмотрим, что происходит при добавлении нового имени *new_val*, которому присвоим *val*:

```
new_val = val
```

В данном случае не будет создаваться новый объект, а имя *new_val* будет ссылаться на уже существующий объект, на который ссылается сам *val*, что увеличит счетчик ссылок этого объекта на 1:

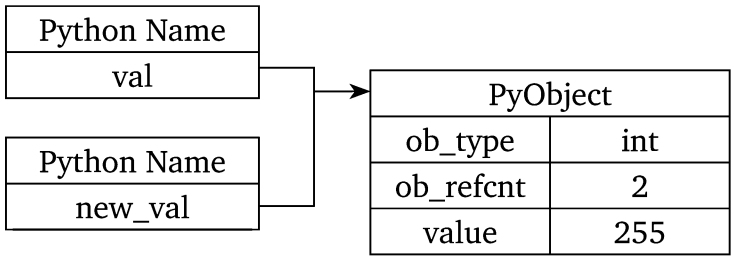

Рис. 1.10. **Объявление новой «переменной» в Python и ее инициализация значением из старой «переменной»**

Проверка имен на эквивалентность *new_val is val* вернет значение *True*, что явно указывает на то, что они ссылаются на один и тот же объект.

Теперь прибавим к *new_val* число 3:

```
new_val += 3
```

Этот код уже вызовет цепочку с созданием нового объекта и теперь область памяти будет выглядеть так, как показано на рис. 1.11.

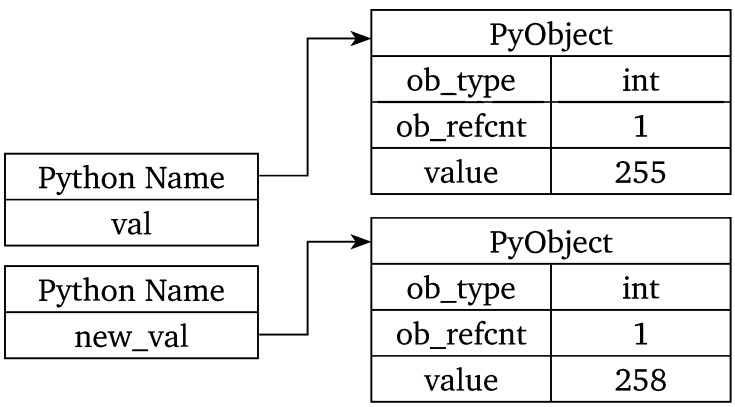

Рис. 1.11. **Изменение значения новой «переменной» в Python**

Данное поведение характерно для работы с неизменяемыми типами данных. Для примера работы с изменяемым типом возьмем список, который инициализируется следующим набором элементов:

```
my_list = [1, 3, 6]
```

Интерпретатор использует PyVarObject вместо PyObject для создания объектов переменной длины. От предыдущего он отличается

тем, что хранит еще и число вложенных в него элементов в переменной *ob_size*. Структура PyVarObject представлена ниже:

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; // Количество элементов в объекте
} PyVarObject;
```

А с учетом того, что для каждого типа создаваемого объекта существует своя структура, список на уровне кода интерпретатора на языке C описывается следующей структурой:

```
typedef struct {
    PyVarObject ob_base;
    PyObject **ob_item; // указатель на реальные данные
    Py_ssize_t allocated;
} PyListObject;
```

Поле *ob_item* — массив указателей на PyObject, а *allocated* — значение, под какое количество элементов зарезервирован размер массива *ob_item* до его увеличения.

Отсюда можно проследить закономерность, что *PyListObject* является расширением *PyVarObject*, который в свою очередь расширяет функционал *PyObject*. Условный вид области памяти в случае работы со списком представлен на рис. 1.12.

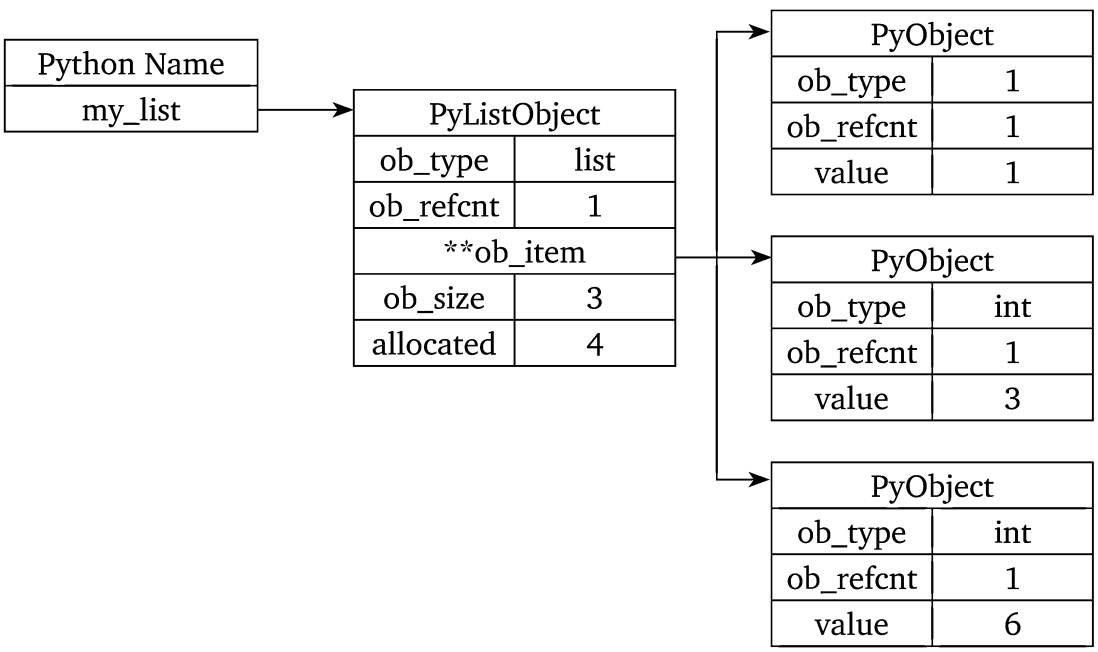

Рис. 1.12. **Объявление и инициализация списка в Python**

Теперь добавим в созданный список еще один элемент (рис. 1.13):

```
my_list.append('Oo')
```

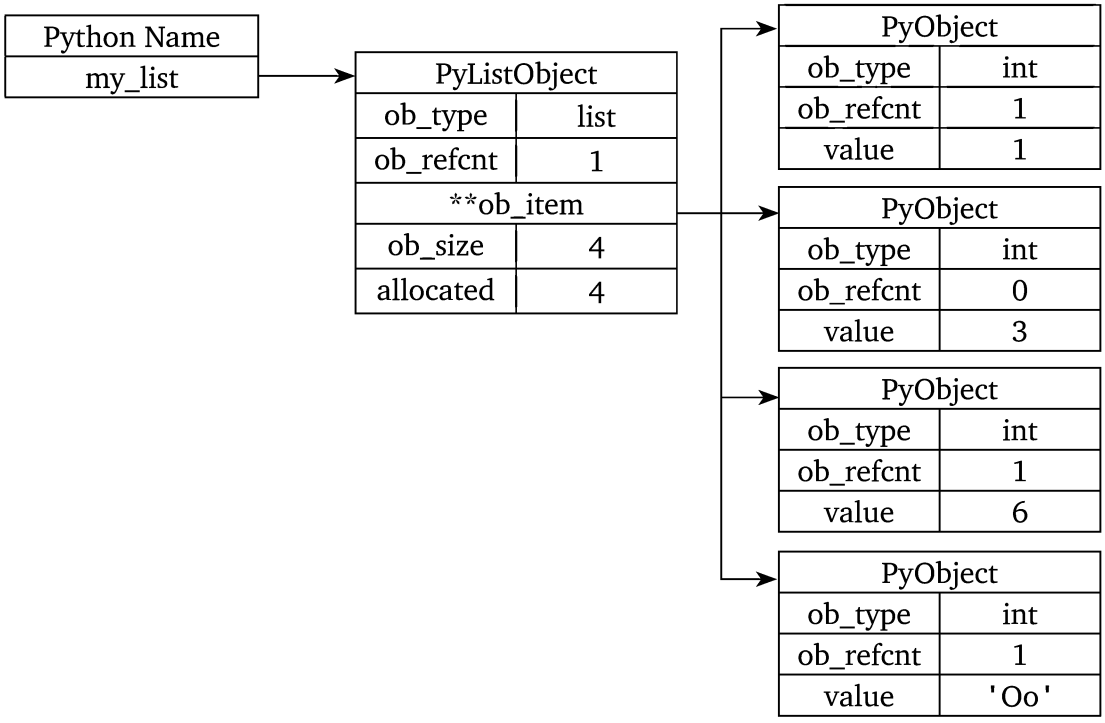

Рис. 1.13. **Добавление элемента в список**

Или изменим значение элемента списка (рис. 1.14):

```
my_list[1]='-_-'
```

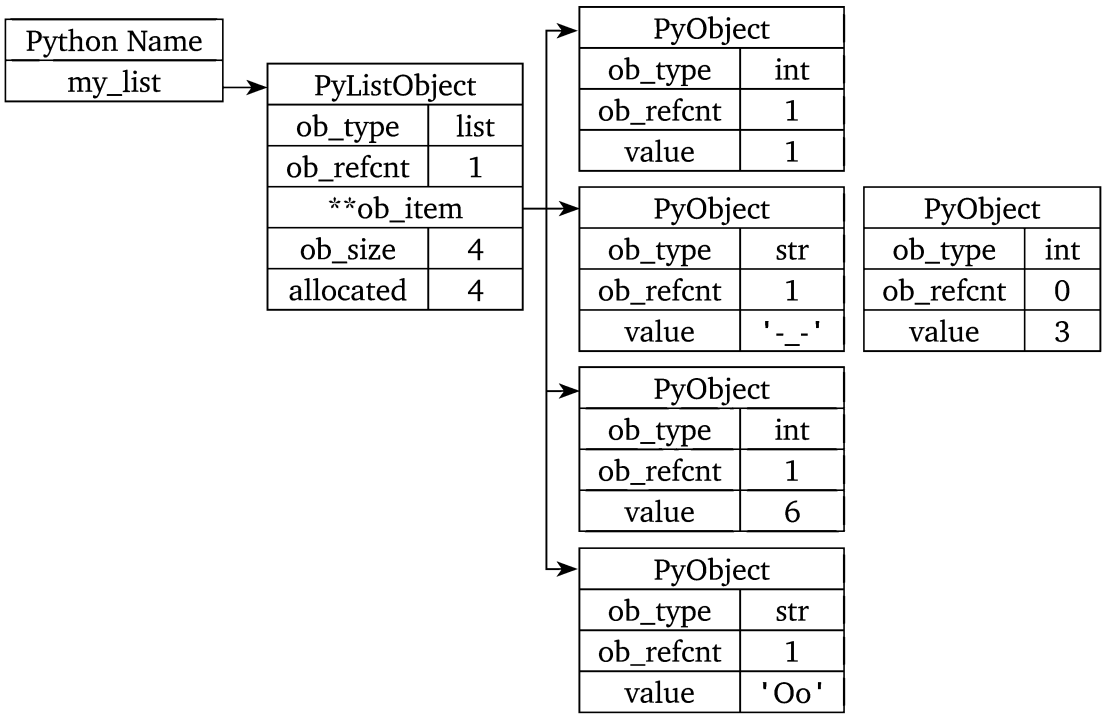

Рис. 1.14. **Изменение значения элемента списка**

При проведении данных операций новый объект не создавался (только на верхнем уровне, на уровне элементов списка создавались новые объекты), и работа производилась с первоначальным PyObject на который ссылается имя *val*. Далее посмотрим, как пове-

дет себя программа при добавлении нового имени и присваивании ему *my_list* (рис. 1.15):

```
bad_list = my_list
```

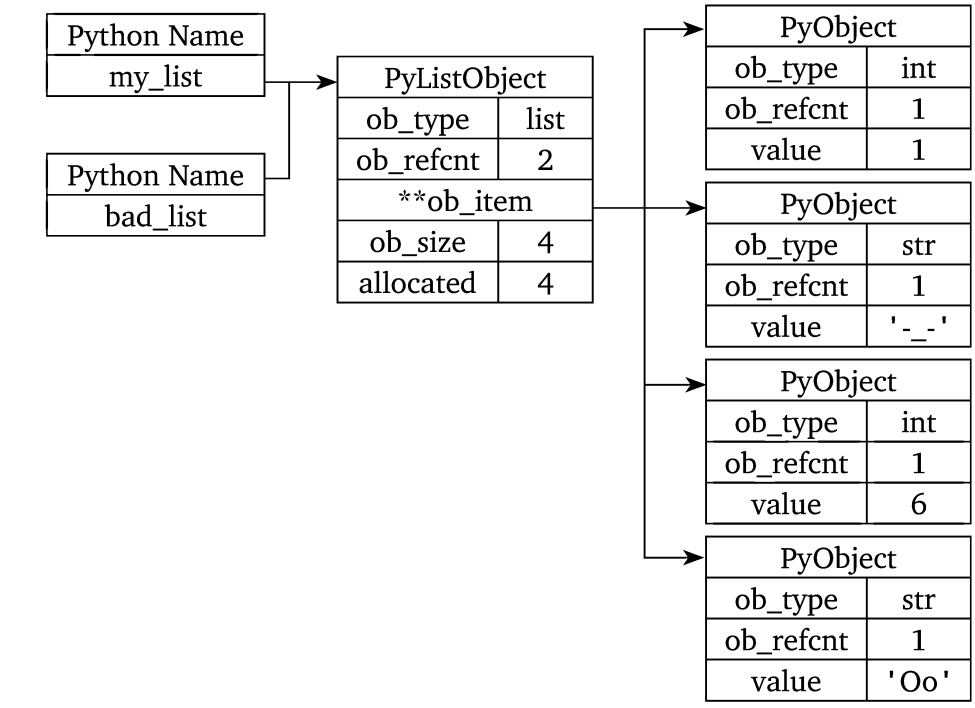

Рис. 1.15. **Связывание нового имени с существующим списком**

Теперь рассмотрим, как изменится область памяти при изменении первого объекта во вновь объявленном списке *bad_list* (рис. 1.16):

```
bad_list[0] = 5
```

или добавлении еще одного элемента (рис. 1.17):

```
bad_list.append(3.6)
```

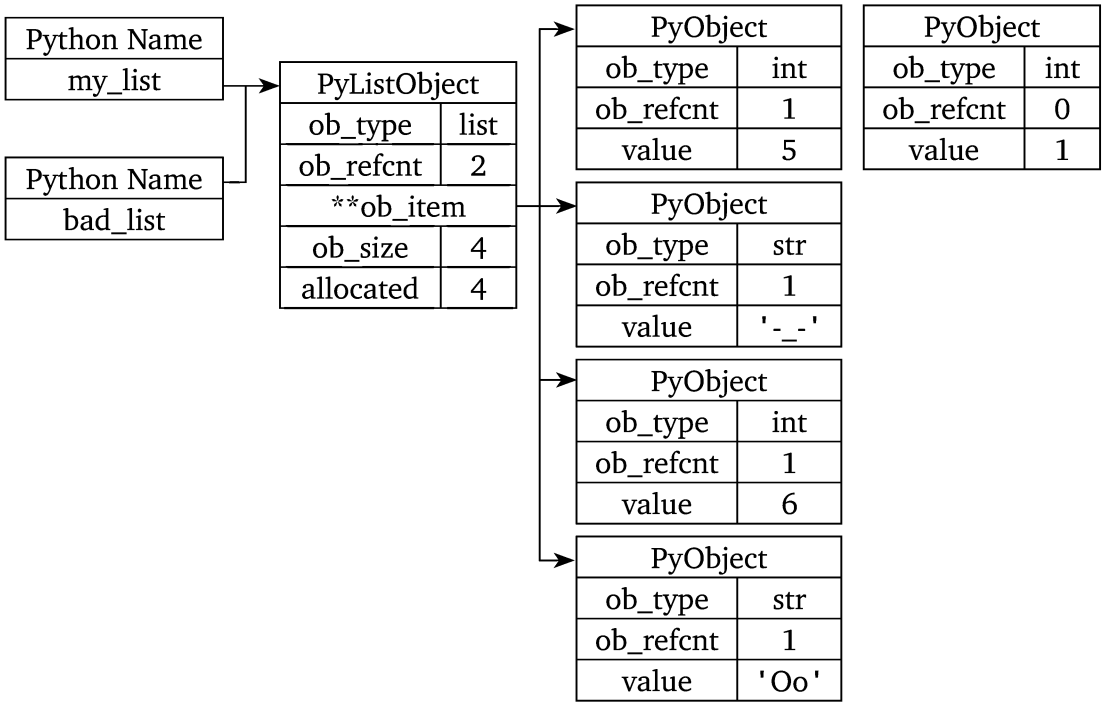

Рис. 1.16. **Изменение значения элемента списка**

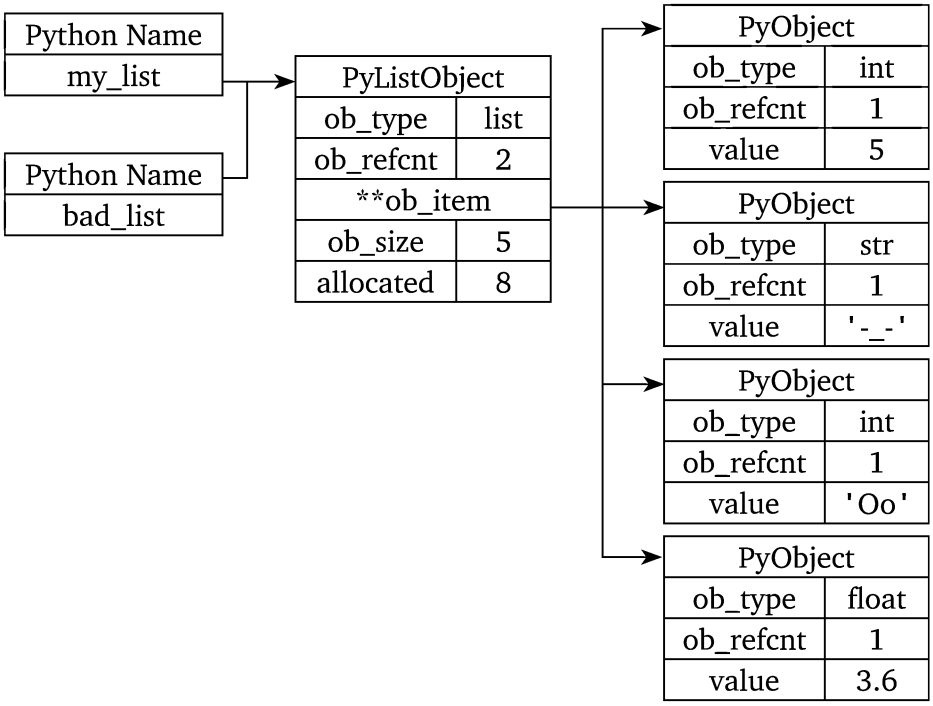

Рис. 1.17. **Добавление элемента в список**

В связи с этой особенностью изменяемых типов данных необходимо с особой осторожностью подходить к операции присваивания, так как имени присваивается ссылка на существующий объект и при изменении элементов это не отображается на верхнем уровне, с которым происходит «связывание» имени. Если есть необходимость работать с копией изменяемого типа данных, то операцию копирования необходимо вызывать явно:

```
# поверхностное копирование
bad_list = my_list.copy()
```

При этом следует учитывать тот факт, что когда объект изменяемого типа хранит (как один из своих элементов) другой объект изменяемого типа, необходимо использовать глубокое, а не поверхностное копирование. Это связано с тем, что при поверхностном копировании сохранится ссылка на вложенный объект, а не его копия:

```
# поверхностное копирование
my_list = [2, 3, 4, [1, 4]]
print(my_list) # [2, 3, 4, [1, 4]]
bad_list = my_list.copy()
bad_list.append(10)
print(bad_list) # [2, 3, 4, [1, 4], 10]
print(my_list) # [2, 3, 4, [1, 4]]
bad_list[3].append('Oo')
print(bad_list) # [2, 3, 4, [1, 4, 'Oo'], 10]
print(my_list) # [2, 3, 4, [1, 4, 'Oo']]

# глубокое копирование
import copy
```

```
my_list = [2, 3, 4, [1, 4]]
print(my_list) # [2, 3, 4, [1, 4]]
bad_list = copy.deepcopy(my_list)
bad_list.append(10)
print(bad_list) # [2, 3, 4, [1, 4], 10]
print(my_list) # [2, 3, 4, [1, 4]]
bad_list[3].append('Oo')
print(bad_list) # [2, 3, 4, [1, 4, 'Oo'], 10]
print(my_list) # [2, 3, 4, [1, 4]]
```

1.4. Интернированные (intern) объекты в Python

Интернированный объект — это объект, копия которого хранится в единичном экземпляре в памяти программы. Так, например, при старте интерпретатора Python в памяти предварительно создается подмножество таких объектов, доступ к которым можно получить через глобальное пространство имен.

В CPython к интернированным объектам относят:

— целые числа в диапазоне от –5 до 256;

— строки (< 20 символов), которые содержат только: ASCII-буквы, цифры или знаки подчеркивания.

Это сделано из-за того, что переменные с такими значениями часто используются при разработке программ. А путем их интернирования Python исключает выделение дополнительной памяти под объекты, которые хранят аналогичное значение. Вместо этого имени переменной присваивается ссылка на уже имеющийся в памяти объект с этим значением.

Давайте разберемся, для чего осуществляется интернирование объектов с некоторыми значениями. Как вам известно, переменная типа int (int x = 15;) в C++ занимает в памяти 4 байта (32 бита). А сколько тогда в байтах будет занимать в памяти следующая запись на CPython (3.7)?

```
import sys

x = 15 # объем в байтах?
print(sys.getsizeof(x)) # 28
y = 15
print(x is y)
# True # x и y ссылаются на один и тот же объект
# объем в байтах пустого объекта?
print(sys.getsizeof(object())) # 16
```

Даже пустой объект в CPython занимает в 4 раза больше объема в памяти, нежели переменная типа int в языке программирования C++. А разница записи *int x = 15*; (C++) с *x = 15* (CPython) 28/4 = 7 раз (размер объекта может варьироваться от реализации Python)!!!

Теперь приведем ряд примеров для более подробного погружения в принципы работы интернирования в Python:

```
x = 320
y = 320
print(f'Xid = {id(x)}; Yid = {id(y)}')
print(x is y)
# Xid = 1754587687760; Yid = 1754587687856
# False
x = 25
y = 24
print(f'Xid = {id(x)}; Yid = {id(y)}')
print(x is y)
# Xid = 140721606730896; Yid = 140721606730864
# False
y += 1
print(f'Xid = {id(x)}; Yid = {id(y)}')
print(x is y)
# Xid = 140721606730896; Yid = 140721606730896
# True
x = 540
y = 539
y += 1
print(f'Xid = {id(x)}; Yid = {id(y)}')
print(x is y)
# Xid = 1754587686992; Yid = 1754587687408
# False
```

Если приведенный выше пример исполняется в IDE Pycharm, а не в Jupyter, то $x = 320$ и $y = 320$ при их проверке на эквивалентность *x is y* вернут *True*. Это связано с тем, что компилятор CPython выполняет оптимизацию, которая помогает по мере возможности экономить память и шаги исполнения кода.

Далее рассмотрим пример интернирования строк:

```
str1 = "Itsfantastic"
str2 = "Itsfantastic"
print(f'S1id = {id(str1)}; S2 = {id(str2)}')
# S1id = 1754574647408; S2 = 1754574647408
print(str1 is str2)# True
```

Но если в строку вставить пробел, одиночные кавычки (' ') или вопросительный знак, то получится другой результат:

```
str1 = "Its fantastic"
str2 = "Its fantastic"
print(f'S1id = {id(str1)}; S2 = {id(str2)}')
# S1id = 1754583817968; S2 = 1754582600688
print(str1 is str2) # False
```

Это различие следует запомнить! Интерируются только строки, которые удовлетворяют следующим условиям: строка состоит только из ASCII-букв, цифр или знаков подчеркивания, а ее длина меньше 20 символов.

Для того, чтобы *str1* и *str2* ссылались на один и тот же интернированный объект, используется функция **intern()** модуля **sys**:

```
import sys
str1 = sys.intern("It's fantastic!")
str2 = sys.intern("It's fantastic!")
print(f'S1id = {id(str1)}; S2 = {id(str2)}')
# S1id = 1754558281968; S2 = 1754558281968
print(str1 is str2) # True
```

В таком случае при интернировании строки обязательно использовать **sys.intern()** для каждого имени переменной, иначе будет создан новый объект, и проверка выдаст следующий результат:

```
import sys
str1 = sys.intern("It's fantastic!")
str2 = "It's fantastic!"
print(f'S1id = {id(str1)}; S2 = {id(str2)}')
# S1id = 1754558281968; S2 = 1754580540080
print(str1 is str2) # False
```

Интернирование строк позволяет слегка повысить производительность при поиске по словарю. Когда ключ в словаре и искомый ключ интернированы, то поиск (сравнение хешированных ключей) осуществляется с помощью сравнения указателей, а не строк.

1.5. Глобальная блокировка интерпретатора

Глобальная блокировка интерпретатора (*Global Interpreter Lock*, GIL) — глобальный механизм интерпретатора CPython, который направлен на решение проблемы, возникающей при работе с разделяемыми ресурсами (например, памятью) [9]. Эта проблема может возникать, когда два потока пытаются одновременно изменить данные в одном и том же ресурсе (файле, объекте и т. д.). Это чревато тем, что в результате ни один из потоков не выполнит как подобает своей работы, а данные в разделяемом ресурсе будут пребывать в «хаосе».

В качестве аналогии можно привести пример с двумя писателями и одной записной книгой. Каждый из писателей делает запись когда захочется, но книга-то у них одна, и заранее не обговорен порядок, в котором каждый будет производить в нее запись. Пока запись производится по очереди, нет ничего страшного. Когда же писатели захотят сделать запись в книгу одновременно, произойдет коллапс! В том, что получится, не удастся разобраться даже самому привередливому критику! Что уже говорить об обычных читателях или самих писателях.

GIL же в приведенном примере выступит в качестве арбитра, который будет решать, какой из писателей получит возможность произвести запись в книгу. Да так, что другой писатель никак не сможет этому помешать или же начать запись в книгу в тот же момент

времени и получит доступ к книге только тогда, когда первый писатель завершит свою запись.

В CPython для достижения этой цели механизм GIL блокирует весь интерпретатор. Таким образом, ничто не в силах помешать работе текущего потока, то есть GIL позволяет работать только одному потоку даже в многопоточном приложении. Это обстоятельство порождает неутихающие споры в Python-сообществе. По своей сути GIL представляет собой мьютекс. А сам расчет блокировки производится в основном цикле построения байткода CPython, для того чтобы установить, какой поток в текущий момент времени должен выполнять инструкции.

Первая попытка улучшения GIL была предпринята в Python 3.2 путем добавления механизма подсчета потоков, которые нуждаются в GIL. Поскольку обычно приоритет отдавался процессорным потокам, это отражалось на работе потоков ввода-вывода, которые не могли получить доступ к GIL. После добавления данного улучшения раз в 5 миллисекунд GIL останавливает текущий поток и дает отработать следующему в очереди потоку (даже если его нет).

Вторая попытка заключается в том, что процессы CPython могут иметь несколько интерпретаторов, а, следовательно, несколько блокировок (у каждого своя). Это позволяет использовать несколько интерпретаторов кода в рамках одного процесса, которые будут иметь свои собственные GIL. На текущий момент времени эта функциональность пребывает в черновом варианте и проходит обсуждения (PEP 554) [10]. Если судить из описания документа, то предварительный анонс данного улучшения намечен в Python 3.9.

А пока приходится использовать следующие обходные пути при работе с GIL, когда он вызывает проблемы:

1) использование процессов вместо потоков. Несмотря на то, что происходит повышение производительности по сравнению с многопоточной версией, на управление процессами также расходуется процессорное время. Это связано с тем, что работа с несколькими процессами более сложна, чем с несколькими потоками;

2) использование альтернативных интерпретаторов Python.

Для демонстрации всего вышесказанного приведем несколько примеров. Сначала сделаем замер по времени выполнения следующего кода в однопоточном исполнении:

```
import time
def countup(N):
    n = 0
    while n < N:
        n += 1

if __name__ == '__main__':
    st_time = time.time()
    countup(30000000)
```

```
    end_time = time.time()

    print(f'Время выполнения: {end_time-st_time}')
# Время выполнения: 1.5710163116455078
```

Далее «разобьем» его на два потока:

```
import time
from threading import Thread
def countup(N):
    n = 0
    while n < N:
        n += 1
if __name__ == '__main__':
    max_for_thread = 30000000//2
    first_thread = Thread(target=countup,
                          args=(max_for_thread,))
    second_thread = Thread(target=countup,
                         args=(max_for_thread,))
    st_time = time.time()
    first_thread.start()
    second_thread.start()
    first_thread.join()
    second_thread.join()
    end_time = time.time()
    print(f'Время выполнения: {end_time-st_time}')
# Время выполнения: 1.5899989604949951
```

Как видно из времени выполнения, между ними практически нет никакой разницы! Теперь попробуем использовать механизм работы с потоками из PySide2 и PyQt5:

```
import time
VER_LIB = 'PyQt'
if VER_LIB == 'PyQt':
    from PyQt5.QtCore import QThread
else:
    from PySide2.QtCore import QThread

class TestQtThread(QThread):
    def __init__(self, n, parent=None):
        super(TestQtThread, self).__init__(parent)
        self.N = n

    def run(self):
        n = 0
        while n < self.N:
            n += 1
if __name__ == '__main__':
    max_for_thread = 30000000//2
    st_time = time.time()
    first_thread = TestQtThread(max_for_thread)
    second_thread = TestQtThread(max_for_thread)
    first_thread.start()
    second_thread.start()
```

```
    first_thread.wait()
    second_thread.wait()
    end_time = time.time()
    print(f'Время выполнения: {end_time - st_time}')
# PyQt5          Время выполнения: 6.59580135345459
# PySide2         Время выполнения: 2.4407031536102295
```

Поскольку и PySide2, и PyQt5 являются сторонними библиотеками, использование их механизма работы с потоками накладывает на программу дополнительные вычислительные затраты. Но сама разница между выходными данными программы поражает. Официальный порт Qt на Python (PySide2) дает куда лучший результат.

А теперь реализуем данный код на основе многопроцессного, а не многопоточного подхода:

```
from multiprocessing import Pool
import time
def countup(N):
    n = 0
    while n < N:
        n += 1
if __name__ == '__main__':
    max_for_process = 30000000//2
    process_pool = Pool(processes=2)
    st_time = time.time()
    first_process = process_pool.apply_async(countup,
                                [max_for_process])
    second_process = process_pool.apply_async(countup,
                                [max_for_process])
    process_pool.close()
    process_pool.join()
    end_time = time.time()
    print(f'Время выполнения: {end_time-st_time}')
# Время выполнения: 0.9137446880340576
```

Все плюсы и минусы каждого из подходов будут более подробно рассматриваться позже. На текущий момент времени просто необходимо запомнить наличие GIL в CPython и то, на какой аспект работы программы он влияет.

1.6. Подходы к сборке мусора в Python

При написании кода на Python, как и в ряде других языков программирования (Java, C# и т. д.), нет необходимости волноваться о сборке мусора (например, неиспользуемых объектов) и самой работе с памятью. Когда объекты становятся более не нужными, Python автоматически начинает освобождение памяти из-под них [11—13].

При этом для работы с объектами, размер которых не превышает 512 байт, в Python используется дополнительный менеджер памяти. Он выделяет, резервирует и очищает блоки памяти, в которых

может храниться множество таких объектов. В связи с этим, когда происходит удаление объекта, объем памяти, которую он ранее занимал, не переходит операционной системе, а остается зарезервированным для новых объектов. Если в одном из выделенных блоков памяти менеджера больше нет объектов, то Python может вернуть этот блок памяти под управление операционной системы. В основном такое происходит при наличии большого числа вре́менных объектов в локальной области видимости какой-либо функции или в скрипте.

Из-за такого механизма управления памятью ситуация, когда программа начинает использовать больше памяти, чем обычно, не всегда означает наличие проблем с утечкой памяти.

Интерпретатор CPython использует сразу два подхода к управлению высвобождением памяти:

1) подсчет ссылок;
2) Garbage Collector (GC).

1.6.1. Алгоритм подсчета ссылок

В Python существует два типа ссылок: сильные и слабые. Алгоритм подсчета ссылок работает только с сильным типом ссылок. При данном подходе объект удаляется, как только счетчик ссылок на него становится равным нулю. Если удаляемый объект содержит ссылки на другие объекты, эти ссылки также удаляются, что может повлечь за собой удаление других объектов. Так как на объект может ссылаться множество имен, то увеличение значения счетчика ссылок происходит при следующих операциях:

1) присваивание «=»;
2) передача аргументов (в метод, функцию и т. д.);
3) вставка нового элемента в последовательность;

Объекты, которые объявляются вне функций, классов и блоков, имеют глобальную область видимости. Их жизненный цикл равен времени жизни программы (процесса), а количество ссылок на такие объекты в процессе работы никогда не падает до нуля.

Объекты, которые объявляются внутри блока (функции, класса), имеют локальную видимость (т. е. они видны только внутри блока). Их жизненный цикл не столь долог. Как только интерпретатор выходит из блока, он уничтожает все ссылки на локальные объекты.

Рассмотрим пример подсчета ссылок на объект в ходе выполнения программы:

```
import sys
one = two = three = object()
print(sys.getrefcount(one)) # 4
```

В данном случае инициализируется пустой объект и связывается с тремя разными именами: *one*, *two* и *three*. Каждое имя связывается с объявленным объектом путем копирования ссылки на него. При

вызове функции, выводящей количество ссылок на данный объект, вместо значения «3» выводится «4». Это связано со вторым правилом увеличения счетчика ссылок, так как одно из имен выступило в качестве аргумента для **sys.getrefcount()**.

Теперь имени *three* присвоим объект с другим значением:

```
three = 4
print(sys.getrefcount(one)) # 3
```

К минусам описываемого алгоритма обычно относят проблемы с циклическими ссылками, блокирование потоков, дополнительные накладные расходы на память и процессор. А плюсом является то, что объекты удаляются сразу, как только они становятся не нужными.

1.6.2. Garbage Collector (GC)

Основная задача GC — бороться с наличием циклических ссылок, которые приводят к тому, что объекты продолжают оставаться в памяти, даже когда становятся недоступны в коде.

Чтобы более подробно разобраться в том, как получаются циклические ссылки, приведем следующий пример:

```
import ctypes
import gc

if __name__ == '__main__':
    # используется для доступа к объектам по их адресу в памяти
    class PyObject(ctypes.Structure):
        _fields_ = [("refcnt", ctypes.c_long)]
    gc.disable()  # выключаем GC

    my_list = []
    my_list.append(my_list)
    my_list_address = id(my_list)
    del my_list

    first_dict = {}
    second_dict = {}
    first_dict['second'] = second_dict
    second_dict['first'] = first_dict
    my_dict_address = id(first_dict)
    del first_dict, second_dict
    # gc.collect() # ручной запуск  сборки мусора
    # проверка счетчика ссылок
    print(f'Кол-во ссылок на список = '
          f'{PyObject.from_address(my_list_address).refcnt}')
# Кол-во ссылок на список = 1
    print(f'Кол-во ссылок на словарь = '
          f'{PyObject.from_address(my_dict_address).refcnt}')
# Кол-во ссылок на словарь = 1
```

При включении GC количество ссылок, выводимых программой, станет равно нулю. Важно понимать, что циклические ссылки получаются только в «контейнерных» структурах данных, таких как класс, список, словарь, кортеж и т. д. Ввиду этого GC не следит за наличием циклических ссылок у неизменяемых объектов, за исключением кортежей (при выполнении ряда условий). Со всем остальным спокойно справляется алгоритм подсчета ссылок.

Garbage Collector разделяет все объекты на 3 поколения (в зависимости от длины их времени жизни). Новые объекты сразу же классифицируются как первое поколение, и если один из них «выживает» в процессе сборки мусора, то он перемещается во второе. Чем старше поколение, тем реже GC выполняет обход хранящихся в нем объектов. Это связано с тем, что у новых объектов более короткий жизненный цикл (временные объекты), чем у тех, которые уже прошли через несколько этапов сборки мусора.

В каждом поколении ведется свой специальный счетчик и имеется свой порог срабатывания, при достижении которого начинается процесс сборки мусора. Если сразу несколько поколений преодолели порог срабатывания, то из них выбирается наиболее старшее. Это сделано из-за того, что старшие поколения также сканируют все предыдущие.

1.6.3. Слабые ссылки

Как уже говорилось ранее, как только счетчик ссылок на объект обращается в нуль, сборщик мусора уничтожает этот объект. Но бывают случаи, когда полезно иметь такую ссылку на объект, которая не удерживает его в памяти дольше, чем необходимо. Примером ее использования может служить кэш.

Таким инструментом в Python являются слабые ссылки на объект (за эту функциональность отвечает модуль *weakref*), которые не увеличивают его счетчик ссылок, то есть не препятствуют уничтожению объекта сборщиком мусора:

```
import weakref
class TestWeakRef:
    def say(self, name="NoName"):
        print(f'Hi,  {name}!')

strong_ref = TestWeakRef()
weak_ref = weakref.ref(strong_ref)
weak_ref().say() # Hi,  NoName!
weak_ref().say('Alex') # Hi,  Alex!
print(weak_ref() is strong_ref)   # True
del strong_ref  # удаляем сильную ссылку
print(weak_ref()) # None
```

Как видно из последних строчек кода, после того, как произошло удаление единственной сильной ссылки на экземпляр класса, он уничтожился, а слабой ссылке присвоилось значение None!

В большинстве случаев при разработке программ хватает коллекций из модуля *wreakref* (WeakKeyDictionary — словарь со слабыми ссылками на ключи, WeakValueDictionary — словарь со слабыми ссылками на значения, WeakSet) и функции **finalize**, избавляющих программиста от необходимости вручную создавать и обрабатывать экземпляры **weakref.ref** [14].

Не всякий объект в Python может быть объектом слабой ссылки. Например, объектом слабой ссылки могут быть подклассы, но не сами экземпляры классов **list** и **dict**:

```
class MyList(list):
    pass

my_list = MyList(range(10))
print(my_list)
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
ref_list = weakref.ref(my_list)
print(ref_list())
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

new_list = list(range(10))
ref_new_list = weakref.ref(new_list)
'''
----------------------
TypeError Traceback (most recent call last) in
      8
      9 new_list = list(range(10))
---> 10 ref_new_list = weakref.ref(new_list)

TypeError: cannot create weak reference to 'list' object
'''
```

Встроенные неизменяемые типы данных также не поддерживают слабые ссылки. Большинство из этих ограничений связаны с реализацией CPython, и к другим интерпретаторам Python, возможно, и не относятся.

Также при объявлении объекта слабой ссылки вторым аргументом **weakref.ref()** можно передать функцию, которая будет вызвана при его финализации:

```
import weakref

class TestWeakRef(object):
    def __del__(self):
        print('Class Deleted')
```

```
    def say(self, name="NoName"):
        print(f'Hi,  {name}!')

def callback(ref):
    """Invoked when referenced object is deleted"""
    print("I'am freee!!!")

strong_ref = TestWeakRef()
weak_ref = weakref.ref(strong_ref, callback)
weak_ref().say() # Hi,  NoName!
weak_ref().say('Alex') # Hi,  Alex!
del strong_ref
# Class Deleted
# I'am freee!!!
print(' weak_ref ():', weak_ref())
# weak_ref(): None
```

Пока не забивайте себе голову вопросом, что же тут творится. Главное — запомнить сам принцип, для чего это используется.

А теперь давайте начнем рассматривать основы Python с того, какие типы данных в этом языке программирования присутствуют и как с ними работать.

1.7. Встроенные типы данных Python

Поскольку в Python все является объектом, то не удивительно, что, в отличие от ряда других языков программирования, помимо пользовательских типов (классов) существуют такие типы данных, как функция, файл, модуль, метод и скомпилированный код.

К стандартным (встроенным) типам Python относят [15]:

1) *none* (аналог nullptr из C++);
2) булевы значения;
3) числа (*Numeric Type*):
— *int* — целое число,
— *float* — число с плавающей точкой,
— *complex* — комплексное число;
4) списки (*Sequence Type*):
— *list* — список,
— *tuple* — кортеж,
— *range* — диапазон;
5) строки (*Text Sequence Type*) — *str*;
6) бинарные списки (*Binary Sequence Types*):
— *bytes* — байты,
— *bytearray* — массивы байтов,
— *memoryview* — специальные объекты для доступа к внутренним данным объекта через protocol buffer;

7) множества (*Set Types*):
— *set* — множество,
— *frozenset* — неизменяемое множество;
8) словари (*Mapping Types*) — *dict*.

Все типы данных в Python делятся на две категории: изменяемые (*mutable*) и неизменяемые (*immutable*).

К неизменяемым типам данных относят: символьные строки, числа, булевы значения, неизменяемые множества и кортежи. Остальные типы данных относятся к изменяемым.

Разница заключается в том, что в случае объявления имени переменной неизменяемого типа ей задается идентификатор, который невозможно изменить. Приведем следующий пример:

```
n = 10
b = n
print('id n =',id(n))
print('id b =',id(b))
```

В данном случае *id* переменных будет одинаковый — 140719951553200, и если изменить значение имени переменной *n*, то будет создан новый объект с новым значением, на который будет ссылаться переменная *n*. Соответственно изменится и ее идентификатор:

```
n = 5
print('id n =',id(n))
print('id b =',id(b))
```

Вывод данного сценария представлен ниже:

```
id n = 140719951553040
id b = 140719951553200
```

В случае работы с изменяемыми типами данных не будет создаваться новый объект, а идентификатор переменной останется без изменения. Рассмотрим данный аспект на примере работы со списком:

```
n = [1, 2]
b = n
print('id n =',id(n))
print('id b =',id(b))
b[0]=5
n[1]=4
print('id n =',id(n))
print('id b =',id(b))
```

Вывод данного сценария представлен ниже:

```
id n = 2216118937864
id b = 2216118937864

id n = 2216118937864
id b = 2216118937864
```

Поскольку в Python динамическая типизация, можно не указывать тип объявляемой переменной. Он выведется в процессе компиляции программы в байт-код, и далее будет выполняться проверка, чтобы операции производились над объектами одного типа (сильная типизация). Начиная с версии 3.6 в Python появилась возможность использовать аннотацию типов. Таким образом, у программистов появилась возможность задавать тип объявляемой переменной (т. е. использовать статическую типизацию, но для проверки соответствия типа переменной с используемой аннотацией необходимо использовать дополнительную библиотеку, например, *mypy* [16]):

```
from typing import List, Dict
n : int = 4
b : float = 3.5
myList : List[int] = [1,3,4]
myDict : Dict[str, float] = {'sd':3.4,'rd1':4.76}
```

Далее будут более подробно рассмотрены такие типы данных, как: строки, список, словарь, кортежи и множества.

1.7.1. Строки

Строковый тип данных применяется для хранения текстовой информации и произвольных значений байтов. Строки относятся к так называемым последовательностям — позиционно упорядоченным коллекциям объектов, в которых объекты располагаются в порядке слева направо, сохраняют свои позиции и могут извлекаться по их относительному номеру.

Таким образом строки представляют собой последовательность из односимвольных строк (также к последовательностям в Python относятся списки и кортежи) и поддерживают операции, которые предполагают наличие позиционного порядка среди элементов: индексация, срез, конкатенация и повторение.

Для примера используем переменную строкового типа *my_str*, инициализировав ее значением «Test»:

```
my_str = 'Test'
# также можно инициализировать переменную через
# двойные кавычки my_str = " Test"
print(len(my_str)) # Длина строки = 4
print(my_str[0]) # T
# Выводим первый элемент my_str, так как индексация
# начинается с нуля
```

Индексация в Python представляет собой смещение относительно первого (нулевого) элемента последовательности и может быть как положительной (от 0 до n – 1), так и отрицательной (от –1 до –n):

```
print(my_str[-1]) # Первый элемент с конца = t
print(my_str[-3]) # Третий элемент с конца = e
```

Помимо простой позиционной индексации, последовательности поддерживают ее более общую форму — срез. Срез представляет собой способ извлечения целой части за один шаг. В общем виде концепцию среза можно представить как *X*[*i*:*j*]. Такая форма записи обозначает, что будут извлечены значения из последовательности X, начиная с позиции *i* до позиции *j*, не включая ее (т. е. *j* – 1):

```
print(my_str[1:3])  # es
# Срез my_str со смещения 1 до 2 включительно (не 3)
```

При использовании среза его левая граница по умолчанию принимается равной нулю, а правая — длине нарезаемой последовательности (то есть длине строки). Это приводит к нескольким распространенным вариантам применения:

```
# Вся строка my_str
print(my_str[:]) # Test
# Всё после первого элемента
print(my_str[1:]) # est
# Кроме последнего элемента
print(my_str[0:3]) # Tes
# То же, что и my_str[0:3]
print(my_str[:3]) # Tes
# Аналогична предыдущей записи
print(my_str[:-1])  # Tes
```

Помимо приведенного формата записи среза, можно указывать еще и его шаг: *X*[*i*:*j*:*n*]. Такая форма записи обозначает, что будут извлечены значения из последовательности *X*, начиная с позиции *i* до позиции *j*, не включая ее (т. е. *j* – 1), с шагом *n*:

```
# От первого до последнего элемента с шагом 1
print(my_str[::1]) # Test
# От первого до последнего элемента с шагом 2
print(my_str[::2]) # Ts
# Аналогична предыдущей записи
print(my_str[0:4:2])  # Ts
#  От второго до последнего элемента с шагом 2
print(my_str[1::2])  # et
# Перевернуть последовательность
print(my_str[::-1])  tseT
```

Конкатенация и повторение последовательностей производятся следующим образом:

```
#Конкатенация (значение my_str не изменяется)
print(my_str+'Ko') # TestKo
# Повторение (значение my_str не изменяется)
print(my_str * 3) # TestTestTest
# Конкатенация
print([1,2,3] + [4,5]) #  [1, 2, 3, 4, 5]
# Повторение
print([1, 2, 3] * 3)  # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Так как строки относятся к неизменяемому типу данных, их нельзя модифицировать, обращаясь по индексу элемента (на месте):

```
my_str[1] = 'E'
'''
...текст сообщения об ошибке...
File "F:/code/python/myProjRep/main.py", line 27, in <module>
my_str[1] = E
TypeError: 'str' object does not support item assignment
'''
```

Но можно создавать новые объекты:

```
my_str = my_str[1:] + 'E'
print(my_str)
estE
```

Имеется возможность модифицировать текстовые данные на месте. Для этого необходимо развернуть их в список индивидуальных символов и объединить вместе с пустым разделителем или использовать для этого тип *bytearray*:

```
my_str = 'Test'
# Развернуть в список: [. . . ]
my_list = list(my_str)
print(my_list) # ['T', 'e', 's', 't']
my_list[0] = 'F'    # Изменить на месте
# Объединить с пустым разделителем
print('' .join(my_list))   # Fest

# Гибрид байтов/списка
my_array = bytearray(b'Test')
my_array.extend(b'Ko')
print(my_array)  # bytearray(b'TestKo')
# Преобразовать в обычную строку
print(my_array.decode())    TestKo
```

Далее рассмотрим методы, которые специфичны для строкового типа данных и не распространяются на последовательности. Данные методы возвращают измененную строку, но не модифицируют ту, над которой производится операция. К ним относят:

— *find*;
— *replace*;
— *split*;
— *upper*;
— *rstrip и т. д.*

Рассмотрим их принцип работы:

```
my_str = 'Test'
# Поиск смещения подстроки в my_str
print(my_str.find('es'))   # 1
# Замена вхождения подстроки в my_str
# другой подстрокой
```

```
print(my_str.replace('es', ' XFQ')) # T XFQt
print(my_str) # Test

line = 'ffff,ttt,nn,d'
# Разбить строку по разделителю ',' в список подстрок
print(line.split(','))
['ffff', 'ttt', 'nn', 'd']
# Преобразовать символы строки в верхний регистр
print(my_str.upper()) # TEST
line = 'ffff,ttt,nn,d\n'
# Удалить пробельные символы с правой стороны
print(line.rstrip())# ffff ,ttt ,nn ,d
```

Методы можно чередовать:

```
# Замена подстроки в строке с последующим её
#разделением в список
print(line.replace('tt', 'XFQ').split(','))
['ffff ', 'XFQt ', 'nn ', 'd\n']
```

Более подробный список методов для работы с встроенными типами данных приведен в [17].

1.7.2. Списки

Списки — позиционно упорядоченные коллекции объектов произвольных типов, не имеющие фиксированного размера. В отличие от строк списки можно модифицировать на месте путем присваивания по индексу или вызовом некоторых списковых методов. Это позволяет использовать списки как довольно гибкий инструмент для представления произвольных коллекций, таких как: перечня продуктов в магазине, учащихся в школе и т. д.

К основным характеристикам списков можно отнести то, что они:

— являются упорядоченными коллекциями произвольных объектов;

— поддерживают доступ по смещению;

— имеют переменную длину, разнородны и допускают произвольно глубокое вложение;

— относятся к категории «изменяемая последовательность»;

— представляют собой массивы ссылок на объекты.

Существует несколько способов создания пустого списка:

```
type1 = []
type2 = list()
```

Если список необходимо инициализировать переменными в момент его объявления, используют следующие конструкции:

```
type1 = ['one', 'two']
type2 = list(['one', 'two'])
```

Добавление в список осуществляется через вызов у него метода **.append(*val*)**, где *val* — добавляемое в конец списка значение:

```
my_list = list(['one', 'two'])
print(my_list) # ['one', 'two']
my_list.append('three')
print(my_list) # ['one', 'two', 'three']
```

При этом не обязательно, чтобы в списке содержались или в него добавлялись данные одного типа:

```
my_list.append(4)
my_list.append(5)
print(my_list) # ['one', 'two', 'three', 4, 5]
```

Казалось бы, при объявлении типа посредством механизма аннотации проблема будет решена, и в случае добавления объекта другого типа произойдет ошибка:

```
my_new_list: List[int] = [4, 5]
```

Но нет:

```
my_new_list.append('add')
print(my_new_list) # [4, 5, 'add']
```

В отличие от C++ или Java ошибка будет выдана не на этапе компиляции, а на этапе выполнения программы, если у элемента списка нет вызываемого разработчиком метода. Например, выполним вычитание из каждого элемента списка по единице:

```
for it in my_new_list:
    it = it-1
```

В таком случае будет выдана ошибка типа:

```
'''
File "F:/code/python/myProjRep/main.py", line 12, in <module>
    it = it-1
TypeError: unsupported operand type(s) for -: 'str' and 'int'
'''
```

Это связано с тем, что типы в Python определяются автоматически во время выполнения, а не при их объявлении в коде. Получается, что разработчик никогда не объявляет переменные заблаговременно (эту концепцию проще уловить, если постоянно помнить о том, что все сводится к переменным, объектам и связям между ними).

Если бы строковый тип поддерживал операцию вычитания, разработчик мог бы так и не узнать о допущенной им ошибке. Давайте реализуем файл со сценарием Python «main.py», предварительно скачав анализатор кода Python (линтер) для статической проверки типов (pip install mypy):

```
from typing import List

def foo(a: str, b: int):
    return len(a) - b
```

```
my_new_list: List[int] = [4, 5]
my_new_list.append('add')
print(my_new_list)
for it in my_new_list:
    it = it-1
print(foo(False, 'sdasd'))
```

Перед запуском кода сценария в терминале IDE (*Integrated Development Environment*) выполним следующую команду:

```
(venv) F:\code\python\myProjRep>mypy main.py
```

Муру выполнит проверку подаваемого ему на вход файла с кодом на соответствие типов объектов и выдаст следующие замечания, которые явно указывают на допущенные ошибки:

```
'''
main.py:7: error: Argument 1 to "append" of "list" has
incompatible type "str"; expected "int"
main.py:11: error: Argument 1 to "foo" has incompatible type
"bool"; expected "str"
main.py:11: error: Argument 2 to "foo" has incompatible type
"str"; expected "int"
Found 3 errors in 1 file (checked 1 source file) '''
```

Конечно, такой подход имеет свои недостатки, но позволяет избежать утомительных поисков в случае несоответствия типов объектов, которые хранятся в списках, словарях, подаются на вход функции или возвращаются ею и т. д.

Помимо метода **append()**, который осуществляет добавление объекта в конец списка, можно использовать метод **.insert(i,val)**. Он осуществляет добавление объекта в список на указанную позицию, где все элементы, следующие за индексом *i*, сдвигаются вправо [18]:

```
names = ['Ivan', 'Maxim', 'Alex']
print(names) # ['Ivan', 'Maxim', 'Alex']
names.insert(1,'Jon')
print(names) # ['Ivan', 'Jon', 'Maxim', 'Alex']
```

Метод **extend(L)** добавляет существующие элементы из списка *L* в правую часть текущего списка:

```
one = [1, 2]
two = [3, 4]
one.extend(b)
print(one) # [1, 2, 3, 4]
```

Удаление элемента списка можно осуществлять по номеру его индекса, а также используя методы **remove()** или **.pop([i])**, где *i* — индекс извлекаемого из списка элемента (если его не задавать, то произойдет извлечение последнего элемента в списке). А метод **.clear()** (равносильно **del my_list[:]**) полностью очищает список:

```
# равносильно names.remove('Ivan') или names.pop(0)
del names[0]
print(names) # ['Jon', 'Maxim', 'Alex']
```

Сортировка элементов списка осуществляется следующим образом:

```
names.sort()
print(names) # ['Alex', 'Jon', 'Maxim']
```

При этом необходимо запомнить, что при сортировке списка с разнотипными элементами может произойти ошибка:

```
new_array = [2, 'abc', 'ttt', '10', 3.6]
new_array.sort()
File "F:/code/python/myProjRep/main.py", line 13, in <module>
    new_array.sort()
TypeError: '<' not supported between instances of 'str' and
'int'
```

Если подать на вход метода **.sort()** параметр *key* со значением, равным функции **str** (приведение элемента списка к строковому типу), то ошибки удастся избежать:

```
new_array.sort(key=str)
print(new_array) # ['10', 2, 3.6, 'abc', 'ttt']
```

Обращение по несуществующему индексу к элементу списка считается ошибкой и приводит к сообщению: ***IndexError: list assignment index out of range***.

Если нужно посчитать, сколько раз по значению повторяется элемент в списке, то необходимо использовать метод **.count(x)**:

```
my_list=[1, 2, 2, 3, 2]
print(my_list.count(2)) # 3
```

Так как список относится к изменяемым типам данных в Python, то при присваивании уже существующего списка новой переменной копирование самого списка не выполняется. Новой переменной будет присвоена ссылка на уже существующий список:

```
my_array = [2, 3, 5, 7, 8, 10]
print(my_array) # [2, 3, 5, 7, 8, 10]
my_new_array = my_array
my_new_array.append(12)
print(my_array) # [2, 3, 5, 7, 8, 10, 12]
```

В тех случаях, когда через новую переменную планируется работать с копией уже существующего списка, во избежание изменений исходного списка необходимо использовать метод **.copy()** (**my_array[:]**, **list(my_array)** и т. д.):

```
my_array = [2, 3, 5, 7, 8, 10]
print(my_array) # [2, 3, 5, 7, 8, 10]
my_new_array = my_array.copy()
my_new_array .append(12)
print(my_array) # [2, 3, 5, 7, 8, 10]
print(my_new_array) # [2, 3, 5, 7, 8, 10, 12]
```

Проверить, содержится ли в списке объект с определенным значением, можно следующим способом:

```
my_array = [2, 3, 5, 7, 8, 10]
print(2 in my_array) # True
print(0 in my_array) # False
```

1.7.3. Словари

Словарь — наиболее гибкий тип данных в Python. Если списки можно рассматривать как упорядоченные коллекции объектов, то словари — неупорядоченные коллекции. Их главное отличие от списков состоит в том, что в словарях элементы сохраняются и извлекаются по ключу, а не по позиционному смещению. По своей сути словари замещают собой записи, поисковые таблицы и объединения иного вида, где имена элементов более содержательны, чем их позиции.

Перечислим основные характеристики словарей. Они:

— поддерживают доступ по ключу, а не по смещению;

— являются неупорядоченными коллекциями произвольных объектов;

— имеют переменную длину, разнородны и допускают произвольно глубокое вложение;

— относятся к категории «изменяемое отображение»;

— представляют собой таблицы ссылок на объекты (хеш-таблицы).

В форме литерального выражения словарь записывается как ограниченная фигурными скобками серия пар «ключ:значение», разделенных запятыми. Словари можно вкладывать в коллекции, записывая один словарь в виде значения внутри другого словаря либо внутри списка или кортежа.

Объявить словарь можно следующим образом:

```
my_dict = {} # Пустой словарь
# Двухэлементный словарь
my_dict = {'name': 'Alex', 'course': 3}
# Вложение
my_dict = {'info': {'name': 'Alex', 'course': 3}}
# Альтернативные способы создания: ключевые слова,
# пары "ключ:значение", упакованные пары
# "ключ:значение", списки ключей
my_dict = dict(name='Alex', course=3)
my_dict = dict([('name', 'Alex'), ('course', 3)])
my_dict = dict(zip(keyslist, valueslist))
my_dict = dict.fromkeys(['name', 'course'])
```

Ниже представлен один из способов добавления элементов в словарь:

```
my_dict = {}
my_dict['name'] = 'Alex'
my_dict['course'] = 3
print(my_dict) # {'name': 'Alex', 'course': 3}
```

Вывод значения по его ключу осуществляется следующим образом:

```
print(my_dict['name']) # Alex
```

При обращении к значению элемента по его ключу стоит быть осторожным, так как элемент может отсутствовать в словаре:

```
print(my_dict['info'])
'''
File "F:/code/python/myProjRep/main.py", line 9, in <module>
    print(my_dict['info'])
KeyError: 'info'
'''
```

Перед таким обращением к элементу можно проверить, существует ли запись с данным ключом в словаре:

```
print('info' in my_dict) # False
print('name' in my_dict) # True
```

Проверка на основе оператора включения *in* избавит от ошибки в процессе выполнения программы, но разработчику в таком случае необходимо постоянно проводить проверки. Обойти эту проблему можно, используя метод **.setdefault(key, def_val)** или **.get(key, def_val)**:

```
print(my_dict.get('name', 'Maxim')) # Alex
print(my_dict.get('info', 'no')) # no
my_dict.setdefault('info', 'default')
print(my_dict['info']) # default
my_dict.setdefault('name', 'default')
print(my_dict['name']) # Alex
```

Если в словаре нет элемента с данным ключом, метод **.setdefault(key, def_val)** добавит в словарь пару «ключ:значение по умолчанию», в ином случае никаких операций со словарем производиться не будет. Метод **.get(key, def_val)** в случае отсутствия ключа в словаре вернет значение по умолчанию, иначе будет возвращено значение, хранящееся по заданному ключу.

Бывают моменты, когда требуется получить список ключей, значений, или кортеж пар «ключ:значение» словаря. Для этого используются следующие методы:

```
my_dict = {'name': 'Alex', 'course': 3, 'info':
           {'age': 21, 'country': 'rus'}}
# Вывести все ключи
print(list(my_dict.keys()))
# ['name', 'course', 'info']
# Вывести все значения
print(list(my_dict.values()))
# ['Alex', 3, {'age': 21, 'country': 'rus'}]
# Кортежи пар "ключ:значение"
print(list(my_dict.items()))
# [('name', 'Alex'), ('course', 3), ('info', {'age':
#  21, 'country': 'rus'})]
```

Копировать словарь можно следующими способами:

```
my_dict ={'name': 'Alex', 'course': 3}
my_new_dict = my_dict.copy() # my_new_dict = dict(my_dict)
my_new_dict['info'] = 'None'
print(my_dict) # {'name': 'Alex', 'course': 3}
print(my_new_dict)
# {'name': 'Alex', 'course': 3, 'info': 'None'}
```

Приведенным выше способом осуществляется поверхностное копирование, а это значит, что при наличии в виде значения у словаря списка, вложенного словаря или изменяемых типов будут скопированы ссылки на эти объекты. Таким образом, при их изменении изменятся и значения у объектов в исходном словаре:

```
my_dict = {'name': 'Alex', 'course': 3, 'info':
           {'age': 21, 'country': 'rus'}}
print(my_dict)
# {'name': 'Alex', 'course': 3, 'info': {'age': 21,
#  'country': 'rus'}}
#my_new_dict = dict(my_dict)
my_new_dict = my_dict.copy()
my_new_dict['info']['age'] = 1
print(my_dict)
# {'name': 'Alex', 'course': 3, 'info': {'age': 1,
#  'country': 'rus'}}
```

Чтобы избежать данных проблем, следует использовать глубокое копирование:

```
import copy
my_dict = {'name': 'Alex', 'course': 3, 'info':
           {'age': 21, 'country': 'rus'}}
my_new_dict = copy.deepcopy(my_dict)
my_new_dict['info']['age'] = 1
print(my_dict)
#{'name': 'Alex', 'course': 3, 'info': {'age': 21,
# 'country': 'rus'}}
print(my_new_dict)
#{'name': 'Alex', 'course': 3, 'info': {'age': 1,
# 'country': 'rus'}}
```

Удаление или извлечение пар «ключ:значение» в словаре осуществляется следующими способами:

```
my_dict ={'name': 'Alex', 'course': 3, 'test1': 10, 'test2':
'prob', 'test3': 3.56}
print(my_dict)
#{'name': 'Alex', 'course': 3, 'test1': 10, 'test2':
# 'prob', 'test3': 3.56}
# если ключа 'test3' нет, словарь останется
# без изменения и выведется 10
print(my_dict.pop('test3', 10)) # 3.56
print(my_dict)
# {'name': 'Alex', 'course': 3, 'test1': 10,
```

```
#  'test2':'prob'}
print(my_dict.pop('test3', 10)) # 10
# удаление/возвращение любой пары (ключ:значение)
print(my_dict.popitem()) # ('test2', 'prob')
del my_dict['name'] #Удаление элементов по ключу
print(my_dict) # {'course': 3, 'test1': 10}
```

Для объединения двух словарей по ключам используется следующий подход:

```
my_dict ={'name': 'Alex', 'course': 3}
my_new_dict = { 'test1': 10, 'test2': 'prob',
            'test3': 3.56}
my_dict.update(my_new_dict)
print(my_dict)
#{'name': 'Alex', 'course': 3, 'test1': 10, 'test2':
# 'prob', 'test3': 3.56}
```

Бывают случаи, когда ключи в одном и другом словаре совпадают. Тогда произойдет замена значений по ключам у того словаря, которому вызывается метод **.update()**:

```
my_dict ={'name': 'Alex', 'course': 3}
my_new_dict = { 'test1': 10, 'name': 'prob',
                'test3': 3.56}
my_dict.update(my_new_dict)
print(my_dict)
#{'name': 'prob', 'course': 3, 'test1': 10,
# 'test3': 3.56}
```

Ниже приведено несколько дополнительных указаний и памяток по работе со словарями:

1) операции над последовательностями не работают, так как словари ими не являются. Поэтому понятие порядка среди их элементов отсутствует, а операции вроде конкатенации (упорядоченное объединение) и среза (извлечение непрерывной секции) попросту неприменимы;

2) присваивание по новым индексам добавляет элементы;

3) ключи не обязаны всегда быть строками. До сих пор в примерах в качестве ключей использовались строки, но допускается применять любые неизменяемые объекты. Например, целые числа. Это сделает словарь очень похожим на список (во всяком случае, при индексировании). Кортежи также могут применяться как ключи словаря, позволяя составным значениям ключей, таким как даты и IP-адреса, иметь ассоциированные значения. Ключами могут выступать объекты классов, определяемых разработчиком, при наличии в них подходящих протокольных методов. Грубо говоря, необходимо сообщить Python о том, что их значения являются «хешируемыми» и потому изменяться не будут, так как иначе они оказались бы бесполезными как фиксированные ключи. Изменяемые

объекты вроде списков, множеств и других словарей не могут быть ключами, но разрешены в качестве значений.

1.7.4. Кортежи

Кортежи (*tuples*) — неизменяемые последовательности. Кортеж можно создать с использованием синтаксиса литерала, заключив его компоненты в круглые скобки и разделив их запятыми. Также для создания кортежа можно использовать класс **tuple**, которому для построения нового кортежа в конструктор класса подается последовательность (список):

```
my_tuple = ('Alex', 'Jon')
my_tuple = tuple(['Alex', 'Jon'])
my_tuple = 'Alex', 'Jon'
my_tuple = ()
```

При создании кортежей необходимо запомнить следующее правило: *если в круглые скобки заключен один элемент, то Python рассматривает их как обычные, а если заключено несколько элементов, разделенных запятыми, Python рассматривает их как кортеж*:

```
my_tuple = (2+3)*2
print(type(my_tuple)) # <class 'int'>
my_tuple = (2, 3)
print(type(my_tuple)) # <class 'tuple'>
```

К основным характеристикам кортежей можно отнести то, что они:

— являются упорядоченными коллекциями произвольных объектов;

— поддерживают доступ по смещению;

— относятся к категории «неизменяемая последовательность»;

— имеют фиксированную длину, разнородны и допускают произвольно глубокое вложение;

— представляют собой массивы ссылок на объекты.

Как и рассмотренные ранее списки, кортежи (как последовательности) поддерживают ряд общих методов работы с ними:

— индекс, индекс индекса, срез, длина (my_tuple[i], my_tuple[i][j], my_tuple[0:n], len(my_tuple));

— конкатенация, повторение (my_tuple + new_my_tuple, my_tuple*3);

— итерация (for it in my_tuple: print (x));

— членство ('ff' in my_tuple);

— подсчет (my_tuple.count(x)) и т. д.

В тех случаях, когда необходимо изменить значения объектов, находящихся в кортеже, применяют следующий подход. На основе существующего кортежа создается список, содержащий в себе объекты, хранимые в кортеже. Далее производится работа со списком

и на последнем шаге из данного списка создается новый кортеж, и ссылка на него присваивается переменной, через чье имя осуществляется работа с исходным кортежем:

```
my_tuple = 2, 3, 4, 5
print(my_tuple) # (2, 3, 4, 5)
my_list = list(my_tuple)
print(my_list) # [2, 3, 4, 5]
my_list = list(map(lambda x: x*2, my_list))
print(my_list) # [4, 6, 8, 10]
my_tuple = tuple(my_list)
print(my_tuple) # (4, 6, 8, 10)
```

Если попытаться изменить значение элемента кортежа, то в процессе работы программы будет выведена следующая ошибка:

```
my_tuple[0] = 3
''' File "F:/code/python/myProjRep/main.py", line 10,
in <module>
    my_tuple[0] = 3
TypeError: 'tuple' object does not support item assignment '''
```

Зачем же применять кортежи, если есть списки? Дело в том, что неизменяемость кортежей обеспечивает определенную степень целостности. То есть разработчик может быть уверен, что кортеж не изменится через ссылку где-то в другом месте программы. Для списков же такая гарантия отсутствует.

Поскольку номера полей в кортеже несут меньше осмысленной информации, чем имена ключей в словаре, то можно использовать именованные кортежи, лишенные данного недостатка. Для этого необходимо из библиотечного модуля **collections** импортировать класс **namedtuple**:

```
from collections import namedtuple
ItemRec = namedtuple('ItemRec', ['name',
                     'age', 'jobs'])

my_tuple = ItemRec('Alex', age=21,
                   jobs=['student', 'developer'])
print(my_tuple)
# ItemRec(name='Alex', age=21,
# jobs=['student', 'developer'])
print(my_tuple[0], my_tuple[2])
# Alex ['student', 'developer']
print(my_tuple.name, my_tuple.jobs)
# Alex ['student', 'developer']

     my_dict = my_tuple._asdict()
print(my_dict['name'], my_dict['jobs'])
# Alex ['student', 'developer']
print(my_dict)
#OrderedDict([('name', 'Alex'), ('age', 21),
#             ('jobs', ['student', 'developer'])])
```

1.7.5. Файлы

Файлы считаются основным типом, потому что создаются встроенной функцией **open()**, но отличаются от чисел и последовательностей, а также не реагируют на операции выражений. Файлы предоставляют только методы для решения распространенных задач их обработки. Большинство файловых методов занимаются выполнением ввода из внешнего файла и вывода в файл, ассоциированный с файловым объектом. Остальные методы позволяют переходить в новую позицию внутри файла, сбрасывать буферы вывода и т. д.

Перечислим некоторые особенности работы с файлами в Python:

— файловые итераторы лучше всего подходят для чтения строк;
— содержимое является строками, а не объектами;
— файлы буферизируются и поддерживают позиционирование;
— вызов метода **close()** часто необязателен: файл автоматически закрывается при сборке мусора.

Для открытия файла необходимо использовать встроенную функцию **open()** с указанием пути и имени внешнего файла, а также режима обработки. Вызов возвращает файловый объект, имеющий методы для передачи данных:

```
my_file = open (file_path_and_name, access_mode)
```

В качестве входного параметра *access_mode* можно указать один из следующих режимов работы [19], приведенных в табл. 1.1.

Таблица 1.1

Список режимов открытия файла

Режим доступа	Описание
r	Только для чтения
w	Только для записи, или создается новый файл в случае его отсутствия в каталоге
rb	Только для чтения (бинарный)
wb	Только для записи (бинарный или создается новый файл в случае его отсутствия в каталоге)
r+	Для чтения и записи, которые производятся с начала файла
rb+	Для чтения и записи (бинарный)
w+	Для чтения и записи, или создается новый файл в случае его отсутствия в каталоге. Поток работы с файлом будет установлен в его начало
wb+	Для чтения и записи (бинарный), или создается новый файл в случае его отсутствия в каталоге

Окончание табл. 1.1

Режим доступа	Описание
a	Откроет для добавления нового содержимого или создаст новый файл в случае его отсутствия в каталоге. Добавление будет осуществляться в конец открываемого файла
a+	Откроет файл для чтения или добавления нового содержимого или создаст новый файл в случае его отсутствия в каталоге. Поток работы с файлом будет начинаться с конца открываемого файла
ab	Откроет для добавления нового содержимого (бинарный) или создаст новый файл в случае его отсутствия в каталоге. Добавление будет осуществляться в конец открываемого файла
ab+	Откроет файл для чтения или добавления нового содержимого (бинарный) или создаст новый файл в случае его отсутствия в каталоге. Поток работы с файлом будет начинаться с конца открываемого файла

В табл. 1.2 приведены основные методы для работы с файлами.

Таблица 1.2

Распространенные файловые операции

Метод	Описание
close()	Вручную закрывает файл (это делается автоматически, когда файловый объект подвергается сборке мусора)
fileno()	Возвращает целочисленный дескриптор файла
flush()	Сбрасывает буфер вывода на диск, не закрывая файл
isatty()	Возвращает True, если файл привязан к терминалу
next()	Возвращает следующую строку файла
read()	Читает целый файл в одиночную строку
read(n)	Чтение первые *n* символов файла
readline()	Читает следующую строку файла (включая символ новой строки \n) в строку
readlines()	Читает целый файл в список строк (с символами \n)
seek(N)	Изменяет позицию на N для следующей операции
tell()	Возвращает текущую позицию в файле относительно его начала
write(str)	Записывает строку str в файл
writelines(lines)	Записывает все строки из списка в файл

Рассмотрим простой пример, демонстрирующий основные принципы работы с файлами:

```
myfile = open('myfile.txt', 'w')
myfile.write('Пятнадцать человек на сундук
              мертвеца.\n')
myfile.write('Йо-хо-хо, и бутылка рому!\n')
myfile.close()
myfile = open('myfile.txt', 'r')
print(myfile.readline())
# Пятнадцать человек на сундук мертвеца.
print(myfile.readline())
# Йо-хо-хо, и бутылка рому!
print(myfile.readline())
# Пустая строка: конец файла
# Чтение сразу всего файла
print(open('myfile.txt', 'r').read())
# Пятнадцать человек на сундук мертвеца.
# Йо-хо-хо, и бутылка рому!
```

Для построчной работы с файлом применяется подход, в котором временный файловый объект, созданный функцией **open**, будет автоматически читать и возвращать одну строку на каждой итерации цикла:

```
myfile = open('myfile.txt', 'r')
for line in myfile:
    print(line, end='')
# Пятнадцать человек на сундук мертвеца.
# Йо-хо-хо, и бутылка рому!
```

Теперь разберем процесс записи разнообразных объектов Python в текстовый файл. Здесь следует обращать внимание, что объекты должны быть преобразованы в строки с применением инструментов преобразования. Зачастую данные файла являются строками, и методы записи не делают никакого их автоматического форматирования:

```
a, b, c = 43, 23, 45
mytext = 'Test'
mydict = { 'a' : 1, 'b' : 2}
mylist = [1, 2, 3]
myfile = open('datafile.txt' , 'w')
myfile.write(mytext  + '\n')
# Преобразование чисел в строки
myfile.write('%s,%s,%s\n' % (a, b, c))
# Преобразование и разделение посредством $
myfile.write(str(mylist ) + '$' + str(mydict) + '\n')
myfile.close()
```

После того, как файл создан, мы можем просмотреть его содержимое, открыв и прочитав в строковый объект:

```
print(open('datafile.txt', 'r').read())
# Test
# 43,23,45
# [1, 2, 3]${'a': 1, 'b': 2}
```

Для полноценного считывания данных из файла нужно выполнить следующие шаги:

```
myfile = open('datafile.txt', 'r')
mytext = myfile.readline()
# Удаление символа конца строки
mytext = mytext.rstrip()
print(mytext) # Test
my_numbers_str = myfile.readline()
# Разбиение (разбор) по запятым
my_numbers_str = my_numbers_str.split(',')
# Преобразование всего списка за раз
my_numbers = [int(it) for it in my_numbers_str]
#int(P) - приведение элемента списка к типу int
print(my_numbers) # [43, 23, 45]
```

Чтобы преобразовать список и словарь, сохраненные в третьей строке файла, можно прогнать их через **eval()** — встроенную функцию, которая трактует строку как порцию исполняемого программного кода (формально как строку, содержащую выражение Python):

```
mydict_and_list = myfile.readline().split('$')
print(mydict_and_list)
# ['[1, 2, 3]', "{'a': 1, 'b': 2}\n"]
print(eval(mydict_and_list[1])) # {'a': 1, 'b': 2}
objects = [eval(P) for P in mydict_and_list]
print(objects) # [[1, 2, 3], {'a': 1, 'b': 2}]
```

Поскольку конечным результатом всего разбора и преобразования является список объектов Python, а не строк, теперь к ним можно применять списковые и словарные операции.

Применение метода **eval()** к строкам файла не является безопасным, так как он выполнит любое выражение Python, подаваемое на вход. Например, это может быть код, удаляющий все файлы в каталоге. К еще одному небезопасному способу загрузки объектов Python из файла относится стандартный библиотечный модуль **pickle**. Этот модуль позволяет сохранять почти любой объект Python в файл напрямую, не требуя каких-либо преобразований в строку. В то же самое время нужно быть внимательным при извлечении данных из файла, так как там может содержаться вредоносный код, который выполнится на вашем компьютере:

```
import pickle
mydict = { 'a' : 1, 'b' : 2}
myfile = open('datafile.pkl', 'wb')
pickle.dump(mydict, myfile)
myfile.close()
```

Ниже приведен код для считывания словаря из файла:

```
import pickle
myfile = open('datafile.pkl', 'rb')
mydict = pickle.load(myfile)
print(mydict) # {'a': 1, 'b': 2}
```

Модуль **pickle** выполняет сериализацию объектов (преобразование объектов в строки байтов и обратно) и требует совсем незначительного объема работы со стороны разработчика. Более подробно про данный модуль и работу с ним можно прочитать в [20].

В случае работы с файлом из ненадежных источников рекомендуется использовать более безопасные форматы сериализации, такие как JSON [21, 22]:

```
import json
name = dict(first='Ivan', last='Ivanov')
my_dict = dict(name=name, job=['developer', 'student'],
age=21.5)
print(my_dict)
# {'name': {'first': 'Ivan', 'last': 'Ivanov'},
#  'job': ['developer', 'student'], 'age': 21.5}
json.dump(my_dict, fp=open('testjson.txt', 'w'),
          indent=4)
"""print(open ('testjson.txt').read())
{
    "name": {
        "first": "Ivan",
        "last": "Ivanov"
    },
    "job": [
        "developer",
        "student"
    ],
    "age": 21.5
} """
my_json_obj = json.load(open('testjson.txt'))
print(my_json_obj)
# {'name': {'first': 'Ivan', 'last': 'Ivanov'},
#  'job': ['developer', 'student'], 'age': 21.5}
```

Кроме того, в инструментальном наборе Python доступны дополнительные средства для работы с внешними файлами, это:

— стандартные потоки данных;
— дескрипторные файлы в модуле **os**;
— сокеты, конвейеры и очереди FIFO;
— файлы с доступом по ключу, поддерживаемые модулем **shelve**;
— потоки данных командной оболочки.

1.7.6. Множества

Множество — неупорядоченная совокупность объектов, в которой не может быть дубликатов. Как и кортеж, множество можно создать на базе списка или других последовательностей, элементы которых можно перебирать. Однако, в отличие от списков и кортежей, для множеств неважен порядок элементов. Множества часто используются для двух целей: для удаления дубликатов и для проверки принадлежности. Так как механизм поиска основан на оптимизированной функции хеширования, реализованной для слова-

рей, операция поиска занимает очень мало времени даже для очень больших множеств.

Объявить множество можно следующим способом:

```
mylist = [0, 1, 1, 2, 3, 9, 4, 5, 6, 6, 7, 8, 9]
my_set = set() # пустое множество
my_set = set(mylist)
my_set = {0, 1, 1, 2, 3, 9, 4, 5, 6, 6, 7, 8, 9}
print(my_set) # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

При создании множества все дубликаты удаляются, что видно из приведенного выше кода. Для добавления элементов используют следующие методы:

```
my_set.add(102)
print(my_set) # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 102}
my_set.update([2, 100, 99, 4, 5, 6])
# или my_set.update({2, 100, 99, 4, 5, 6})
print(my_set)
# {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 102, 100, 99}
```

Метод **update()** принимает на вход последовательность (список итерируемых неизменяемых объектов) и добавляет все элементы к исходному множеству. Как можно заметить из результата выполнения кода, повторяющиеся значения, добавляемые в множество, игнорируются.

Удаление элементов из множества производится следующим образом:

```
my_set.remove(2)
print(my_set)
# {0, 1, 3, 4, 5, 6, 7, 8, 9, 102, 100, 99}
my_set.discard(100)
print(my_set)
# {0, 1, 3, 4, 5, 6, 7, 8, 9, 102, 99}
```

При этом следует учитывать следующие моменты поведения программы при удалении элементов из множества: в случае удаления элемента, которого нет в множестве, метод **.discard()**, в отличие от **.remove()**, не выдаст ошибки, и программа продолжит работать дальше. Для примера повторим те же самые операции удаления элементов из множества:

```
print(my_set)# {0, 1, 3, 4, 5, 6, 7, 8, 9, 102, 99}
my_set.discard(100)
print(my_set) # {0, 1, 3, 4, 5, 6, 7, 8, 9, 102, 99}
my_set.remove(2)
print(my_set)
File "F:/code/python/myProjRep/main.py", line 13, in <module>
    my_set.remove(2)
KeyError: 2
```

Извлечение элемента из множества можно также производить, используя метод .pop(), но следует учитывать, что при применении данного метода к пустому множеству произойдет ошибка.

Python поддерживает обычные математические операции над множествами [23, 24]:

Таблица 1.3

Операции над множествами

Операция	Описание
`A | B` `A.union(B)`	Возвращает объединение множеств A и B
`A |= B` `A.update(B)`	Добавляет в множество A все элементы из множества B
`A & B` `A.intersection(B)`	Возвращает пересечение множеств A и B
`A &= B` `A.intersection_update(B)`	Оставляет в множестве A только те элементы, которые есть в множестве B
`A - B` `A.difference(B)`	Возвращает разность множеств A и B
`A-=B` `A.difference_update(B)`	Удаляет из множества A все элементы, не входящие в множество B
`A^B` `A.symmetric_difference(B)`	Возвращает симметрическую разность множеств A и B
`A^=B` `A.symmetric_difference_update(B)`	Записывает в A симметрическую разность множеств A и B
`A <= B` `A.issubset(B)`	Возвращает True, естли A является подмножеством B
`A >= B` `A.issuperset(B)`	Возвращает True, если A является надмножеством B
`A < B`	Эквивалентно A <= B and A != B
`A > B`	Эквивалентно A >= B and A != B

На рис. 1.18 приведены наглядные примеры операций над множествами:

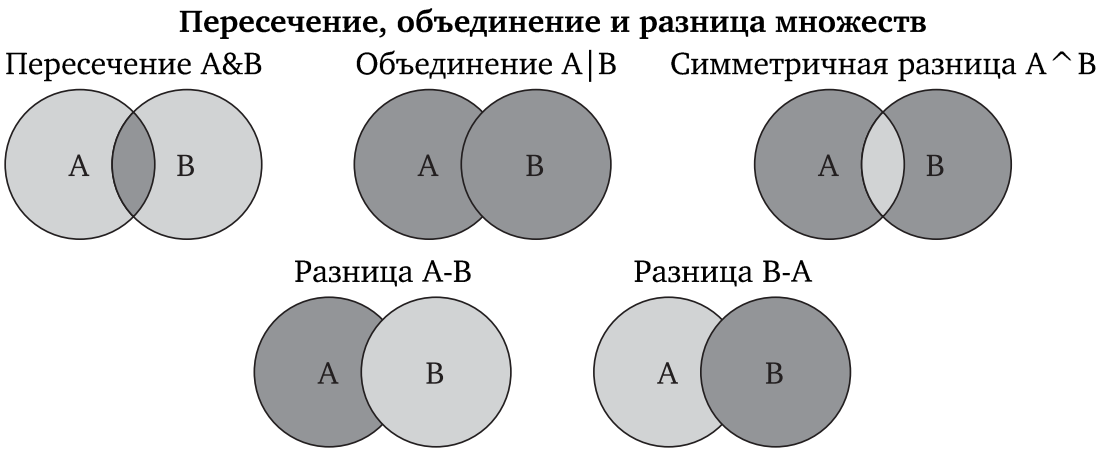

Рис. 1.18. **Операции над множествами**

Объявим несколько множеств и произведем над ними приведенные на рис. 1.18 операции:

```
A = {0, 1, 1, 2, 3, 9, 4, 5, 6, 6, 7, 8, 9}
B = {1, 3, 6, 10, 15, 21, 28, 36, 45}
# объединение
new_set = A | B # A.union(B)
print(new_set)
# {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 21,
#  28, 36, 45}
# пересечение
new_set = A & B # A.intersection(B)
print(new_set) # {1, 3, 6}
# симметрическая разность
new_set = A ^ B # A.symmetric_difference(B)
print(new_set)
# {0, 2, 36, 4, 5, 7, 8, 10, 9, 45, 15, 21, 28}
# разность A - B
new_set = A - B # A.difference(B)
print(new_set) # {0, 2, 4, 5, 7, 8, 9}
# разность B - A
new_set = B - A # B.difference(A)
print(new_set) # {36, 10, 45, 15, 21, 28}
```

При этом три из приведенных операций симметричны:

```
# объединение
check = (A | B) == (B | A)
print(check) # True
# пересечение
check = (A & B) == (B & A)
print(check) # True
# симметрическая разность
check = (A ^ B) == (B ^ A)
print(check) # True
# разность
check = (A - B) == (B - A)
print(check) # False
```

Проверка на то, является ли множество A подмножеством множества B, и наоборот (множество B является надмножеством A), выполняется следующим способом:

```
A = {1, 2, 3}
B = {1, 2, 3, 4}
# A является подмножеством B?
print(A.issubset(B)) # True
# B является надмножеством A?
print(B.issuperset(A)) # True
A.add(5)
print(A.issubset(B)) # False
print(B.issuperset(A)) # False
```

Проверка на то, входит ли элемент в множество, осуществляется посредством использования оператора *in* (*not in*):

```
B = {1, 2, 3, 4}
print(0 in B) # False
print(1 in B) # True
print(0 not in B) # True
print(1 not in B) # False
```

Так как множества должны вычислять хеш-код для каждого элемента, в них могут храниться только хешируемые элементы. В языке Python изменяемые элементы не являются хешируемыми. Это означает, что хешировать список или словарь невозможно. А для хеширования пользовательских классов необходимо реализовывать методы **__hash__** и **__eq__**.

Единственным отличием типа **set** (множество) от **frozenset** (неизменяемое множество) является то, что frozenset — неизменяемый тип данных.

Резюме

В первой теме учебного пособия, помимо краткой истории языка программирования Python, были рассмотрены его основные особенности на примере интерпретатора CPython (3.7), типы данных и способы работы с ними. Этого должно хватить для базового представления о возможностях Python.

Приведенные способы работы со списками, словарями и т. д. постоянно будут встречаться в ходе работы над реальными проектами, так что лучше их изучить в самом начале пути и не заглядывать каждый раз в справочник для уточнения по методам и способам работы с ними.

Вопросы и задания для самопроверки

1. Почему в Python нет переменных?
2. Для чего используется счетчик ссылок?
3. В чем разница между сильными и слабыми ссылками?
4. Что такое интернированные объекты?
5. Как можно интернировать строку, содержащую не ASCII-символы?
6. Что такое GIL? В чем его особенность?
7. Как работает алгоритм подсчета ссылок?
8. Опишите принцип работы Garbage Collector.
9. Зачем используется подход с поколениями при работе Garbage Collector?
10. Для чего можно использовать слабые ссылки?
11. Назовите четыре основных типа данных Python.
12. Что означает «неизменяемость» и какие три основных типа Python считаются неизменяемыми?

13. Что означает «последовательность» и какие три типа входят в эту категорию?

14. Каким будет значение выражения 2 * (3 + 4) в Python?

15. Каким будет значение выражения 2 * 3 + 4 в Python?

16. Каким будет значение выражения 2 + 3 * 4 в Python?

17. Назовите два способа построения списка, содержащего пять целочисленных нулей.

18. Назовите два способа построения словаря с двумя ключами, 'a' и 'b', с каждым из которых ассоциировано значение 0.

19. Назовите четыре операции, которые изменяют списковый объект на месте.

20. Назовите четыре операции, которые изменяют словарный объект на месте.

21. Почему вы можете использовать словарь вместо списка?

22. Как определить размер кортежа? Почему этот инструмент находится там, где он есть?

23. Напишите выражение, которое изменяет первый элемент в кортеже. В процессе кортеж (4,5,6) должен стать (1,5,6).

24. Какой модуль вы бы использовали для сохранения объектов Python в файле без предварительного преобразования их в строки?

25. Как бы вы копировали сразу все части вложенной структуры (словарь в словаре, список в словаре и т. д.)?

26. Можно ли использовать строковый метод **find** для поиска в списке?

27. Можно ли применять выражение среза строки для списка?

28. Как бы вы могли изменить строку в Python?

29. Для заданной строки S со значением «s,pa,in» назовите два способа извлечения двух символов из середины строки.

Упражнения

Перед тем, как приступать к выполнению упражнений — обратите внимание на то, что они сформулированы таким образом, что материал по части из них либо не в полной мере рассматривался в этом разделе, либо не рассматривался вовсе. Это сделано специально! Не забывайте о том, что один из навыков, которыми вы должны обладать, — поиск информации в интернете и ее корректное применение в своем проекте. Как уже обсуждалось во введении, не следует довольствоваться просто найденным ответом и бездумно копировать найденный код. Лучше перепечатайте его вручную и более подробно разберитесь в том, как работает один из вариантов (или все варианты) ответов.

1. Что будет результатом следующих выражений?

```
"agility"[2:5] + "taxonomy"[3:6]
[115,202,192,334,257][:4]
len("crazy"[3:3+4])
[9,8,7,6,5,4,3,2,1][-3:]
type([False,True,False,True][2:3])
"---".join( "this is important".split() )
int( ''.join( "7/7/07".split('/') ) )
"too soon to tell".replace('o','*').replace('* ','')
```

2. Напишите скрипт, вычисляющий двумя способами длину строки.

3. Напишите скрипт, который позволяет из строки собрать другую по следующим правилам: новая строка должна состоять из двух первых и двух последних элементов исходной строки. Если длина исходной строки меньше двух, то результатом будет пустая строка.

4. Напишите скрипт, который заменяет произвольный символ/букву в строке на «$».

5. Напишите скрипт, который позволяет инвертировать последовательность элементов в строке.

6. Напишите скрипт, который считает количество вхождений символа в строку. Например: «google.com» — {'o': 3, 'g': 2, '.': 1, 'e': 1, 'l': 1, 'm': 1, 'c': 1}

7. Напишите скрипт, позволяющий из исходной строки собрать две новые. Первая строка должна состоять только из элементов с нечетными индексами исходной строки, а вторая — с четными.

8. Напишите скрипт, который удаляет задаваемый произвольный символ в строке.

9. Напишите скрипт, который позволяет переводить все символы исходной строки в верхний и нижний регистры.

10. Напишите скрипт, выводящий все элементы строки с их номерами индексов.

11. Напишите скрипт, проверяющий, содержится ли произвольный символ (элемент) или слово в строке.

12. Выведите символ, который встречается в строке чаще, чем остальные.

13. Поменяйте регистр элементов строки.

14. Вычислите сумму элементов (чисел) в списке двумя разными способами.

15. Умножьте каждый элемент списка на произвольное число.

16. Найдите максимальное и минимальное числа, хранящиеся в списке.

17. Напишите скрипт, удаляющий все повторяющиеся элементы из списка.

18. Скопируйте список двумя различными способами.

19. Напишите скрипт для слияния (конкатенации) двух списков различными способами.

20. Напишите скрипт, меняющий позициями элементы списка с индексами n и $n + 1$.

21. Напишите скрипт, переводящий список из различного количества числовых целочисленных элементов в одно число. Пример списка: [15, 23, 150], результат: 1523150

22. Объявите и инициализируйте словарь различными способами.

23. Добавьте еще несколько пар «ключ: значение» в следующий словарь: {0: 10, 1: 20}.

24. Напишите скрипт, который из трех словарей создаст один новый.

25. Напишите скрипт, проверяющий, существует ли заданный ключ в словаре.

26. Напишите скрипт для удаления элемента словаря.

27. Напишите скрипт, который выводит максимальное и минимальное числа среди значений словаря.

28. Объявите и инициализируйте кортеж различными способами, после чего распакуйте его.

29. Напишите скрипт для добавления элементов в кортеж.

30. Напишите скрипт, конвертирующий список в кортеж.

31. Конвертируйте кортеж в словарь.

32. Напишите скрипт, подсчитывающий количество элементов типа кортеж в списке.
33. Объявите и инициализируйте множество различными способами.
34. Добавьте элемент в множество.
35. Удалите элемент из множества.
36. Удалите повторяющиеся элементы из списка.
37. Напишите скрипт для объединения двух множеств.
38. Напишите скрипт, находящий длину множества двумя различными способами.
39. Напишите скрипт для проверки, входит ли элемент в множество.
40. Напишите скрипт для записи текста в файл.
41. Напишите скрипт для чтения текста из файла.
42. Напишите скрипт для добавления текста в файл и отображения содержимого файла.
43. Напишите скрипт для чтения последних n строк файла.
44. Напишите скрипт, подсчитывающий количество строк в файле.
45. Напишите скрипт, позволяющий найти самое встречаемое слово в файле.
46. Напишите скрипт, копирующий содержимое одного файла в другой.
47. Запишите словарь в файл посредством модуля pickle и прочитайте его.
48. Запишите список в файл посредством модуля pickle и прочитайте его.
49. Запишите словарь в файл посредством модуля json и прочитайте его.

Тема 2
СИНТАКСИС, ОПЕРАТОРЫ И УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

В данной теме рассматривается синтаксис языка программирования Python, а именно: что такое отступы в парадигме Python и зачем они нужны, основные операторы, циклы, вложения, условные конструкции и т. д.

Отдельно рассмотрим, как происходит итерация по объектам Python на примере использования циклов и какие инструменты, невидимые разработчику, при этом используются.

Также коснемся вопроса документации кода и тех стандартных инструментов документирования, которые изначально входят в состав Python.

В результате изучения данной темы обучающиеся должны:

знать

- синтаксис и основные операторы языка программирования Python,
- механизм итерации последовательностей,
- как создаются списки посредством механизма списковых включений;

уметь

- применять условные конструкции и циклы,
- применять циклы для перебора элементов последовательностей;

владеть

- навыками документирования кода,
- навыками использования существующего механизма документирования в Python,
- навыками написания кода с использованием условных конструкций, циклов и списковых включений.

2.1. Основные операторы в Python

Каждый оператор Python имеет собственное специфическое назначение и собственный специфический синтаксис — правила, определяющие его структуру.

Операторы в Python классифицируются следующим образом:

— процедурные операторы;

— операторы функций;

— операторы классов;
— операторы исключений;
— операторы модулей;
— арифметические операторы;
— операторы сравнения;
— операторы присваивания;
— побитовые операторы;
— логические операторы;
— операторы членства;
— операторы тождественности.

В табл. 2.1 приведен перечень операторов, относящихся к ключевым словам Python:

Таблица 2.1

Операторы (ключевые слова) Python

Оператор	Роль	Пример
Присваивания	Создание ссылок	`a, b = 'good', 'bad'`
Вызовы и другие выражения	Выполнение функций	`log.write("spam, ham")`
Вызовы print	Вывод объектов	`print ('test', val)`
if/elif/else	Выбор действий	`if "python" in text:` `  print(text)`
for/else	Итерация	`for x in mylist: print(x)`
while/else	Универсальные циклы	`while X > Y:` `  print('hello! ')`
pass	Пустой заполнитель	`while True:` `  pass`

break	Выход из цикла	`while True:`
		`  if exittest():`
		`  break`

continue	Продолжение цикла	`while True:`
		`  if skiptest () :`
		`  continue`

def	Функции и методы	`def f(a, b, c=1, *d):` `  print(a + b + c + d[0])`
return	Результаты функций	`def f (a, b, c=1, *d) :` `  return a+b+c+d[0]`
yield	Генераторные функции	`def gen (n) :` `  for i in n: yield i*2`

global	Пространства имен	`x = 'old'`
		`def function():`
		`  global x,y; x = 'new'`

Окончание табл. 2.1

Оператор	Роль	Пример

nonlocal	Пространства имен	`def outer():`
		`  x = 'old'`
		`  def function():`
		`  nonlocal x; x = 'new'`

import	Доступ к модулям	`import sys`
from	Доступ к атрибутам	`from sys import stdin`

class	Построение объектов	`class Subclass(Superclass):`
		`  staticData = [ ]`
		`  def method (self) :`
		`  pass`

try/except/finally	Перехват исключений	`try:`
		`  action()`
		`except:`
		`  print('action error')`

raise	Генерация исключений	`raise EndSearch(location)`
assert	Отладочные проверки	`assert X > Y, 'X too small'`
with/as	Диспетчеры контекста	`with open ('data ') as myfile:` `  process(myfile)`

del	Удаление ссылок	`del data[k]`
		`del data[i:j]`
		`del obj.attr`
		`del variable`

Формально **print**, начиная с Python3, не является ни зарезервированным словом, ни оператором, а вызовом встроенной функции; но поскольку **print** почти всегда будет выполняться в виде оператора с выражением (и часто занимать отдельную строку), такой вызов в общем случае трактуется как разновидность оператора.

Аналогично и с **yield**. Начиная с Python 2.5 оно считается выражением, а не оператором, но аналогично **print** оно обычно используется как оператор с выражением и было включено в таблицу.

Арифметические операторы представлены в табл. 2.2, операторы сравнения — в табл. 2.3.

Таблица 2.2

Арифметические операторы

Оператор	Описание	Примеры
+	Сложение	`10 + 5 = 15` `10 + -3 = 7`

Окончание табл. 2.2

Оператор	Описание	Примеры

-	Вычитание	`15 - 5 = 10`
		`25 - -3 =28`
		`11.98 - 7 = 4.98`

*	Умножение	`2 * 2 = 4`
		`7 * 3.2 =  22.4`
		`-2 * 4 = -8`

/	Деление	`12 / 4 = 3` `7 / 3 = 2.334`

%	Деление по модулю	`4 % 2 = 0`
		`9 % 2 = 1`
		`13.2 % 4 = 1.199`

**	Возведение в степень	`5 ** 2 = 25`
		`4 ** 3 = 64`
		`9**(1/2) = 3.0`

//	Целочисленное деление	`17 // 5 = 3` `10 // 3 = 3`

Таблица 2.3

Операторы сравнения

Оператор	Описание	Примеры

==	Проверка на равенство	`1 == 1  (True)`
		`True == False (False)`
		`"test" == "test" (True)`

!=	Проверка на неравенство	`1 != 2 (True)`
		`False != False (False)`
		`"test" != "Test" (True)`

>	Проверка на то, что значение левого операнда больше правого	`3 > 2 (True)` `2 > 3 (False)`
<	Проверка на то, что значение левого операнда меньше правого	`3 < 2 (False)` `2 < 3 (True)`

>=	Проверка на то, что значение левого операнда больше или равно правому	`3 >= 1 (True)`
		`3 >= 3 (True)`
		`"C" >= "D" (False)`

<=	Проверка на то, что значение левого операнда меньше или равно правому	`5 <= 5 (True)` `-4 <= -21 (False)`

Перечень операторов присваивания в Python представлен в табл. 2.4.

Таблица 2.4

Операторы присваивания

Оператор	Описание	Примеры

=	Оператор присваивания	`a = 3`
		`a = "4frt"`
		`a = 2.3`

+=	a += b, что равносильно a = a + b	`a = 3` `a +=2 #5`
-=	a -= b, что равносильно a = a - b	`a = 3` `a -= -2 #5`
*=	a *= b, что равносильно a = a*b	`a = 3` `a *= -2 # -6`
/=	a /= b, что равносильно a = a/b	`a = 8` `a /= 4 # 2`
%=	a %= b, что равносильно a = a%b	`a = 8` `a %= 3 # 2`
**=	a **= b, что равносильно a = a**b	`a = 8` `a **= 2 # 64`
//=	a //= b, что равносильно a = a**b	`a = 8` `a //= 3 # 2`

Для рассмотрения работы побитовых операторов (табл. 2.5) необходимо ввести два значения, a и b. Примем $a = 59$ в десятичной системе счисления, что эквивалентно 0011 1011 в двоичной системе счисления, и $b = 91$ (0101 1011).

Таблица 2.5

Побитовые операторы

Оператор	Описание	Примеры
&	Побитовое логическое «И» между операндами	`a & b = 0001 1011 (27)`
|	Побитовое логическое «ИЛИ» между операндами	`a | b = 0111 1011 (123)`
^	Побитовое логическое «исключающее ИЛИ» между операндами	`a ^ b = 0110 0000 (96)`
~	Логическое отрицание. Инвертирует значения бит операнда, к которому применяется	`~a = 1100 0100 (-60)`
<<	Побитовый сдвиг влево. Применяется к одному операнду. Эквивалентно умножению на 2^n, где n — число, на которое производится сдвиг	`b << 2 = 0001 0110 1100 (91 * 4 = 364)`
>>	Побитовый сдвиг вправо. Применяется к одному операнду. Эквивалентно целочисленному делению на 2^n	`b >> 2 = 00010 110 (91 // 4 = 22)`

К логическим операторам (табл. 2.6) относят обычные логические операции, как «И», «ИЛИ» и «НЕ»:

Таблица 2.6

Логические операторы

Оператор	Описание	Примеры
and	Логическое «И» между операндами	`True and True = True` Во всех остальных случаях False
or	Логическое «ИЛИ» между операндами	`False and False = False` Во всех остальных случаях True
not	Логическое отрицание	`not False = True` `not True = False`

Операторы включения (членства) используются для проверки, входит ли элемент в список, строку, множество и т. д. (табл. 2.7).

Таблица 2.7

Операторы включения (членства)

Оператор	Описание	Примеры

in	Возвращает True, если элемент присутствует в последовательности, иначе False	`"es" in "Test" (True)`
		`5 in [1, 3, 5, 7] (True)`
		`"first" in {"first":10,"one":1} (True)`
		`10 in {"first":10,"one":1} (False)`

not in	Возвращает False, если элемент присутствует в последовательности, иначе True	Результат работы противоположен работе оператора in

Операторы тождественности используются для проверки размещения двух операндов в памяти (их идентификаторов, табл. 2.8)

Таблица 2.8

Операторы тождественности

Оператор	Описание	Примеры
is	Возвращает True, если оба оператора указывают на один объект (имеют одинаковый идентификатор)	`a is b == True` (если id(a) ==id(b))
not is	Возвращает False, если оба оператора указывают на один объект (имеют одинаковый идентификатор)	`a is b == True` (если id(a) !=id(b))

Как и в любом другом языке программирования, в Python существует приоритет применения (выполнения) приведенных выше

операторов. Ниже приведен список приоритетов исполнения операторов (в порядке убывания):

— возведение в степень (**);
— унарный плюс, минус и побитовое отрицание (~, +, -);
— умножение, деление, целочисленное и деление по модулю (*, /, %, //);
— сложение и вычитание (+, -);
— побитовый сдвиг вправо и влево (>>, <<);
— побитовое логическое «И» (&);
— побитовое логическое «ИЛИ» и «исключающее ИЛИ» (|, ^);
— операторы сравнения (<=, <, >, >=);
— операторы равенства (==, !=);
— операторы присваивания (=, %=, /=, //=, -=, +=, *=, **=);
— тождественные операторы (is, is not);
— операторы членства (in, not in);
— логические операторы (not, or, and).

2.2. Использование отступов в Python

При написании кода на Python следует обращать внимание на отступы, так как они имеют значение. Именно отступы определяют границы блока (условного оператора, метода, класса, цикла и т. д.) выполнения кода и какой код в них входит.

Приведем пример:

```
def connet_stud_with_test(st_dict, ts_dict):
    bd_table = {}
    print("Связь id студентов с назначенными им id
           вариантов:")
    var_amount = len(ts_dict.keys())
    current_var = list(ts_dict.keys())[0]
    for id_stud in st_dict.keys():
        bd_table[id_stud] = current_var % var_amount
        current_var += 1
        print(id_stud, bd_table[id_stud])
    print("\n\n")
    return bd_table

x = 3
y = 5
if a > b:
    print("x > y")
    print(x - y)
else:
    print("y >= x")
    print(y - x)
```

Python понимает, какие строки кода относятся к блоку if на основе отступов. Выполнение блока if a > b заканчивается, когда встречается строка с тем же отступом (в данном случае *else*:).

В качестве отступов могут использоваться табуляция или пробелы (***их смешивания необходимо избегать! Начиная с версии Python 3.x оно запрещено.***). Их количество в одном блоке должно быть одинаковым. Также для форматирования кода можно использовать пустые строки. Это делается для упрощения чтения.

С более подробными рекомендациями по оформлению кода в процессе его написания можно ознакомиться в PEP (*Python Enhanced Proposal* — заявки на улучшение языка Python) [25 — 34]. Наиболее известный из них — PEP8, руководство по стилю написания кода на Python (*The Style Guide for Python Code*). Данный документ создан на основе рекомендаций Гвидо ван Россума с добавлениями идей от других разработчиков, входящих в программный комитет.

2.3. Комментарии

В том случае, когда написанный вами код будет поддерживаться еще и сторонними разработчиками, необходимо его комментировать. Это позволяет описать особенности его работы, что сэкономит как ваше время при последующем к нему обращении, так и время того человека, который должен разобраться в том, как код работает.

Комментарии в Python могут быть однострочными и многострочными. В первом случае используется ключевой символ «#», после которого идет комментарий, который не переносится на следующую строку:

```
# комментарий
a = 10 # еще один
```

Если комментарий будет занимать несколько строк, нужно использовать многострочный формат записи комментария. Для этого можно использовать три двойные или три одинарные кавычки:

```
"""
Сверх
длинный комментарий
"""
x = "text"
```

Комментарии можно использовать не только для того, чтобы комментировать происходящее в коде. Например, посредством комментариев можно исключить выполнение определенной строки или блока кода (то есть закомментировать их).

2.4. Правила именования переменных (имен)

В Python переменные начинают свое существование, когда им присваиваются значения, но есть несколько правил, которым необходимо следовать при выборе имен для объектов программы.

1. ***Синтаксис*: (*подчеркивание или буква*) + (*любое количество букв, цифр или подчеркиваний*).** Такая запись подразумевает, что имена должны начинаться с подчеркивания или буквы, за которой может следовать любое количество букв, цифр или подчеркиваний. Например, _test, test и Test__1 — допустимые имена, но 1_Test, test$ — нет.
2. ***Регистр символов имеет значение***. TEST — не то же самое, что и test.
3. ***Зарезервированные слова не разрешены***. Например, если выбрать для переменной имя class, то Python сообщит об ошибке, но имена klass и Class будут допустимыми.

Помимо этих правил существует еще набор соглашений по именованию. Это правила, которые считаются не обязательными, но соблюдаются разработчиками в повседневной практике:

— имена, которые начинаются с одиночного подчеркивания (_X), не импортируются оператором from module import *;

— имена с двумя подчеркиваниями в начале и конце (__X__) являются системными именами, которые имеют особый смысл для интерпретатора;

— имена, начинающиеся с двух подчеркиваний, но не оканчивающиеся ими (__X), локализованы («искажены») для включения в себя классов, то есть являются псевдо-приватными;

— имя, состоящее из одиночного подчеркивания (_), хранит результат последнего выражения при работе в интерактивном сеансе.

2.5. Оператор if

Оператор *if* в Python выбирает действия, которые будут выполняться в процессе работы программы в зависимости от условий. Он является основным инструментом выбора в Python и представляет большой объем логики.

Самая общая форма конструкции *if-then-else* в Python выглядит так:

```
if условие1:
    блок1
elif условие2:
    блок 2
...
elif условие(n-1):
    блок (n-1)
else:
    блок (n)
```

Если результат проверки *условия1* равен True, то выполняется *блок1*. В противном случае, если результат проверки *условия2* равен True, выполняется *блок2*, и так далее, пока не будет найдено условие, которое возвращает True, или не будет достигнута секция *else* (тогда выполняется *блок*(*n*)). Если *else* отсутствует и все проверки условий вернули False, то ни один из блоков ветвления не выполнится.

Блок кода после команды *if* является обязательным. А когда необходимо указать, что в данном блоке ничего не выполняется, то используется команда *pass*, которая размещается там, где должна находиться команда, но никаких действий она не выполняет:

```
if a < 25:
    pass
else:
    a = 25
```

К сожалению, в Python отсутствует оператор вроде *switch* или *case*. Разработчики Python обычно обходятся цепочкой *if… elif… elif… else*, а если она становится слишком громоздкой, то используется следующий способ со словарем функций:

```
def first_function():pass
#Обработка a
def second_function():pass
#Обработка b
def default():pass
#Обработка default
func_dict = {'a': first_function,
    'b': second_function,
    'c': lambda: …}
#выполнить функцию из словаря
func_dict.get(var, default)()
```

Также в Python нет тернарного оператора «?:» как такового. Вместо него используется конструкция *if-else*. Для примера преобразуем следующий код:

```
if A:
    Z = B
else:
    Z = C
```

посредством тернарной записи в следующую строку:

```
Z = B if A else C
```

Приведенное выражение дает в точности такой же результат, что и предшествующий четырехстрочный оператор *if*, но проще в написании.

2.6. Цикл while

Оператор (цикл) *while* используется для итераций в языке Python. Он многократно выполняет блок операторов до тех пор, пока проверка условия в заголовочной части возвращает True. Когда результат проверки становится False, управление переходит на оператор, следующий после блока *while*. Если с самого начала результат условия выполнения цикла вернет False, то тело цикла никогда не выполнится (оператор *while* пропускается).

В общем виде оператор *while* состоит из строки заголовка с условием выполнения цикла, тела цикла с одним или большим количеством операторов с отступами и необязательной части else, которая выполняется, если цикл не прерывается оператором *break*:

```
while условие: # Проверка цикла
    операторы # Тело цикла
else: # Необязательная часть else
    операторы
    # Выполняются, если не произведен
    # выход из цикла с помощью break
```

При объявлении цикла необходимо удостовериться, что из него предусмотрен выход. Иначе он получится бесконечным:

```
while True:
    print('Я вечен!!!')
```

Такой цикл будет выполняться до тех пор, пока в терминале не будет нажато сочетание клавиш Ctrl+C, прекращающее выполнение программы.

В Python нет оператора цикла «do while», имеющегося в других языках. Однако его можно эмулировать посредством проверки и оператора break в конце тела цикла, так что тело цикла всегда выполняется хотя бы раз:

```
while True:
    #тело цикла.
    if isExitLoop(): break
```

В следующем примере производится подсчет значений от x до y:

```
x= 0
y= 5
while x < y:
    print(x, end=' ')
    x += 1  # Или x = x + 1
# 0 1 2 3 4
```

Цикл с конструкцией *else*. Рассмотрим работу цикла с конструкцией *else* на основе предыдущего примера:

```
x = 0
y = 5
```

```
while x < y:
    print(x, end=' ')
    x += 1  # Или x = x + 1
else:
    print('Happy End!')
# 0 1 2 3 4 Happy End!
```

Конструкция *else* выполняется после окончания цикла в том случае, если выход из него не был выполнен посредством оператора *break*:

```
x= 0
y= 5
while x < y:
    print(x, end=' ')
    x += 1
    if x > y-2: break
else:
    print('Happy End!')
# 0 1 2 3
```

2.6.1. Работа цикла с операторами break, continue, pass

Теперь рассмотрим два простых оператора, которые достигают своих целей, только когда вложены внутри циклов — *break* и *continue*, а также пустой оператор-заполнитель *pass* (который не привязан к циклам как таковой, но относится к общей категории простых однословных операторов). Описание указанных операторов Python приведено ниже:

— *break* — переходит за пределы ближайшего заключающего его цикла;

— *continue* — переходит в начало ближайшего заключающего его цикла (на строку заголовка цикла);

— *pass* — ничего не делает (это пустой оператор-заполнитель).

С учетом операторов *break* и *continue* расширенный формат цикла *while* выглядит следующим образом:

```
while проверка:
    # тело цикла
    if проверка: break
    # выход из цикла с пропуском else (если есть)
    if проверка: continue
    # переход на проверку в начале цикла
else: # выполняется, если не было break
    #блок else
```

Операторы *break* и *continue* могут появляться где угодно внутри тела цикла *while* (или *for*). Обычно их записывают с дополнительным отступом в операторе *if*.

Далее будут рассмотрены несколько простых примеров применения этих операторов на практике.

Оператор pass. Оператор *pass* — заполнитель, который используется в ситуациях, когда синтаксис требует оператора, но нет возможности выполнить что-либо полезное.

Для примера его работы перепишем ранее приведенный бесконечный цикл, чтобы он ничего не выводил:

```
while True: pass
```

Иногда оператор *pass* обозначает место, подлежащее заполнению в будущем, что служит временной заглушкой для тела функции. Это связано с тем, что нельзя оставить тело функции пустым, не получив синтаксической ошибки:

```
def test():
    pass

def test():
# ошибка
File "", line 1
    def test():
               ^
SyntaxError: unexpected EOF while parsing
```

Начиная с python3 вместо *pass* можно указать многоточие, которое записывается как «. . .» (просто три следующие друг за другом точки):

```
def test():
    ...
```

Оператор continue. Оператор *continue* используется для немедленного перехода в начало цикла. В некоторых случаях он позволяет избежать вложения операторов:

```
a = 15
while a:
    x = a-1 # a -= 1
    if a % 2 != 0: continue
    print(a, end=' ')
```

Из-за того, что *continue* инициирует переход в начало цикла, нет необходимости вкладывать *print* внутрь проверки *if*, так как он выполнится только в том случае, если не будет выполнен *continue*. Приведенный выше пример можно записать и без использования оператора *continue*:

```
a = 15
while a:
    x = a-1 # a -= 1
    if a % 2 == 0:
        print(a, end=' ')
```

Оператор break. Оператор *break* используется для немедленного выхода из цикла. Поскольку при его достижении код, который нахо-

дится за ним в теле цикла, не выполняется, за счет включения *break* иногда можно избежать вложения:

```
my_sum = 0
while True:
    val = int(input('Введите значение: '))
    if val == 0: break
    my_sum += val
    print('Текущая сумма: ', my_sum)
print('Конечная сумма: ', my_sum)
""" Введите значение: 6
Текущая сумма:  6
Введите значение: 10
Текущая сумма:  16
Введите значение: 20
Текущая сумма:  36
Введите значение: 0
Конечная сумма:  36 """
```

В текущем примере функция **input** используется для считывания вводимого значения с клавиатуры. Поскольку данные всегда считываются в строковом формате *str*, введенное значение необходимо преобразовать в числовой (целочисленный) формат данных:

```
int(input('Введите значение: '))
```

2.7. Цикл for

Цикл (оператор) *for* перебирает значения, возвращаемые любым итерируемым объектом, то есть любым объектом, который может сгенерировать последовательность значений. Например, цикл *for* может перебрать элементы списка, кортежа или строки. Итерируемый объект может быть создан функцией **range** или специальной разновидностью функций — генератором:

```
for элемент in последовательность:
    ... #тело цикла
else: # необязательная часть else
    ...
# если для выхода из цикла не использовался break
```

Тело цикла выполнится по одному разу для каждого элемента последовательности. Необязательная часть *else* используется достаточно редко и работает аналогично циклу *while*. То же самое относится к командам *break* и *continue*. Таким образом, более полный формат цикла *for* может быть записан следующим образом:

```
for элемент in последовательность:
    # тело цикла
    if проверка: break
    if проверка: continue
else: #блок else
```

Приведем пример поэлементного перебора списка с выводом значений текущего итерируемого элемента в терминал:

```
for it in ['first', 'second', 4, 5.9, 'finish']:
    print(it, end='  ')
# first second 4 5.9 finish
```

Цикл *for* позволяет проводить поэлементный перебор строк:

```
my_str = 'в далёкой-далёкой галактике...'
for it in my_str:
    print(it, end='-')
# в- -д-а-л-ё-к-о-й---д-а-л-ё-к-о-й-
# -г-а-л-а-к-т-и-к-е-.-.-.-
```

Итерация по элементам кортежей осуществляется следующим образом:

```
my_str = ('в', 'далёкой-','далёкой', 'галактике...')
for it in my_str:
    print(it, end=' ')
# в далёкой- далёкой галактике...
```

Для итерации по списку кортежей используют следующий подход:

```
my_tuple_list = [(3,6), (0,1), (4,5), ('O',3.9)]
for (a,b) in my_tuple_list:
    print(a, b)
# 3 6
# 0 1
# 4 5
# O 3.9
```

Кортежи в циклах *for* также становятся полезными при итерации сразу по ключам и значениям в словарях с применением метода **items** вместо прохода в цикле по ключам и индексации для извлечения значений вручную:

```
# извлечение значений через итерируемый ключ
my_dict = {'a':1, 'b': 10, 'c': 3.8}
for key in my_dict:
    print(key, '=>', my_dict[key])
# a => 1
# b => 10
# c => 3.8

# итерация по паре "ключ:значение"
for key, value in my_dict.items():
    print(key, '=>', value)
# a => 1
# b => 10
# c => 3.8
```

Каждый цикл может содержать в себе еще один цикл и т. д. Такие циклы называются вложенными:

```
items = ("aaa", 111, (4, 5), 2.01)
tests = [(4, 5) , 3.14]
for key in tests:
    for item in items:
        if item == key:
            print (key, 'найден')
            break
else:
    print(key, "не найден!")
# (4, 5) найден
# 3.14 не найден!
```

2.8. Различные способы написания циклов

Существуют ситуации, когда требуется выполнить проход по последовательности более специализированным способом. Допустим, нужно посетить каждый второй или каждый третий элемент в списке, либо попутно изменять список, или произвести обход более одной последовательности параллельно в том же самом цикле *for* и т. д.

Такие специфичные итерации всегда можно реализовать с помощью цикла *while* и ручной индексации, но Python предлагает набор встроенных функций, которые позволяют специализировать итерацию в цикле *for*:

— встроенная функция **range** (доступная начиная с Python 0.x) производит серию последовательных целых чисел с задаваемым направлением (возрастает или убывает) и шагом, которые могут использоваться в качестве индексов в цикле *for*;

— встроенная функция **zip** (доступная начиная с версии Python 2.0) возвращает серию кортежей из параллельных элементов, которые могут применяться для обхода множества последовательностей в цикле *for*;

— встроенная функция **enumerate** (доступная начиная с версии Python 2.3) генерирует значения и индексы элементов в итерируемом объекте.

2.8.1. Использование встроенной функции *range*

Встроенная функция **range** является универсальным инструментом, который может использоваться в разнообразных контекстах. Она наиболее часто применяется для генерации индексов в цикле *for*, но ее можно использовать где угодно, когда требуется серия целых чисел. Начиная с python3 **range** является итерируемым объ-

ектом, который генерирует элементы по запросу, и ее базовый синтаксис может быть записан следующим образом:

```
range(from [, to, step])
```

где элементы в [] являются необязательными.

Приведем пример работы с различными значениями, подаваемыми на вход функции **range**:

```
print(range(10)) # range(0, 10)
print(list(range(10)))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(range(0,6)) # range(0, 6)
print(list(range(0, 6))) # [0, 1, 2, 3, 4, 5]

print(range(0, 6, 2)) # range(0, 6, 2)
print(list(range(0, 6, 2))) # [0, 2, 4]
```

С одним аргументом **range** генерирует список целых чисел, начиная с нуля и заканчивая (но не включая) значением аргумента. В случае передачи двух аргументов первый считается нижней границей. Необязательный третий аргумент может задавать «шаг», когда он указан. Python добавляет выбранный шаг к каждому последующему целому числу в результате (по умолчанию шаг равен +1). При желании диапазоны могут быть неположительными и невозрастающими:

```
print(list(range(-5, 5)))
# [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

print(list(range(5, -5, -1)))
# [5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Таким образом, чтобы вывести три строки в цикле *for*, с помощью **range** можно сгенерировать соответствующее количество целых чисел:

```
for i in range (3) :
    print(i, 'Test')
# 0 Test
# 1 Test
# 2 Test
```

Встроенная функция **range** также может использоваться для непрямого прохода по последовательности:

```
my_str = 'Test'
for it in my_str:
    print(it, end=' ') # Простая итерация
# T e s t

print(len(my_str)) # 4
print(list(range(len(my_str)))) # [0, 1, 2, 3]
```

```
#Ручная итерация посредством range/len
for it in range(len(my_str)):
    print(my_str[it], end=' ')
# T e s t
```

Во второй части примера производится проход по списку смещений в строке *my_str*, а не по ее действительным элементам, ввиду чего для извлечения каждого элемента необходима индексация в *my_str*.

При работе с циклами в Python необходимо ориентироваться на следующие правила:

— во всех случаях, когда это возможно, вместо цикла *while* стоит использовать *for*;

— использовать вызовы **range** в циклах *for* только в крайних или специализированных случаях, то есть для обхода последовательностей предпочтительнее использовать запись вида *for it in list*.

2.8.2. Использование встроенной функции zip

Встроенная функция **zip**, в отличие от **range**, дает возможность применять циклы *for* для просмотра множества последовательностей параллельно — без совмещения во времени, но в течение того же самого цикла. В базовом виде функция **zip** принимает одну или большее количество последовательностей в качестве аргументов и возвращает серию кортежей, объединяющих в пары параллельные элементы из указанных последовательностей:

```
first_list = [1,2,3,4,5]
second_list = [6,7,8,9,10]
print(list(zip(first_list, second_list)))
# [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

Таким образом, встроенная функция **zip** в сочетании с циклом *for* предоставляет возможность поддержки параллельных итераций:

```
for a, b in zip(first_list, second_list):
    print(a, '+', b, '=' , a+b)
# 1 + 6 = 7
# 2 + 7 = 9
# 3 + 8 = 11
# 4 + 9 = 13
# 5 + 10 = 15
```

Эффект от такого подхода заключается в том, что в цикле просматриваются оба списка: *first_list* и *second_list*. Похожего результата можно добиться используя цикл *while*, который поддерживает индексацию вручную. Но этот способ потребовал бы большего объема кода и выполнялся несколько медленней, чем использование связки for-zip.

Ранее было сказано, что функция **zip** может принимать на вход любое количество последовательностей. Главное, чтобы объекты,

подаваемые ей на вход, представляли собой итерируемый объект. Приведем пример с тремя аргументами. В этом случае функция **zip** построит список трехэлементных кортежей с элементами из каждой последовательности, по существу проецируя по столбцам (формально для N аргументов мы получаем N-арный кортеж):

```
one, two, three = (1,2,3) , (4,5,6), (7,8,9)
print(list(zip(one, two, three)))
# [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
for a, b, c in zip(one, two, three):
    print('({} - {}) * {} = {}'.format(b, a, c,
                                       (b-a)*c))
# (4 - 1) * 7 = 21
# (5 - 2) * 8 = 24
# (6 - 3) * 9 = 27
```

Если на вход подаются последовательности различной длины, результирующие кортежи нормируются по длине самой короткой из них:

```
one, two, three = (1,2,3,5,9), (4,5,6), (7,8,9,23)
print(list(zip(one, two, three)))
# [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

2.8.3. Использование встроенной функции enumerate

В ряде случаев бывает необходимо производить итерацию по элементам последовательности и знать, какой индекс у текущего итерируемого элемента. Традиционно такая задача решается посредством простого цикла *for* с дополнительным счетчиком индекса итерируемого элемента:

```
my_str = 'Test'
index = 0
for it in my_str:
    print('Индекс элемента ',it,' = ' , index)
    index +=1
# Индекс элемента  T  =  0
# Индекс элемента  e  =  1
# Индекс элемента  s  =  2
# Индекс элемента  t  =  3
```

Функция **enumerate** возвращает *генераторный объект* — разновидность объекта, поддерживающая протокол итерации. Такой объект имеет метод, вызываемый встроенной функцией **next**, который на каждом проходе цикла возвращает кортеж {*индекс*, *значение*}:

```
my_str = 'Test'
for index, it in enumerate(my_str):
    print('Индекс элемента ',it,' = ' , index)
# Индекс элемента  T  =  0
# Индекс элемента  e  =  1
# Индекс элемента  s  =  2
# Индекс элемента  t  =  3
```

Цикл *for* автоматически проходит по всем кортежам, возвращаемым функцией **enumerate**, что позволяет распаковывать их значения почти так же, как это делается для **zip**:

```
test = enumerate(my_str)
print(next(test)) # (0, 'T')
print(next(test)) # (1, 'e')
print(next(test)) # (2, 's')
print(next(test)) # (3, 't')
```

2.9. Итерации и включения

2.9.1. Итератор и итерируемый объект

Как уже отмечалось ранее, цикл *for* работает с любым итерируемым объектом. Это же касается и всех итерационных инструментов Python, которые просматривают объекты слева направо: списковых включений, проверок членства *in*, встроенной функции **map** и т. д.

Концепция «итерируемых объектов» в Python буквально пронизывает всю языковую модель и представляет собой обобщение понятия «последовательность». Большинство контейнеров и множество других типов данных (файлы, сокеты и т. д.) являются итерируемыми объектами. В то время как контейнеры обычно содержат конечное количество элементов, просто итерируемый объект может представлять источник бесконечных данных [35].

В действительности итерация основана на двух объектах, которые используются на двух отдельных шагах:

— итерируемый объект, для которого запрашивается итерация, чей метод **__iter__** запускается методом *i***ter**;

— объект итератора, возвращенный итерируемым объектом, который фактически производит значения во время итерации, чей метод **__next__** запускается методом **next**, и генерирует исключение *StopIteration*, когда завершает выдачу результатов.

На рис. 2.1 приведен в упрощенном виде пример итерации по итерируемому объекту или генератору, используемый циклом *for*.

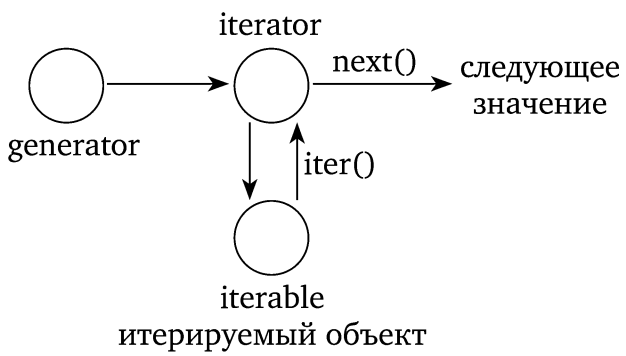

Рис. 2.1. **Пример итерации по итерируемому объекту, используемый циклом *for***

Говоря простыми словами, итерируемый объект — это любой объект, который может предоставить итератор, возвращающий отдельные элементы этого объекта:

```
my_list = [1, 2, 3]
first_iter = iter(my_list)
second_iter = iter(my_list)
print(next(first_iter)) # 1
print(next(first_iter)) # 2
print(next(second_iter)) # 1
print(next(first_iter)) # 3
print(next(first_iter))
------------------------------------------------------------
StopIteration     Traceback (most recent call last)
 in
----> 8 print(next(y))
StopIteration:
```

my_list — это итерируемый объект (список), а *first_iter* и *second_iter* — два отдельных экземпляра итератора, производящего значения из итерируемого объекта *my_list*. Из примера видно, что *first_iter* и *second_iter* сохраняют состояние между вызовами **next()**, и когда итерация по списку посредством *first_iter* становится невозможной, генерируется исключение *StopIteration*.

Из описанного выше следует, что код:

```
my_list = [1, 2, 3]
for it in my_list:
    pass
```

выполняет набор действий, показанный на рис. 2.2.

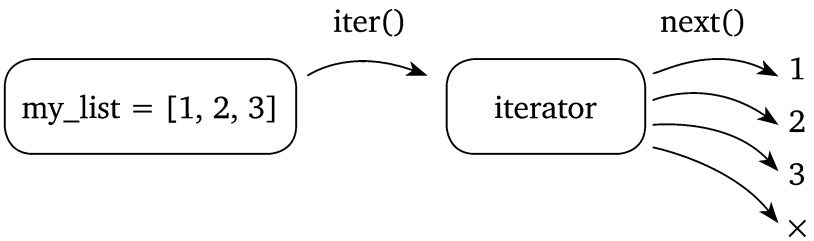

Рис. 2.2. **Итерация в цикле *for* по списку *my_list***

Также при работе с итератором можно напрямую вызывать метод **__next__**:

```
my_list = [1, 2, 3]
first_iter = iter(my_list)
print(first_iter.__next__()) # 1
```

2.9.2. Списковые включения (list comprehension)

В дополнение к операциям над последовательностями и списковым методам в Python имеется более сложная операция, известная как выражение спискового включения (*list comprehension*) и являю-

щаяся мощным способом обработки структур. Вместе с циклами *for* списковые включения входят в число наиболее известных контекстов, в которых применяется итерирование.

Списковые включения происходят от системы обозначения множеств и предоставляют способ построения нового списка за счет выполнения выражения на каждом элементе в последовательности, по одному за раз (слева направо) [36]:

```
new_list = [expression for member in iterable]
```

Под ***expression*** понимается вычисление, вызов метода или любое другое допустимое выражение, которое возвращает значение; ***member*** является объектом или значением в списке или итерируемым объекте (***iterable***).

Все это позволяет следующий код:

```
my_list = [1, 2, 3, 4, 5]
for i in range (len(my_list)) :
    my_list[i] += 10
print(my_list) # [11, 12, 13, 14, 15]
```

записать следующим образом:

```
my_list = [1, 2, 3, 4, 5]
my_list = [it+10 for it in my_list]
print(my_list) # [11, 12, 13, 14, 15]
```

Следует отметить, что списковые включения не являются обязательными, так как всегда можно построить список результатов выполнения выражения вручную, используя цикл *for*:

```
my_list = [1, 2, 3, 4, 5]
new_list = []
for it in my_list:
    new_list.append(it + 10)
print(new_list) # [11, 12, 13, 14, 15]
```

Их часто называют «синтаксическим сахаром» для цикла *for*, но по времени выполнения они даже выигрывают из-за того, то не вызывают метод **append()** у списка [37].

Наиболее распространенный способ использовать условную логику при работе со списковыми включениями — добавить условное выражение в конец выражения:

```
new_list = [expression for member in iterable (if conditional)]
```

Например:

```
new_list = [x for x in range(10) if x % 2 != 0]
print(new_list) # [1, 3, 5, 7, 9]
```

В случае, когда необходимо использовать полноценный блок ветвления *if-else*, используется следующий вид спискового включения:

```
new_list = [expression (if conditional) for member in iterable]
```

Например, заменим отрицательные значения на 0, а положительные оставим без изменения:

```
my_list = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
new_list = [i if i > 0 else 0 for i in my_list]
print(new_list) # [1.25, 0, 10.22, 3.78, 0, 1.16]
```

Если необходимо использовать более сложную логику, то вместо выражения можно использовать функцию:

```
def test_func(value):
    return value if value > 0 else 0

my_list = [1.25, -9.45, 10.22, 3.78, -5.92, 1.16]
new_list = [test_func(i) for i in my_list]
print(new_list) # [1.25, 0, 10.22, 3.78, 0, 1.16]
```

Начиная с версии 3.8 в Python введен моржовый оператор walrus. В отличие от операции присваивания (=), результат моржового оператора также является выражением и может быть использован, например, для сравнения:

```
import random
def get_weather_data():
     return random.randrange(90, 110)

too_hot = [temp for _ in range(20) if (temp :=
             get_weather_data()) >= 100]
print(too_hot)
# [102, 106, 108, 103, 106, 103, 100, 108,
#  100, 102, 100]
```

Как и циклы, включения могут содержать вложенные включения. Рассмотрим пример такого вложения для формирования матрицы:

```
matrix = [[i for i in range(4)] for _ in range(4)]
print(matrix)
# [[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3],
#  [0, 1, 2, 3]]
```

Включения можно использовать также для словарей (*dictionary comprehensions*) и множеств (*set comprehensions*):

```
my_list = [10, 34, 56 ,1 ,1, 2, 4, 3, 102, 102]
my_set = {i for i in my_list if i%2==0}
print(my_set) # {2, 34, 4, 102, 10, 56}

my_dict = {i: i * i for i in range(10)}
print(my_dict)
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25,
#  6: 36, 7: 49, 8: 64, 9: 81}
```

2.10. Источники документации Python

Часто бывает необходимо найти информацию по той или иной функции, методу объекта и т. д. Именно из-за этого документация и является настолько важным инструментом при практическом программировании. В табл. 2.9 приведены источники документации Python.

Таблица 2.9

Источники документации Python

Формат источника	Описание
Комментарии «#»	Внутрифайловая документация
Функция **dir**	Списки атрибутов, доступных в объектах
Строки документации: *__doc__*	Внутрифайловая документация, присоединяемая к объектам
PyDoc: функция **help**	Интерактивная справка для объектов
PyDoc: отчеты HTML	Документация по модулям, отображающаяся в браузере
Стандартный набор руководств и Web-ресурсы	Официальные описания языка, библиотек, онлайновые руководства, примеры и т. д.
Печатные издания	Коммерческие справочники

Комментарии «#». Комментарии «#» являются наиболее базовым способом документирования кода. Python просто игнорирует весь текст, следующий за символом # (до тех пор, пока он не находится внутри строкового литерала), поэтому после # можно помещать любые слова и описания. Однако такие комментарии доступны только в файлах исходного кода, а для написания более широко доступных комментариев понадобится применять строки документации ('"Файл делает то-то"').

Функция dir. Встроенная функция **dir** предоставляет легкий способ получения перечня всех атрибутов, доступных внутри объекта (т. е. его методов и более простых элементов данных). Ее можно вызывать без аргументов, чтобы получить список переменных в области видимости вызывающего кода. Но более удобно то, что **dir** можно вызывать на любом объекте, имеющем атрибуты, в числе которых импортированные модули и встроенные типы, а также имена типов данных:

```
print(dir())
['In','Out', '_', '__', '___', '__builtin__', '__builtins__',
'__doc__', '__loader__', '__name__', '__package__', '__spec__',
'_dh', '_i', '_i1', '_ih', '_ii', '_iii', '_oh', 'exit', 'get_
ipython', 'os', 'quit', 'sys']

print(dir(dict))
```

```
['__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__
init__', '__init_subclass__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__
str__', '__subclasshook__', 'clear', 'copy', 'fromkeys', 'get',
'items', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']

#- количество атрибутов в словаре
print(len(dir({}))) # 40
```

Чтобы отфильтровать элементы, например, с двумя подчеркиваниями в выводимом списке атрибутов, которые обычно не представляют интереса для разработчика, можно воспользоваться включениями:

```
print([it for it in dir(dict) if not it.startswith
        ('__')])
['clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

Строки документации __doc__. Строки документации в Python автоматически присоединяются к объектам и сохраняются во время выполнения программы. Синтаксически такие комментарии записываются как строки ('''Файл делает то-то''', """Метод делает то-то""") в начале файлов модулей, а также операторов функций и классов, их методов и т. д., перед любым другим исполняемым кодом.

Python автоматически помещает текст таких строк в атрибуты *__doc__*соответствующих объектов (файл testdoc.py):

```
'''
Документация модуля
Бла-бла-бла
'''

spam =40
def square(x):
    '''
    Документация функции
    Дебажить? Или заявить, что это фича?
    '''
    return x**1 #квадрат

class Employee:
    "Документация класса"
    pass

print(square (4)) # 4
print(square.__doc__)
    # Документация функции
    # Дебажить? Или заявить, что это фича?
```

Весь смысл соблюдения такого подхода к документации в том, что после импортирования файла комментарии хранятся в атрибутах *__doc__*. Таким образом, для отображения строк документации, ассоциированных с модулем и его объектами, необходимо импортировать файл и вывести его атрибуты *__doc__*, где Python сохранил текст:

```
import testdoc

print(testdoc.__doc__)
# Документация модуля
# Бла-бла-бла
print(testdoc.square.__doc__)
    # Документация функции
    # Дебажить? Или заявить, что это фича?
print(testdoc.Employee.__doc__)
# Документация класса
```

Встроенные модули и объекты в Python применяют похожие методы для присоединения к ним документации. Например, чтобы увидеть фактическое описание встроенного модуля, предназначенное для человека, необходимо его импортировать и вывести его строку *__doc__*:

```
import copy
print(copy.deepcopy.__doc__)
"""Deep copy operation on arbitrary Python objects.
  See the module's __doc__ string for more info. """
print(list.remove.__doc__)
# Remove first occurrence of value.
print(dict.__doc__)
""" dict() -> new empty dictionary
dict(mapping) -> new dictionary initialized from a
mapping object's
    (key, value) pairs
dict(iterable) -> new dictionary initialized as
if via:
    d = {}
    for k, v in iterable:
        d[k] = v
dict(**kwargs) -> new dictionary initialized with
     the name=value pairs
    in the keyword argument list.  For example:
  dict(one=1, two=2)
"""
```

PyDoc: функция help. Стандартный инструмент PyDoc представляет собой код Python, которому известно, как извлекать строки документации вместе с ассоциированной структурной информацией и форматировать их в аккуратно организованные отчеты различных видов.

Существует несколько способов запуска PyDoc, включая параметры сценария командной строки, которые могут сохранять результирующую документацию для просмотра в будущем. Самыми востребованными интерфейсами PyDoc обычно являются встроенная функция **help**, а также средства отображения отчетов в формате HTML с графическим пользовательским интерфейсом:

```
import testdoc
print(help(list.remove))
"""Help on method_descriptor:

remove(self, value, /)
    Remove first occurrence of value.

    Raises ValueError if the value is not present.

None """

print(help(dict.update))
"""Help on method_descriptor:

update(...)
    D.update([E, ]**F) -> None.  Update D from
              dict/iterable E and F.
    If E is present and has a .keys() method, then
           does:  for k in E: D[k] = E[k]
    If E is present and lacks a .keys() method, then
           does:  for k, v in E: D[k] = v
    In either case, this is followed by: for k in F:
          D[k] = F[k]

None """

print(help(testdoc))
"""Help on module testdoc:

NAME
    testdoc

DESCRIPTION
    Документация модуля
    Бла-бла-бла

CLASSES
    builtins.object
        Employee

    class Employee(builtins.object)
     |  Документация класса
     |
     |  Data descriptors defined here:
     |
     |  __dict__
```

```
    |      dictionary for instance variables (if
    |            defined)
    |
    |  __weakref__
    |      list of weak references to the object
    |         (if defined)

FUNCTIONS
    square(x)
        Документация функции
        Дебажить? Или заявить, что это фича?

DATA
    spam = 40

FILE
    f:\code\python\myprojrep\testdoc.py """
```

PyDoc: отчеты HTML. Для запуска данного режима работы документации необходимо в терминале (из каталога проекта) запустить следующую команду:

```
(venv) F:\code\python\myProjRep>python -m pydoc -b
Server ready at http://localhost:58003/
Server commands: [b]rowser, [q]uit
server>
```

Результатом выполнения приведенной команды будет запуск браузера с вкладкой документации, как показано на рис. 2.3.

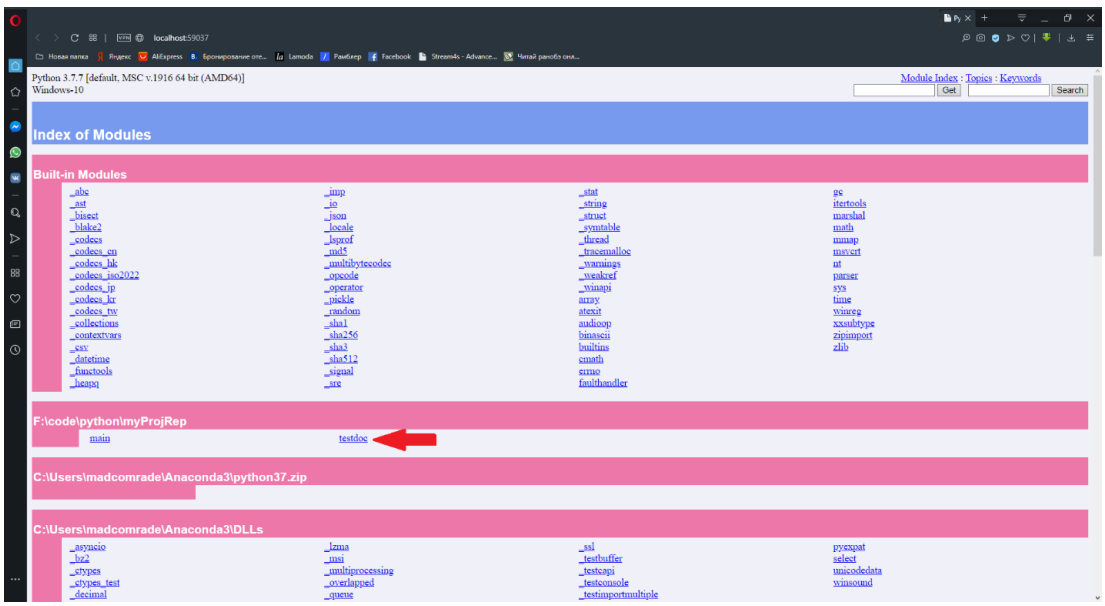

Рис. 2.3. **Сформированная HTML-документация**

Если нажать на написанный ранее модуль testdoc.py, то произойдет переход на страницу с отображением его документации (рис. 2.4).

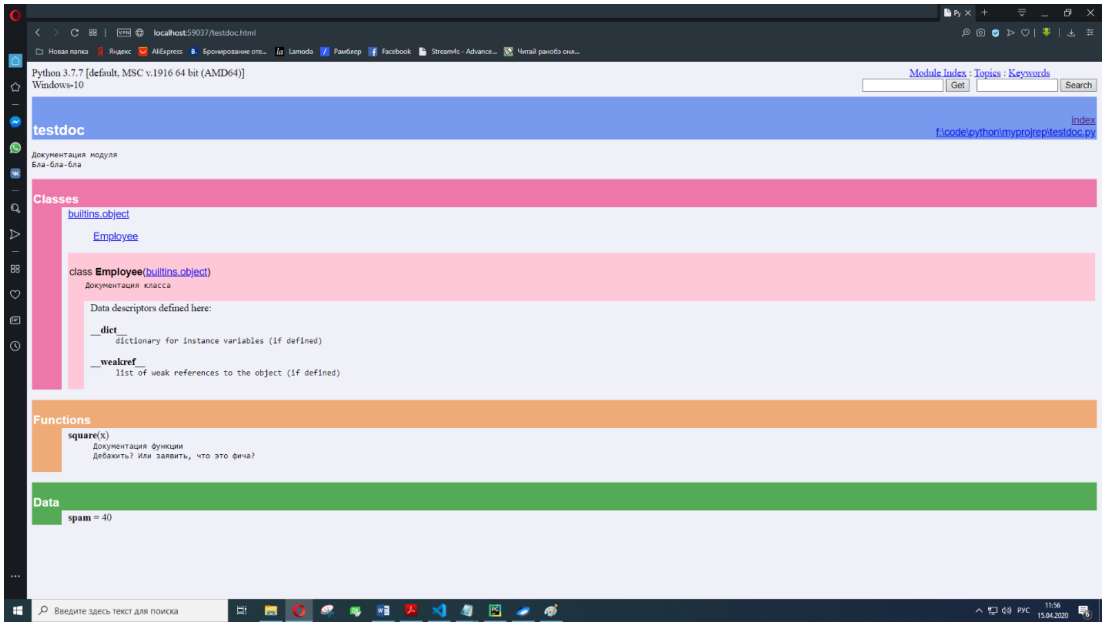

Рис. 2.4. **Сформированная HTML-документация модуля testdoc**

Стандартный набор руководств и Web-ресурсы. На официальном веб-сайте Python (http://www.python.org) помимо документации [38] по языку программирования (имеются онлайн- и офлайн-версии) можно найти ссылки на разнообразные ресурсы, где раскрываются специальные темы, тонкости применения и т. д.

Помимо этого, в наши дни значительно развит онлайн-сегмент образовательных блогов, курсов, вебинаров и прочих ресурсов. Надо только уметь пользоваться поисковиком и не пренебрегать им в случае возникновения вопросов или затруднений при написании кода.

Печатные издания. Также можно выбрать из коллекции прошедших профессиональную редактуру и опубликованных справочников (самоучителей) по Python. При таком подходе стоит обратить внимание на то, что книги имеют тенденцию отставать от самых современных изменений Python. Это связано отчасти с объемом работы по созданию таких книг и частично с естественными задержками, присущими издательскому циклу. Как правило, к моменту выхода английская версия книги отстает на три и больше месяцев от текущего состояния Python, что уж говорить о ее издании на русском языке.

Резюме

Данная часть была посвящена больше синтаксису Python и вопросам документации кода. Были рассмотрены циклы, вложения, условные конструкции и т. д. Отдельно рассматривался процесс итерации по циклу и какие инструменты, невидимые разработчику, при этом используются.

Чтобы в дальнейшем избежать проблем, связанных с синтаксисом языка и стилем написания кода, приведем список подсказок, к которому можно обратиться в любой момент времени.

1. *Не забывайте о двоеточиях*. Постоянно помните о необходимости набора символа «:» в конце заголовка составного оператора — первой строки if, while, for и т. д.
2. *Начинайте свой код без отступов*. Убедитесь, что начинаете код верхнего уровня (не вложенный) без отступов.
3. *Делайте отступы согласованно*. Избегайте смешивания табуляций и пробелов в отступах. В противном случае то, что вы видите в своем редакторе, может быть не тем, что видит Python, когда он пересчитывает табуляции в пробелы.
4. *Не пишите код Python в стиле C*. Напоминание программистам на C/C++: нет необходимости помещать в круглые скобки выражения проверок в заголовках if и while (например, if (X==1) :). При желании поступать так можно (заключать в круглые скобки разрешено любое выражение), но в данном контексте они совершенно излишни.
5. *Используйте простые циклы for вместо while или range*. Простой цикл for (скажем, for x in seq:) почти всегда проще в написании и часто быстрее в выполнении, чем цикл с подсчетом на основе while или range.
6. *Будьте осторожны с изменяемыми объектами в присваиваниях*. Нужно проявлять осмотрительность при использовании изменяемых объектов в групповом присваивании (a = b = [ ]), а также в дополненном присваивании (a += [1, 2]). В обоих случаях изменения на месте могут повлиять на другие переменные.
7. *Не ожидайте получить результаты от функций, которые изменяют объекты на месте*.
8. *Всегда применяйте круглые скобки для вызова функции*. Чтобы вызвать функцию, после ее имени потребуется указать круглые скобки независимо от того, принимает она аргументы или нет.
9. *Не используйте расширения или пути при импортировании и перезагрузке*. Опускайте пути по каталогам и файловые расширения в операторах *import*. Используйте *import mod*, а не *import mod.py*.

Вопросы для самопроверки

1. Почему необходимо соблюдать осторожность в случае присваивания трем переменным изменяемого объекта?
2. Как вы могли бы закодировать множественное ветвление в Python?
3. Как вы могли бы закодировать оператор if/else в форме выражения в Python?
4. Как вы могли бы распространить одиночный оператор на несколько строк?

5. Что означают слова True и False?
6. Каковы главные функциональные отличия между while и for?
7. В чем разница между break и continue?
8. Когда выполняется конструкция else цикла?
9. Как можно записать цикл с подсчетом в Python?
10. Для чего может применяться range в цикле for?
11. Как связаны циклы for и итерируемые объекты?
12. Как связаны циклы for и списковые включения?
13. Когда должны использоваться строки документации вместо комментариев #?
14. Назовите три способа, которыми можно просматривать строки документации.
15. Как можно получить список доступных атрибутов в объекте?
16. Как можно получить список доступных модулей на компьютере?
17. Какую книгу, посвященную Python, вы должны приобрести после этой?

Упражнения

1. Выведите все символы из строки «*Данная часть была посвящена больше синтаксису Python и вопросам документации кода*», значения индексов которых делятся на 2.
2. Выведите все символы из строки «*Данная часть была посвящена больше синтаксису Python и вопросам документации кода*», значения индексов которых без остатка делятся на 3, но не делятся на 4.
3. Выведите все символы из строки «*Данная часть была посвящена больше синтаксису Python и вопросам документации кода*», значения индексов которых при делении на 6 дают остаток 2, 4, и 5.
4. Выведите числа из диапазона от 1 до 10, используя цикл *for* и *while*.
5. Выведите числа из диапазона от –20 до 20 с шагом 3, используя цикл *for* и *while*.
6. Посчитайте количество вхождений элемента со значением «3» в следующем списке: [*3 0 1 3 0 4 3 3 4 56 6 1 3*], используя цикл *for*, *while* и метод *count*.
7. Сформируйте список из элементов строки «*список доступных атрибутов*», используя механизм списковых включений и цикл *for*.
8. Сформируйте единичную матрицу *N*×*N*, используя механизм списковых включений.
9. Напишите программу, выводящую элементы списка [*3 0 1 3 0 4 3 3 4 56 6 1 3*] в обратной последовательности.
10. Напишите программу, которая выводит числе в диапазоне от 1 до 9, кроме 5 и 7.
11. Напишите программу, выводящую сумму элементов списка [*3 0 1 3 0 4 3 3 4 56 6 1 3*], используя цикл *for*, *while* и метод *sum*.
12. Напишите программу, выводящую сумму элементов списка [*3 0 1 3 0 4 3 3 4 56 6 1 3*], значения индексов которых делятся на без остатка на 3, используя цикл *for* и *while*.
13. Сформируйте список, значения элементов которого находятся в диапазоне от 23 до 35.

14. Сформируйте список, значения элементов которого находятся в диапазоне от 3 до 15 с шагом 4.

15. Сформируйте список, значения элементов которого находятся в диапазоне от 3 до 25 и без остатка делятся на 3.

16. Сформируйте словарь из двух списков [*3 0 1 3 0 4 3 3 4 56 6 1 3*] и [*2, 4, 7, 26, 33*], используя встроенную функцию *zip*. Выведите словарь в консоль и объясните, почему он получился такого вида.

17. Выведите различными способами в консоль элементы списка [*3 0 1 3 0 4 3 3 4 56 6 1 3*] с их индексами.

18. Напишите программу, которая считывает целое число (месяц), после чего выводит сезон к которому этот месяц относится.

19. Напишите программу, выводящую среднее из трех значений.

20. Напишите программу, выводящую таблицу умножения для задаваемого пользователем числа от 1 до 9 (включительно).

Тема 3
ФУНКЦИИ В PYTHON

В результате изучения данной темы обучающиеся должны:

знать

- принципы объявления и использования функций,
- механизм замыканий и правило LEGB,
- принцип работы с рекурсивными функциями,
- чем отличаются генераторы от списковых включений;

уметь

- применять механизм декорирования функций,
- использовать генераторные функции и выражения при работе с циклом *for*,
- передавать аргументы в функцию и получать возвращаемый ею результат;

владеть

- навыками написания приложений с использованием функций,
- навыками использования декораторов и генераторов в процессе написания приложений.

Функции в Python относятся к объектам первого класса. Их можно присваивать переменным, хранить в структурах данных, передавать в качестве аргументов другим функциям и даже возвращать в качестве значений из других функций [33]. Они могут принимать произвольное количество аргументов или не принимать их вовсе.

Функции позволяют использовать код, заключенный в них, сколько угодно раз в процессе выполнения программы и являются альтернативой любимого метода начинающих программистов: «copy&paste». Они также позволяют разбивать сложные системы на составные части, что упрощает процесс проектирования систем.

В табл. 3.1 приведены примеры всех операторов и выражений, которые тем или иным образом связаны с функциями.

Таблица 3.1

Операторы и выражения, связанные с функциями

Оператор или выражение	Пример
Выражения вызовов	`my_test_func('text', meat=ham, *rest)`
def	`def print_text(message) :` `  print('Hi, ' + message)`

Оператор или выражение	Пример
return	`def add_test(a, b=1, *c):` `  return a + b + c[0]`

global	`x = 'old'`
	`def changer():`
	`  global x; x = 'new'`

nonlocal	`def outer():`
	`  x = 'old'`
	`  def changer ():`
	`  nonlocal x; x = 'new'`

yield	`def test(x):` `  for i in range (x): yield i ** 2`
lambda	`funcs = [lambda x: x**2, lambda x: x**3]`

Функции и методы класса в Python начинаются с оператора *def*. В отличие от функций в компилируемых языках, таких как C, *def* представляет собой исполняемый оператор. Это означает, что написанная в коде функция не существует до тех пор, пока интерпретатор Python не встретит и не выполнит ее. Функции можно вкладывать внутрь операторов *if*, циклов *while* и даже других функций. При типовом применении операторы *def* помещаются в файлы модулей и естественным образом выполняются для генерации функций, когда файл модуля, где они находятся, импортируется в первый раз.

Когда интерпретатор Python достигает оператора *def* и выполняет его, генерируется новый объект, который присваивается имени функции. Как и со всеми присваиваниями, имя функции становится ссылкой на объект функции. Объекты функций также могут иметь произвольные определяемые пользователем атрибуты, присоединяемые к ним для регистрации данных.

Помимо этого, функции можно также создавать с помощью *lambda*-выражений — средства, которое позволяет встраивать определения функций в места, где оператор *def* синтаксически не допускается.

Когда функция вызывается, вызывающий код приостанавливается до тех пор, пока функция не завершит свою работу и не возвратит управление обратно. Функции, которые вычисляют значение, возвращают его вызывающему коду посредством оператора *return* (*return* без значения просто возвращает управление в вызывающий код и отправляет стандартный результат *None*).

yield — выражение, но оно обычно используется как оператор и отправляет результирующий объект вызывающему коду, при этом запоминая место, где он остановился. В функциях, известных как генераторы, можно также использовать *yield*, чтобы посылать об-

ратно значение и предохранять их состояние, так что они смогут возобновлять работу позже для производства серии результатов.

Оператор *global* объявляет переменные уровня модуля, предназначенные для присваивания. По умолчанию все объявляемые переменные в функции являются локальными и существуют только во время ее выполнения. Чтобы присвоить значение переменной из включающего модуля, необходимо расширить ее область видимости посредством оператора *global* внутри функции.

Оператор *nonlocal* объявляет переменные включающей функции, предназначенные для присваивания. Проще говоря, он позволяет функции присваивать значения переменной, которая уже существует в области видимости функции более высокого порядка, которая включает в себя текущую функцию.

3.1. Области видимости

Помимо упаковки кода, для многократного использования функции добавляют к программам дополнительный уровень, чтобы свести к минимуму возможность возникновения конфликтов между переменными с одним и тем же именем — по умолчанию все имена, которым выполнено присваивание внутри функции, ассоциируются с пространством имен данной функции и никаким другим:

— имена, присвоенные внутри *def*, могут быть видны только внутри этого оператора *def*. Ссылаться на такие имена извне функции нельзя;

— имена, присвоенные внутри *def*, не конфликтуют с переменными за пределами *def*, даже если те же самые имена применяются где-то в другом месте.

Переменные могут присваиваться в трех разных местах, соответствующих трем разным областям видимости:

— если переменная присваивается внутри *def*, то она будет локальной в этой функции;

— если переменная присваивается в объемлющем *def*, тогда она будет нелокальной в отношении вложенных функций;

— если переменная присваивается за пределами всех *def*, то она будет глобальной в целом файле.

Приведем описание схемы распознавания имен Python, которая иногда называется правилом LEGB, согласно названиям областей видимости.

1. Когда внутри функции указывается неуточненное имя, Python ищет его максимум в четырех местах — в локальной (L, *local*) области видимости, затем в локальных областях видимости любых объемлющих (E, *enclosing*) операторов *def* и *lambda*, далее в глобальной (G, *global*) области видимости и, наконец, во встроенной (B, *built-in*) области видимости. Поиск останавливается на первом же месте, где

обнаруживается имя. Если в результате такого поиска имя найти не удалось, Python сообщит об ошибке.

2. Когда внутри функции выполняется присваивание имени, Python всегда создает либо изменяет имя в локальной области видимости, если только оно не было объявлено в этой функции как глобальное или нелокальное.

3. Когда производится присваивание имени за пределами всех функций (т. е. на верхнем уровне файла модуля или в интерактивной подсказке), локальная область видимости совпадает с глобальной — пространством имен модуля.

Данные области видимости проиллюстрированы на рис. 3.1.

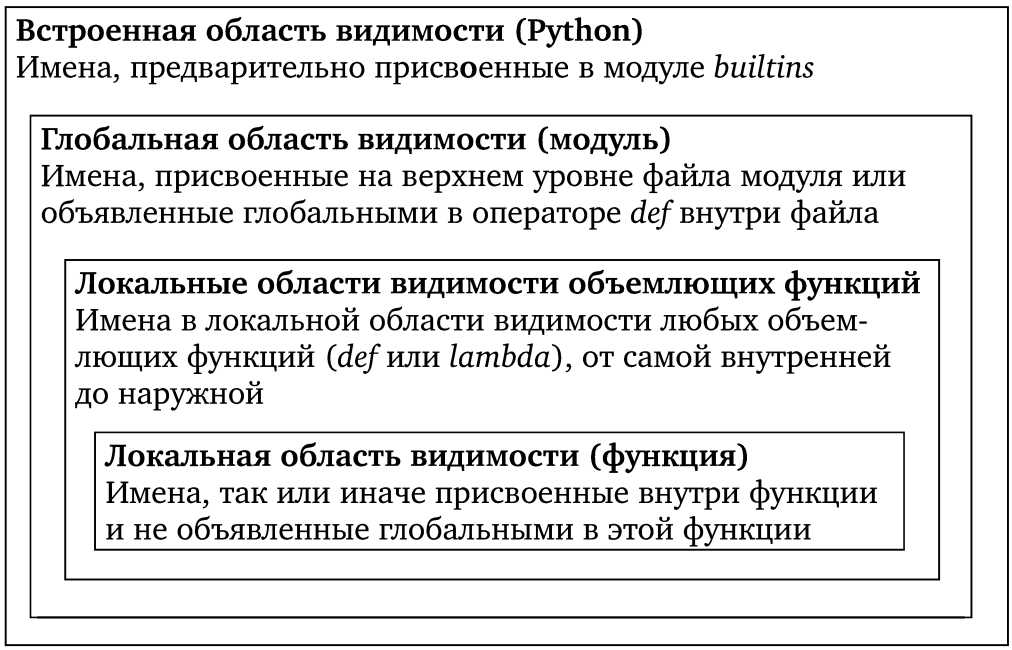

Рис. 3.1. **Области видимости Python**

В дополнение к приведенным выше областям видимости существует еще три: временные переменные циклов в ряде включений, переменные ссылок на исключения в некоторых обработчиках *try* и локальные области видимости в операторах *class*.

3.1.1. Области видимости и вложенные функции

Каждая вложенная функция имеет свою локальную (вложенную) область видимости, и с их добавлением слегка усложняются правила поиска переменных.

1. Ссылка (X) ищет имя X сначала в текущей локальной области видимости (функция); затем в локальных областях видимости любых лексически объемлющих функций в исходном коде, от внутренней до наружной; далее в текущей глобальной области видимости (файл модуля); и, наконец, во встроенной области видимости (модуль *builtins*). Объявления *global* заставляют поиск начинаться в глобальной области видимости (файл модуля).

2. Присваивание (X = значение) по умолчанию создает либо изменяет имя X в текущей локальной области видимости. Если имя X объявлено глобальным внутри функции, тогда присваивание создает или изменяет имя X в области видимости включающего модуля. Если же имя X объявлено нелокальным внутри функции, то присваивание изменяет имя X в локальной области видимости ближайшей объемлющей функции.

3.1.2. Пример работы с областями видимости

Для начала приведем простой пример, где X — глобальная переменная и функция производит суммирование передаваемого ей значения со значением этой переменной:

```
X = 4
def test_func(y):
    print(X + y)
test_func(4) # 8
```

Как видим из кода, все работает по описанным выше принципам и, так как переменная X не была объявлена в области видимости функции, Python обращается при ее поиске к области видимости более высокого порядка.

Теперь добавим в функцию локальную переменную X = 3 и посмотрим, что изменится:

```
X = 4
def test_func(y):
    X = 3
    print(X + y)
test_func(4) # 7
print(X) # 4
```

Так как переменная X была объявлена локально и нет указаний рассматривать ее как глобальную переменную (оператор *global*), то Python будет считать, что нашел необходимую для выражения переменную с именем X.

Рассмотрим, как работает оператор *global*:

```
X = 4
def test_func(y):
    global X
    X = 3
    print(X + y)
test_func(4) # 7
print(X) # 3
```

Оператор *global* дает указание интерпретатору Python начинать поиск переменной X с глобальной области видимости. Потом происходит изменение значения этой переменной, что можно заметить из результирующего вывода программы.

3.1.3. Замыкания

Суть данного подхода заключается в том, что указанный объект функции помнит значения из объемлющих областей видимости, невзирая на то, присутствуют ли еще эти области видимости в памяти. Фактически объект имеет присоединенные пакеты данных из памяти (известные как сохранение состояния), которые локальны для каждой созданной копии вложенной функции и часто выступают в качестве простой альтернативы классам в такой роли.

Замыкания иногда применяются в программах, которым необходимо генерировать обработчики событий на лету в ответ на условия, сложившиеся во время выполнения. Например, представим себе графический пользовательский интерфейс, где нужно определять действия согласно данным, введенным пользователем, которые невозможно предугадать на этапе построения интерфейса. В таких ситуациях нам требуется функция, создающая и возвращающая другую функцию с информацией, которая может варьироваться в зависимости от создаваемой функции.

Ниже представлен пример работы с замыканиями:

```
def maker(N):
    def action(X):  # Создание и возвращение функции action
        return X ** N
    return action

my_func = maker(3)
print(my_func(2)) # 8
print(my_func(4)) # 64
print(my_func(5)) # 125
```

В приведенном примере функция **action** сохраняет значение N из объемлющей области видимости (т. е. области видимости функции **maker**).

Также замыкание можно сделать с использованием *lambda*-функции, которую будет возвращать основная (оборачивающая) функция:

```
def maker(N):
    return lambda X: X ** N
my_func = maker(3)
print(my_func(2)) # 8
print(my_func(4)) # 64
print(my_func(5)) # 125
```

Рассмотрим принцип работы оператора *nonlocal* на следующем примере кода:

```
def counter(start):
    state = start
    def adder(X):
        print(X + state)
    return adder
```

```
F = counter(5)
F(3) # 8
F(5) # 10
```

В тех случаях, когда мы захотим в процессе выполнения замыкания изменить значение переменной *state*, при выполнении кода будет сгенерировано следующее исключение:

```
def counter(start):
    state = start
    def adder(X):
        print(X + state)
        state += 1
    return adder
F = counter(0)
F(3)
"""Traceback (most recent call last):
  File "F:/code/python/myProjRep/main.py", line 9, in <module>
    F(3)
  File "F:/code/python/myProjRep/main.py", line 4, in adder
    print(X + state)
UnboundLocalError: local variable 'state' referenced before
assignment"""
```

Это связано с тем, что изменение значения из области видимости объемлющей функции по умолчанию запрещено. Для того, чтобы обойти этот запрет, и используется оператор *nonlocal*:

```
def counter(start):
    state = start
    def adder(X):
        nonlocal state
        print(X + state)
       state += 1
    return adder
F = counter(5)
F(3) # 8
F(5) # 11
```

Следует отметить тот факт, что нелокальные (*nonlocal*) переменные уже должны существовать в области видимости объемлющей функции (*def*), иначе произойдет ошибка во время работы программы:

```
def counter(start):
    def adder(X):
        nonlocal state
        state = start
        print(X + state)
        state += 1
    return adder
F = counter(5)
File "F:/code/python/myProjRep/main.py", line 3
    nonlocal state
    ^
SyntaxError: no binding for nonlocal 'state' found
```

Кроме того, оператор *nonlocal* ограничивает область поиска переменной только областями видимостей объемлющих функций (*def*):

```
state = 10
def counter():
    def adder(X):
        nonlocal state
        print(X + state)
        state += 1
    return adder
F = counter()
File "F:/code/python/myProjRep/main.py", line 4
    nonlocal state
    ^
SyntaxError: no binding for nonlocal 'state' found
```

3.2. Аргументы функции

При передаче аргументов в функции необходимо запомнить следующие ключевые моменты.

1. Аргументы передаются в функцию путем автоматического присваивания объектов именам локальных переменных. Аргументы функций, т. е. ссылки на (возможно) разделяемые объекты, отправленные вызывающим кодом, представляют собой еще один пример присваивания в Python. Поскольку ссылки реализованы в виде указателей, все аргументы в действительности передаются через указатели, в связи с чем объекты, передаваемые как аргументы никогда автоматически не копируются.
2. Когда функция выполняется, имена аргументов в заголовке функции становятся новыми локальными именами в области видимости функции. Никакого совмещения имен аргументов функции и имен переменных в области видимости вызывающего кода не происходит.
3. При модификации внутри функции передаваемого ей аргумента, являющегося изменяемым объектом, его значение изменится и в вызывающем коде.

Разница в передаче изменяемых и неизменяемых объектов в качестве аргументов в функции заключается в том, что:

— неизменяемые аргументы фактически передаются «по значению». Объекты, подобные целым числам и строкам, передаются по ссылке на объекты, а не путем копирования, но из-за того, что модифицировать на месте неизменяемые объекты невозможно, эффект во многом похож на создание копий;

— изменяемые аргументы фактически передаются «по указателю». Объекты вроде списков и словарей также передаются по ссыл-

ке на объекты, что аналогично способу передачи массивов как указателей в языке C, и их можно модифицировать на месте.

Для рассмотрения особенностей передачи объектов в функцию используем следующий код:

```
def test(a, b, c):
    print('a = {}; b = {}; c = {}'.format(a, b, c))

x, y, z = (1, [2, 10], 'Hi!')
test(x, y, z)  # a = 1; b = [2, 10]; c = Hi!
print('x = {}; y = {}; z = {}'.format(x, y, z))
# x = 1; y = [2, 10]; z = Hi!
```

Как уже обозначили в предыдущем разделе, аргументам функции присваиваются ссылки на объекты передаваемых на ее вход имен переменных. Более наглядно это показано на рис. 3.2.

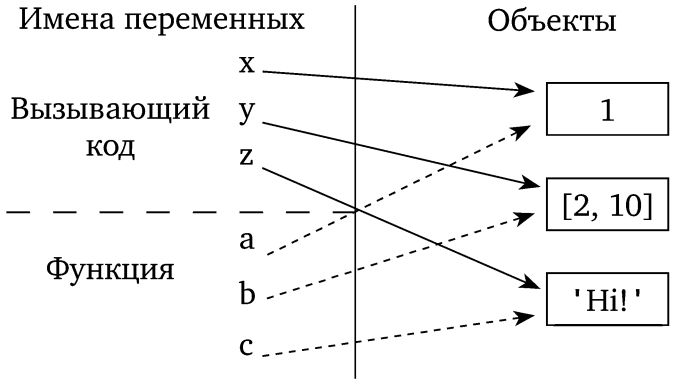

Рис. 3.2. **Присваивание ссылок на объекты в функции**

Из рис. 3.2 видно, что независимо от того, изменяемый или неизменяемый тип данных используется в качестве аргументов функции, в случае ее вызова аргументам присваиваются ссылки на объекты из вызывающей области. Чтобы убедиться в этом окончательно, используем оператор *id*(), который возвращает уникальный идентификатор объекта, присваиваемый в момент выделения под него область памяти:

```
def test(a, b, c):
    print('a = {}; b = {}; c = {}'.format(id(a),
          id(b), id(c)))

x, y, z = (1, [2, 10], 'Hi!')
test(x, y, z)
# a = 140721789706640; b = 2740459779272;
# c = 2740486981488
print('x = {}; y = {}; z = {}'.format(id(x),
       id(y), id(z)))
# x = 140721789706640; y = 2740459779272;
# z = 2740486981488
```

Теперь попробуем изменить значения объектов в функции и посмотрим, что из этого получится:

```
def test(a, b, c):
    a = 5
    b[0] = 'Oo'
    c = '(-_-)'
    print('a = {}; b = {}; c = {}'.format(a, b, c))
    print('a = {}; b = {}; c = {}'.format(id(a),
          id(b), id(c)))

x, y, z = (1, [2, 10], 'Hi!')
test(x, y, z) # a = 5; b = ['Oo', 10]; c = (-_-)
# a = 140721789706768; b = 2740487405256;
# c = 2740482508848
print('x = {}; y = {}; z = {}'.format(x, y, z))
# x = 1; y = ['Oo', 10]; z = Hi!
print('x = {}; y = {}; z = {}'.format(id(x), id(y), id(z)))
# x = 140721789706640; y = 2740487405256;
# z = 2740479988528
```

В случае присваивания нового значения переменной, которая хранит ссылку на объект неизменяемого типа, создается новый объект, на который теперь и будет ссылаться данная переменная. Это видно по работе с аргументами функции, такими как «a» и «c». При изменении значения объекта изменяемого типа новый объект создаваться не будет (аргумент функции «b»).

На рис. 3.3 приведен пример описанного выше принципа работы Python.

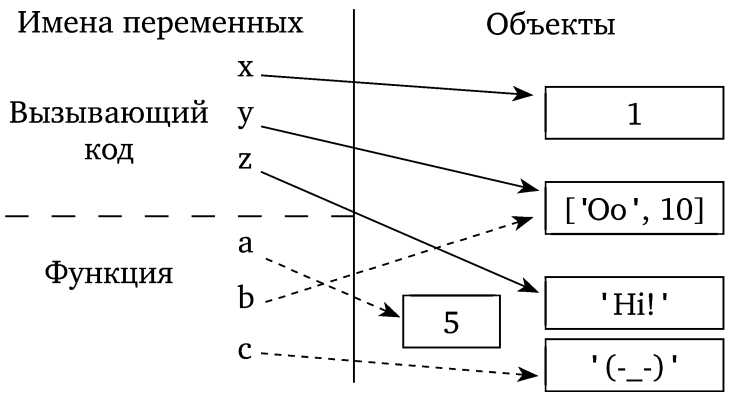

Рис. 3.3. **Изменение значений аргументов функции**

Такой же принцип работы используется и при присваивании переменным уже существующих значений:

```
x, y, z = 1, [2, 10], 'Hi!'
a, b, c = x, y, z
print('a = {}; b = {}; c = {}'.format(id(a), id(b), id(c)))
# a = 140721789706640; b = 2740487423880;
# c = 2740485781232
print('x = {}; y = {}; z = {}'.format(id(x), id(y), id(z)))
```

```
# x = 140721789706640; y = 2740487423880;
# z = 2740485781232
a = 5
b[0] = 'Oo'
c = '(-_-)'
print('a = {}; b = {}; c = {}'.format(a, b, c))
# a = 5; b = ['Oo', 10]; c = (-_-)
print('x = {}; y = {}; z = {}'.format(x, y, z))
# x = 1; y = ['Oo', 10]; z = Hi!
print('a = {}; b = {}; c = {}'.format(id(a), id(b), id(c)))
# a = 140721789706768; b = 2740487423880;
# c = 2740481511792
print('x = {}; y = {}; z = {}'.format(id(x), id(y), id(z)))
# x = 140721789706640; y = 2740487423880;
# z = 2740485781232
```

При присваивании переменным маленьких целочисленных значений (от –5 до 256) оператор *id*() будет возвращать одно и то же значение, даже в том случае, когда новая переменная не ссылается на уже объявленный ранее объект. Это связано с оптимизацией CPython, где для улучшения производительности небольшие числа представляются одним объектом:

```
one = 4
two = one
three = 4
print('one = {}; two = {}; three = {}'.format(
                      id(one), id(two), id(three)))
# one = 140721789706736; two = 140721789706736;
# three = 140721789706736
print(id(one) == id(two) == id(three)) # True
```

3.2.1. Значения аргументов по умолчанию

Python, как и ряд других языков программирования, поддерживает вызов функций и методов класса с их аргументами, значения которых заданы по умолчанию. Для этого при объявлении функции необходимо ее аргументу явно присвоить значение:

```
def test_default(a=10):
    print(a)
test_default('Hi!') # Hi!
test_default() # 10
```

Наличие в функции аргументов, которые имеют значения по умолчанию, позволяет производить ее вызов с меньшим числом параметров:

```
def test_default2(a=10, b=12):
    print(a+b)

test_default2() # 22
test_default2(4) # 16
test_default2(4,2) # 6
```

Из приведенных выше примеров видно, что данные аргументы не являются обязательными при вызове функции. Но если среди аргументов должны быть обязательные, которые необходимо передавать ей на вход при каждом вызове, их необходимо указывать в начале списка аргументов:

```
def test_default2(a, b=12):
    print(a+b)
test_default2(4) # 16
test_default2(4,2) # 6
```

Если первыми аргументами поставить необязательные (значения которых заданы по умолчанию), а следом за ними обязательные аргументы, то интерпретатор Python сообщит об ошибке:

```
def test_default2(b=12, a):
    print(a+b)

test_default2(4,2)
  File "", line 1
    def test_default2(b=12, a):
                      ^
SyntaxError: non-default argument follows default argument
```

Правила расстановки обязательных и необязательных аргументов в функции рассматриваются в следующем параграфе.

3.2.2. Режимы сопоставления аргументов функции

Помимо уже рассмотренных ранее способов передачи аргументов в функции, Python предлагает дополнительные инструменты, которые позволяют функции принимать необязательные аргументы, благодаря чему можно создавать гибкие API в модулях и классах.

Базовый вид такого объявления функции можно записать следующим образом:

```
def func_test([arg1, ..., argN, *args, kw1, ...,
               kwN, **kwargs]):
    ...
```

где в квадратные скобки заключены необязательные элементы, arg1 — argN — позиционные аргументы, которые должны следовать друг за другом; *args — итерируемая последовательность позиционных аргументов; kw1 — kwN — ключевые аргументы (аргументы, значения которых задаются по умолчанию); **kwargs — словарь ключевых элементов.

Далее рассмотрены примеры объявления функций с различными способами сопоставления их аргументов и объектов вызывающего кода.

1. Позиционная передача аргументов (сопоставляются слева направо).

```
def custom_function(a, b, c):
    print(a + b + c)
custom_function(10,4,8) # 22
```

При таком подходе выполняется сопоставление переданных значений аргументов с именами аргументов в заголовке функции по позиции, слева направо.

2. Передача по имени аргумента (сопоставляются по имени аргумента).

```
def custom_function(a, b, c=2):
    print(a + b + c)
custom_function(b=2, c=1, a=4) # 7
```

В качестве альтернативы позиционному способу сопоставления в вызывающем коде можно указывать, какой аргумент в функции получает значение посредством синтаксиса «имя=значение».

3. Переменное количество позиционных аргументов (сначала сопоставляются единичные позиционные элементы, потом итерируемый список позиционных элементов, а далее аргументы сопоставляются по имени).

```
def custom_function(a, b, c, *args):
    print(a + b + c + sum(*args))
custom_function(1, 1, 1, [10, 20, 30])
# 63

custom_function(a=1, c=4, b=11, [10, 20, 30])
""" File "", line 5
    custom_function(a=1, c=4, b=11, [10, 20, 30])
                                    ^
SyntaxError: positional argument follows keyword argument"""

def custom_function(a, b, c, *args, d):
    print(a + b + c + sum(*args) - d)
custom_function(1, 1, 1, [10, 20, 30], d=4) # 59
custom_function(1, 1, 1, [10, 20, 30], 4)
"""--------------
TypeError                                 Traceback (most
recent call last)
 in
      2     print(a + b + c + sum(*args)-d)
      3
----> 4 custom_function(1, 1, 1, [10, 20, 30], 4)

TypeError: custom_function() missing 1 required keyword-only
argument: 'd' """
```

4. Передача произвольного количества позиционных и ключевых элементов.

```
def custom_function4(*args, **kwargs):
    if args:
        print(args)
    if kwargs:
        print(kwargs)
custom_function4(10, 20, key='Hi', Oo = 100)
# (10, 20)
# {'key': 'Hi', 'Oo': 100}
```

При этом стоит учитывать, что имена аргументов *args и **kwargs не является обязательным. Например:

```
def custom_function4(*a, **b):
    if a:
        print(a)
    if b:
        print(b)
custom_function4(10, 20, key='Hi', Oo = 100)
# (10, 20)
# {'key': 'Hi', 'Oo': 100}
```

5. Передача произвольного числа ключевых аргументов.

```
def custom_function5(**kwargs):
    print(kwargs)
custom_function5(key='Hi', Oo = 100)
# {'key': 'Hi', 'Oo': 100}
```

Шаги, выполняемые Python для сопоставления аргументов перед присваиванием, могут быть грубо описаны следующим образом:

1) присваивание не ключевых аргументов по позиции;
2) присваивание ключевых аргументов по совпадающим именам;
3) присваивание добавочных не ключевых аргументов кортежу *name;
4) присваивание добавочных ключевых аргументов словарю **name;
5) присваивание стандартных значений не присвоенным аргументам в заголовке.

Затем Python проверяет, передается ли каждому аргументу только одно значение, и если нет, возникает ошибка. Когда сопоставление завершено, Python присваивает именам аргументов переданные для них объекты.

3.3. Возвращение результатов выполнения функцией

В случае передачи в качестве аргумента функции переменной изменяемого типа необходимо быть осторожным и не менять значение аргумента без необходимости. В ряде случаев бывает лучше скопировать значения объекта, на который ссылается аргумент, и присвоить полученный объект новой переменной, с которой в дальнейшем и будет производиться работа в функции. В то же самое время это свойство изменяемых типов можно использовать, когда необходимо изменить значение передаваемого объекта в качестве аргумента функции и в вызывающем коде продолжить работать с измененным значением.

Для возвращения результатов работы функции в вызывающий ее код используется оператор *return*. При его отсутствии в теле функции по умолчанию возвращается None:

```
def test_return(a, b):
    c = a + b

print(test_return(2, 4)) # None
```

Следующий пример возвращает в вызывающий код результат суммы аргументов функции:

```
def test_return(a, b):
    return a + b

print(test_return(2, 4)) # 6
```

Бывают случаи, когда нужно вернуть не одно, а несколько значений. Здесь на помощь приходят кортежи:

```
def test_return(a, b):
    c = a + b
    m = a * b
    e = a ** b
    d = a / b
    s = a - b
    return c, m, e, d, s

print(test_return(4, 2)) # (6, 8, 16, 2.0, 2)
```

Чтобы присвоить значения из возвращаемого функцией кортежа переменным, можно применить следующие методы распаковки последовательностей:

```
x, y, q, w, z = test_return(4, 2)
print(x, y, q, w, z) # 6 8 16 2.0 2 #1
_, *my_list, _ = test_return(4, 2)
print(my_list) # [8, 16, 2.0] #2
y, *my_list, q = test_return(4, 2)
print(y, my_list, q) # 6 [8, 16, 2.0] 2 #3
w, _, _, *my_list = test_return(4, 2)
print(w, my_list) # 6 [2.0, 2] #4
```

Символ «_» в данном случае используется как одноразовое имя переменной и действует в качестве заполнителя для тех элементов последовательности (кортежа), которые нам безразличны.

Когда нет уверенности, сколько элементов будет содержать возвращаемый функцией кортеж, то при его распаковке можно использовать переменную, которая соберет список значений. Это делается путем размещения звездочки перед ее именем.

В примере «#1» все элементы кортежа распаковались и присвоились переменным именно в той последовательности, в которой располагались в нем. В примере «#2» отбрасываются крайние элементы кортежа и распаковываются в список элементы, располагающиеся

между ними. В примере «#3» крайние значения крайних элементов записываются в переменные с именами «y» и «q», а в остальном распаковка ведет себя, как и в предыдущем примере. Четвертый пример тем интересен, что отбрасываются при распаковке второй и третий элемент кортежа.

Если при распаковке не используется подход сбора части элементов последовательности в список, то главное — придерживаться правила, что количество переменных, которым присваиваются значения распаковываемой последовательности, должны равняться количеству элементов этой самой последовательности:

```
w, _ = test_return(4, 2)
print(w, my_list)
"""-----------------
ValueError                                Traceback (most
recent call last)
 in
----> 1 w, _ = test_return(4, 2)
      2 print(w, my_list)

ValueError: too many values to unpack (expected 2) """
```

Также запрещается использовать несколько имен переменных с приставкой в виде символа «*»:

```
*first, *second = test_return(4, 2)
  File "", line 4
SyntaxError: two starred expressions in assignment
```

Вместо кортежей можно также использовать словарь.

3.4. Рекурсия

Рекурсивные функции — функции, которые вызывают самих себя либо прямо, либо косвенно с целью организации цикла. Рекурсия относительно редко используется при написании кода в Python, отчасти из-за того, что процедурные операторы включают более простые циклические структуры.

Рекурсия позволяет программам обходить структуры, которые имеют произвольную непредсказуемую форму и глубину, например, при планировании маршрутов в путешествии, анализе языка и т. д.

Рассмотрим различные варианты написания рекурсивной функции на примере задачи суммирования элементов последовательности:

```
def my_sum (L) :
    print (L)
    if not L:
        return 0
    else:
        return L[0] + my_sum(L[1:])
```

```
print(my_sum([1, 2, 3, 4, 5, 20, 30]))
"""
[1, 2, 3, 4, 5, 20, 30]
[2, 3, 4, 5, 20, 30]
[3, 4, 5, 20, 30]
[4, 5, 20, 30]
[5, 20, 30]
[20, 30]
[30]
[]
65
"""
```

Этот же код можно записать с использованием тернарного выражения *if/else*:

```
def my_sum1(L):
    return 0 if not L else L[0] + my_sum1(L[1:])
print(my_sum1([1, 2, 3, 4, 5, 20, 30])) # 65

def my_sum2(L):
    return L[0] if len(L) == 1 else L[0] +
                                my_sum2(L[1:])
print(my_sum2([1, 2, 3, 4, 5, 20, 30])) # 65

def my_sum3(L):
    first, *rest = L
    return first if not rest else first +
                                    my_sum3(rest)
print(my_sum3([1, 2, 3, 4, 5, 20, 30])) # 65
```

Как уже отмечалось, рекурсия может быть прямой или косвенной. Ранее рассматривался прямой вызов функцией самой себя. Косвенная рекурсия организуется следующим образом:

```
def my_sum4(L) :
    if not L: return 0
    return nonempty(L)

def nonempty(L) :
    return L[0] + my_sum4(L[1:])

print(my_sum4([1, 2, 3, 4, 5, 20, 30])) # 65
```

В случаях, когда не получается выйти из рекурсии определенное количество шагов, Python прервет выполнение кода и оповестит пользователя об исчерпании лимита рекурсивных вызовов функции.

3.5. Аннотация функций

Начиная с версии 3.6 функциям можно задавать типы аргументов и тип возвращаемого значения. Синтаксически аннотация функции записывается в строке ее заголовка. Для аргументов анно-

тация указывается после двоеточия, следующего непосредственно за его именем, а для возвращаемых значений после символа «->», следующих за списком аргументов:

```
def sum_ab(a: int, b: float) -> float:
    return a+b
```

Вызывается аннотированная функция, как и обычная, но при наличии аннотаций Python накапливает их в словаре и присоединяет его к самому объекту функции посредством атрибута *__annotations__*:

```
print(sum_ab.__annotations__)
# {'a': <class 'int'>, 'b': <class 'float'>,
# 'return': <class 'float'>}
```

Как видно из приведенного примера, имена аргументов и аннотация возвращаемого значения, если имеется (сохраняется под ключом *return*), записываются в качестве ключа словаря, а в качестве значений записываются выражения аннотации.

Поскольку аннотации являются всего лишь объектами Python (словарем), присоединенными к другому объекту (функции), пройтись по их элементам можно следующим способом:

```
for arg in sum_ab.__annotations__:
    print(arg, '=>', sum_ab.__annotations__[arg])
# a => <class 'int'>
# b => <class 'float'>
# return => <class 'float'>
```

Но аннотации функций могут содержать не только типы аргументов и возвращаемого значения, ведь это только один из способов их использования. Согласно PEP8, проверка типов в Python не обязательна и является отдельным инструментом. По умолчанию интерпретаторы Python не должны выдавать никаких сообщений проверки типов и не должны изменять свое поведение на основе аннотаций.

Перепишем аннотацию объявленной ранее функции следующим образом:

```
def sum_ab(a: 'what?', b: 99, c) -> 'holy python!!!':
    return a+b
print(sum_ab.__annotations__)
# {'a': 'what?', 'b': 99, 'return': 'holy python!!!'}
for arg in sum_ab.__annotations__:
    print(arg, '=>', sum_ab.__annotations__[arg])
# a => what?
# b => 99
# return => holy python!!!
```

Или так:

```
def sum_ab(a:'what?'=2, b:(4, 5)=99, c:float=3.5) -> 'holy
python!!!':
    return a+b+c
print(sum_ab.__annotations__)
```

```
# {'a': 'what?', 'b': (4, 5), 'c': <class 'float'>,
# 'return': 'holy python!!!'}
for arg in sum_ab.__annotations__:
    print(arg, '=>', sum_ab.__annotations__[arg])
# a => what?
# b => (4, 5)
# c => <class 'float'>
# return => holy python!!!
```

В показанном выше коде запись аргумента и его аннотация «a:'what?'=2» означают, что аргумент «a» имеет стандартное значение 4 и аннотирован строкой *'what?'*. Аргумент «b» аннотирован кортежем с элементами (4, 5) и т. д.

Таким образом, эту функцию можно спокойно вызывать со значениями по умолчанию или передавать на ее вход значения аргументов:

```
print(sum_ab()) # 104.5
print(sum_ab(1,3)) # 7.5
print(sum_ab(c=-5)) # 96
```

3.6. Лямбда-функции (выражения)

Ключевое слово **lambda** в Python предоставляет краткую форму для объявления небольших анонимных функций. Лямбда-функции ведут себя точно так же, как обычные функции, объявляемые ключевым словом **def**, и могут использоваться всякий раз, когда требуются объекты-функции.

Например, следующий код:

```
my_add = lambda x, y: x+y
print(my_add(1, 2)) # 3
```

эквивалентен:

```
def my_add(x, y):
    return x+y
print(my_add(1, 2)) # 3
```

Концептуально лямбда-выражение

```
lambda x, y: x + y
```

аналогично объявлению функции при помощи ключевого слова **def**, только записывается в одну строку. Основное отличие здесь в том, что перед использованием не производится связывание объекта-функции с ее именем.

Ввиду этого приведенное выше лямбда-выражение можно выполнить сразу же при объявлении:

```
print((lambda x, y: x+y)(1, 2)) # 3
```

Следующее различие между определениями лямбд и обычных функций заключается в том, что лямбда-функция ограничена одним-единственным выражением. То есть в них не могут применяться инструкции или аннотации (даже инструкция *return*). При этом всегда существует неявная инструкция *return*, которая автоматически возвращает результат выражения.

Этот обеспечивает удобную и краткую форму для определения функции в Python там, где одним из аргументов функции или метода является объект-функция. Примером может служить написание кратких и сжатых функций для сортировки итерируемых объектов по альтернативному ключу, где к каждому элементу последовательности применяется лямбда-функция:

```
list_tuples = [('d', 4), ('b', 2), ('a', 1),
               ('c', 3), ('a', 0)]
print(sorted(list_tuples))
#[('a', 0), ('a', 1), ('b', 2), ('c', 3), ('d', 4)]
print(sorted(list_tuples, key=lambda x: x[1]))
#[('a', 0), ('a', 1), ('b', 2), ('c', 3), ('d', 4)]
print(sorted(list_tuples, key=lambda x: x[0]))
#[('a', 1), ('a', 0), ('b', 2), ('c', 3), ('d', 4)]
```

Или можно использовать так:

```
print(sorted(range(-5, 6), key=lambda x: x * x))
# [0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
print(sorted(range(-5, 6), key=lambda x: x ** x))
# [-1, -3, -5, -4, -2, 0, 1, 2, 3, 4, 5]
```

Если с лямбда-функцией получается сложный (или близкий к этому) код, то лучше использовать обычную функцию с подходящим именем.

3.7. Декораторы

Декораторы в Python позволяют расширять и изменять поведение вызываемых объектов (функций, методов и классов) без необратимой модификации самих вызываемых объектов. Поэтому любая достаточно универсальная функциональность, которую можно присоединить к разрабатываемому классу или функции, является отличным кандидатом для декорирования:

— ведение протокола операций (логирование);
— контроль за доступом и аутентификацией;
— хронометраж;
— ограничение частоты вызова API;
— кэширование и др.

В основе декораторов лежит нескольких продвинутых концепций Python, включая свойства функций первого класса:

1) функции являются объектами — их можно присваивать переменным, передавать в другие функции и возвращать из других функций;

2) функции могут быть определены внутри других функций — и дочерняя функция может захватывать локальное состояние родительской функции (лексические замыкания).

Самый простой декоратор можно записать следующим образом:

```
def free_decorator(function):
    return function
```

Реализованный выше декоратор — это по сути вызываемый объект (функция), который на входе принимает еще один вызываемый объект и возвращает тот же самый вызываемый объект без его изменения. Чтобы показать, как происходит декорирование, объявим еще одну функцию и обернем ее в уже реализованный ранее декоратор:

```
def my_func():
    return "It's work!!!"
my_func = free_decorator(my_func)
my_func() # "It's work!!!"
```

Python предоставляет дополнительный синтаксис «@» для декорирования функции. Благодаря ему нет надобности явным образом вызывать декоратор с передаваемой ему с функцией и затем снова присваивать его переменной:

```
@free_decorator
def my_func():
    return "It's work!!!"
my_func() # "It's work!!!"
```

«@» декорирует функцию непосредственно во время ее определения, что делает очень трудным получение доступа к недекорированному оригиналу без различных «фокусов». Поэтому для сохранения способности вызывать как декорированную, так и исходную функцию используют «ручной» режим декорирования.

Далее рассмотрим пример того, как декоратор может менять поведение оборачиваемой в него функции. Для этого реализуем следующий декоратор:

```
def up_register(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper

@up_register
def my_func():
    return "It's work!!!"
my_func() # "IT'S WORK!!!"
```

Замыкание *wrapper* имеет доступ к недекорированной входной функции, и оно свободно может выполнить дополнительный программный код до и после ее вызова. Также, в отличие от предыдущего примера декоратора, текущий при декорировании функции возвращает не исходный, а другой объект-функцию. Таким образом, единственный способ повлиять на «будущее поведение» входной функции, которую декорируем, состоит в том, чтобы подменить (или обернуть) входную функцию замыканием.

Теперь рассмотрим применение нескольких декораторов к функции на примере HTML-разметки:

```
def h1_html(function):
    def wrapper():
        return '<h1>' + function() + '</h1>'
    return wrapper

def body(function):
    def wrapper():
        return '<body>' + function() + '</body>'
    return wrapper

@body
@h1_html
def my_func():
    return "It's work!!!"
print(my_func())
# <body><h1>It's work!!!</h1></body>
```

Результат этой функции ясно показывает, что декораторы были применены снизу вверх. То есть сначала входная функция была обернута декоратором *@h1_html*, и затем результирующая (декорированная) функция была обернута декоратором *@body*.

В «ручном» режиме декорирования такой подход выглядит следующим образом:

```
my_decfunc = body(h1_html(my_func))
print(my_decfunc ())
# <body><h1>It's work!!!</h1></body>
```

На практике в глубоких уровнях декорирования функций нет проблем, но это может сказаться на производительности при работе над вычислительно емким программным кодом, в котором декорирование применяется часто.

Но такой подход к декорированию не сработает, когда начальная функция имеет аргументы:

```
def body(function):
    def wrapper():
        return '<body>' + function() + '</body>'
    return wrapper

@body
def my_func(name):
```

```
    return name+", It's work!!!"
my_func('Alex')
"""----------------
TypeError                                 Traceback (most
recent call last)
 in
      7 def my_func(name):
      8     return name+", It's work!!!"
----> 9 my_func('Alex')

TypeError: wrapper() takes 0 positional arguments but 1 was
given """
```

Чтобы избежать этой ошибки, замыкающая функция также должна принимать аналогичный перечень аргументов:

```
def body(function):
    def wrapper(name):
        return '<body>' + function(name) + '</body>'
    return wrapper

@body
def my_func(name):
    return name+", It's work!!!"
my_func('Alex')
# <body>Alex, It's work!!!</body>
```

Далее рассмотрим, как декорировать функцию с произвольным количеством аргументов. В данном случае на помощь приходят такие функциональные средства языка Python как **args* и ***kwargs* для работы с неизвестными количествами аргументов:

```
def proxy(function):
    def wrapper(*args, **kwargs):
        return function(*args, **kwargs)
return wrapper
```

Теперь перепишем функцию **my_func** следующим образом:

```
def my_func(name = "Alex", line="It's work!!!"):
    return f'{name}, {line}'
```

и функцию **wrapper** так:

```
def proxy(func):
    def wrapper(*args, **kwargs):
        print(f'ТРАССИРОВКА: вызвана'
              f' {func.__name__}() '
              f'с {args}, {kwargs}')
        original_result = func(*args, **kwargs)
        print(f'ТРАССИРОВКА: {func.__name__}() '
                f'вернула {original_result!r}')
        return original_result
    return wrapper
```

При декорировании функции и последующем ее вызове с различным количеством аргументов и их заданием будут выведены следующие значения:

```
print(my_func())
# ТРАССИРОВКА: вызвана my_func() с (), {}
# ТРАССИРОВКА: my_func() вернула "Alex,It's work!!!"
# Alex, It's work!!!
print(my_func('Jon'))
# ТРАССИРОВКА: вызвана my_func() с ('Jon',), {}
# ТРАССИРОВКА: my_func() вернула "Jon, It's work!!!"
# Jon, It's work!!!
print(my_func(line="^_^", name = 'Jon'))
# ТРАССИРОВКА: вызвана my_func() с (),
# {'line': '^_^', 'name': 'Jon'}
# ТРАССИРОВКА: my_func() вернула 'Jon, ^_^'
# Jon, ^_^
print(my_func("Fine", "you win!!!"))
# ТРАССИРОВКА: вызвана my_func() с ('Fine',
# 'you win!!!'), {}
# ТРАССИРОВКА: my_func() вернула 'Fine, you win!!!'
# Fine, you win!!!
```

3.8. Генераторы

По своей сути генераторные функции (функции с оператором *yield*) или выражения (похожи на списковые включения) в Python являются синтаксическим сахаром для простого написания итератора и позволяют откладывать создание результатов всякий раз, когда это возможно. Из-за того, что ни та, ни другая конструкция не создает сразу весь результирующий список, они экономят пространство памяти и позволяют распределить время вычислений по запросам результатов.

3.8.1. Генераторные функции

Генераторные функции — это функции, которые способны отправлять обратно значение и позже возобновлять работу с места, на котором они остановились (генерируют последовательность значений с течением времени).

Генераторные функции во многих отношениях похожи на нормальные функции и фактически записываются с помощью обычных операторов. Однако при их создании они специально компилируются в объект, который поддерживает протокол итерации. В отличие от нормальных функций, которые возвращают значение и прекращают работу, генераторные функции автоматически приостанавливают и возобновляют свое выполнение и состояние во-

круг точки генерации значений. По этой причине они часто будут служить удобной альтернативой заблаговременному вычислению полной серии значений с последующим сохранением и восстановлением вручную состояния посредством классов. Состояние, которое генераторные функции сохраняют, когда приостанавливаются, содержит местоположение в коде и полную локальную область видимости. Следовательно, их локальные переменные помнят информацию между выпуском результатов и делают ее доступной при возобновлении выполнения функциями. В генераторных функциях вместо инструкции возврата *return* используется инструкция *yield*.

Рассмотрим принцип работы генераторной функции на примере генерации возрастающей последовательности чисел от нуля до 5:

```
def gen_test():
    N = 0
    while N < 5:
        yield N
        N += 1

for x in gen_test():
    print(x, end=' ')
# 0 1 2 3 4
```

Чтобы лучше понять принцип работы генераторов в Python, реализуем функцию, тело которой состоит только из *yield*, и результат работы не отличается от того, что приведен в примере выше:

```
def gen_surprise_test():
    yield 0
    yield 1
    yield 2
    yield 3
    yield 4

for x in gen_surprise_test():
    print(x, end=' ')
# 0 1 2 3 4
```

В ходе итерации сначала генераторная функция вернула значение ноль, запомнила точку в теле функции, из которой произошел возврат значения, и перешла в режим ожидания следующего ее вызова. На следующем шаге цикла генераторная функция начнет свою работу с того места, на котором остановилась в прошлый раз, вернет единицу, запомнит точку в теле функции, из которой произошел возврат значения, и перейдет в режим ожидания следующего ее вызова. И так до крайнего *yield* в функции. Как только поток управления достигает конца генераторной функции, порождается

исключение *StopIteration*, сигнализируя о том, что у нее больше нет значений, которые она могла бы предоставить:

```
iterator = gen_surprise_test()
print(iterator)
<generator object gen_surprise_test at 0x0000023BE25DF448>
print(next(iterator))
0
print(iterator.__next__()) # 1
print(next(iterator)) # 2
print(next(iterator)) # 3
print(next(iterator)) # 4
print(next(iterator)) # StopIteration
```

Таким образом, генераторные функции позволяют избежать выполнения всей работы заранее, что особенно полезно, когда результирующие списки большие или получение каждого значения требует длительных вычислений.

Отдельно при работе с генераторными функциями следует уделить внимание методу **send**. Он осуществляет переход на следующий элемент в серии результатов, в точности как **__next__**, но также снабжает вызывающий код возможностью взаимодействия с генератором для влияния на его работу:

```
def gen_send_test(N):
    for x in range(N):
        Y = yield x
        print(Y)

iterator = gen_send_test(2)
print(next(iterator)) # 0
print(iterator.send('-_-'))
# -_-
# 1
print(iterator.send('o_o'))
# o_o
# StopIteration
```

Генераторная функция также позволяет перехватить прерывание итерации по генерируемой последовательности:

```
def exept_gen_test(N):
    try:
        for x in range(N):
            yield x
    except GeneratorExit:
        print('Exception!!!')
    print('happy end')

for it in exept_gen_test(30):
    if it >= 10:
        break
    print(it, end=' ')
```

```
# 0 1 2 3 4 5 6 7 8 9 Exception!!!
# happy end
# при (exept_gen_test(9))
# 0 1 2 3 4 5 6 7 8 happy end
```

3.8.2. Генераторные выражения

Поскольку функциональность генераторных функций оказалась удобной, со временем понятия итерируемых объектов и списковых включений были объединены в новый инструмент — генераторные выражения. Синтаксически генераторные выражения похожи на нормальные списковые включения и поддерживают весь их синтаксис, в том числе фильтры *if* и вложение циклов, но они помещаются в круглые скобки, а не в квадратные:

```
print([x+2 for x in range(5)]) # [2, 3, 4, 5, 6]
print((x+2 for x in range(5)))
# <generator object <genexpr> at 0x0000023BE310FBC8>
```

Для более наглядного примера перепишем следующий код, использующий генераторную функцию:

```
def gen_func(N):
    for it in range(N):
        yield '-_-'

func_iter = gen_func(4)
for it in func_iter:
    print(it)
# -_-
# -_-
# -_-
# -_-
```

на код, использующий генераторное выражение:

```
new_func_iter = ('-_-' for _ in range(4))
for it in new_func_iter:
    print(it)

#или

for it in ('-_-' for _ in range(4)):
    print(it)
```

Добавим в данный пример фильтрацию на четность:

```
for it in ('-_-' for x in range(4) if x % 2 == 0):
    print(it)
# -_-
# -_-
```

Так же, как и генераторные функции, выражения обеспечивают оптимизацию расхода памяти — они не требуют создания сразу всего результирующего списка.

На практике генераторные выражения могут выполняться несколько медленнее списковых включений, а потому их лучше всего применять для очень крупных результирующих наборов или в приложениях, которые не могут ожидать генерации полных результатов.

Резюме

В данном разделе были рассмотрены функции и все, что с ними связано в Python. При их объявлении необходимо обращать внимание на следующие моменты:

— применяйте глобальные переменные, только когда они по-настоящему нужны;

— не модифицируйте изменяемые аргументы, если только такое изменение не ожидается вызывающим кодом;

— каждая функция должна иметь единственное назначение.

Декораторы определяют структурные блоки многократного использования, которые можно применять к вызываемому объекту с целью модификации его поведения без необратимого изменения самого вызываемого объекта. Синтаксис с использованием «@» — сокращенная запись для вызова декоратора с входной функцией. Многочисленные декораторы, размещенные над одной-единственной функцией, применяются снизу вверх.

Генераторные функции (функции-генераторы) — «синтаксический сахар» для написания объектов, которые поддерживают протокол итератора. Инструкция *yield* позволяет временно приостанавливать исполнение генераторной функции и возвращать из нее значения. Исключения *StopIteration* вызываются после того, как поток управления покидает генераторную функцию каким-либо иным способом, кроме инструкции *yield*.

Генераторные выражения (выражения-генераторы) похожи на включения в список, но не конструируют объекты-списки. Вместо этого они генерируют значения «точно в срок» подобно тому, как это делают итераторы на основе класса или генераторные функции. Как только генераторное выражение было использовано, оно не может быть перезапущено или использовано заново.

Вопросы и задания для самопроверки

1. В чем смысл написания функций?
2. В какой момент Python создает функцию?
3. Что функция возвращает, если в ней нет ни одного оператора return?

4. Когда выполняется код, вложенный внутрь оператора определения функции?

5. Что выведет следующий код и почему?

```
X = 'Spam'
def func () :
    print(X)
func()
```

6. Что выведет следующий код и почему?

```
X = 'Spam'
def func () :
    X = 'N1!'
func ()
print (X)
```

7. Что выведет следующий код и почему?

```
X = 'Spam'
def func () :
    X = 'N1'
    print(X)
func()
print(X)
```

8. Что выведет следующий код и почему?

```
X = 'Spam'
def func () :
    global X
    X = 'N1'
func()
print(X)
```

9. Что выведет следующий код в Python 3.x и почему?

```
def func () :
    X = 'N1'
    def nested() :
        nonlocal X
        X = 'Spam'
    nested()
    print(X)
func()
```

10. Каким будет вывод следующего кода и почему?

```
def func(a, b=4, c=5) :
    print (a, b, c)
func(1, 2)
```

11. Каким будет вывод такого кода и почему?

```
def func(a, b, c=5) :
    print(a, b, c)
func(1, c=3, b=2)
```

12. Что насчет следующего кода: каким будет вывод и почему?

```
def func(a, *pargs) :
    print(a, pargs)
func(1, 2, 3)
```

13. Что выведет этот код и почему?

```
def func(a, **kargs) :
    print(a, kargs)
func(a=1, c=3, b=2)
```

14. Каким будет вывод следующего кода и почему?

```
def func(a, b, c=3, d=4) : print (a, b, c, d)
    func(1, *(5, 6))
```

15. Каким образом связаны друг с другом лямбда-выражения и операторы *def*?
16. Какой смысл применять лямбда-выражения?
17. Что такое аннотации функций и как их использовать?
18. Что такое рекурсивные функции и как их применять?
19. Назовите несколько универсальных принципов проектирования функций.
20. Назовите три или больше способов передачи результатов функций вызывающему коду.
21. В чем разница между помещением спискового включения в квадратные скобки и в круглые скобки?
22. Как связаны между собой генераторы и итераторы?
23. Как определить, является ли функция генераторной?
24. Что делает оператор *yield*?

Упражнения

1. Напишите функцию, вычисляющую максимальное из трех чисел.
2. Напишите функцию, которая возвращает сумму элементов списка.
3. Напишите функцию, которая возвращает произведение элементов списка.
4. Напишите функцию, которая возвращает инвертированную строку, подаваемую ей на вход.
5. Напишите функцию для вычисления факториала задаваемого числа.
6. Напишите функцию, которая проверяет, входит ли задаваемое значение в определенный диапазон или нет.
7. Напишите функцию, которая подсчитывает количество элементов в нижнем и верхнем регистрах у входной строки.
8. Напишите функцию, удаляющую повторяющиеся элементы в списке.
9. Напишите функцию, выводящую элементы списка с четным индексом.
10. Напишите функцию, которая проверяет, является ли строка палиндромом (читается одинаково как слева направо, так и наоборот).
11. Напишите декоратор, обертывающий возвращаемое строковое значение функции в тег « ».
12. Декорируйте функцию из первого упражнения таким образом, чтобы возвращаемое ей значение возводилось в квадрат.

13. Декорируйте функцию из четвертого упражнения таким образом, чтобы возвращалась строка в верхнем регистре.

14. Напишите декоратор, выводящий значения аргументов, подаваемых на вход декорируемой функции.

15. Напишите генераторную функцию, позволяющую проводить итерацию по значениям в диапазоне от 23 до 37.

16. Напишите генераторную функцию, позволяющую проводить итерацию по значениям в диапазоне от 5 до 37 с шагом 4.

17. Напишите генераторное выражение, позволяющее итерироваться по последовательности чисел от 0 до 15.

18. Напишите генераторное выражение, позволяющее итерироваться элементам списка [3 0 1 3 0 4 3 3 4 56 6 1 3] возводя их при этом в квадрат.

19. Напишите генераторную функцию, позволяющую проводить итерацию по значениям в диапазоне от 0 до 100 с шагом, регулируемым в процессе ее работы.

20. Сформируйте список, используя для этого реализованные ранее генераторные функции из упражнений 15 и 16.

Тема 4
МОДУЛИ И ПАКЕТЫ

В результате изучения данной темы обучающиеся должны:

знать

- принципы организации модулей и пакетов,
- существующие механизмы импортирования модулей и пакетов, а также их отдельных атрибутов;

уметь

- создавать и импортировать собственные модули,
- создавать и импортировать собственные пакеты;

владеть

- навыками написания приложений с использованием модулей и пакетов.

Модуль в Python — это единица организации программ наивысшего уровня, которая упаковывает программный код и данные для многократного использования и предоставляет изолированные пространства имен, сводящие к минимуму конфликты имен переменных внутри программ. В Python каждый файл является модулем, а модули импортируют другие модули, чтобы задействовать определяемые в них имена. Для работы с ними используются два оператора и одна функция:

1) *import* — позволяет импортирующему коду извлечь модуль как единое целое;
2) *from* — позволяет клиентам извлекать отдельные имена из модуля;
3) imp.*reload* — предоставляет способ перезагрузки кода модуля без остановки интерпретатора Python.

Модули позволяют довольно простым способом организовывать отдельные компоненты в единую систему, выступая в качестве изолированных пакетов переменных, которые известны как пространства имен. Все имена, определенные на верхнем уровне файла модуля, становятся атрибутами объекта импортированного модуля.

В основном модули используют для следующего:

1) многократное использование кода;
2) разбиение пространства имен системы. В основном модули можно рассматривать как изолированные пакеты имен, которые

запрещают доступ к имени в другом файле, пока этот файл явно не импортируется. Такое решение помогает избежать конфликтов имен в разрабатываемых программах. В данном случае необходимо запомнить следующую особенность Python — абсолютно все «существует» в модуле: запускаемый код и создаваемые объекты всегда неявно заключаются в модули;

3) реализация совместно используемых служб или данных. Модули удобны для реализации компонентов, которые совместно используются в рамках системы и потому требуют только одной копии. Например, если необходимо предоставить глобальный объект, применяемый в нескольких функциях или файлах, тогда его можно реализовать в модуле, который затем будет импортироваться многими клиентами.

4.1. Принцип работы импортирования

Импортирование в Python — не просто вставка одного текста внутрь другого, а операция со своим временем выполнения, которая протекает за три отдельных шага, при условии, что модуль импортируется впервые:

1) поиск файла модуля;
2) компиляция импортируемого модуля в байт-код (при необходимости);
3) выполнение кода модуля для создания объектов, которые в нем определены.

Следует запомнить, что все три описанных выше шага выполняются только при первом импортировании модуля во время выполнения программы. При последующем их импортировании (того же самого модуля) во время выполнения программы интерпретатор просто обходит эти шаги стороной, так как результат их выполнения уже загружен в память. Python хранит информацию по загруженным модулям в словаре по имени sys.modules. Если данные по загружаемому модулю отсутствуют в этом словаре, тогда запускается описанный выше процесс.

4.1.1. Поиск файла подключаемого модуля

В большинстве случаев конфигурирование пути поиска файлов не требуется. Однако если есть необходимость импортировать файлы из сторонних каталогов, то необходимо знать, как его настраивать. Грубо говоря, путь поиска модулей Python формируется объединением следующих главных компонентов, часть которых задана заранее, а другие можно конфигурировать для указания, где проводить поиск.

1. Домашний каталог программы. Поиск всегда начинается с домашнего каталога. В этом случае, если все модули программы расположены в единственном каталоге, не требуется какая-либо конфигурация путей. Все будет работать автоматически.

2. Каталоги PYTHONPATH (если установлены). PYTHONPATH — список определяемых пользователем имен каталогов, которые содержат файлы с кодом, являющийся системной переменной операционной системы с таким же названием (PYTHONPATH). Устанавливать PYTHONPATH нужно только при импорте файлов из-за границ домашнего каталога программы, либо при написании программ значительного размера.

3. Каталоги стандартной библиотеки. Следом за каталогами, указанными в PYTHONPATH, Python автоматически выполняет поиск в каталогах, где располагаются стандартные библиотечные модули. Поиск в них происходит всегда, что не требует добавления данных каталогов в переменную PYTHONPATH или их включения в файлы конфигурации путей (см. далее).

4. Содержимое любых файлов .pth (при их наличии). Это еще один инструмент, который позволяет пользователям добавлять каталоги в путь поиска модулей, просто перечисляя их по одному в строке внутри текстового файла с расширением .pth (от path — путь). Такие файлы конфигурации путей являются несколько более развитым средством и предлагают альтернативу настройкам переменной PYTHONPATH.

5. Каталог сторонних расширений «Lib\site-packages» (устанавливается автоматически). Самым последним Python проверяет наличие искомого файла в подкаталоге модулей (site-packages) своей стандартной библиотеки. Сюда автоматически устанавливается большинство сторонних расширений. Именно из-за того, что этот каталог всегда является частью пути поиска модулей, разработчик спокойно может импортировать модули устанавливаемых расширений безо всяких настроек путей.

Все приведенные пути поиска хранятся в sys.path. Это изменяемый список строк с именами каталогов:

```
import sys
if __name__ == '__main__':
    print(sys.path)
"""['F:\\code\\python\\myProjRep', 'F:\\code\\python\\
myProjRep', 'C:\\Users\\madcomrade\\Anaconda3\\python37.
zip', 'C:\\Users\\madcomrade\\Anaconda3\\DLLs', 'C:\\Users\\
madcomrade\\Anaconda3\\lib', 'C:\\Users\\madcomrade\\
Anaconda3', 'F:\\code\\python\\myProjRep\\venv', 'F:\\code\\
python\\myProjRep\\venv\\lib\\site-packages', 'F:\\code\\
python\\myProjRep\\venv\\lib\\site-packages\\setuptools-40.8.0-
py3.7.egg', 'F:\\code\\python\\myProjRep\\venv\\lib\\site-
packages\\pip-19.0.3-py3.7.egg'] """
```

В большинстве случаев sys.path не нуждается в изменениях, но всегда есть исключение. Например, сценарии, запускаемые на веб-серверах, зачастую запускаются от имени пользователя *nobody* (никто). Это делается для того, чтобы ограничить доступ. При таком подходе до выполнения любых операторов import sys. path устанавливается вручную. Для этого вполне хватает метода **sys.path.append()** или **sys.path.insert()** (действует в течение только одиночного запуска программы).

При наличии файлов с одинаковыми именами в разных каталогах загрузится тот, который встречается первым при проходе путей поиска, прописанных в sys.path (слева направо). Такое же правило действует и в случае, когда файлы находятся в одном каталоге. Однако не факт, что такой подход гарантированно останется одинаковым с течением времени либо не будет отличаться в разных реализациях Python. В связи с этим назначайте модулям несовпадающие имена или конфигурируйте путь поиска модулей так, чтобы сделать предпочтения выбора модулей явными.

4.1.2. Компиляция импортируемого модуля в байт-код

После того, как файл был найден, Python при необходимости компилирует его в байт-код. Во время операции импорта осуществляется проверка времени модификации подключаемых файлов исходного кода и уже существующего байт-кода, а также номер версии Python байт-кода. В первом случае используются временны́е отметки файлов, а во втором — имя файла, которое содержит содержат версию Python, которой он компилировался. На основе этих проверок принимается одно из следующих решений:

1) компилировать (сгенерировать байт-код заново), если файл байт-кода старше файла исходного кода (т. е. исходный код был модифицирован) или был создан другой версией Python;

2) не компилировать, если файл байт-кода .рус не старше файла исходного кода и создан текущей версией интерпретатора Python.

Если при поиске файла импортируемого модуля находится его файл байт-кода, но не сам файл исходного кода, то Python загружает байт-код. Такой механизм делает возможной поставку программы в виде только файлов байт-кода (без передачи исходного) и ускоряет начальный запуск программы, так как этап компиляции пропускается.

Обратите внимание, что компиляция происходит во время импортирования файла. Из-за этого вы обычно не будете видеть файл байт-кода .рус для файла верхнего уровня своей программы, если только он также не импортируется где-то в другом месте — лишь импортированные файлы оставляют после себя файлы . рус на компьютере. Байт-код файлов верхнего уровня применяется внутренне

и отбрасывается; байт-код импортированных файлов сохраняется в файлах для ускорения будущих операций импорта.

4.1.3. Выполнение кода модуля

Все операции в файле выполняются по очереди, от начала до конца, и любые присваивания именам на данном шаге генерируют атрибуты результирующего объекта модуля. Например, операторы *def* в файле запускаются на стадии импортирования для создания объектов функций и их присваивания атрибутам внутри объекта модуля. Функции затем вызываются в файлах, импортирующих этот модуль.

4.2. Создание и использование модулей

Создать модуль достаточно просто, так как каждый файл с расширением .py Python автоматически воспринимает как модуль. Все имена, которые объявлены на верхнем уровне модуля, будут восприниматься как его атрибуты. Например, создадим файл «my_module1.py», содержащий следующий код:

```
a = 15
str1 = "test_string"

def printer(name):
    print(f'Hello, {name}!')
```

Поскольку имена модулей становятся именами переменных внутри программы Python, при их создании необходимо следовать обычным правилам именования переменных (особенно в случае ключевых слов и операторов). Так, например, если создать следующий файл «if.py», то его будет невозможно импортировать, а попытка это сделать (*import if*) приведет к синтаксической ошибке. То же правило распространяется и на имена каталогов (они могут содержать только буквы, цифры и подчеркивания).

Использовать только что написанный файл модуля в модуле более верхнего уровня можно за счет оператора *import* или *from*. Что в первом, что во втором случае осуществляются поиск, компиляция и запуск кода файла модуля, если он еще не был загружен. Основное отличие заключается в том, что оператор *import* загружает модуль как объект с атрибутами, поэтому обращение к его атрибутам необходимо производить через имя самого модуля. А при использовании оператора *from* извлекаются (или копируются) специфические имена из модуля (его атрибуты).

В случае импортирования написанного ранее модуля посредством оператора *import* его имя «my_module1» служит двум разным целям: 1) с его помощью идентифицируется внешний файл, под-

лежащий загрузке; 2) имя модуля становится переменной, которая ссылается на объект модуля после того, как файл загружен. Ниже приведен пример импортирования модуля оператором *import*:

```
import my_module1

if __name__ == "__main__":
    my_module1.printer('Alex') # Hello, Alex!
    print(my_module1.str1) # test_string
```

Оператор *from* является расширением оператора *import*. Он импортирует файл модуля обычным образом, после чего копирует одно или несколько имен из файла. Таким образом осуществляется более прямой доступ к именам импортируемого модуля:

```
from my_module1 import printer

if __name__ == "__main__":
    printer('Alex') # Hello, Alex!
    print(str1)
"""Traceback (most recent call last):
  File "F:/code/python/part4_module/main.py", line 5,
in <module>
    print(str1)
NameError: name 'str1' is not defined"""
```

Запись вида *from my_module1 import ** копирует все имена на верхнем уровня импортируемого модуля:

```
from my_module1 import *

if __name__ == "__main__":
    printer('Alex') # Hello, Alex!
    print(str1) # test_string
```

Импорт является довольно затратной операцией, из-за чего Python производит его только один раз на файл модуля (однократно на процесс). При последующих операциях импортирования происходит извлечение имен (объекта) из уже загруженного ранее модуля.

Import и *from*, как и оператор *def*, выступают в роли явного присваивания:

— *import* связывает имя (название модуля) с объектом целого модуля;

— *from* связывает одно или несколько имен с объектами с такими же именами из импортируемого модуля.

Таким образом, результатом работы оператора *from* становятся имена в локальной области видимости импортирующего модуля, ссылающиеся на совместно используемые объекты. Повторное присваивание скопированному имени какого-либо значения не оказывает влияния на сам модуль, из которого оно было скопировано. В то же самое время модификация совместно используемого объ-

екта изменяемого типа данных через скопированное имя изменит его в импортируемом модуле. Изменение значения объекта неизменяемого типа может произойти при непосредственном обращении к нему через его глобальное имя в другом файле, для чего необходимо использовать оператор *import*.

Также операторы *import* и *from* могут:

— вкладываться внутрь проверок *if* для выбора среди нескольких вариантов;

— находиться внутри операторов *def* функций, чтобы загружать только по вызову;

— использоваться в операторах *try*.

По своей сути запись типа:

```
from import a, str1
```

эквивалентна следующей последовательности операторов:

```
import my_module1
a = my_module1.a
str1 = my_module1.str1
del my_module1
```

Оператор *import* необходимо всегда применять в случае, когда в различных импортируемых модулях на верхнем уровне используется одно и то же имя (если необходимо работать с обоими):

```
# first.py
def function():
    print('first')

# second.py
def function():
    print('second')

# main.py
import first
import second

if __name__ == "__main__":
    first.function() # first
    second.function() # second
```

Так, например, при использовании оператора *from* при импортировании имен произойдет следующее:

```
# main.py
from first import function
from second import function
# from second переписывает имя function,
# извлеченное из first

if __name__ == "__main__":
    function() # second
```

Выйти из такой ситуации можно при помощи расширения *as*:

```
# main.py
from first import function as  ffunction
from second import function as sfunction

if __name__ == "__main__":
    ffunction() # first
    sfunction() # second
```

4.2.1. Пространства имен модулей

В рамках Python модули можно рассматривать как пространства имен, а существующие в них на верхнем уровне имена — как атрибуты. Доступ к пространству имен модуля можно получить через его атрибут *__dict__* или используя встроенную функцию **dir()**. Поскольку модули представляют собой локальную область видимости, то они следуют правилу поиска LEGB, но без двух первых уровней (L и E).

Самое важное, что следует запомнить: когда модуль загружен, его глобальная область видимости становится словарем атрибутов объектов модуля. В качестве примера рассмотрим подключение модуля следующего вида:

```
print('start load')

import sys
name_a = 33
def function():
    print('func use')
class TestClass:
    class_name = 30

print('finish load')
```

Как уже говорилось ранее, при первом импортировании модуля Python выполнит все операторы в нем. Часть из этих операторов создаст имена в пространстве имен модуля, а другие будут выполняться, пока не завершится операция импорта:

```
# main.py
import namespases_ex

if __name__ == "__main__":
    pass
# start load
# finish load
```

Далее рассмотрим созданный в процессе импортирования словарь атрибутов модуля:

```
# main.py
import namespases_ex
```

```
if __name__ == "__main__":
    attr_keys = namespases_ex.__dict__.keys()
    print(list(attr_keys))
"""
start load
finish load
['__name__', '__doc__', '__package__', '__loader__', '__
spec__', '__file__', '__cached__', '__builtins__', 'sys',
'name_a', 'function', 'TestClass'] """
```

Имена, присвоенные в файле модуля, становятся ключами словаря. В процессе операции импорта Python автоматически добавляет в пространство имен модуля несколько дополнительных имен (имена с двумя нижними подчеркиваниями с разных сторон имени в Python принято называть *дандерами*). Давайте выведем только те ключи словаря, которые присваиваются только в написанном ранее коде:

```
# main.py
import namespases_ex

if __name__ == "__main__":
    attr_keys = namespases_ex.__dict__.keys()
    attr_keys = [name for name in attr_keys if not name.
startswith('__')]
    print(attr_keys)
# start load
# finish load
# ['sys', 'name_a', 'function', 'TestClass']
```

Теперь попробуем обратиться к одному из имен модуля через его атрибут и словарь *__dict__*:

```
# main.py
import namespases_ex

if __name__ == "__main__":
    print(namespases_ex.name_a) # 33
    print(namespases_ex.__dict__['name_a']) # 33
```

В Python доступ к атрибутам любого объекта, имеющего атрибуты, производится с применением синтаксиса уточнения вида «*объект.атрибут*» (извлечение атрибутов). Уточнение представляет собой выражение, которое возвращает значение, присвоенное имени атрибута и ассоциированное с объектом.

Важно помнить о том, что уточнение атрибутов не имеет ничего общего с правилами поиска в областях видимости и является независимой концепцией. Когда уточнение используется для доступа к именам, Python получает явный объект, из которого необходимо извлечь указанные имена. Ниже приведены основные правила, которые распространяются на уточнения.

1. X означает поиск имени X в текущих областях видимости.
2. Уточнение X. Y означает поиск X в текущих областях видимости, затем поиск атрибута Y в объекте X (не в областях видимости).

3. Путь уточнения X.Y.Z означает поиск имени Y в объекте X, затем поиск Z в объекте X.Y.

4. Уточнение работает для всех объектов с атрибутами: модулей, классов, типов расширений и т. д.

Помимо описанного выше, операции импорта никогда не обеспечивают восходящую видимость коду в импортируемых файлах. Это значит, что импортируемый файл не может видеть имена из файла, в котором он импортируется, то есть:

— функции не могут видеть имена в других функциях, если только они физически не вкладываются в них;

— код модуля не может видеть имена в других модулях, если только он явно не импортирует их.

4.2.2. Перезагрузка модулей

Код модуля выполняется только один раз при запуске программы (или процесса). Для его принудительной перезагрузки и повторного выполнения используется встроенная функция **reload**. Механизм перезагрузок позволяет делать разрабатываемые системы более динамичными.

Таким образом, функция **reload** делает возможным изменение частей программы без ее остановки, позволяя незамедлительно наблюдать эффекты от изменений в компонентах. Естественно, перезагрузка не может использоваться абсолютно в любой ситуации, но там, где это возможно, это значительно сокращает цикл разработки.

Существуют следующие отличия **reload** от *import* и *from*:

— это функция, а не оператор Python;

— в **reload** в качестве аргумента передается существующий объект модуля, а не новое имя;

— функция **reload** находится внутри модуля в Python 3.x и должна быть импортирована.

Из перечисленного выше следует, что прежде чем осуществить перезагрузку модуля, он должен быть предварительно успешно загружен. Синтаксис операторов *import* и *from* отличается от **reload**: функция перезагрузки требует наличия круглых скобок. Обобщенный вид перезагрузки модулей представлен ниже:

```
import module
# Первоначальная операция импорта
# ... использовать атрибуты module ...
# Внести изменения в файл модуля
from imp import reload
# Получить саму функцию reload, (в Python 3. X)
reload(module)
# Получить обновленные атрибуты
... использовать атрибуты module ...
```

К ключевым моментам работы с механикой перезагрузки модуля можно отнести то, что:

— функция reload запускает новый код файла модуля в текущем пространстве имен модуля. Повторное выполнение кода файла модуля переписывает его существующее пространство имен, а не удаляет и воссоздает его;

— присваивания верхнего уровня в файле заменяют имена новыми значениями;

— перезагрузка воздействует на всех клиентов, которые применяют *import* для извлечения модулей;

— перезагрузка воздействует только на клиентов, которые применят *from* в будущем. То есть она не окажет влияние на клиентов, которые использовали *from* для извлечения атрибутов в прошлом, и они по-прежнему будут иметь ссылки на старые объекты, извлеченные перед перезагрузкой;

— перезагрузка применяется только к одиночному модулю.

Из этого следует, что если перезагружается модуль A, в котором импортируются модули B и C, то перезагрузка применяется только к модулю A (к B и C не применяется). Операторы внутри A, которые импортируют B и C, в течение перезагрузки выполняются повторно, но они всего лишь извлекают объекты уже загруженных модулей B и C (предполагая, что они были импортированы раньше).

4.2.3. Запуск модуля как автономной программы

Каждый модуль имеет встроенный атрибут *__name__*, который в процессе запуска программы или импортирования модуля автоматически создается интерпретатором Python, после чего ему присваивается значение. При этом, если файл модуля запускается как программа верхнего уровня, тогда во время старта атрибут *name* устанавливается в строку «__main__». В ином случае, когда модуль импортируется, атрибут *__name__* устанавливается в имя файла модуля.

Это позволяет модулю проверять собственный атрибут *__name__* для выяснения, запущен он или импортирован. Например, пусть создан следующий файл модуля по имени «*test_run.py*», экспортирующий единственную функцию **run**:

```
# test_run.py
def run():
    print('Let's go!')

if __name__ == '__main__': # если запущен как модуль верхнего
уровня
    run()
```

Если запустить написанный модуль посредством команды, то функция выполнится автоматически:

```
python test_run.py
# Let's go!
```

В сущности, встроенный атрибут модуля *__name__* служит флагом режима его использования и позволяет коду быть задействованным как в качестве импортируемой библиотеки, так и в качестве сценария верхнего уровня.

Самый распространенный вариант применения проверки атрибута *__name__* предназначен для самотестирования кода. Это позволяет использовать файл модуля как в клиентах — импортируя его, так и запускать его в командной оболочке или посредством другой схемы запуска для тестирования его логики работы. Такой подход является, вероятно, наиболее часто применяемым и простым способом модульного тестирования в Python. Это в разы проще и удобнее, чем повторно набирать все тесты.

Помимо этого, такой подход часто используется при создании файлов модулей, которые могут применяться как утилиты командной строки или как библиотеки инструментов.

4.2.4. Внутренние имена в модулях

Имена, которые начинаются с символа нижнего подчеркивания «_», не импортируются командой *from module import*. Это позволяет разработчикам писать модули, предназначенные для импортирования этой командой, и при этом ограничивать импортирование некоторых функций или переменных. Отсюда следует, что ***внутренние имена (те имена, которые должны быть доступны только в пределах модуля) должны начинаться с символа нижнего подчеркивания*** «_». Такой подход гарантирует, что команда *from module import ** импортирует только те имена, которые должны быть доступны пользователю.

Для примера создадим модуль с названием «private_names.py»:

```
# private_names.py
def function(x):
    print(x)

def _internal_func(x):
    print(x)

first = 4
_second = 2
```

Далее импортируем его и попробуем обратиться к различным именам модуля:

```
from private_names import *

if __name__ == "__main__":
    function(20) # 20
    _internal_func(20)
""" Traceback (most recent call last):
  File "F:/code/python/part4_module/main.py", line 5,
```

```
in <module>
    _internal_func(20)
NameError: name '_internal_func' is not defined """

print(first)# 4
    print(_second)
"""Traceback (most recent call last):
  File "F:/code/python/part4_module/main.py", line 7,
in <module>
    print(_second)
NameError: name '_second' is not defined """
```

Из приведенного выше примера видно, что такие имена, как *function* и *first*, спокойно импортируются, а *_internal_func* и *_second* остаются скрытыми за пределами модуля private_names.

Важно запомнить, что такое поведение при импортировании возможно только с командой вида *from ... import **. При обычном импортировании, через оператор *import*, пользователь получает доступ и к внутренним именам:

```
import private_names

if __name__ == "__main__":
    private_names._internal_func(20) # 20
    print(private_names._second) # 2
```

Такого же эффекта можно добиться, если использовать встроенный атрибут модуля (дандер модульного уровня) *__all__*, который является списком и идентифицирует имена, подлежащие копированию, в то время как запись вида *_X* обозначает имена, которые копироваться не должны. Python сначала ищет список *__all__* в модуле и копирует его имена вне зависимости от наличия в них подчеркиваний:

```
# priv_names_v2.py
__all__ = ['_internal_func', 'first', '_second']

def function(x):
    print(x)

def _internal_func(x):
    print(x)

first = 4
_second = 2
```

Далее импортируем написанный выше модуль и обратимся к различным именам модуля:

```
from priv_names_v2 import *

if __name__ == "__main__":
    _internal_func(20) # 20
```

```
    print(_second) # 2
    function(20)
"""Traceback (most recent call last):
  File "F:/code/python/part4_module/main.py", line 6,
in <module>
    function(20)
NameError: name 'function' is not defined"""
```

Подобно соглашению *_X* использование списка *__all__* имеет смысл только при использовании оператора *from ** при импортировании имен модуля.

4.3. Создание и использование пакетов модулей

При импортировании в дополнение к имени модуля можно указывать путь к каталогу с кодом, который в Python называется пакетом. При этом само импортирование пакетов превращает каталог в еще одно пространство имен Python с атрибутами, соответствующими подкаталогам и файлам модулей, которые он содержит.

Пакеты являются естественным расширением концепции модуля и предназначены для очень больших проектов. По аналогии с тем, как модули группируют взаимосвязанные функции, классы и переменные, пакеты группируют взаимосвязанные модули. Там, где в операторах *import* находится имя простого файла модуля, взамен него можно указать путь с именами, разделенными точками:

```
import myproj.dir1.dir2.example
```

или

```
from myproj.dir1.dir2.example import x
```

Такое указание пути соответствует иерархии каталогов на диске к файлу *example.py*. Если каталог не является домашним или не находится в подкаталоге site-packages с установленными сторонними расширениями, его необходимо вручную добавить в список *sys.path*.

До версии Python 3.3 каждый каталог, указанный внутри пути в операторе импортирования пакета, должен был содержать файл по имени __init__.py. Несмотря на то, что теперь наличие этих файлов в каталогах пакетов является не обязательным, их использование обеспечивает выигрыш в производительности.

Файлы __init__.py могут содержать код Python подобно нормальным файлам модулей, и их код выполняется автоматически при первом импортировании каталога. При этом __init__.py объявляет каталог пакетом Python, генерирует пространство имен для каталога и реализует поведение операторов *from ** (т. е. *from … import **)

в случае применения с операциями импорта каталогов. Также эти файлы могут быть пустыми.

Рассмотрим пакет следующей структуры:

```
myproj/
    __init__.py
    dir1/
        __init__.py
        value.py
        dir2/
            __init__.py
            example1.py
            example2.py
```

В Python-файлах приведенного пакета записан следующий код:

```
# myproj/__init__.py
print("myproj init")
__all__ = ['dir1']
version = 1.03

#myproj/dir1/__init__.py
__all__ = ['value']
print("myproj.dir1 init")

#myproj/dir1/value.py
val = 3.99

#myproj/dir1/dir2/__init__.py
print("myproj.dir1.dir2 init")

#myproj/dir1/dir2/example1.py
from myproj import version
from myproj.dir1 import value
from myproj.dir1.dir2.example2 import function
def ex1_finction():
    print("Package version: ", version)
    print(function())

#myproj/dir1/dir2/example2.py
def function():
    return "Function in example2"
```

При импортировании пакета произойдет только выполнение файла *myproj/__init__.py*:

```
import myproj

if __name__ == "__main__":
    print(myproj.version)
# myproj init
# 0.1
```

Если вызвать функцию **ex1_finction**, определенную в файле *myproj/dir1/dir2/example1.py*, то по наличию ошибки станет очевид-

но, что импортирование пакета не осуществляет импортирование его субпакетов:

```
import myproj

if __name__ == "__main__":
    print(myproj.dir1.dir2.example1)
"""Traceback (most recent call last):
  File "F:/code/python/part4_module/main.py", line 4,
in <module>
    print(myproj.dir1.dir2.example1)
AttributeError: module 'myproj' has no attribute 'dir1' """
```

Из приведенного выше примера следует, что загрузки модуля верхнего уровня пакета недостаточно для загрузки всех его субмодулей. Это соответствует философии Python, согласно которой ничего не должно происходить у вас за спиной. Ясность важнее лаконичности.

Вызвать функцию пакета можно следующим образом:

```
import myproj.dir1.dir2.example1

if __name__ == "__main__":
    myproj.dir1.dir2.example1.ex1_finction()
# myproj init
# myproj.dir1 init
# myproj.dir1.dir2 init
# Package version:  0.1
# Function in example2
```

Получается, что прежде чем Python сможет импортировать *myproj.dir1.dir2.example1*, он должен импортировать *myproj.dir1*, а затем *myproj.dir1.dir2*. Из чего следует, что файлы в пакете не получают автоматического доступа к объектам, которые были определены в других файлах пакета, и для явного обращения к объектам из других файлов пакетов их необходимо импортировать.

Если атрибут *__all__* присутствует в файле *__init__.py*, он возвращает список строк, определяющий те имена, которые должны импортироваться при использовании записи вида *from ... import ** для данного пакета:

```
from myproj import *

if __name__ == "__main__":
    pass
# myproj init
# myproj.dir1 init
```

Так как список *__all__* в файле *myproj/__init__.py* содержит запись «dir1», то осуществляется импортирование субпакета «dir1». В том случае, когда словарь *__all__* отсутствует, ничего не импортируется.

Для проектирования пакетов необходимо следовать следующим рекомендациям.

Пакеты не должны использовать структуры каталогов с большой глубиной вложенности. То есть для большинства пакетов достаточно одного-двух каталогов верхнего уровня. Исключением может являться проектирование действительно огромных библиотек.

Несмотря на то, что использование механизма с включением в модуль списка *__all__* может использоваться для сокрытия имен при их импортировании посредством *from ... import **, лучше использовать нижний префикс «_» для этих целей.

Резюме

В данном разделе рассмотрены основы модулей, атрибутов и импортирования, а также принципы создания и использования пакетов модулей. При импортировании Python ищет указанный файл в пути поиска модулей, компилирует его в байт-код и выполняет все имеющиеся внутри операторы для генерации содержимого. Пути поиска файлов модулей можно конфигурировать, чтобы иметь возможность импортировать из каталогов, отличающихся от домашнего каталога, и т. д. Код в одном модуле изолирован от кода в другом, и ни один файл не может даже видеть имена, определенные в другом файле, если только не были выполнены явные операторы *import*. По этой причине модули сводят к минимуму конфликты имен между разными частями программы.

Поскольку операция импортирования и модули лежат в самой основе программной архитектуры Python, то обычно проектируемые программы разбиваются на множество файлов, которые с помощью импортирования связываются вместе во время выполнения.

Вопросы и задания для самопроверки

1. Как оператор *from* связан с оператором *import*?
2. Как функция **reload** связана с операциями импортирования?
3. Когда вы обязаны использовать *import* вместо *from*?
4. Назовите три потенциальных затруднения, связанных с оператором from.
5. Каким образом файл исходного кода модуля становится объектом модуля?
6. Затем вам может понадобиться установка переменной среды PYTHONPATH?
7. Назовите пять главных компонентов в пути поиска импортируемых модулей.
8. Назовите четыре типа файлов, которые Python может загрузить в ответ на операцию импорта.

9. Что такое пространство имен и что содержит пространство имен модуля?

10. Что важно знать о переменных на верхнем уровне модуля, чьи имена начинаются с одиночного подчеркивания?

11. Что означает наличие у переменной *__name__* модуля строки «__main__»?

12. Чем изменение sys.path отличается от установки PYTHONPATH для модификации пути поиска модулей?

13. Для чего предназначен файл __init__.py в каталоге пакета модуля?

14. Какие каталоги требуют наличия файлов __init__.py?

15. Когда с пакетами обязательно использовать оператор *import*, а не *from*?

Упражнения

1. Напишите модуль, содержащий функции, которые выполняют следующие арифметические операции: сложение, вычитание, умножение.

2. Напишите модуль, содержащий функции, которые выполняют следующие операции: проверка наличия элемента в списке, подсчет частоты вхождения элемента в список.

3. Напишите модуль, содержащий функции, которые выполняют следующие операции: проверку, является ли строка палиндромом, подсчет длины строки, перевод всех символов в нижний регистр.

4. Напишите модуль, содержащий функции, которые выполняют следующие операции: подсчет площади круга, прямоугольника и треугольника.

5. Напишите модуль, содержащий функции, которые выполняют следующие операции: подсчет количества элементов в словаре, проверку на наличие ключа в словаре.

6. Реализуйте пакет, объединяющий модули из первого и второго упражнений.

7. Реализуйте пакет, объединяющий модули из третьего и четвертого упражнений.

8. Реализуйте пакет, объединяющий пакеты из шестого и седьмого упражнений.

9. Напишите модуль, содержащий внутренние имена, значения которых можно получить через функции верхнего уровня модуля.

10. Импортируйте имена из написанных модулей (пакетов) и проверьте их работу.

Тема 5
КЛАССЫ И ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

В данной теме внимание будет уделено таким пользовательским типам данных, как классы, механизму их реализации в Python, а также насколько сам язык программирования поддерживает принципы объектно-ориентированного программирования (ООП).

Отдельно коснемся таких понятий, как: экземпляр класса, переменная или метод класса, переменная или метод экземпляра класса и т. д.

В результате изучения данной темы обучающиеся должны:

знать

- принципы объектно-ориентированного программирования,
- механизм реализации классов в Python и работы с ними,
- способы работы с перечислениями (Enum),
- существующие отличия между переменными класса и переменными экземпляра класса;

уметь

- создавать пользовательские классы;
- объявлять (абстрактные) базовые классы и их производные классы,
- осуществлять перегрузку методов класса;

владеть

- навыками написания кода на языке программирования Python в объектно-ориентированной парадигме.

5.1. Определение класса

Классы в Python являются пользовательским типом данных и представляют собой описание объекта предметной области с его состояниями и поведением. Они также используются для декомпозиции кода на составные части с целью минимизации избыточности.

За создание классов отвечает оператор *class*:

```
class TestClass:
    pass
```

Тело класса может состоять лишь из оператора-заполнителя *pass* или из необязательной последовательности команд и определений функций.

По соглашениям Python имя классов записывается в «верблюжьей нотации», и их можно рассматривать как пакеты функций, которые используют и обрабатывают объекты встроенных типов. Несмотря на это, классы предназначены для создания новых объектов, реализации их логики работы и поддерживают наследование — механизм настройки и многократного применения кода.

Экземпляр объекта типа объявленного ранее класса **TestClass** создается следующим образом:

```
instance = TestClass()
```

Отличительной чертой экземпляров классов Python от структур C или классов Java является то, что их атрибуты (поля данных) не обязательно объявлять заранее. Они могут добавляться «в процессе работы». Давайте рассмотрим пример, в котором определяется класс с именем **TestClass**, создается его экземпляр и полю *magic* экземпляра присваивается значение:

```
class TestClass:
    pass

if __name__ == "__main__":
    my_test = TestClass()
    my_test.magic = 10
    print(my_test.magic) # 10
```

Для автоматической инициализации атрибутов (полей) класса поля в его тело включают метод инициализации **__init__**. Этот метод выполняется при каждом создании экземпляра класса и относительно похож на конструктор в языке Java/C++. При этом, в отличие от классов Java/C++, метод **__init__** ничего не конструирует (только инициализирует поля класса) и не может перегружаться. То есть классы в Python могут иметь только один метод **__init__**:

```
class TestClass:
    def __init__(self):
        self.magic = 10

if __name__ == "__main__":
    my_test = TestClass()
    print(my_test.magic) # 10
```

По правилам Python первый аргумент метода **__init__** — *self*, которому присваивается создаваемый экземпляр при выполнении **__init__**. Помимо метода инициализации **__init__** в Python существует еще один аналог конструктора — метод **__new__**. Он вызывается при создании объекта и возвращает неинициализированный объект.

Классы не всегда необходимы в Python, в ряде случаев может хватить и функции (или группы функций). Класс следует использовать для программного представления физических или концептуальных

объектов и при его написании рекомендуется задуматься и найти ответы на следующие вопросы:

1) Какое имя носит описываемый объект?
2) Какими свойствами обладает описываемый объект?
3) Присущи ли эти свойства всем экземплярам класса? А именно:
— Какие свойства являются общими для класса в целом?
— Какие из этих свойств уникальны для каждого экземпляра?
4) Какие операции выполняет описываемый объект?

5.2. Имена (переменные) экземпляров класса

В рассмотренном ранее примере имя (переменная) *magic* является элементом (атрибутом) экземпляра класса **TestClass**. Это значит, что каждый создаваемый экземпляр класса будет иметь собственную копию этой переменной, и значение, с которым осуществляется работа через нее, будет отличаться от значений аналогичных переменных других экземпляров класса. Как уже показывалось ранее, Python позволяет создавать атрибуты экземпляров класса в процессе работы кода по мере необходимости путем присваивания значения имени атрибута экземпляра класса [39]:

```
my_test.new_ magic = value
```

Для обращения к имени атрибута экземпляра класса для начала необходимо явно указать имя экземпляра созданного класса, после чего обратиться к его атрибуту (*имя_экземпляра.имя_атрибута*). Если же обращение к имени атрибута экземпляра класса будет производиться без указания имени экземпляра, то поиск имени будет производиться в локальной области выполняемого метода, функции и т. д.

5.3. Методы экземпляра класса

Метод класса — функция, которая связана с конкретным классом:

```
class TestClass:
    def __init__(self, magic=10):
        self.magic = magic

    def square_magic(self):
        return self.magic ** 2

if __name__ == "__main__":
    my_test1 = TestClass()
    print(my_test1.square_magic()) # 100
    my_test2 = TestClass(5)
    print(my_test2.square_magic()) # 25
```

Существует несколько способов вызова метода класса: связный или несвязный. Приведенный выше способ вызова является связным. Несвязный способ вызова метода менее удобен и реже применяется на практике, так как при таком вызове метода в его первом аргументе должен передаваться экземпляр класса, что делает код менее понятным:

```
print(TestClass.square_magic(my_test2)) # 25
```

В первом аргументе любого метода передается экземпляр, для которого он вызывается — *self*, который является аналогом *this* во многих языках программирования.

Python преобразует вызов метода класса в обычный вызов функции по следующим правилам:

— поиск имени метода в пространстве имен экземпляра (my_test2);

— если метод не обнаружен в пространстве имен экземпляра, поиск осуществляется в типе класса экземпляра (TestClass);

— если метод не найден, то поиск продолжается в суперклассе;

— вызов найденного метода как обычной функции Python.

При вызове такого метода экземпляр передается в его первом аргументе, а все остальные аргументы сдвигаются на одну позицию вправо. Получается, что запись вида *instance.method*(*arg1*, *arg2*, …) превращается в *class.method*(*instance*, *arg1*, *arg2*, …).

5.4. Имена (переменные) класса

Переменная класса связана непосредственно с классом, а не с его конкретным экземпляром, и доступна всем его экземплярам. Она может использоваться для отслеживания информации на уровне класса или хранить константные значения описываемого классом объекта и создается присваиванием в теле класса, а не в методе __*init*__. При их использовании необходимо следить за возможными конфликтами имен между переменными классов и переменными экземпляров.

После создания переменной класса она становится видимой для всех его экземпляров:

```
class TestClass:
    pow_val = 3 # переменная класса
    def __init__(self, magic=10):
        self.magic = magic

    def square_magic(self):
        return self.magic ** TestClass.pow_val

if __name__ == "__main__":
    my_test1 = TestClass()
    print(my_test1.square_magic()) # 1000
```

```
my_test2 = TestClass(5)
print(my_test2.square_magic()) # 125
```

Имя переменной класса жестко фиксируется в методах этого класса. Но также к нему можно обратиться через специальный атрибут экземпляра класса — *__class__*. Этот атрибут вернет класс, к которому принадлежит экземпляр:

```
print(TestClass) # <class '__main__.TestClass'>
print(my_test2.__class__)
#  <class '__main__.TestClass'>
```

Таким образом, посредством атрибута *__class__* можно получить значение *TestClass.pow_val* без явного указания имени класса **TestClass**:

```
print(my_test2.__class__.pow_val) # 3
```

Этот подход позволяет избавиться от внутреннего упоминания класса **TestClass** путем замены ссылки: *TestClass.pow_val* заменяется на *self.__class__.pow_val*:

```
def new_square_magic(self):
   return self.magic ** self.__class__.pow_val

print(my_test2.new_square_magic()) # 125
```

При этом, если изменить значение переменной *pow_val* посредством одного из экземпляров класса (my_test3) следующим образом:

```
my_test3 = TestClass()
my_test4 = TestClass()
my_test3.pow_val = 10
print(my_test3.pow_val) # 10
print(my_test4.pow_val) # 3
```

то данный экземпляр класса будет содержать собственную копию *pow_val*, отличную от копии *TestClass.pow_val*, к которой обращается **my_test4**. Это связано с тем, что присваивание *my_test3.pow_val* создает переменную экземпляра в **my_test3**.

При последующем обращения к *my_test3.pow_val* возвращается значение переменной из экземпляра класса, а при обращении к *my_test4.pow_val* Python ищет переменную экземпляра *pow_val* в *my_test4*, не находит ее и переходит к поиску значения переменной класса в **TestClass**. Если имеется необходимость изменить значение переменной класса, необходимо обратиться к ней по имени класса, а не через переменную экземпляра self:

```
my_test5 = TestClass()
TestClass.pow_val = 40
print(my_test5.pow_val) # 40
```

5.5. Статические методы

Как и в C++ или Java, в Python статические методы можно вызывать, даже если ни один экземпляр класса не создан. Чтобы объявить метод как статический, используется декоратор *@staticmethod*:

```
class TestClass:
    """Класс пыщ-пыщ"""
    all_created_clases = []
    def __init__(self, magic=10):
        """Создание экземпляра класса"""
        self.magic = magic
        TestClass.all_created_clases.append(self)
        #self.__class__.all_created_clases.append(self)

    def square_magic(self):
        """Возведение в квадрат"""
        return self.magic ** 2

    @staticmethod
    def sum_all_square_magic():
        """Статистический метод подсчета квадратов всех
TestClass"""
        rezult = 0
        for it in TestClass.all_created_clases:
            rezult += it.square_magic()
        return rezult

if __name__ == "__main__":
    my_test1 = TestClass()
    my_test2 = TestClass(5)
    my_test4 = TestClass(15)
    print(TestClass.sum_all_square_magic()) # 350
    my_test5 = TestClass(1)
    print(TestClass.sum_all_square_magic()) # 351
```

5.6. Методы класса

Методы классов, так же как и статические методы, могут вызываться до того, как будет создан объект класса. Также они могут использоваться с указанием экземпляра класса. Отличие заключается в том, что у методов класса первым аргументом неявно идет класс (*cls*), к которому они принадлежат. В связи с этим у них более простое написание. Для этого используется декоратор *@classmethod*:

```
class TestClass:
    """Класс пыщ-пыщ"""
    all_created_clases = []
    def __init__(self, magic=10):
        """Создание экземпляра класса"""
        self.magic = magic
```

```
        TestClass.all_created_clases.append(self)
        #self.__class__.all_created_clases.append(self)

    def square_magic(self):
        """Возведение в квадрат"""
        return self.magic ** 2

    @classmethod
    def sum_all_square_magic(cls):
        """Статистический метод подсчета квадратов всех
TestClass"""
        rezult = 0
        for it in cls.all_created_clases:
            rezult += it.square_magic()
        return rezult

if __name__ == "__main__":
    my_test1 = TestClass()
    my_test2 = TestClass(5)
    my_test4 = TestClass(15)
    print(TestClass.sum_all_square_magic()) # 350
    my_test5 = TestClass(1)
    print(TestClass.sum_all_square_magic()) # 351
```

Использование метода класса вместо статического метода позволяет всем производным классам класса **TestClass** вызвать метод **sum_all_square_magic** и обращаться к своим полям и методам (а не к полям и методам **TestClass**).

5.7. Приватные методы и переменные

В C++, Java и других языках программирования приватные переменные или методы не видны за пределами методов класса, в котором они определяются. То есть к ним нельзя обратиться через экземпляр класса. Наличие модификаторов доступа позволяет использовать инкапсуляцию при проектировании программы, а сами приватные переменные и методы при этом повышают уровень безопасности и надежности за счет ограничения доступа к важным или критичным частям реализации объекта.

Во многих языках программирования для объявления приватной переменной или метода используется модификатор доступа *private*. В Python же к приватным относятся только переменные или методы, которые начинаются с двойного подчеркивания «__x».

В то же самое время необходимо понимать, что в Python инкапсуляция присутствует только на словах, так как у пользователя все равно есть возможность получить доступ к приватным переменным или методам класса. Такое положение дел накладывает дополнительные обязательства на разработчика, который должен помнить

об этом и всегда придерживаться условности, что переменные, начинающиеся с двойного подчеркивания и не заканчивающиеся им, — приватные:

```
class PrivateTest:
    def __init__(self):
        self.public_x = 2
        self.__private_y = "I'm invisible"
        self._private_z = "I'm not invisible"

    def print_y(self):
        print(self.__private_y)

    def __get_y(self):
        return  self.__private_y

if __name__ == "__main__":
    private_test = PrivateTest()
    print(private_test.public_x) # I'm public
    print(private_test._private_z)
# I'm not invisible
    print(private_test.__private_y)
"""Traceback (most recent call last):
  File "F:/code/python/module5/private_val_and_method.py", line
14, in <module>
    print(private_test.__private_y)
AttributeError: 'PrivateTest' object has no attribute '__
private_y'"""
```

Казалось бы, все ожидаемо, и напечатать значение, хранящееся в приватной переменной *__private_y*, можно только посредством метода **print_y()**:

```
    private_test.print_y() # I'm invisible
```

Однако вспомните, о чем говорилось ранее. Когда компилятор формирует байт-код запускаемого скрипта и встречается с переменной или методом, которые начинаются с двойного подчеркивания, он создает атрибут класса не с именем самого метода или переменной, а с именем «*_ИмяКласса__ИмяАтрибута*», как показано в следующем коде:

```
    print(private_test._PrivateTest__private_y)
# I'm invisible
    print(private_test._PrivateTest__get_y())
# I'm invisible
```

5.8. Наследование

Поскольку Python — язык программирования с неявной динамической типизацией, в нем механизм наследования проще и гибче, чем в компилируемых языках (например, Java или C++).

В качестве примера того, как используется наследование в Python, рассмотрим классический пример обобщения геометрических фигур. Для этого определим по классу для квадратов и окружностей:

```
class Square:
    def __init__(self, side=1, x=0, y=0):
        self.side = side

class Circle:
    def __init__(self, radius=1, x=0, y=0):
        self.radius = radius
```

Допустим, что эти классы планируется использовать при написании графического редактора. Исходя из этого, добавим координаты в виде переменных *x* и *y* для хранения информации о том, в каком месте «виртуальной поверхности» изображения располагается каждый экземпляр:

```
class Square:
    def __init__(self, side=1, x=0, y=0):
        self.side = side
        self.x = x
        self.y = y

class Circle:
    def __init__(self, radius=1, x=0, y=0):
        self.radius = radius
        self.x = x
        self.y = y
```

К сожалению, такой подход приводит к появлению большого количества дублирующего кода, если придется увеличивать количество классов геометрических фигур. Это связано с тем, что для каждой такой фигуры придется хранить информацию о ее текущей позиции. Такие проблемы являются стандартной ситуацией для применения наследования.

Вместо того чтобы определять переменные *x* и *y* в каждом классе геометрической фигуры, их можно абстрагировать в некоторый обобщенный класс **Shape**, от которого будет производится наследование остальных классов геометрических фигур:

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

class Square(Shape):
    def __init__(self, side=1, x=0, y=0):
        super().__init__(x, y)
        self.side = side
```

```
class Circle(Shape):
    def __init__(self, radius=1, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius
```

В большинстве случаев для использования наследования необходимо выполнить (обычно) два требования:

1) определить иерархию наследования, то есть в круглых скобках непосредственно за именем определяемого класса указать имя класса, от которого производится наследование;

2) явно вызвать метод **__init__** классов, используемых при наследовании. Это связано с тем, что Python не делает этого автоматически, и приходится использовать функцию **super**. В приведенном коде за это отвечает строка вызова *super*().*__init__*(*x*,*y*), которая из производного класса вызывает функцию инициализации родительского класса **Shape** с инициализируемым экземпляром и правильными аргументами.

Вместо использования функции **super** можно вызвать метод **__init__** класса **Shape** с явным указанием родительского класса: *Shape.__init__*(*self*, *x*, *y*). Однако этот способ будет менее гибким в долгосрочной перспективе, потому что он жестко фиксирует имя родительского класса, что может создавать проблемы в будущем при изменении структуры и иерархии наследования.

Далее определим в классе **Shape** еще один метод с именем **move**, который сдвигает фигуру, изменяя ее координаты x и y на величину, определяемую аргументами метода, а также по методу **__str__** для вывода состояния значений переменных каждого из класов посредством функции **print**:

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

    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y

    def __str__(self):
        return f'x: {self.x}, y: {self.y}'

class Square(Shape):
    def __init__(self, side=1, x=0, y=0):
        super().__init__(x, y)
        self.side = side

    def __str__(self):
        return super().__str__() +
               f', side: {self.side}'
```

```
class Circle(Shape):
    def __init__(self, radius=1, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def __str__(self):
        return super().__str__() +
               f', side: {self.radius}'
```

Теперь создадим по экземпляру классов **Square** и **Circle** и изменим их стартовые координаты:

```
if __name__ == "__main__":
    my_circle = Circle(x=10, y=5)
    my_square = Square(5, 25, 40)
    print(my_circle) #  x: 10, y: 5, side: 1
    print(my_square) #  x: 25, y: 40, side: 5
    my_square.move(35, 50)
    my_circle.move(10, 10)
    print(my_circle) #  x: 20, y: 15, side: 1
    print(my_square) # x: 60, y: 90, side: 5
```

Классы **Circle** и **Square** изначально не содержали собственного определения метода **move**. С другой стороны, так как они наследуются от класса **Shape**, где этот метод реализован, то все экземпляры **Circle** и **Square** могут использовать метод **move**. Это связано с тем, что все методы Python являются виртуальными. Таким образом, если метод не существует в текущем классе, то Python ищет его по списку атрибутов родительского класса и использует первый найденный метод.

Давайте рассмотрим, как при наследовании обстоит дело с переменными экземпляров и классов:

```
class BaseTextTest:
    name = "Test"

    def set_text(self):
        self.text = "Class BaseTextTest"

    def print_base(self):
        return self.text

    def __str__(self):
        return self.text

class NewTextTest(BaseTextTest):
    def set_new_text(self):
        self.text = "Class NewTextTest"

    def print_new(self):
        return self.text

    def __str__(self):
        return self.text
```

```
if __name__ == "__main__":
    new_class = NewTextTest()
    new_class.set_text()
    print(new_class) #Class BaseTextTest
    print(new_class.print_base())
#Class BaseTextTest
    print(new_class.print_new())
#Class BaseTextTest
    new_class.set_new_text()
    print(new_class.print_new()) #Class NewTextTest
    print(new_class) #Class NewTextTest
```

Из приведенного примера видно, что обращения к переменным экземпляров класса с одинаковыми именами будут производиться к одной переменной. Именно поэтому после объявления переменной *text* посредством метода родительского класса попытка вывести ее значения посредством метода **print_new()** выводит «Class BaseTextTest», а после переопределения ее значения методом дочернего класса **set_new_text()** выводится «Class NewTextTest», и наоборот.

Так как переменные классов наследуются, необходимо помнить об их обобщенном поведении. Например, переменную класса *name* определили для родительского класса **BaseTextTest**, и к ней можно обратиться несколькими способами: через экземпляр *new_class*, через дочерний класс **NewTextTest** или через родительский класс **BaseTextTest**:

```
print(f'new_class = {new_class.name},
      f' NewTextTest = {NewTextTest.name}, '
      f'BaseTextTest = {BaseTextTest.name}')
# new_class = Test, NewTextTest = Test,
# BaseTextTest = Test
```

При попытке задать значение переменной класса *name* через класс **NewTextTest** для него создается новая переменная класса с таким именем. Это не влияет на значение переменной класса **BaseTextTest**. В тоже время при последующих обращениях через экземпляр или сам класс **NewTextTest** пользователю будет доступна только эта новая переменная, а не оригинал из **BaseTextTest**:

```
NewTextTest.name = "Not bad"
print(f'new_class = {new_class.name},
      f'NewTextTest = {NewTextTest.name}, '
      f'BaseTextTest = {BaseTextTest.name}')
# new_class = Not bad, NewTextTest = Not bad,
# BaseTextTest = Test
```

Тот же самый эффект будет наблюдаться, если задать новое значение переменной *name* через экземпляр класса **NewTextTest**.

То есть создастся новая переменная экземпляра, и уже будет не две разные переменные, а три:

```
BaseTextTest.name = "Not good"
print(f'new_class = {new_class.name},
      f' NewTextTest = {NewTextTest.name}, '
      f'BaseTextTest = {BaseTextTest.name}')
# new_class = Not bad, NewTextTest = Not bad,
# BaseTextTest = Not good
```

5.9. Множественное наследование

В Python, как и C++, разрешается множественное наследование, а в таких языках программирования как Java оно запрещено, хотя имеется возможность расширять функциональность класса различным количеством интерфейсов.

Что значит «множественное наследование»? Это когда производный класс может быть унаследован от любого количества базовых классов. По умолчанию, если не происходит явного указания базового класса при наследовании, то класс наследуется от *object*.

Приведем простейший пример множественного наследования с использованием «пустых» классов, которые не содержат переменных или методов с одинаковыми именами:

```
class FirstBaseClass:
    ...
class SecondBaseClass:
    ...
class ThirdBaseClass:
    ...
class FirstTestClass(FirstBaseClass,
                     ThirdBaseClass):
    ...
class SecondTestClass(SecondBaseClass):
    ...
class TopTestClass(FirstTestClass,
                   SecondTestClass):
    ...
```

Так как в приведенных классах нет одноименных методов, экземпляр класса **TopTestClass** может использоваться, как если бы он был экземпляром любого из классов. В то же самое время, если некоторые из приведенных классов будут содержать одноименные методы, трудно даже предположить, не зная способов обхода базовых классов при поиске метода, какой из них должен считаться правильным:

```
class FirstBaseClass:
    ...
class SecondBaseClass:
    ...
```

```
class ThirdBaseClass:
    def test_method(self):
        print("ThirdBaseClass")

class FirstTestClass(FirstBaseClass,
                     ThirdBaseClass):
    ...

class SecondTestClass(SecondBaseClass):
    def test_method(self):
        print("SecondTestClass")
class TopTestClass(FirstTestClass,
                   SecondTestClass):
    ...

if __name__ == "__main__":
    my_class = TopTestClass()
    my_class.test_method() # ??
```

В ходе попытки вызова метода **test_method()** для экземпляра класса **TopTestClass** Python начнет перебирать его базовые классы слева направо. Также он всегда просматривает всех предков одного базового класса перед тем, как переходить к следующему базовому классу. Давайте восстановим последовательность поиска интерпретатором выполняемого метода:

1) сначала поиск метода будет выполняться в классе объекта, для которого он вызывается, то есть в **TopTestClass**;

2) поскольку в **TopTestClass** не определен метод **test_method()**, поиск начнет осуществляться в его базовых классах. Первым базовым классом для **TopTestClass** является **FirstTestClass**, поэтому именно в нем Python продолжит поиск;

3) так как в **FirstTestClass** аналогично предыдущему случаю тоже не определен метод **test_method()**, поиск продолжится уже в его базовых классах, и первым из них будет **FirstBaseClass**;

4) поскольку в **FirstBaseClass** не найден определенный метод **test_method()**, после проверки на отсутствие у него базовых классов поиск прерывается. Python переходит к классу **FirstTestClass** и обращается к его следующему базовому классу — **ThirdBaseClass**;

5) в классе **ThirdBaseClass** Python находит вызываемый метод **test_method()**, и поскольку это первый найденный метод с заданным именем, именно он будет использован. Аналогичный метод, определенный в классе **SecondTestClass**, будет проигнорирован.

Считается, что множественное наследование — плохой способ организации архитектуры приложения. Но оно иногда способствует предотвращению слишком глубоких иерархий, которые могут встречаться при использовании обычного наследования или ис-

пользуются для создания классов примесей (mixin). Примеси представляют собой класс, в котором у части или всех методов нет реализации, а обязанность по обеспечению недостающей реализации ложится на их производный класс.

5.10. Абстрактные классы и переопределение методов

Абстрактный (интерфейсный) класс — класс, описывающий (предоставляющий) интерфейс взаимодействия с объектом, реализация части методов которого описывается в классах, наследующихся от него, и который имеет хотя бы один чисто виртуальный метод, т. е. у объявленного метода нет реализации. Основной чертой абстрактного класса является то, что его экземпляр нельзя создать. Хотя к Python это утверждение в ряде случаев может быть неприменимо, лучше придерживаться данного подхода при проектировании архитектуры приложения, библиотеки и т. д.

Посредством абстрактного класса реализуется один из основных принципов ООП — полиморфизм. Без использования модуля *abc* абстрактный класс в Python можно реализовать двумя способами.

1. **С использованием assert:**

```
class AbstractBaseClass:
    def print_name(self):
        print("AbstractBaseClass")

    def action(self):
        assert False, "method not define"

if __name__ == "__main__":
    my_class = AbstractBaseClass()
    my_class.print_name() # AbstractBaseClass
    my_class.action()
# AssertionError: method not define
```

2. **С использованием исключения *NotImplementedError*:**

```
class AbstractBaseClassEx:
    def print_name(self):
        print("AbstractBaseClass")

    def action(self):
        raise NotImplementedError("method not define")

if __name__ == "__main__":
    #my_class = AbstractBaseClass()
    my_class = AbstractBaseClassEx()
    my_class.print_name() # AbstractBaseClassEx
    my_class.action()
# NotImplementedError: method not define
```

Как видно из приведенных примеров, реализованные абстрактные классы не удовлетворяют одной из главных особенностей — их экземпляры можно создать! Чтобы интерпретатор не сообщал об ошибках в производном классе, по отношению к которому абстрактный класс выступает базовым, необходимо переопределить реализацию метода **action()**:

```
class SubClass(AbstractBaseClassEx):
    def action(self):
        print("method was defined")

if __name__ == "__main__":
    my_class = SubClass()
    my_class.print_name() # AbstractBaseClassEx
    my_class.action() # method was defined
```

В следующем примере для написания абстрактного класса используем возможности модуля *abc*:

```
from abc import ABCMeta, abstractmethod

class AbstractBaseClass(metaclass=ABCMeta):
    def print_name(self):
        print("AbstractBaseClass with abc
                              module help")

    @abstractmethod
    def action(self):
        ...

if __name__ == "__main__":
    my_class = AbstractBaseClass()
'''TypeError: Can't instantiate abstract class
AbstractBaseClass with abstract methods action'''
```

Реализация данного абстрактного класса уже удовлетворяет основному требованию — его экземпляр нельзя создать. Далее попробуем создать его производный класс без переопределения метода **action()**:

```
class FirstSubClass(AbstractBaseClass):
    ...

if __name__ == "__main__":
    my_class = FirstSubClass()
'''TypeError: Can't instantiate abstract class FirstSubClass
with abstract methods action'''
```

Все верно! До тех пор, пока пользователь не напишет реализацию данного метода, должна выводиться ошибка:

```
class SecondSubClass(AbstractBaseClass):
    def action(self):
        print("method was defined")
```

```
if __name__ == "__main__":
    my_class = SecondSubClass()
    my_class.print_name()
# AbstractBaseClass with abc module help
    my_class.action()
# method was defined
```

Такой подход лишен недостатков способов написания абстрактного класса с использованием assert или исключений.

Несмотря на наличие в Python утиной типизации (если объект выглядит как утка и крякает как утка, то объект является уткой), использование абстрактных классов является предпочтительным при написании сложных систем. Они позволяют создать такой вид иерархии при наследовании, который обязывает производные классы переопределить и реализовать часть его методов. Абстрактный класс можно назвать интерфейсом, когда в нем определен только набор методов, реализацию которых надо будет полностью описывать в производных классах.

5.11. Перегрузка операций

Под перегруженной операцией подразумевается автоматический вызов методов экземпляра класса, когда он встречается в коде со встроенными операциями (+, -, * и т. д.) и возвращаемое значение перегруженного метода становится результатом этой операции. Таким образом, когда класс перегружает особым образом именованные методы, Python автоматически вызывает их в случае появления экземпляров этого класса в ассоциированных с ними выражениях.

В табл. 5.1 приведены наиболее распространенные методы для перегрузки операций:

Таблица 5.1

Наиболее распространенные методы перегрузки операций

Метод	Описание	Для чего используется
__init__	Конструктор	Создание экземпляра объекта my_class = MyClass(args)
__del__	Деструктор	Удаление экземпляра объекта
__add__	Операция сложения	A + B, A += B, если отсутствует __iadd__
__or__	Операция побитовое «ИЛИ»	A | B, A I = B, если отсутствует __ior__
__sub__	Операция вычитания	A - B, A -= B
__repr__ __str__	Вывод, преобразования	print(X), repr(X), str(X)
__call__	Вызовы функций	function(*args, **kargs)

Метод	Описание	Для чего используется
__getattr__	Извлечение атрибута	X.undefined
__setattr__	Присваивание атрибута	X.any = value
__delattr__	Удаление атрибута	del X.any
__getattribute__	Извлечение атрибута	X.any
__getitem__	Индексирование, срез, итерация	X[key], X [i: j], циклы for и другие итерационные конструкции, если отсутствует__iter__
__setitem__	Присваивание по индексу и срезу	X[key] = value, X[i: j] = iterable
__delitem__	Удаление по индексу и срезу	del X[key], del X[i:j]
__len__	Длина	len (X), проверки истинности, если отсутствует__bool__
__bool__	Булевские проверки	bool (X), проверки истинности
__lt__ __gt__ __le__ __ge__ __eq__ __ne__	Сравнения	A < B, A > B, A <= B, A >= B, A == B, A ! = B
__radd__	Правосторонние операции	Other + X
__iadd__	Дополненные на месте операции	A += B (либо иначе__add__)
__iter__ __next__	Итерационные контексты	I=iter(X), next (I); циклы for, in, если отсутствует __contains__, все включения, map (F,X), остальные
__contains__	Проверка членства	item in X (любой итерируемый объект)
__index__	Целочисленное значение	hex(X), bin(X), oct(X), O[X], O[x:]
__enter__ __exit__	Диспетчер контекста	with obj as var:
__get__ __set__ __delete__	Атрибуты дескриптора	X.attr, X.attr = value, del X.attr
__new__	Создание	Создание объекта, перед __init__

Даже в том случае, когда методы перегрузки операций не определяются, они могут быть унаследованы от базового класса, как

и любые другие методы. Также они являются необязательными, а попытка применения операций к экземпляру класса, в котором перегрузка операции не была реализована, приведет к генерированию исключения.

5.11.1. Перегрузка __add__, __or__, __sub__

В примере ниже рассмотрим перегрузку таких операций, как: сумма, вычитание и исключающее «ИЛИ», реализовав класс **Number** с последующей перегрузкой соответствующих методов:

```
class Number:
    def __init__(self, val):
        self.value = val

    def __sub__(self, other):
        return Number(self.value - other)

    def __add__(self, other):
        return Number(self.value + other)

    def __or__(self, other):
        return Number(self.value | other)

    def __str__(self):
        return f'{self.value}'

if __name__ == "__main__":
    my_number = Number(4)
    addValue = my_number + 10
    print(addValue) # 14
    subValue = addValue - 13
    print(subValue) # 1
    orValue = subValue | 4
    print(orValue) # 5
```

5.11.2. Перегрузка __getitem__ и __setitem__

Когда метод **__getitem__** определен в классе, он автоматически вызывается для операций индексирования экземпляров этого класса. Например, в случае обращения «X[i]» интерпретатор Python вызывает метод **__getitem__**, с передачей X в первом аргументе, а во втором аргументе — значения индекса или среза в квадратных скобках. Метод присваивания по индексу **__setitem__** похожим образом перехватывает присваивания по индексу и срезу:

```
class CustomList:
    def __init__(self, elements_amount=1):
        self.__custom_list = [0] * elements_amount
        self.size = elements_amount

    def __str__(self):
        return f'{self.__custom_list}'
```

```
    def __getitem__(self, item):
        if not self.__index_check(item):
            new_list = CustomList(self.size)
            new_list.__custom_list =
                          self.__custom_list[item]
            return new_list
        return self.__custom_list[item]

    def __setitem__(self, key, value):
        if self.__index_check(key):
            self.__custom_list[key] = value
        else:
            my_range = range(key.start if key.start
                        else 0, key.stop,
                        key.step if key.step else 1)
            for it in my_range:
                self.__custom_list[it] = value

    def __index_check(self, index):
        if isinstance(index, int):
            print(f'current index = {index}')
            return True
        elif isinstance(index, slice):
            print(f'slice: start = {index.start}, '
                  f'stop = {index.stop}, '
                  f'step = {index.step}')
            return False
        else:
            raise IndexError("Bad index")

if __name__ == "__main__":
    my_list = CustomList(8)
    print(my_list) # [0, 0, 0, 0, 0, 0, 0, 0]
    my_list[4] = 35 # current index = 4
    print(my_list) # [0, 0, 0, 0, 35, 0, 0, 0]
    my_list[0:4] = 50
# slice: start = 0, stop = 4, step = None
    print(my_list) # [50, 50, 50, 50, 35, 0, 0, 0]
    new_list = my_list[0:6:2]
# slice: start = 0, stop = 6, step = 2
    print(new_list) # [50, 50, 35]
   my_list['2'] # IndexError: Bad index
```

Также существует не всегда очевидный способ использования **__getitem__**. При отсутствии более специфических методов итерации оператор *for* работает с данным методом путем многократного индексирования последовательности от нуля и далее, пока не сгенерируется исключение выхода за границы *IndexError*. То есть если этот метод определен, цикл *for* на каждом проходе вызывает **__getitem__** класса с последовательно увеличивающимися смещениями:

```
    for it in new_list:
        print(it)
'''
```

```
current index = 0
50
current index = 1
50
current index = 2
35
current index = 3
'''
```

Таким образом, если метод **__getitem__** определен, то проверка членства *in*, списковые включения, встроенная функция **map**, присваивания списков и кортежей, а также конструкторы типов будут автоматически его вызывать.

5.11.3. Перегрузка __iter__ и __next__

Перед **__getitem__** при работе с итерационным контекстом Python будет сначала пытаться вызвать метод **__iter__**. Это связано с тем, что для многократного индексирования объекта обычно выбирается протокол итерации.

Можно сказать, что итерационные контексты работают путем передачи итерируемого объекта встроенной функции **iter**, который вызывает метод **__iter__** экземпляра класса, возвращающий новый итерируемый объект. Когда же перегружается метод **__next__**, Python будет многократно вызывать его для прохода по элементам итерируемого экземпляра класса до тех пор, пока не сгенерируется исключение *StopIteration*.

Для примера напишем класс, который реализует логику, близкую к работе встроенной функции **range()** при положительных значениях:

```
class MyRange:
    def __init__(self, start, stop, step=1):
        self.start = start
        self.stop = stop
        self.step = step
        self.count_value = 0

    def __iter__(self):
        self.count_value = self.start - self.step
        return self

    def __next__(self):
        if self.count_value+self.step >= self.stop:
            raise StopIteration
        self.count_value += self.step
        return self.count_value

if __name__ == "__main__":
    my_range = MyRange(0, 4)
    for it in my_range:
        print(it, end=' ') # 0 1 2 3
    print()
    for it in MyRange(0, 12, 4):
        print(it, end=' ') # 0 4 8
```

В данном примере объект итератора, возвращаемый **__iter__**, является экземпляром класса — *self*. Это связано с тем, что метод **__next__** является частью самого класса **MyRange**. Для более сложных сценариев итерирования объект итератора определяется как отдельный класс с собственной информацией о состоянии для поддержки множества активных итераций по тем же самым данным.

При этом ручное итерирование работает с такими экземплярами итерируемых объектов аналогично работе со встроенными типами:

```
my_iter = iter(my_range)
print(next(my_iter)) # 0
print(next(my_iter)) # 1
print(next(my_iter)) # 2
print(next(my_iter)) # 3
print(next(my_iter)) # StopIteration
```

Теперь перепишем класс **MyRange** таким образом, чтобы свести к минимуму объем написанного кода за счет комбинирования метода **__iter__** и оператора генераторных функций *yield*:

```
class MyGeneratorRange:
    def __init__(self, start, stop, step=1):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        count_value = self.start - self.step
        while count_value+self.step < self.stop:
            count_value += self.step
            yield count_value

if __name__ == "__main__":
    print("MyGeneratorRange") # MyGeneratorRange
    my_range = MyGeneratorRange(0, 4)
    for it in my_range:
        print(it, end=' ')
# 0 1 2 3
    print()
    for it in MyGeneratorRange(0, 12, 4):
        print(it, end=' ')
# 0 4 8
    print()
    my_iter = iter(my_range)
    my_iter = iter(my_range)
    print(next(my_iter)) # 0
    print(next(my_iter)) # 1
    print(next(my_iter)) # 2
    print(next(my_iter)) # 3
    print(next(my_iter)) # StopIteration
```

Поскольку генераторные функции автоматически сохраняют состояние локальных переменных и создают обязательные методы ите-

раторов, они хорошо подходят для этой роли и дополняют предохранение состояния и другие полезные вещи, получаемые от классов.

В следующем примере рассмотрим более сложный сценарий реализации итерируемого объекта и определим отдельный класс с собственной информацией о состоянии, чтобы более прямо поддерживать множество активных итераций по тем же самым данным. Чтобы понять, зачем это нужно, попробуем использовать экземпляр написанного ранее класса **MyRange** в контексте вложенных циклов:

```
    test_range = MyRange(0, 3)
    for firtst_it in test_range:
        for second_it in test_range:
            print(firtst_it*second_it, end=' ')
# 0 0 0
```

Как видим, результат получился совсем не тот, что ожидали увидеть (***0 0 0 0 1 2 0 2 4***). Это связано с тем, что по завершении первого полного прохода вложенного цикла каждое последующее обращение к итератору будет возвращать *StopIteration*, а так как имена *firtst_it* и *second_it* ссылаются на один и тот же объект, то внешний цикл выполнит только одну итерацию, тогда как вложенный отработает один полный проход:

```
    test_range = MyRange(0, 3)
    for firtst_it in test_range:
        print(firtst_it)
        for second_it in test_range:
            print(f'second_it = {second_it}, '
                  f'firtst_it*second_it =
                           {firtst_it*second_it}')
'''
firtst_it = 0
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 0
second_it = 2, firtst_it*second_it = 0
'''
```

Реализация с использованием генератора лишена этого недостатка:

```
    test_range = MyGeneratorRange(0, 3)
    for firtst_it in test_range:
        print(f'firtst_it = {firtst_it}')
        for second_it in test_range:
            print(f'second_it = {second_it}, '
                  f'firtst_it*second_it = '
                  f'{firtst_it*second_it}')
'''
firtst_it = 0
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 0
second_it = 2, firtst_it*second_it = 0
```

```
firtst_it = 1
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 1
second_it = 2, firtst_it*second_it = 2
firtst_it = 2
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 2
second_it = 2, firtst_it*second_it = 4
'''
```

Теперь перепишем реализацию класса **MyRange**, добавив в его модуль еще один класс — **MyRangeIterator**:

```
class MyRange:
    def __init__(self, start, stop, step=1):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        return MyRangeIterator(self)

class MyRangeIterator:
    def __init__(self, myrange_object):
        self.my_range = myrange_object
        self.count_value = myrange_object.start - myrange_
object.step

    def __next__(self):
        if self.count_value+self.my_range.step >= self.my_
range.stop:
            raise StopIteration
        self.count_value += self.my_range.step
        return self.count_value

if __name__ == "__main__":
    print("MyRange")
    my_range = MyRange(0, 4)
    for it in my_range:
        print(it, end=' ') # 0 1 2 3
    print()
    for it in MyRange(0, 12, 4):
        print(it, end=' ') # 0 4 8
 print()

    test_range = MyRange(0, 3)
    for firtst_it in test_range:
        print(f'firtst_it = {firtst_it}')
        for second_it in test_range:
            print(f'second_it = {second_it}, '
                  f'firtst_it*second_it = '
                  f' {firtst_it*second_it}')
'''
firtst_it = 0
```

```
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 0
second_it = 2, firtst_it*second_it = 0
firtst_it = 1
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 1
second_it = 2, firtst_it*second_it = 2
firtst_it = 2
second_it = 0, firtst_it*second_it = 0
second_it = 1, firtst_it*second_it = 2
second_it = 2, firtst_it*second_it = 4
'''
```

5.11.4. Перегрузка __contains__

В области итераций классы реализуют операцию проверки членства *in* в виде итерации с применением методов **__iter__** или **__getitem__**. Но для поддержки более специфической операции проверки членства предусмотрен метод **__contains__**. Если он присутствует в классе, то ему всегда отдается предпочтение перед рассмотренными ранее методами:

```
class TestContains:
    def __init__(self, value):
        self.data = value

    def __contains__(self, x):
        return x in self.data

if __name__ == '__main__' :
    X = TestContains([1, 2, 3, 4, 5])
    print(3 in X) # True
    print(10 in X) # False
    print('d' in X) # False
```

5.11.5. Перегрузка __getattr__ и __setattr__

Классы Python имеют возможность перехватывать обращение к их атрибутам. Например, для объекта *X*, являющегося экземпляром класса, может быть реализована следующая операция — *X.атрибут*, которая участвует в контекстах ссылки, присваивания и удаления.

Метод **__getattr__** перехватывает обращение к несуществующему атрибуту класса, то есть он не вызывается в тех случаях, когда Python может найти атрибут с применением процедуры поиска в дереве наследования. Данный метод удобен в качестве привязки, которая обеспечивает реагирование на запросы атрибутов в обобщенной манере и используется при делегировании вызовов вложенным объектам из промежуточного объекта контроллера. Кроме того, он может применяться для адаптации классов к интерфейсу.

Если метод **__setattr__** определен или унаследован, тогда он всегда будет перехватывать все обращения к атрибуту *self.атрибут = значение*. Он позволяет классу выполнять желаемые проверки достоверности или преобразования. При его использовании нужно быть предельно внимательным, так как любое присваивание атрибуту класса в стиле *self.атрибут = значение* внутри метода снова вызовет его и тем самым станет причиной бесконечной рекурсии, которая остановится после генерации исключения, связанного с переполнением стека. Избежать такого поведения можно используя присваивания значений атрибутам экземпляра в виде присваиваний ключам словаря атрибутов. То есть внутри данного метода вместо записи *self.атрибут = значение* необходимо использовать *self.__dict__["атрибут"] = значение*.

Запомните: метод **__setattr__** без исключений перехватывает все присваивания значений атрибутам!

Ниже приведен пример перегрузки методов **__getattr__** и **__setattr__**:

```
class TestAttribute:
    def __init__(self, name):
        self.name = name

    def __getattr__(self, item):
        print("in __getattr__")
        if item == "age":
            self.__dict__[item] = 3
            return self.__dict__[item]
        else:
            raise AttributeError(item)

    def __setattr__(self, key, value):
        print(f"in __setattr__, key = {key},
              f" value = {value}")
        if key == "age":
            self.__dict__[key] = value + 1
        elif key == "name":
            self.__dict__[key] = value
        else:
            raise AttributeError(value + "not allowed")

if __name__ == "__main__":
    my_test = TestAttribute("Alex")
    print(my_test.name) # Alex
    print(my_test.age) # in __getattr__, 3
    print(my_test.age) # 3
    #print(my_test.country)
# in __getattr__, AttributeError: country
```

```
    my_test.name = "Jon"
# in __setattr__, key = name, value = Jon
    my_test.age = 10
# in __setattr__, key = age, value = 10
    print(my_test.name) # Jon
    print(my_test.age) # 11
    my_test.country = "Germany"
# AttributeError: Germanynot allowed
```

Таким образом, используя метод **__setattr__**, можно реализовать механизм, который позволяет каждому производному классу иметь собственный список закрытых имен, которым экземпляры не могут присваивать значения:

```
class PrivateExc(Exception): pass

class PrivatAttribute:
    def __setattr__(self, key, value):
        print(f"check key = {key}, value = {value}")
        if key in self.privates:
            raise PrivateExc(key, self)
        else:
            self.__dict__[key] = value

class FirstTestAttribute(PrivatAttribute):
    privates = ["name", "city"]

class SecondTestAttribute(PrivatAttribute):
    privates = ["age"]

    def __init__(self, name):
        self.name = name

if __name__ == "__main__":
    first_test = FirstTestAttribute()
    second_test = SecondTestAttribute("Alex")
# check key = name, value = Alex
    second_test.city = "SPb"
# check key = city, value = SPb
    first_test.country = "Germany"
# check key = country, value = Germany
    #first_test.name = "Jon"
# check key = name, value = Jon
''' __main__.PrivateExc: ('name', <__main__.FirstTestAttribute
object at 0x0000024C8DC67048>)
    second_test.age = 20 # check key = age, value = 20
__main__.PrivateExc: ('age', <__main__.SecondTestAttribute
object at 0x000001F8A49B7088>) '''
```

5.11.6. Перегрузка __repr__ и __str__

Если у класса перегружены сразу два этих метода, то метод **__str__** будет вызываться первым при передаче экземпляра класса в функцию **print** или при его приведении к строковому типу

данных посредством встроенной функции **str**. Метод **__repr__** используется во всех остальных контекстах и даже в том случае, когда метод **__str__** отсутствует. Эти методы должны возвращать строку, которую можно было бы применять для воссоздания объекта или для детального отображения значений атрибутов экземпляра класса при отладке.

Без перегрузки этих методов стандартное отображение не приносит особой пользы:

```
class Number:
    def __init__(self, val):
        self.value = val

    def __sub__(self, other):
        return self.value - other

if __name__ == "__main__":
    number = Number(10)
    print(number)
# <__main__.Number object at 0x000001B5DBF76748>
```

Согласитесь, хотелось бы увидеть значение, которое хранит объект, а не то, что получили в итоге. Для этой цели и используется перегрузка методов **__repr__** и **__str__**. Приведем пример с **__repr__**:

```
class NewNumber(Number):
    def __repr__(self):
        return f'{self.value}'

if __name__ == "__main__":
    newNumber = NewNumber(15)
    print(newNumber) # 15
    newNumber = newNumber - 3
    print(newNumber) # 12
```

5.11.7. Перегрузка __call__

Метод **__call__** вызывается каждый раз при вызове экземпляра класса, в котором он был перегружен. Python выполняет его для выражений вызова функций, применяемых к экземпляру класса, передавая ему любые позиционные или ключевые аргументы, которые были отправлены:

```
class CallTest:
    def __call__(self, *args, **kwargs):
        print("__call__")
        print(f"args = {args} \nkwargs = {kwargs}")


if __name__ == "__main__":
    test = CallTest()
    test(10, 'Oo', 32.1)
# __call__
```

```
# args = (10, 'Oo', 32.1)
# kwargs = {}
    test(10, 'Oo', 32.1, a=2, b='-_-')
# __call__
# args = (10, 'Oo', 32.1)
# kwargs = {'a': 2, 'b': '-_-'}
```

При этом метод **__call__** поддерживает все режимы передачи аргументов (см. тему 3):

```
class CallTest2:
    def __call__(self, a, b, c=3):
        print("__call__")
        print(f"a = {a}, b = {b}, c = {c}")

if __name__ == "__main__":
    test2 = CallTest2()
    test2(4, '^_^') # a = 4, b = ^_^, c = 3
    test2(b=4, c=45.3, a='^_^')
# a = ^_^, b = 4, c = 45.3
    test2(8)
# TypeError: __call__() missing 1
# required positional argument: 'b'
```

Такой перехват выражений вызовов позволяет экземплярам класса эмулировать внешний вид и поведение сущностей вроде функций, используя их в качестве аргументов для различных API или библиотек, которые ожидают на вход функцию. А для сокрытия информации о состоянии или наследовании используется перегрузка метода **__call__**:

```
class Persone:
    def __init__(self, name="Ivan"):
        self.name = name

    def __call__(self, profession):
        print(f"{self.name} is a {profession}")

class ProfessionName:
    def __init__(self, persone, prof="unemployed"):
        self.persone = persone
        self.profession = prof

    def __setattr__(self, key, value):
        self.__dict__[key] = value
        if key == "profession":
            self.persone(self.__dict__[key])

if __name__ == "__main__":
    persone = Persone("Alex")
    prof = ProfessionName(persone)
# Alex is a unemployed
    prof.profession = "doctor" # Alex is a doctor
    prof.profession = "student" # Alex is a student
```

5.11.8. Перегрузка методов сравнения

Классы могут перегружать все шесть операций сравнения: <, >, <=, >=, == и ! =. Ниже приведен пример перегрузки операторов сравнения:

```
class Number:
    def __init__(self, val):
        self.value = val

    def __lt__(self, other):
        return self.value < other

    def __gt__(self, other):
        return self.value > other

    def __le__(self, other):
        return self.value <= other

    def __ge__(self, other):
        return self.value >= other

    def __eq__(self, other):
        return self.value == other

    def __ne__(self, other):
        return self.value != other

if __name__ == "__main__":
    first = Number(10)
    second = Number(11)
    print(first < second) # True
    print(first > second) # False
    print(first <= second) # True
    print(first >= second) # False
    print(first == second) # False
    print(first != second) # True
```

5.11.9. Перегрузка __len__ и __bool__

Каждый объект в Python может являться истинным или ложным. Это означает, что объекты, у которых перегружены или реализованы методы **__len__** или **__bool__**, могут возвращать значения True или False экземпляра класса по запросу.

Python сначала пробует получить булево значение с помощью метода **__bool__**, в ином случае он делает это посредством метода **__len__**, пытаясь вывести значение истинности из длины объекта.

```
class MyTruth:
    def __bool__(self):
        return True

class MyTruth2:
    def __len__(self):
        return 0
```

```
class MyTruth3:
    def __bool__(self):
        return True

    def __len__(self):
        return 0

if __name__ == "__main__":
    my_truth = MyTruth()
    if my_truth: print("True") # True
    my_truth2 = MyTruth2()
    if not my_truth2: print("True") # True
    my_truth3 = MyTruth3()
    if my_truth3: print("True") # True
```

Если ни один из этих методов не определен, объект возвращает истину:

```
class TestTruth: ...

if __name__ == "__main__":
    test = TestTruth()
    print(bool(test)) # True
```

5.11.10. Декоратор @property или свойства

Вместо дескрипторов **__get__**, **__set__** и **__delete__**, которые предлагают свой способ перехвата доступа к атрибутам, рассмотрим использование встроенной функции **property** (свойства) в виде декоратора. Она представляет собой упрощенный способ создания особого типа дескриптора, который запускает функции методов при доступе к атрибуту, а также позволяет вычислять или изменять присваиваемые значения «на ходу»:

```
class Persone:
    def __init__(self, name="Alex", age=22):
        self.__persone_name = name
        self.__persone_age = age

    @property
    def name(self):
        print("get name", end=' ')
        return self.__persone_name

    @property
    def age(self):
        print("get age", end=' ')
        return self.__persone_age

    @name.setter
    def name(self, new_name):
        print(f"set new name - {new_name}")
        self.__persone_name = new_name
```

```
    @age.setter
    def age(self, new_age):
        self.__persone_age = new_age + 1
        print(f"set new age - {new_age}")


if __name__ == "__main__":
    my_persone = Persone()
    print(my_persone.name) # get name Alex
    print(my_persone.age) # get age 22
    my_persone.name = "Maxim" # set new name - Maxim
    my_persone.age = 10 # set new age - 11
    my_persone.name = 20 # set new name – 21
```

Если не добавить один из декорированных методов для установки возраста, то Python при попытке присвоить данному атрибуту новое значение выдаст ошибку: AttributeError: can't set attribute.

Главное преимущество использования свойств заключается в том, что в ходе разработки могут использоваться обычные переменные экземпляров, которые затем будут легко преобразованы в свойства там, где это понадобится, без изменения клиентского кода и способа обращения к атрибутам экземпляра класса.

5.12. Вложенные классы и пространство имен

Несмотря на то, что классы обычно определяются на верхнем уровне модуля, иногда их вкладывают в генерирующие их функции (фабричный метод) с похожими ролями сохранения состояния, как и в случае с замыканиями у функций. Сам по себе оператор *class* вводит новую локальную области видимости подобно операторам *def* функций и следует правилу LEGB.

После выполнения оператора *class*, как и в случае с модулями, локальная область видимости класса превращается в пространство имен атрибутов. Это позволяет классу иметь доступ к областям видимости объемлющих функций, но при этом сами объемлющие функции не действуют в качестве объемлющей области видимости для кода, вложенного внутрь класса. С этим связано то, что Python ищет имена, на которые произведена ссылка, в объемлющих функциях, но никогда не выполняет их поиск в объемлющих классах.

То есть класс является локальной областью видимости, которая имеет доступ к объемлющим локальным областям видимости, но в то же самое время не служит объемлющей локальной областью видимости для дальнейшего вложенного кода:

```
TestVal = 5

def test_function():
    print(TestVal) # 5
```

```
    class NestedClass:
        print(TestVal) # 5

        def first_method(self):
            print(TestVal) # 5

        def second_method(self):
            TestVal = 3 # скрывает глобальное имя
            print(TestVal) # 3

    my_class = NestedClass()
    my_class.first_method()
    my_class.second_method()

if __name__ == "__main__":
    print(TestVal) # 5
    test_function()
```

Теперь попробуем изменить значение *X* в теле объемлющей функции сразу после ее объявления:

```
TestVal = 5

def test_function():
     TestVal = 1   # скрывает глобальное имя
    print(TestVal) # 1

    class NestedClass:
        print(TestVal) # 1

        def first_method(self):
            print(TestVal) # 1

        def second_method(self):
            TestVal = 3
# скрывает имя из объемлющей функции
            print(TestVal) # 3

    my_class = NestedClass()
    my_class.first_method()
    my_class.second_method()

if __name__ == "__main__":
    print(TestVal) # 5
    test_function()
```

Остается еще рассмотреть случай, когда изменяется значение *X* в теле вложенного класса **NestedClass** сразу после его объявления:

```
TestVal = 5

def test_function():
    TestVal = 1 # скрывает глобальное имя
    print(TestVal) # 1
```

```
    class NestedClass:
        TestVal = 4
# скрывает имя из объемлющей функции
        print(TestVal) # 4

        def first_method(self):
            print(TestVal) # 1
            print(self.TestVal) # 4

        def second_method(self):
            TestVal = 3
# Скрывает имя из объемлющей области видимости, а не класса
            print(TestVal) # 3
            self.TestVal = 9
            print(self.TestVal) # 9

    my_class = NestedClass()
    my_class.first_method()
    my_class.second_method()

if __name__ == "__main__":
    print(TestVal) # 5
    test_function()
```

Основное правило, которое следует запомнить из рассмотренных примеров — **поиск простых имен вроде *TestVal* никогда не производится во включающих операторах *class*, а только в операторах *def*, модулях и встроенной области видимости**.

5.13. Перечисления (Enum)

Перечисление — набор символьных имен (членов), привязанных к уникальным постоянным значениям. Так как Python не поддерживает специального синтаксиса для работы с перечислениями, необходимо использовать его библиотечный модуль *enum* [40]. Этот модуль определяет четыре класса перечисления:

1) класс **enum.Enum** — базовый класс для создания перечисляемых констант;

2) класс **enum.IntEnum** — базовый класс для создания перечисляемых констант, которые также являются подклассами **int**;

3) класс **enum.IntFlag** — базовый класс для создания перечисляемых констант, которые можно комбинировать с помощью побитовых операторов без потери их членства в перечислении. Члены **IntFlag** также являются подклассами **int**;

4) класс **enum.Flag** — базовый класс для создания перечисляемых констант, которые можно комбинировать, используя побитовые операции, не теряя членства в перечислении.

Помимо классов перечислений модуль определяет декоратор **unique()** и один помощник *auto*. Декоратор обеспечивает привязку только одного имени к какому-либо одному значению. Помощник заменяет экземпляры соответствующими значениями для членов *Enum*. Начальное значение начинается с 1.

Создать перечисление можно следующим способом:

```
from enum import Enum

class Color(Enum):
    BLACK = 0
    WHITE = 1
    RED = 2
    BLUE = 3

if __name__ == "__main__":
    print(Color.BLACK) # Color.BLACK
    print(Color.BLACK.name) # BLACK
    print(Color.BLACK.value) # 0
```

По перечислениям можно итерироваться в порядке определения их атрибутов:

```
    for it in Color:
        print(it)
# Color.BLACK
# Color.WHITE
# Color.RED
# Color.BLUE
```

В случае дублирования имени атрибута перечисления Python сообщит об ошибке:

```
class TestEnum(Enum):
    ONE = 0
    ONE = 1
# TypeError: Attempted to reuse key: 'ONE'
```

Однако надо быть внимательным при присваивании значений атрибутам перечисления. Так, если значения должны быть неповторяемыми, то приведенный ниже код может при его использовании вызвать ряд проблем:

```
class TestNewEnum(Enum):
    ZERO = 0
    ONE = 1
    TWO = 0

if __name__ == "__main__":
    print(TestNewEnum.ZERO.value) # 0
    print(TestNewEnum.ONE.value) # 1
    print(TestNewEnum.TWO.value) # 0
```

Чтобы найти эту ошибку на этапе компиляции, необходимо использовать декоратор *unique*:

```
from enum import Enum, unique

@unique
class TestNewEnum(Enum):
    ZERO = 0
    ONE = 1
    TWO = 0
# ValueError: duplicate values found in <enum 'TestNewEnum'>:
# TWO -> ZERO
```

либо использовать помощник *auto*:

```
from enum import Enum, auto

class TestNewEnum(Enum):
    ZERO = auto()
    ONE = auto()
    TWO = auto()

if __name__ == "__main__":
    for it in TestNewEnum:
        print(repr(it))
# <TestNewEnum.ZERO: 1>
# <TestNewEnum.ONE: 2>
# <TestNewEnum.TWO: 3>
```

Если перечисление имеет уже описанные атрибуты, то оно не может выступать в качестве базового класса при наследовании:

```
from enum import Enum

class Color(Enum):
    BLACK = 0
    WHITE = 1

class NewColor(Color):
    GREEN = 99
# TypeError: Cannot extend enumerations
```

Для использования класса **Color** в качестве базового можно разделить некоторое общее поведение между ним и его производным классом:

```
from enum import Enum

class Color(Enum):
    def magic(self): ...

class NewColor(Color):
    BLACK = 0
    WHITE = 1
    GREEN = 99
```

Еще один способ создания перечисления — создание экземпляра класса **Enum**:

```
    myColor = Enum('MyColor', 'BLACK WHITE RED BLUE')
    for it in myColor:
        print(repr(it))
# <MyColor.BLACK: 1>
# <MyColor.WHITE: 2>
# <MyColor.RED: 3>
# <MyColor.BLUE: 4>
```

Сравнивать члены перечисления можно следующим образом:

```
print(myColor.BLACK is myColor.BLACK) # True
print(myColor.BLACK is myColor.WHITE) # False
print(myColor.BLACK is not myColor.WHITE) # True
print(myColor.BLACK == myColor.BLACK) # True
print(myColor.BLACK != myColor.WHITE) # True
print(myColor.BLACK == myColor.WHITE) # False
print(myColor.BLACK == 1) # False
```

Для сравнения посредством операций: >, >= < и т. д. необходимо для класса перегружать рассматриваемые ранее методы сравнения.

Еще важно запомнить, что члены перечисления хэшируемые, поэтому их можно использовать в словарях и множествах.

5.13.1. IntEnum

Члены *IntEnum* можно сравнить с целыми числами, а также целочисленные перечисления разных типов можно сравнивать друг с другом:

```
from enum import IntEnum

class FirstEnum(IntEnum):
     GO = 1
     STOP = 2

class SecondEnum(IntEnum):
     TURN = 1
     DOWN = 2

if __name__ == "__main__":
    print(FirstEnum == 1) # False
    print(FirstEnum.GO == 1) # True
    print(FirstEnum.GO == SecondEnum.TURN) # True
```

Однако их нельзя сравнивать со стандартными перечислениями *Enum*:

```
from enum import IntEnum, Enum

class SecondEnum(IntEnum):
     TURN = 1
     DOWN = 2
```

```
class Color(Enum):
    RED = 1
    BLACK = 2

if __name__ == "__main__":
    print(Color.RED == SecondEnum.TURN) # Flase
```

5.13.2. IntFlag

Текущий вариант **Enum**, как и предыдущий, основан на **int**. Различие состоит в том, что члены **IntFlag** могут быть объединены с использованием побитовых операторов (&, |, ^, ~):

```
from enum import IntFlag

class TestFlag(IntFlag):
    A = 1
    B = 2
    C = 3
    D = 4

if __name__ == "__main__":
    print(repr(TestFlag.A | TestFlag.B))
# <TestFlag.C: 3>
    RW = TestFlag.A | TestFlag.D
    print(repr(RW)) # <TestFlag.D|A: 5>
    print(TestFlag.A in RW) # True
    print(repr(TestFlag.D & TestFlag.C))
# <TestFlag.0: 0>
    print(repr(TestFlag.D ^ TestFlag.C))
# <TestFlag.D|C|B|A: 7>
```

Другое важное различие между **IntFlag** и **Enum** заключается в том, что если флаги не установлены (значение равно 0), приведение экземпляра класса **IntFlag** к логическому типу вернет False:

```
print(bool(TestFlag.D & TestFlag.C))  # False
```

Поскольку члены **IntFlag** также являются подклассами **int**, их можно комбинировать с ними:

```
print(repr(TestFlag.D ^ 12)) # <TestFlag.8: 8>
print(repr(TestFlag.A | 6)) # <TestFlag.D|C|B|A: 7>
```

5.13.3. Flag

Как и **IntFlag**, члены **Flag** можно комбинировать с помощью побитовых операторов (&, |, ^, ~), но в отличие от **IntFlag**, члены не могут ни комбинироваться, ни сравниваться ни с другими перечислениями **Flag**, ни с **int**.

В случае работы с **Flag** рекомендуется использовать *auto* для присвоения значения, а не назначать его вручную:

```
from enum import Flag, auto

class BoolFlag(Flag):
    A = auto()
    B = auto()
    C = auto()
    D = auto()

if __name__ == "__main__":
    print(repr(BoolFlag.A | BoolFlag.B))
# <BoolFlag.B|A: 3>
    print(repr(BoolFlag.D & BoolFlag.C))
# <BoolFlag.0: 0>
    print(repr(BoolFlag.D ^ BoolFlag.C))
# <BoolFlag.D|C: 12>
```

Отдельные флаги должны иметь значения степеней двойки (1, 2, 4, 8,…), в то время как комбинации флагов — нет:

```
from enum import Flag, auto

class NewBoolFlag(Flag):
    A = auto()
    B = auto()
    C = auto()
    D = auto()
    E = (A|B|C)^D

if __name__ == "__main__":
    print(repr(NewBoolFlag.E)) # <NewBoolFlag.E: 15>
```

5.13.4. Расширенные возможности перечислений

Так как перечисления реализуются посредством классов, их можно расширять собственными методами или атрибутами:

```
from enum import Enum

class Color(Enum):
    BLACK = 0
    WHITE = 1
    RED = 2
    BLUE = 3

class CarBrand(Enum):
    LADA = 0
    FORD = 1
    KAMAZ = 2

class Car(Enum):
    MY_WORK_CAR = ("e006xx127", Color.BLUE,
                   CarBrand.KAMAZ)
```

```
    MY_HOME_CAR = ("H126yx172", Color.WHITE,
                   CarBrand.LADA)
    MY_WIFE_CAR = ("H125yx001", Color.BLACK,
                   CarBrand.FORD)

    def __init__(self, number, color, brand):
        self.number = number
        self.color = color
        self.brand = brand

if __name__ == "__main__":
    for it in Car:
        print(repr(it))
'''
<Car.MY_WORK_CAR: ('e006xx127', <Color.BLUE: 3>, <CarBrand.
KAMAZ: 2>)>
<Car.MY_HOME_CAR: ('H126yx172', <Color.WHITE: 1>, <CarBrand.
LADA: 0>)>
<Car.MY_WIFE_CAR: ('H125yx001', <Color.BLACK: 0>, <CarBrand.
FORD: 1>)>
'''
    print(Car.MY_HOME_CAR.number) # H126yx172
    print(Car.MY_HOME_CAR.brand.name) # LADA
    print(Car.MY_HOME_CAR.brand.value) # 0
    print(Car.MY_HOME_CAR.color.name) # WHITE
    print(Car.MY_HOME_CAR.color.value) # 1
    print(Car.MY_WORK_CAR.color is Color.BLUE)
# True
    print(Car.MY_WORK_CAR.color is Color.RED)
# False
```

Такое поведение перечислений близко по духу к тому, что имеется в языке программирования Java.

Резюме

В текущем разделе были рассмотрены классы и степень поддержки принципов ООП языком программирования Python. Фактически определение класса равносильно созданию нового типа данных.

Отдельно стоит выделить тот факт, что у классов нет конструктора с возможностью его перегрузки, как в ряде других языков программирования, а для инициализации атрибутов (переменных) класса используется метод **__init__**, вызываемый при создании нового экземпляра класса.

Инкапсуляция поддерживается условно. Ее соблюдение лежит полностью на плечах разработчика и зависит от того, придерживается он соглашений для обеспечения инкапсуляции в коде или нет.

Такой параметр класса, как *self*, ссылается на его текущий экземпляр и передается в первом параметре методов класса. Также вместо него методам класса может передаваться параметр *cls*, который является ссылкой на класс, а статические методы могут вызываться

без создания экземпляра класса и не нуждаются в передаче им в качестве аргумента параметра *self*.

Следует запомнить, что все методы классов в Python являются виртуальными. Таким образом, если метод не переопределяется в производном классе и не является приватным для базового класса, он будет доступен для всех производных классов.

Методы и атрибуты (переменные) классов можно объявить «условно» приватными путем добавления в начало их имени двойного нижнего подчеркивания. Это вызывает механизм искажения, который по-другому формирует такой атрибут класса, но не делает его полностью недоступным.

Вопросы для самопроверки

1. Как в Python реализуются основные принципы ООП?
2. Где процедура поиска в иерархии наследования ищет атрибуты?
3. В чем отличие между объектом класса и объектом экземпляра?
4. Почему первый аргумент в функции метода класса является особым?
5. Каким образом создаются экземпляры и классы?
6. Где и как создаются атрибуты класса?
7. Где и как создаются атрибуты экземпляра?
8. Что self означает в классе Python?
9. Каким образом реализовывать перегрузку операций в классе Python? Для чего применяется метод __init__?
10. Когда может понадобиться поддержка перегрузки операций в классах?
11. Что такое абстрактный суперкласс?
12. Что происходит, когда простое присваивание появляется на верхнем уровне оператора class?
13. Почему в классе может возникнуть потребность в ручном вызове метода __init__ базового класса (суперкласса)?
14. Как можно дополнить унаследованный метод, не замещая его полностью?
15. Чем локальная область видимости класса отличается от локальной области видимости функции?
16. Какие два метода перегрузки операций можно использовать для поддержки итерации в классах?
17. Какие два метода перегрузки операций обрабатывают вывод и в каких контекстах?
18. Как можно перехватывать операции нарезания в классе?
19. Как можно перехватывать сложение на месте в классе?
20. Когда должна предоставляться перегрузка операций?
21. Что такое множественное наследование?
22. Как происходит поиск метода при множественном наследовании?

Упражнения

1. Напишите класс, позволяющий сформировать все уникальные подмножества из списка целых чисел, например: [2, 4, 10, 1].

2. Напишите класс, реализующий все арифметические операции над двумя значениями (a и b).

3. Напишите класс, описывающий такой объект, как автомобиль. Продумайте, какие методы и переменные он должен иметь.

4. Напишите класс, описывающий такой объект, как прямоугольник. Перегрузите у реализованного класса методы сравнения (сравнивать по площади), после чего создайте два экземпляра класса и проверьте, как работают перегруженные методы.

5. Напишите класс, описывающий такой объект, как автомобиль. У него может быть различное количество состояний, реализуемых посредством перечислений. Добавьте методы, позволяющие экземпляру класса менять свое текущее состояние (например: остановка, движение, поворот налево и т. д.).

6. Напишите класс, который подсчитывает текущее количество его экземпляров в приложении. Для корректного отображения этого числа перегрузите у класса метод __del__ и напишите необходимую логику.

7. Напишите базовый класс, задающий интерфейс и часть характеристик (если надо) таких объектов, как геометрические фигуры, автомобиль и магазин, животное.

8. Напишите несколько производных классов от базового класса геометрических фигур (например: прямоугольник и квадрат).

9. Напишите несколько производных классов от базового класса автомобилей (например: легковой и грузовой автомобиль).

10. Напишите несколько производных классов от базового класса магазинов (например: ларек и супермаркет).

11. Напишите несколько производных классов от базового класса автомобилей (например: лошадь и тигр).

12. Напишите класс, доступ к атрибутам (переменным) которого осуществляется с помощью декоратора *@property*.

13. Напишите класс, хранящий целое число. Перегрузите у него методы арифметических операций. Объявите два экземпляра класса и проверьте, как работают перегруженные методы.

14. Напишите класс, который позволяет работать с json-файлом, осуществляя его чтение, запись, добавление, удаление и изменение значений.

15. Напишите класс, который находит прямоугольник с максимальной площадью из списка. Список можете формировать из экземпляров класса, реализованного в упражнении 4.

16. Напишите класс, осуществляющий преобразование целого числа из десятичной системы счисления в двоичную и наоборот.

17. Напишите класс, вычисляющий наименьший общий делитель (НОД). Значения, участвующие в поиске НОД должны устанавливаться отдельными методами или посредством декоратора *@property* до вызова метода расчета.

18. Напишите класс, вычисляющий корни квадратного уравнения.

19. Напишите класс, хранящий данные сотрудника фирмы и имеющий метод, возвращающий характеристики текущего сотрудника в виде словаря.

20. Напишите класс, представляющий собой записную книжку. Каждый элемент записной книжки должен содержать следующие поля: ФИО, номер телефона, e-mail, день рождения. Записная книжка может сохраняться на диск в виде json-файла, а также должна иметь метод загрузки данных из файла.

Тема 6
ИСКЛЮЧЕНИЯ (EXCEPTION)

В результате изучения материалов данной темы обучающиеся должны:

знать

- как устроен механизм исключений в Python,
- основные способы генерации и обработки исключений;

уметь

- создавать и генерировать пользовательские исключения,
- использовать операторы *try/except/finally* и *with/as*;

владеть

- навыками перехвата и обработки исключений, возникающих в процессе работы приложений,
- основными принципами работы с исключениями.

Исключения позволяют в случае ошибки в вычислениях или логике работы программы сразу же перейти к ее обработке, при этом отменяя все вызовы функций, которые начались до того, как был совершен вход в данный обработчик. Их можно рассматривать как некоторый структурированный «безусловный переход». Исходя из этого, оператор *try* представляет собой некоторую метку, к которой при генерации исключения перейдет интерпретатор Python, прекратив выполнение любых функций, вызванных после объявления метки (то есть в теле оператора *try*).

Такое поведение обеспечивает согласованный способ реагирования на необычные события в процессе работы программы и позволяет не вводить дополнительные проверки на коды отработки вызываемых функций.

Перечислим основные способы применения исключения.

1. **Обработка ошибок**. Всякий раз, когда интерпретатор Python обнаруживает ошибки в процессе выполнения программы, он генерирует исключение. У пользователя имеется возможность перехватить и обработать это исключение, либо проигнорировать. В последнем случае интерпретатор остановит выполнение программы и выведет соответствующее сообщение об ошибке.
2. **Уведомление о событиях**. В этом случае исключения используются вместо возвращаемых результирующих кодов, например, у функций, что избавляет от необходимости вводить дополнительные проверки.

3. **Обработка особых случаев**. Могут возникать ситуации, когда условие возникновения исключения наступает настолько редко, что его проще обрабатывать на более высоких уровнях, чем дополнительно усложнять код, внося обработку этой ситуации в различные места программы. Также для проверки того, что условия соответствуют ожидаемым, в процессе разработки может применяться оператор *assert*.
4. **Действия при завершении**. Позволяет быть уверенным, что, например, операции закрытия будут выполнены в любом случае, независимо от того, сгенерировалось исключение в процессе работы или нет.
5. **Редкие потоки управления**. Хотя в Python не существует оператора «безусловного перехода», для этих задач может применяться механизм исключений. Как уже рассматривалось ранее, оператор *break* позволяет выйти только из тела цикла, в котором он встречается, а при наличии блока с большим числом вложенных циклов придется вводить множество дополнительных условий, чтобы выйти из него. Самым простым и элегантным решением в данной ситуации будет сгенерировать исключение при достижении условия выхода из блока вложенных циклов.

6.1. Пользовательские исключения

Такие исключения реализуются с помощью классов, унаследованных от встроенного класса исключения. Обычно им выступает класс по имени **Exception**:

```
class TestException(Exception): pass
```

Пользовательские исключения в основном используются при оповещениях о нарушении логики работы программы, отсутствии каких-либо пользовательских данных и т. д.

Генерация пользовательского исключения производится при помощи оператора *raise*, основы работы с которым будут рассмотрены в следующем параграфе:

```
class TestException(Exception): pass

def test_raise(text):
    if text == "exept":
        raise TestException("My Exception")
    print("No Exception")

def test(text):
    try:
        test_raise(text)
    except TestException as my_ex:
        print(my_ex)
```

```
if __name__ == "__main__":
    my_str = "notExept"
    test(my_str) # No Exception
    my_str = "exept"
    test(my_str) # My Exception
```

Пользовательские исключения позволяют хранить добавочную информацию в своих экземплярах и определять собственные методы, что расширяет их возможности в процессе их обработки. Например, код ниже позволяет записать данные исключения в файл журнала (лог):

```
class TestException(Exception):
    logfile = 'dataerror.txt'

    def __init__ (self, line, file) :
        self.line = line
        self.file = file

    def logerror(self):
        log = open(self.logfile, 'a')
        print('Error at:', self.file, self.line, file=log)

def test():
    try:
        raise TestException(22, 'test.log')
    except TestException as my_ex:
        my_ex.logerror()
        print(my_ex)

if __name__ == "__main__":
    test()
```

6.2. Основы обработки и генерации исключений

Для начала напишем следующую функцию:

```
def intro_function(text, index):
    print(text[index])
```

При ее вызове с допустимым значением аргумента *index* не будет генерироваться никаких исключений:

```
if __name__ == "__main__":
    text_text = "Test"
    intro_function(text_text, 2) # s
```

Но стоит передать значение, которое выходит за пределы длины последовательности, сгенерируется встроенное исключение *IndexError* и интерпретатор прекратит свою работу:

```
intro_function(text_text, 10)
IndexError: string index out of range
```

Такое поведение имеет смысл для простых сценариев. Так как ошибки часто могут носить критический характер, то этот сценарий позволяет изучить сообщение об ошибке и внести исправления в код. Но нередко могут возникать ситуации, когда программа должна продолжать свою работу даже после возникновения внутренних ошибок. Рассмотрим, что для этого нужно сделать, на примере написанной ранее функции. Для того, чтобы программа продолжила свою работу, необходимо ее вызов поместить внутрь оператора *try*, чтобы самостоятельно перехватывать исключения и не вызывать стандартный механизм отработки исключений интерпретатора Python, который приводит к остановке программы:

```
    try:
        intro_function(text_text, 10)
    except IndexError:
        print("Перехват исключения IndexError")
    print("Продолжаем работу!")
Перехват исключения IndexError
Продолжаем работу!
```

Конечно, и следующий код отработает аналогично предыдущему:

```
    try:
        intro_function(text_text, 10)
    except:
        print("Перехват исключения IndexError")
    print("Продолжаем работу!")
```

Но обратите внимание на то, что в нем явно не указывается, какое исключение будет обрабатываться в блоке *except*. Это является плохим тоном при разработке и может значительно усложнить жизнь разработчика при проектировании сложных систем.

Далее рассмотрим существующие операторы для обработки и генерации исключений в Python.

6.2.1. Унифицированный оператор try/except/finally

Ниже приведена структура унифицированного оператора *try/except/finally*:

```
try:
    ...#блок try
except ИмяИсключения:
    ... # перехват исключения ИмяИсключения
except (ИмяИсключения2, ИмяИсключения3):
    ...
# перехват любого из перечисленных исключений
except ИмяИсключения4 as переменная:
    ...
# перехват ИмяИсключения4 и присвоение переменной
   # экземпляра класса исключения
except:
    ...
```

```
# если были сгенерированы все остальные исключения
else: # обработчик для случая отсутствия исключений
    ... #блок else
finally: # охватывает все остальное
    ... # блок finally
```

Код, который подлежит выполнению, помещается в блок *try*. Если при его выполнении сгенерируется исключение, оно будет перехвачено блоками *except*. При этом *except* может перехватывать как определенное исключение, так и любое из перечисляемых при его объявлении. Ссылку на экземпляр класса сгенерированного исключения можно присвоить любой переменной посредством оператора *as*. В том случае, если сгенерировалось исключение, которое не было задано, оно будет перехвачено блоком *except*, у которого не объявлено ни одного параметра.

Если в процессе выполнения кода в блоке *try* не генерируется ни одного исключения, отработает блок *else*. Блок *finally* срабатывает в любом случае, независимо от того, будет ли сгенерировано исключение. А сама конструкция *finally* предназначена для указания действий очистки, которые всегда должны совершаться при выходе из *try*.

В приведенном примере структуры блок *else* можно опустить, если код из него перенести в конец блока *try*. Его использование дает возможность убедиться, что код блока *try* выполнился корректно и без генерации исключений.

Унифицированный оператор *try/except/finally* должен записываться полностью и в своей короткой форме, где он обязан иметь блок *except*, либо *finally*:

```
def example_1(text):
    try:
        text[99]
    except IndexError:
        print('except') # выполняется except
    finally:
        print('finally') # выполняется finally
    print('after try')

def example_2(text):
    try:
        text[3]
    except IndexError:
        print('except') # выполняется except
    else:
        print('else')  # выполняется else
    finally:
        print('finally') # выполняется finally
    print('after try')

def example_3(text):
    try:
        text[3]
```

```
    except IndexError:
        print('except') # выполняется except
    finally:
        print('finally') # выполняется finally
    print('after try')

def example_4():
    try:
        my_error = 1 / 0
    except IndexError:
        print('except')  # выполняется except
    finally:
        print('finally')  # выполняется finally
    print('after run')  # после выполнения

def example_5():
    file = open('data', 'w')
    try:
        raise FileNotFoundError # Генерирует исключение
    finally:
         print('finally')  # выполняется finally
        file.close()
# Всегда закрывать файл, чтобы сбросить буферы
    print('after try')

if __name__ == "__main__":
    my_text = "Test"
    print("run example_1") # run example_1
    example_1(my_text)
# except
# finally
# after try
    print("run example_2")# run example_2
    example_2(my_text)
# else
# finally
# after try
    print("run example_3")# run example_3
    example_3(my_text)
# finally
# after try
    # print("run example_4")
    # example_4()
    # print("run example_5")
    # example_5()
    print("Продолжаем работу!") # Продолжаем работу!
```

В приведенном примере показаны способы обработки исключений. При запуске функций **example_4** или **example_5** после отработки блока *finally* интерпретатор прекратит выполнение программы, так как не предусмотрена обработка тех исключений, что генерируются в процессе выполнения программы:

```
    print("run example_4") # run example_4
    example_4()
# finally
# ZeroDivisionError: division by zero

    print("run example_5") # run example_5
    example_5()
# finally
# FileNotFoundError
```

6.2.2. Оператор raise

Данный оператор используется для явного генерирования исключения. Оператор *raise* состоит из слова raise, за которым дополнительно указывается класс или экземпляр класса генерируемого исключения:

```
raise экземпляр # Генерирует экземпляр класса
raise класс # Создает и генерирует экземпляр класса
raise
# Повторно генерирует самое последнее исключение
```

Давайте рассмотрим наиболее интересный из способов генерации, который позволяет повторно сгенерировать текущее обрабатываемое исключение. Такой подход используется в том случае, когда исчезновение обрабатываемого исключения в коде нежелательно:

```
    try:
        raise IndexError("my error")
    except IndexError:
        print("Обработка исключения")
# Обработка исключения
        raise
'''Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 3, in <module>
    raise IndexError("my error")
IndexError: my error'''
```

В некоторых случаях исключения могут генерироваться в ответ на другие исключения. Для этого, начиная с Python 3.x, оператору *raise* добавили конструкцию *from*:

```
class TestException(Exception): pass

if __name__ == "__main__":
    try:
        raise IndexError("my error")
    except IndexError as ex:
        print("Обработка исключения") # Обработка исключения
        raise TestException("Сцепление") from ex
    print("Работаем дальше!")
'''Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 12, in <module>
    raise IndexError("my error")
```

```
IndexError: my error

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 15, in <module>
    raise TestException("Сцепление") from ex
__main__.TestException: Сцепление '''
```

Когда конструкция *from* используется в явном запросе *raise*, следующее за *from* выражение указывает еще один класс или экземпляр для присоединения к атрибуту *__cause__* нового генерируемого исключения. Если сгенерированное исключение не перехвачено, тогда интерпретатор Python выводит оба исключения как часть стандартного сообщения об ошибке.

При таком подходе сцепленные исключения могут иметь произвольную длину:

```
class TestException(Exception): pass
class NewTestException(Exception): pass
if __name__ == "__main__":
    try:
        try:
            raise IndexError("my error")
        except IndexError as ex:
            print("Обработка исключения")
            raise TestException("Сцепление") from ex
    except TestException as ex:
        raise NewTestException("Еще одно сцепление")from ex
    print("Работаем дальше!")
'''Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 12, in <module>
    raise IndexError("my error")
IndexError: my error

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 15, in <module>
    raise TestException("Сцепление") from ex
__main__.TestException: Сцепление

The above exception was the direct cause of the following
exception:

Traceback (most recent call last):
  File "F:/code/python/module6/part3.py", line 17, in <module>
    raise NewTestException("Еще одно сцепление") from ex
__main__.NewTestException: Еще одно сцепление

'''
```

6.2.3. Оператор assert

В процессе разработки приложения для целей отладки можно использовать оператор *assert*, представляющий собой синтаксическое сокращение для оператора *raise* и имеющий следующую форму записи:

```
assert test, data
# Часть data является необязательной
```

В случае, когда результатом вычисления *test* будет *False*, интерпретатор Python сгенерирует исключение, в котором элемент *data* (если предоставлен) используется как аргумент для конструктора класса исключения.

Так как операторы *assert* посредством флага компилятора удаляются из сгенерированного байт-кода, то не рекомендуется в блоках вычисления *test* использовать методы, меняющие внутренние состояния объектов.

Далее приводится код, демонстрирующий принцип использования оператора *assert*:

```
def test_assert(value):
    assert value > 0, 'value должен быть > 0'
    return 30/value

if __name__ == "__main__":
    print(test_assert(10)) # 3.0
    print(test_assert(0))
'''Traceback (most recent call last):
  File "F:/code/python/module6/part4.py", line 7, in <module>
    print(test_assert(0))
  File "F:/code/python/module6/part4.py", line 2, in test_
assert
    assert value > 0, 'value должен быть > 0'
AssertionError: value должен быть > 0'''
```

Самое важное, о чем необходимо помнить при использовании оператора *assert*, — он служит для проверок ограничений (логики), которые определяет разработчик в приложении, а не для перехвата подлинных программных ошибок.

6.2.4. Оператор with/as и протокол управления контекстами

Оператор *with* вместе с его необязательной конструкцией *as* предназначен для работы с объектами диспетчеров контекстов. Он используется как альтернатива оператору *try/finally* и, подобно ему, предназначен для указания действий стадии завершения или «очистки», которые должны выполняться независимо от того, возникало исключение в процессе выполнения кода или нет.

Диспетчер контекста расширяет возможности ряда встроенных инструментов, таких как файлы (автоматически закрываются) и потоки (автоматически блокируются и деблокируются), а также позволяет разработчикам реализовывать собственные диспетчеры контекстов с применением классов.

Базовый формат оператора *with/as* приведен ниже:

```
with выражение [as переменная]:
    блок-with
```

В данном случае *выражение* должно возвращать объект, поддерживающий протокол управления контекстами, а также может возвращать значение, которое будет присвоено имени *переменная* при наличии необязательной конструкции *as*. Следом за этим, объект может выполнить код запуска перед началом *блока-with*, а также код завершения после окончания *блока-with* независимо от того, генерировалось ли исключение.

Приведем пример работы с файлом посредством рассматриваемой конструкции:

```
with open('dataFile', 'r') as myfile:
    for line in myfile:
    print(line)
    ...дополнительный код...
```

После завершения *блока-with* механизм управления контекстами гарантирует автоматическое закрытие файлового объекта, на который ссылается *myfile* (даже если во время обработки файла в цикле *for* возникло исключение).

Рассмотренный пример работы с файлом полностью идентичен следующему коду с использованием оператора *try/finally*:

```
myfile = open('dataFile', 'r')
try:
    for line in myfile:
        print(line)
       ...дополнительный код...
finally:
    myfile.close()
```

Теперь реализуем собственный объект, поддерживающий работу с диспетчером контекста. Для этого у класса необходимо перегрузить следующие методы:

1) *__enter__*, возвращаемое значение, которое присваивается переменной в конструкции as при ее наличии либо попросту отбрасывается;

2) *__exit__*, который отрабатывается при завершении блока *with* или в случае возникновения исключения при выполнении кода в блоке *with*. Когда генерируется исключение, в атрибуты данного метода передается детальная информация по нему. После отработки

исключения рекомендуется, чтобы данный метод возвращал *False*, тем самым исключение сгенерируется повторно и распространится за пределы блока *with*. Если метод вызывается по завершении блока *with*, в качестве атрибутов передается *None*.

Ниже приведен пример работы с классом, у которого перегружены методы *__exit__* и *__enter__*:

```
class MyError(Exception):pass

class TestWith:
    def print_value(self, value):
        print(f"Входное значение: {value}")

    def __enter__(self):
        print("Начало блока with")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type is None:
            print("Успешное завершение!")
        else:
            print("Было сгенерировано исключение: "+str(exc_
type))
            return False

if __name__ == "__main__":
    with TestWith() as test_object:
# Начало блока with
        test_object.print_value("Test 1")
# Входное значение: Test 1
        print("Конец блока with") # Конец блока with
    with TestWith() as test_object:
# Начало блока with
        test_object.print_value("Test 2")
# Входное значение: Test 2
        raise MyError("Ошибка!")
# Было сгенерировано исключение: <class '__main__.MyError'>
        print("Конец блока with")
'''Traceback (most recent call last):
  File "F:/code/python/module6/part5.py", line 24, in <module>
    raise MyError("Ошибка!")
__main__.MyError: Ошибка! '''
```

6.3. Встроенные классы исключений

Все встроенные исключения организованы в неглубокую иерархию с универсальными категориями базовых классов и уточняющими типами производных классов, которые в свою очередь являются предварительно определенными классами, доступными в виде

встроенных имен в модуле builtins. Ниже приводится упрощенная иерархия исключений:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
        AssertionError
        AttributeError
        BufferError
        EnvironmentError
        EOFError
        ImportError
        LookupError
        MemoryError
        NameError
        ReferenceError
        RuntimeError
        SyntaxError
        SystemError
        TypeError
        ValueError
        Warning
```

BaseException — базовый класс верхнего уровня, который предназначен для прямого наследования классами, определяемыми пользователем.

Exception — базовый класс для определяемых пользователем исключений, связанных с приложением. Его рекомендуется использовать вместо **BaseException**, так как тогда в обработчике оператора *try* гарантируется, что программа будет перехватывать все исключения, кроме событий выхода в систему (то есть не перехватываются: *SystemExit*, *KeyboardInterrupt* и *GeneratorExit*).

Поскольку исключения подвержены самому частому изменению, то модуль *exceptions* был удален в Python 3.x, что не позволяет получить по ним документацию посредством встроенной функции *help*. С подробным их описанием можно ознакомиться в [41].

Резюме

Исключения Python являются высокоуровневым механизмом управления потоком выполнения и могут генерироваться интерпретатором или самим программистом «вручную». При этом их можно игнорировать либо перехватывать посредством различных конструкций оператора *try*. Такие операторы, как *raise* и *assert*,

инициируют различные исключения: встроенные или определяемые пользователем.

Использование диспетчеров контекста посредством оператора *with/as* позволяет гарантировать то, что действия на стадии завершения или «очистки» будут выполняться всегда, независимо от того, возникало исключение в процессе выполнения кода или нет.

Поскольку исключения реализуются в виде объектов экземпляров классов, то они поддерживают концепцию иерархии исключений. Что в свою очередь позволяет облегчить сопровождение кода путем добавления дополнительной информации или методов в виде атрибутов определяемого класса исключения. А также разрешает исключениям наследовать данные и поведение от базовых классов. Это способствует тому, что указания базового класса иерархии исключений в операторе *try* обеспечивает перехват исключения этого класса и всех его производных классов.

Вопросы и задания для самопроверки

1. Назовите три случая, для обработки которых хорошо подходят исключения.
2. Что произойдет с исключением, если вы не предпримете ничего специального для его обработки?
3. Как сценарий может восстанавливаться после исключения?
4. Назовите два способа генерации исключений в сценарии.
5. Назовите два способа указания действий, подлежащих выполнению на стадии завершения вне зависимости от того, возникало исключение или нет.
6. Для чего предназначен оператор try?
7. Для чего предназначен оператор raise?
8. Для чего предназначен оператор assert и на какой другой оператор он похож?
9. Для чего предназначен оператор with/as и на какой другой оператор он похож?
10. Назовите два способа присоединения информации контекста к объектам исключений.

Упражнения

1. Напишите класс, реализующий все арифметические операции над двумя значениями (a и b). В случае, если одно из значений при вызове операции равно нулю, генерируется исключение.
2. Напишите класс, реализующий такие арифметические действия, как деление и умножение. Если одно из значений при вызове операции равно нулю, генерируется исключение, значение ноль меняется на 1 и вычисление операции продолжается.
3. Напишите функцию, возводящую строку в верхний регистр. Добавьте проверку на то, что на вход функции подается не пустая строка.

4. Напишите функцию, проверяющую вхождение задаваемого элемента в список. Добавьте проверку на то, что список не пустой.

5. Реализуйте собственный класс исключения, которое будет генерироваться каждый раз, когда в строке, которая является аргументом для функции, присутствует символ «п».

Тема 7
ПОТОКИ, ПРОЦЕССЫ И АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

В первой теме мы познакомились с такой особенностью Python, как глобальная блокировка интерпретатора (GIL). Именно GIL ответственен за то, что в Python единовременно может выполняться только один поток, из-за чего многопоточные программы не могут использовать преимущества многоядерных процессоров при наличии в них вычислений, работающих на CPU.

При этом, если разрабатываемая программа ориентирована на операции ввода-вывода, что типично для сетевых коммуникаций, потоки или асинхронное программирование являются самым оптимальным выбором, так как в этом случае GIL не задействуется.

В результате изучения данной темы обучающиеся должны:

знать

- особенности работы с модулями *threading*, *multiprocessing* и *asyncio*,
- разницу между многопоточным, асинхронным и многопроцессорным программированием в Python, а также когда и какой из этих подходов к разработке программного обеспечения необходимо применять,

уметь

- создавать классы, которые могут запускаться в отдельном потоке или процессе,
- правильно организовать доступ к разделяемым ресурсам,
- организовывать межпроцессорное и многопоточное взаимодействие;

владеть

- навыками разработки и отладки многопоточных, многопроцессорных и асинхронных приложений.

7.1. Многопоточное программирование

Поток — отдельно выполняемый набор инструкций, использующий глобальное состояние (память) совместно с другими потоками. Из приведенного определения может сложиться впечатление, что потоки выполняются одновременно, но обычно они выполняются поочередно на одном процессоре (ядре). В то же самое время корректное использование многопоточности позволяет повысить бы-

стродействие разрабатываемой программы по сравнению с однопоточным подходом.

7.1.1. Модуль threading и класс Thread

Применяемый в данном модуле подход к многопоточному программированию аналогичен используемому в Java. Отличия заключаются в том, что объекты блокировок и состояний моделируются как отдельные объекты, а также нет возможности извне управлять потоками.

Для работы с потоками в модуле threading содержатся следующие классы: **Thread**, **Condition**, **Semaphore**, **Event**, **Barrier**, **Lock**, **RLock**, **Timer** и **BoundedSemaphore**. При этом все объекты, которые предоставляет модуль, атомарные.

Для использования потоков необходимо импортировать класс **Thread** модуля *threading*, с которым можно работать несколькими способами:

1) в процессе создания экземпляра класса **Thread** именованному аргументу *target* можно передать функцию, которая будет использоваться в качестве основной для данного экземпляра;

2) реализовать производный класс от класса **Thread** и переопределить метод **run**.

Для того, чтобы запустить поток, необходимо у созданного экземпляра класса вызвать метод **start**. Его работа завершится после выполнения основной функции. Каждый объявляемый поток может работать в двух режимах: основной режим и режим демона (аргумент *daemon = True*). В первом случае потоки не дадут программе завершиться, пока все они не выполнятся все их основные функции. Во втором случае поток-демон позволяет завершать выполнение программы, даже если потоки остаются активными.

Ниже приведен пример выполнения функции в различных потоках:

```
from threading import Thread

def thread_work(name):
    print(name)

if __name__=="__main__":
    for it in range(5):
        new_thread = Thread(target=thread_work,args=(f"Thread
#{it}",))
        new_thread.start()
# Thread #0
# Thread #1
# Thread #2
# Thread #3
# Thread #4
```

Теперь рассмотрим аналогичный вариант, с тем отличием, что в нем пользовательский класс наследуется от класса **Thread**. Следует отметить, что для отладки программ не следует использовать функцию **print**, а лучше воспользоваться функциональностью, которую предоставляет модуль *logging*. Так, например, включение имен потоков в протоколируемые сообщения позволяет отслеживать источники этих сообщений:

```
from threading import Thread
import logging

class MyThread(Thread):
    def __init__(self, group=None, target=None,
                 name=None, args=(), kwargs=None, *,
                 daemon=None):
        super().__init__(group=group,
                         target=target, name=name,
                         daemon=daemon)
        self.args = args
        self.kwargs = kwargs

    def run(self) -> None:
        logging.debug(f'Thread args = {self.args} '
                      f'and kwargs = {self.kwargs}')

if __name__=="__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='(%(threadName)-10s) %(message)s',
    )

    for it in range(5):
        new_thread = MyThread(args=(it,),
             kwargs={it: it+2, f"{it}": str(it)*2})
        new_thread.start()
''' (Thread-1  ) Thread args = (0,) and kwargs = {0: 2, '0':
'00'}
(Thread-2  ) Thread args = (1,) and kwargs = {1: 3, '1': '11'}
(Thread-3  ) Thread args = (2,) and kwargs = {2: 4, '2': '22'}
(Thread-4  ) Thread args = (3,) and kwargs = {3: 5, '3': '33'}
(Thread-5  ) Thread args = (4,) and kwargs = {4: 6, '4': '44'}
'''
```

7.1.2. Потоки Timer

Экземпляры класса **Timer** начинают работать с некоторой задержкой, определяемой пользователем. Кроме того, их выполнение можно отменить (или завершить) в любой момент периода задержки:

```
from threading import Timer
import logging
import time
```

```
def thread_work():
    logging.debug("Поехали!")

if __name__=="__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='(%(threadName)-10s) %(message)s',
    )

    my_timer1 = Timer(0.3, thread_work)
    my_timer1.setName("MyTreadTimer-1")
    my_timer2 = Timer(0.3, thread_work)
    my_timer2.setName("MyTreadTimer-2")

    logging.debug("Запуск таймеров!")
# (MainThread) Запуск таймеров!
    my_timer1.start()
    my_timer2.start()

    logging.debug(f"Задержка перед отменой
                f'выполнения {my_timer2.getName()}")
# (MainThread) Задержка перед отменой выполнения MyTreadTimer-2
    time.sleep(0.2)
    logging.debug(f"Отмена потока – "
                  f" {my_timer2.getName()}")
# (MainThread) Отмена потока - MyTreadTimer-2
    my_timer2.cancel()
    logging.debug("Завершение")
# (MainThread) Завершение
# (MyTreadTimer-1) Поехали!
```

В приведенном примере второй таймер не успевает вызвать функцию, так как его работу отменили. А первый таймер выполняется после завершения основной части программы, и поскольку этот поток не является демоном, основная программа завершится только после его выполнения.

7.1.3. Класс RLock. Управление доступом к ресурсам

При работе со встроенными структурами данных используются атомарные операции, в то время как простые типы данных (числа, целые или с плавающей точкой) не имеют такой защиты. В этом случае для контроля доступа к разделяемым ресурсам, чтобы избежать потери данных или их повреждения, используется так называемый замок — классы **Lock** или **RLock**.

При работе с **Lock** может возникнуть ситуация самоблокировки из-за обращения к экземпляру класса Lock из потока, в котором уже осуществляется блокировка доступа к ресурсам.

Чтобы избежать таких проблем, рекомендуется использовать **RLock**. Данная реализация замка позволяет выполнить блокиров-

ку только в том случае, если замок удерживает другой поток, что предотвращает нежелательную блокировку:

```
from threading import Thread, RLock
import logging
import time

class DoubleCounter:
    def __init__(self):
        self.first_counter = 1
        self.second_counter = 5
        self.lock = RLock()

    def increment_first_counter(self):
        with self.lock:
# для блокировки используем диспетчер контекста
            self.first_counter += 1
            logging.debug(f"New first counter value is "
                          f"{self.first_counter }")
    def increment_second_counter(self):
        with self.lock:
            self.second_counter += 1
            logging.debug(f"New second counter value is "
                          f"{self.second_counter}")


    def increment(self):
        with self.lock:
            self.increment_first_counter()
            self.increment_second_counter()


def thread_work(counter):
    counter.increment()
    time.sleep(0.2)
    counter.increment_second_counter()
    time.sleep(0.3)
    counter.increment_first_counter()

if __name__ == "__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='(%(threadName)-10s) %(message)s',
    )
    my_counter = DoubleCounter()
    logging.debug("Start")
    for it in range(3):
        new_thread = Thread(target=thread_work, args=(my_
counter,))
        new_thread.start()
    logging.debug("Finish")
'''
(MainThread) Start
(Thread-1  ) New first counter value is 2
```

```
(Thread-1  ) New second counter value is 6
(Thread-2  ) New first counter value is 3
(Thread-2  ) New second counter value is 7
(Thread-3  ) New first counter value is 4
(MainThread) Finish
(Thread-3  ) New second counter value is 8
(Thread-1  ) New second counter value is 9
(Thread-2  ) New second counter value is 10
(Thread-3  ) New second counter value is 11
(Thread-1  ) New first counter value is 5
(Thread-2  ) New first counter value is 6
(Thread-3  ) New first counter value is 7
'''
```

Для того, чтобы понять разницу между **Lock** и **RLock**, замените в примере выше вариант блокировки и запустите скрипт.

7.1.4. Синхронизация потоков

Бывают случаи, когда необходимо синхронизировать операции, выполняемые в двух или более потоках. Для этого модуль *threading* предоставляет механизм событий, который обеспечивает простой способ организации безопасного взаимодействия потоков. Класс **Event** имеет внутренний флаг, который могут устанавливать или сбрасывать другие потоки посредством методов **set** и **clear**. Метод **wait** класса **Event** позволяет приостановить работу потока до тех пор, пока другой поток не установит указанный флаг, или спустя задаваемый период времени (в секундах):

```
from threading import Thread, Event
import logging
import time

def event_work(event):
    logging.debug("run event_work")
    event_wait = event.wait()
    logging.debug("Флаг установлен")

def event_with_timeout(event, time):
    while not event.is_set(): # флаг установлен?
        logging.debug(f"Ожидание установки флага
                    " или истечение "
                    "времени в event_with_timeout")
        event_wait = event.wait(time)
        logging.debug("Флаг установлен")
        if event_wait:
            logging.debug("Обработка события")
        else:
            logging.debug("Флаг не был установлен")

if __name__=="__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='(%(threadName)-10s) %(message)s', )
```

```
    event = Event()
    my_thread1 = Thread(name='event_thread',
                        target=event_work,
                        args=(event,))
    my_thread1.start()
    my_thread2 = Thread(name='event_timeout',
                        target=event_with_timeout,
                        args=(event, 3))
    my_thread2.start()
    logging.debug('Задержка перед установкой флага')
    time.sleep(0.3)
    event.set()
    logging.debug("Флаг установлен")
'''
(event_thread) run event_work
(event_timeout) Ожидание установки флага или истечение времени
в event_with_timeout
(MainThread) Задержка перед установкой флага
(MainThread) Флаг установлен
(event_timeout) Флаг установлен
(event_timeout) Обработка события
(event_thread) Флаг установлен
'''
```

Помимо класса **Event**, для синхронизации потоков еще используются **Condition** и **Barrier**. Класс **Condition** является некоторой комбинацией классов **Event** и **Lock**, что позволяет нескольким потокам ожидать, пока ресурс, с которым производится работа, не будет обновлен. Класс **Barrier** позволяет создавать так называемую «контрольную точку» для задаваемого количества потоков. Каждый из этих потоков, достигая контрольной точки, блокируется до тех пор, пока ее не достигнут все потоки, участвующие в этом механизме блокировки.

Ниже приводится пример использования **Condition**:

```
from threading import Thread, Condition
import logging
import time

def slave(condition):
    logging.debug('Запуск потока ведомого')
    with condition:
        condition.wait()
# ожидаем разблокировки ресурса
        logging.debug('Блокировка с ведомого снята')

def master(condition):
    logging.debug('Запуск потока ведущего')
    with condition:
        logging.debug('Разблокировка ресурса')
        condition.notifyAll()
```

```
if __name__ == "__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='(%(threadName)-3s) %(message)s',
    )

    condition = Condition()
    slave1 = Thread(name='slave1', target=slave,
                    args=(condition,))
    slave2 = Thread(name='slave2', target=slave,
                    args=(condition,))
    master = Thread(name='master', target=master,
                    args=(condition,))
    slave1.start()
    time.sleep(0.1)
    slave2.start()
    time.sleep(0.1)
    master.start()
'''
(slave1) Запуск потока ведомого
(slave2) Запуск потока ведомого
(master) Запуск потока ведущего
(master) Разблокировка ресурса
(slave1) Блокировка с ведомого снята
(slave2) Блокировка с ведомого снята
'''
```

7.1.5. Семафоры

Порой может случиться, что необходимо разрешить одновременный доступ к ресурсу сразу нескольким потокам или иметь возможность ограничить общее количество потоков, которые имеют доступ к ресурсу в один момент времени. Примером такой ситуации может выступать пул потоков, задача которого — поддерживать фиксированное количество соединений, загрузок и т. д.

Для решения таких задач используются семафоры (класс **Semaphore**). Ниже приведен пример работы класса **TestPool**, который обеспечивает удобный способ отслеживания работы потоков. Естественно, в реальных системах на данном пуле лежала бы еще задача распределения соединений и ряда значений между всеми активными потоками и возврат их в пул после завершения потока:

```
from threading import Thread, Semaphore, Lock, current_thread
import logging
import time

class TestPool:

    def __init__(self):
        super(TestPool, self).__init__()
        self.active = []
        self.lock = Lock()
```

```
    def makeActive(self, name):
        with self.lock:
            self.active.append(name)
            logging.debug(f'Потоков в пуле'
                          f' {self.active}')

    def makeInactive(self, name):
        with self.lock:
            self.active.remove(name)
                logging.debug(f'Потоков в пуле'
                              f' {self.active}')



def worker(semaphore, pool):
    logging.debug('Ожидание очереди при '
                  'подключении к пулу')
    with semaphore:
        name = current_thread().getName()
        pool.makeActive(name)
        time.sleep(0.1)
        pool.makeInactive(name)


if __name__=="__main__":
    logging.basicConfig(
        level=logging.DEBUG,
        format='%(asctime)s (%(threadName)-1s) '
                '%(message)s',
        datefmt='%I:%M:%S %p'
    )

    pool = TestPool()
    semaphore = Semaphore(2)
    for it in range(6):
        new_thread = Thread(target=worker,
                            name=str(it),
                            args=(semaphore, pool),
                            )
        new_thread.start()
'''
12:14:04 PM (0) Ожидание очереди при подключении к пулу
12:14:04 PM (0) Потоков в пуле ['0']
12:14:04 PM (1) Ожидание очереди при подключении к пулу
12:14:04 PM (1) Потоков в пуле ['0', '1']
12:14:04 PM (2) Ожидание очереди при подключении к пулу
12:14:04 PM (3) Ожидание очереди при подключении к пулу
12:14:04 PM (4) Ожидание очереди при подключении к пулу
12:14:04 PM (5) Ожидание очереди при подключении к пулу
12:14:05 PM (0) Потоков в пуле ['1']
12:14:05 PM (2) Потоков в пуле ['1', '2']
12:14:05 PM (1) Потоков в пуле ['2']
12:14:05 PM (3) Потоков в пуле ['2', '3']
```

```
12:14:05 PM (2) Потоков в пуле ['3']
12:14:05 PM (4) Потоков в пуле ['3', '4']
12:14:05 PM (3) Потоков в пуле ['4']
12:14:05 PM (5) Потоков в пуле ['4', '5']
12:14:05 PM (5) Потоков в пуле ['4']
12:14:05 PM (4) Потоков в пуле []
'''
```

7.2. Multiprocessing

Процесс — это экземпляр выполняющейся программы. Процессы, которым необходимо участвовать в обмене данными с другими процессами, должны явно организовать такой обмен посредством механизмов межпроцессного взаимодействия (IPC). Они могут передавать данные между друг другом посредством файлов, баз данных, встроенных структур данных, сетевого взаимодействия и т. д.

Модуль *multiprocessing* предоставляет пользователю схожую функциональность в части синхронизации (*Event*, *Condition* и т. д.) и блокировки общих ресурсов (*Lock*, *Semaphore* и т. д.) с модулем *threading*. Поэтому в данном разделе сфокусируем свое внимание на различных вариантах создания процессов и межпроцессорном взаимодействии.

7.2.1. Модуль multiprocessing и класс Process

Для использования процессов необходимо импортировать класс **Process** модуля *multiprocessing*, с которым можно работать, как и с потоками, несколькими способами:

1) в процессе создания экземпляра класса **Process** именованному аргументу *target* можно передать основную функцию, которая будет выполняться в отдельном процессе;

2) реализовать производный класс от класса **Process** и переопределить метод **run**.

Чтобы запустить процесс, необходимо у созданного экземпляра класса вызвать метод **start**. Его работа завершится после выполнения основной функции. Каждый объявляемый процесс может работать в двух режимах: основной режим и режим демона (аргумент *daemon = True*).

Ниже приведен пример выполнения функции различными процессами.

```
from multiprocessing import Process

def process_run(num):
    print(f"Процесс № {num}")
```

```
if __name__ == "__main__":
    for it in range(5):
        my_process = Process(target=process_run,
                             args=(it,))
        my_process.start()
# Процесс № 0
# Процесс № 1
# Процесс № 2
# Процесс № 3
# Процесс № 4
```

Теперь рассмотрим вариант с производным пользовательским классом, от класса **Process**:

```
from multiprocessing import Process

class MyProcess(Process):

    def __init__(self, group=None, target=None,
                 name=None, args=(), kwargs=None,
                 *, daemon=None):
        super().__init__(group=group,target=target,
                         name=name, daemon=daemon)
        self.args = args
        self.kwargs = kwargs


 def run(self) -> None:
        print(f'Process #{self.args[0]} run with '
              f' args = {self.args} '
              f'and kwargs = {self.kwargs}')

if __name__ == "__main__":
    for it in range(5):
        my_process = MyProcess(args=(it,),
                               kwargs={it: it+2,
                               f"{it}": str(it)*2}
                               )
        my_process.start()
'''
Process #0 run with args = (0,) and kwargs = {0: 2, '0': '00'}
Process #1 run with args = (1,) and kwargs = {1: 3, '1': '11'}
Process #2 run with args = (2,) and kwargs = {2: 4, '2': '22'}
Process #3 run with args = (3,) and kwargs = {3: 5, '3': '33'}
Process #4 run with args = (4,) and kwargs = {4: 6, '4': '44'}
'''
```

7.2.2. Взаимодействие между процессами

Самый простой способ взаимодействия между процессами — использование встроенной структуры данных *Queue* из модуля *multiprocessing*:

```
from multiprocessing import Process, Queue

def pow_square(value, queue):
    for i in value:
        queue.put(i*i)

def pow_cube(value, queue):
    for i in value:
        queue.put(i*i*i)

if __name__ == "__main__":
    my_numbers = range(3)

    queue = Queue()
    process_pow_square = Process(target=pow_square,
                                 args=(my_numbers,
                                 queue) )
    process_pow_cube = Process(target=pow_cube,
                               args=(my_numbers,
                                     queue) )
    process_pow_square.start()  # стартуем процесс
    process_pow_cube.start()

    process_pow_square.join()
# ожидаем завершение процесса
    process_pow_cube.join()

    while not queue.empty():
        print(queue.get(), end="  ")
# 0  1  4  0  1  8
```

Еще одним способом взаимодействия между процессами является добавление общего пространства имен. Для этого используется класс **Manager**, который также предоставляет доступ к встроенным типам данных модуля *multiprocessing*:

```
from multiprocessing import Process, Manager, Event

def slave(ns, event):
    ns.my_list.append(1)
    ns.my_list.append(2)
    ns.my_list.append(3)
    ns.my_list.append("new value")
    ns.my_value = 3.14
    event.set()

def master(ns, event):
    print(f'my_list до установки флага события: '
          f'{ns.my_list}')
    print(f'my_value до установки флага события: '
          f'{ns.my_value}')
    event.wait()
```

```
    print(f'my_list после установки флага события: '
          f'{ns.my_list}')
    print(f'my_value после установки флага события:'
          f'{ns.my_value}')

if __name__ == '__main__':
    mgr = Manager()
    namespace = mgr.Namespace()
# общее пространство имен
    namespace.my_list = mgr.list() # создаем список
# объявляем переменную
    namespace.my_value = mgr.Value('d', 0.0)

    event = Event()
    slave_process = Process(
        target=slave,
        args=(namespace, event),
    )
    master_process = Process(
        target=master,
        args=(namespace, event),
    )

    master_process.start()
    slave_process.start()

    master_process.join()
    slave_process.join()
'''
my_list до установки флага события: [1]
my_value до установки флага события: Value('d', 0.0)
my_list после установки флага события: [1, 2, 3, 'new value']
my_value после установки флага события: 3.14
'''
```

Также имеется возможность использовать сетевое взаимодействие для организации передачи данных между потоками. Для этого из *multiprocessing .connection* необходимо импортировать классы **Listener** и **Client**:

```
from multiprocessing.connection import (Listener,
                                        Client)
from multiprocessing import Process
from array import array

def server():
    address = ('localhost', 6000)
# family is deduced to be 'AF_INET'
    with Listener(address,
         authkey=b'secret password') as listener:
        with listener.accept() as conn:
            print(f'connection accepted
                  f'from {listener.last_accepted}')
```

```
            conn.send([2.25, None, 'junk', float])
            conn.send_bytes(b'hello')
            conn.send_bytes(array('i', [42, 1729]))

def client():
    address = ('localhost', 6000)
    with Client(address,
                authkey=b'secret password') as conn:
        print(conn.recv())
# => [2.25, None, 'junk', float]
        print(conn.recv_bytes())  # => b'hello'
        arr = array('i', [0, 0, 0, 0, 0])
        print(conn.recv_bytes_into(arr))  # => 8
        print(arr)
# => array('i', [42, 1729, 0, 0, 0])

if __name__ == "__main__":
    my_server = Process(target=server)
    my_client = Process(target=client)
    my_server.start()
    my_client.start()
```

7.2.3. Пул процессов

Для управления фиксированным количеством процессов, когда можно разделить работу на независимые части, распределяемые между процессами, используется класс **Pool**. В этом случае возвращаемые значения отдельных процессов объединяются и возвращаются в виде списка.

Конструктор класса имеет такие аргументы, как количество процессов и функция, подлежащая выполнению при запуске процесса отдельной задачи. В качестве примера рассмотрим параллельную обработку списка, элементы которого будут возводиться в квадрат. Для этого используем функцию **map** модуля *multiprocessing*. Синхронизацию основного процесса с процессами, выполняющими работу, можно производить с помощью методов **close** и **join**. Такой подход гарантирует выполнение завершающих операций по освобождению неиспользуемых ресурсов:

```
from multiprocessing import Process, \
    current_process, cpu_count, Pool

def square(data):
    return data * data

def start_process():
    print(f'Старт процесса - '
          f'{current_process().name}')

if __name__ == '__main__':
    inputs = list(range(10))
    print(f'Начальный список: {inputs}')
```

```
    builtin_map = map(square, inputs)
    print(f'Встроенная функция map: '
          f' {list(builtin_map)}')

    pool_size = cpu_count() * 2
    print(f'Количество ядер у процессора: '
          f' {cpu_count()}')
    print(f'Количество запускаемых процессов: '
          f'{pool_size}')
    pool = Pool(
        processes=pool_size,
        initializer=start_process,
    )
    pool_map = pool.map(square, inputs)
    pool.close()
    pool.join()

    print(f'Результат работы пула процессов:
          f'{pool_map}')
'''
Начальный список: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Встроенная функция map: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Количество ядер у процессора: 12
Количество запускаемых процессов: 24
Старт процесса - SpawnPoolWorker-1
Старт процесса - SpawnPoolWorker-2
Старт процесса - SpawnPoolWorker-5
...
Старт процесса - SpawnPoolWorker-24
Результат работы пула процессов: [0, 1, 4, 9, 16, 25, 36, 49,
64, 81]
'''
```

7.3. Асинхронное программирование

Модуль асинхронного программирования *asyncio* предоставляет инструменты для создания приложений, основанных на выполнении параллельных вычислений с использованием сопрограмм. То есть он использует подход на основе единственного потока и единственного процесса, в котором отдельные части приложения кооперируются для явного переключения задач в наиболее подходящие для этого моменты времени. Аналогично предыдущим рассмотренным модулям, модуль *asyncio* имеет конструкции для синхронизации выполняемого кода и блокировок ресурсов, а также свой набор встроенных типов данных.

Центральное место в модуле *asyncio* занимает «цикл событий» — объект первого класса, ответственный за эффективную обработку событий ввода-вывода и изменение контекста приложения. Существует несколько вариантов реализации цикла событий. В обычной

ситуации выбирается цикл событий по умолчанию, но у пользователя имеется возможность выбора. Это особенно актуально при запуске программы на различных операционных системах.

Взаимодействие приложения и цикла событий происходит следующим образом. Сначала выполняется регистрация кода, подлежащего асинхронному выполнению, после чего цикл событий выполняет необходимые вызовы в коде приложения, если доступны ресурсы.

Возврат управления циклу событий реализован с помощью сопрограмм — конструкции языка, предназначенной для параллельного выполнения операций [42]. При вызове функции сопрограммы создается ее объект, что позволяет вызывающему коду выполнить код этой функции, используя метод **send**. Приостановить выполнение сопрограммы можно с помощью ключевого слова **await** совместно с именем другой сопрограммы. На протяжении такой паузы состояние сопрограммы сохраняется, что позволяет ей при пробуждении продолжить выполнение с той точки, в которой оно было прервано.

Чтобы функция выполнялась в асинхронном режиме и выступала в роли сопрограммы, перед ее определением необходимо добавить ключевое слово **async**. При этом цикл событий модуля *asyncio* может запустить сопрограмму различными способами. Самый простой вызов у цикла события метода **run_until_complete** с непосредственной передачей ему объекта сопрограммы:

```
import asyncio

async def test_async():
    print(await func()) # Hello, async world!

async def func():
    return "Hello, async world!"

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(test_async())
```

Еще один способ объявления сопрограммы — использование функции-генератора, обернутой декоратором *@asyncio.coroutine*:

```
import asyncio

@asyncio.coroutine
def test_async():
    print((yield from func())) # Hello, async world!

@asyncio.coroutine
def func():
    return "Hello, async world!"
```

```
if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(test_async())
```

Ниже приведен пример асинхронной работы двух функций:

```
import asyncio

async def first_function():
    print("first_function start")
    for it in range(7):
        await asyncio.sleep(1.2)
        print(f"first_function {it}")

async def second_function():
    print("second_function start")
    for it in range(10):
        await asyncio.sleep(0.9)
        print(f"second_function {it}")

if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    my_functions = asyncio.wait([first_function(),
                                 second_function()])
    loop.run_until_complete(my_functions)
```

7.3.1. Планирование времени вызова функций

Цикл событий модуля *asyncio* позволяет планировать вызовы функций, основываясь на значении внутреннего таймера цикла. Существует несколько способов планирования вызова функции.

— «На ближайшее время». Используется метод **call_soon**, который позволяет запланировать вызов callback-функции на следующую итерацию цикла. В качестве первого аргумента функции выступает ссылка на callback-функцию, а после следуют любые дополнительные позиционные аргументы.

— С задержкой во времени. В данном случае используется метод **call_later**. В качестве первого аргумента метода выступает длительность задержки в секундах, следом идет ссылка на callback-функцию и позиционные аргументы.

— На определенное время. Такой способ позволяет запланировать вызов callback-функции на определенное время (метод **call_at**). Чтобы выбрать время для запланированного обратного вызова, необходимо вести отсчет от внутреннего состояния монотонных часов, то есть от момента «сейчас», это позволяет сделать метод **time** цикла событий. Последовательность подаваемых аргументов на вход метода **call_at** аналогична предыдущему способу.

Если необходимо в callback-функцию передать ключевые аргументы, для этого можно использовать функцию **partial** из модуля *functools*:

```
import asyncio
import functools

def callback(arg, kwarg='default'):
    print(f'Вызов callback-функции arg= {arg}  '
          f'kwarg = {kwarg}')

async def main(loop):
    print('Регистрируем callback-функцию')
    loop.call_soon(callback, 1)
    wrapped = functools.partial(callback,
                                kwarg='^_^')
    loop.call_soon(wrapped, "-_-")
    await asyncio.sleep(0.2)

if __name__ == "__main__":
    event_loop = asyncio.get_event_loop()
    try:
        print('Запуск цикла событий')
        event_loop.run_until_complete(main(event_loop))
    finally:
        print('Остановка цикла событий')
        event_loop.close()
'''
Запуск цикла событий
Регистрируем callback-функцию
Вызов callback-функции arg= 1  kwarg = default
Вызов callback-функции arg= -_-  kwarg = ^_^
Остановка цикла событий
'''
```

7.3.2. Асинхронное получение результатов

Для асинхронного извлечения результатов из сопрограмм можно использовать класс **Future** (фьючерс), который сам действует подобно сопрограмме. Именно поэтому к фьючерсам применимы любые приемы, используемые для ожидания завершения сопрограммы. То есть экземпляр класса **Future** можно использовать вместе с ключевым словом *await*.

При этом экземпляры класса **Future** могут не только работать как сопрограммы, но и запускать callback-функции при завершении работы. Перед извлечением результата из экземпляра класса **Future** у него должен вызываться метод **set_result** с передачей ему результирующего значения:

```
import asyncio
import functools

def callback(future, arg):
    print(f'{arg}: результат future = '
          f' {future.result()}')
```

```
async def register_callbacks(all_done):
    print('Регистрируем callback-функцию для '
          ' future')
    all_done.add_done_callback(functools.partial(callback,
                                                 arg=1))
    all_done.add_done_callback(functools.partial(callback,
                                                  arg=2))

async def main(all_done):
    await register_callbacks(all_done)
    print('Установка результирующего '
          'значения future')
    all_done.set_result({"-_-": 10, "^_^": 20})

if __name__ == "__main__":
    event_loop = asyncio.get_event_loop()
    try:
        all_done = asyncio.Future()
        event_loop.run_until_complete(main(all_done))
    finally:
        print(f"Основной процесс: "
              f" {all_done.result()}")
        event_loop.close()
# 1: результат future = {'-_-': 10, '^_^': 20}
# 2: результат future = {'-_-': 10, '^_^': 20}
# Основной процесс: {'-_-': 10, '^_^': 20}
```

7.3.3. Параллельное выполнение задач

Задачи (*Task*) — один из основных способов взаимодействия с циклом событий. Они являются обертками над сопрограммами и отслеживают момент их завершения. Так как задачи являются производным классом от *Future*, то наследуют часть его функциональности, а именно:

— другие сопрограммы могут ожидать завершения задач;

— каждая задача имеет результат, который может быть извлечен после ее завершения.

Чтобы создать экземпляр класса **Task**, необходимо у объекта цикла события вызвать метод **create_task**. Результирующая задача будет выполняться как часть параллельных операций, управляемых циклом событий, до тех пор, пока выполняется цикл и сопрограмма не возвращает управление.

До момента завершения задачи ее можно отменить, вызвав у экземпляра класса метод **cancel** и тем самым сгенерировав исключение *CancelledError*, которое будет необходимо обработать.

Давайте приведем пример работы с задачами:

```
import asyncio

async def task_test():
    print('Выполняем задачу')
    return (30, "-_-", [4.6, "o_O"])
```

```
async def main(loop):
    print('Объявляем задачу')
    task = loop.create_task(task_test())
    print(f'Ожидание выполнения задачи: {task}')
    return_value = await task
    print(f'Завершение задачи: {task}')
    print(f'Результат работы задачи: '
          f'{return_value}')

if __name__ == "__main__":
    event_loop = asyncio.get_event_loop()
    try:
        event_loop.run_until_complete(main(event_loop))
    finally:
        event_loop.close()
'''
Объявляем задачу
Ожидание выполнения задачи: <Task pending coro=<task_test()
running at
F:/code/python/module7/async_part5.py:3>>
Выполняем задачу
Завершение задачи: <Task finished coro=<task_test() done, defined
at
 F:/code/python/module7/async_part5.py:3> result=(30, '-_-',
[4.6, 'o_O'])>
Результат работы задачи: (30, '-_-', [4.6, 'o_O'])
'''
```

7.4. Что и когда использовать?

Если разрабатываемая вами программа интенсивно использует операции ввода-вывода (бóльшая часть времени работы программы тратится на выполнение этих операций), то лучше использовать асинхронное программирование, но только при условии, что операции ввода-вывода будут неблокирующими. В противном случае лучше использовать модуль *threading*.

Если разрабатываемая программа интенсивно производит вычисления на процессоре, модуль *threading* не улучшит производительность программы, так как за дело возьмется GIL. Конечно, на него можно не обращать внимания, если переключиться на Jython и IronPython, которые не используют GIL. Есть и еще одно решение — написать на Си собственное расширение, способное на время вычислений отключать GIL, как, например, библиотека NumPy. В данном случае лучше вспомнить про модуль *multiprocessing* и не бросаться в крайности. Особенно он поможет в ситуациях, когда имеется возможность выполнять программу параллельно или распределенно на нескольких компьютерах.

Резюме

В данном разделе мы рассмотрели возможности таких модулей Python, как *threading*, *multiprocessing* и *asyncio*, их основные свойства, механизмы работы, а также то, когда лучше применять тот или иной подход при распараллеливании операций.

Самое важное, что стоит помнить при использовании любого из этих модулей — работа с ними подвержена ошибкам, трудно поддающимся тестированию и отладке. Поэтому, если нет необходимости, лучше на начальных стадиях разработки программного обеспечения придерживаться однопоточного подхода.

Вопросы и задания для самопроверки

1. Назовите ключевые отличия в принципах работы модулей *threading*, *multiprocessing* и *asyncio*.
2. В каких случаях лучше применять асинхронное программирование?
3. Почему при вычислениях, интенсивно использующих CPU, не следует использовать многопоточное программирование?
4. Назовите способы межпроцессорного взаимодействия.
5. За что отвечает класс **Timer** в модуле *threading*?
6. Для чего используется блокировка доступа к общим ресурсам?
7. Какие имеются способы синхронизации потоков?
8. Какие имеются способы синхронизации процессов?
9. Что такое семафор? В каких задачах применяется?
10. Перечислите основные способы планирования вызова функции при асинхронном программировании.
11. Для чего используется класс **Future**?
12. Принцип работы экземпляров класса **Event** и **Condition**.
13. Для каких задач можно использовать пул процессов?

Упражнения

1. Напишите программу, которая осуществляет чтение данных из файла посредством одного потока и запись этих данных в другом потоке в файл. Названия файлов должны отличаться.
2. Напишите программу, которая осуществляет чтение из файла посредством одного процесса и запись этих данных в другом процессе в файл. Названия файлов должны отличаться.
3. Напишите программу, которая, используя механизм асинхронного программирования, осуществляет чтение данных из одного файла и их запись в другой файл.
4. Напишите программу, осуществляющую перемножение двух матриц с использованием потоков и процессов. Каждая строка новой матрицы должна высчитываться в отдельном потоке (процессе).
5. Напишите программу, в которой несколько процессов осуществляют увеличение значения общего для них счетчика.

6. Напишите программу, реализующую задачу про обедающих философов на основе процессов.

7. Напишите программу, в которой 10 списков заполняются случайными значениями, после чего для каждого из списка в отдельном потоке (процессе) находится медиана.

8. Напишите программу, которая заполняет 20 случайными значениями *Queue*, передаваемую четырем запускаемым потокам (процесса), после чего выведите в консоль извлекаемые из структуры данных значения с информацией о том, в каком потоке (процессе) это произошло.

9. Напишите программу, которая реализует следующую логику. Имеется общий счетчик для двух потоков (процессов), максимальное значение которого равно 50. Каждый из потоков (процессов) ожидает, пока другой увеличит значение счетчика на 5, после чего тот поток переходит в режим ожидания, а текущий начинает увеличивать значение счетчика.

10. Напишите программу, в которой один поток (процесс) осуществляет чтение данных из файла, а три других потока (процесса) осуществляют их запись в файл. У каждого потока (процесса) свой файл для записи данных. При этом учтите, что файлы, в которые производится запись, не должны содержать в себе один и тот же набор данных, то есть прочитанная строка из файла может быть единожды записана в один из трех файлов.

Тема 8
РАЗРАБОТКА ГРАФИЧЕСКОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

В результате изучения данной темы обучающиеся должны:

знать

- основные принципы разработки графических пользовательских интерфейсов (GUI),
- существующие различия между библиотекой PySide2 и PyQt5 для разработки приложений с графическим пользовательским интерфейсом;

уметь

- компоновать виджеты GUI создаваемого приложения,
- использовать механизм сигнал-слотового взаимодействия между классами в рамках фреймворка Qt;

владеть

- навыками разработки приложений с графическим пользовательским интерфейсом.

Существует достаточное количество библиотек или фреймворков, позволяющих разрабатывать графический пользовательский интерфейс (GUI) программ на языке Python. Например, в Python для этих целей изначально встроена библиотека Tkinter, которая подключается импортом пакета *tkinter*. Это не самая популярная библиотека для разработки GUI, но на ней реализовано достаточное количество opensource-проектов.

Данная тема рассматривает процесс разработки GUI посредством одной из реализаций кроссплатформенного фреймворка Qt — PySide2, являющейся официальной реализацией Qt для Python, развитие которой поддерживается The Qt Company. Использование такого инструмента позволяет быстро и эффективно проектировать, разрабатывать, выпускать и сопровождать программные продукты, обеспечивающие единый UX (пользовательский опыт) среди всех используемых платформ [43].

8.1. Установка PySide2

Для установки PySide2 воспользуемся встроенным репозиторием пакетов Python Package Index (PyPI) [44] и в терминале введем следующую команду:

```
pip install PySide2
```

После запуска команды выполнится загрузка зависимостей (если таковые имеются) и самого PySide2:

```
Installing collected packages: shiboken2, PySide2
Successfully installed PySide2-5.15.0 shiboken2-5.15.0
```

8.2. Основы разработки GUI

8.2.1. Hello World

Для создания собственного виджета необходимо создать производный класс от класса **QWidget**, после чего определить переменные и методы экземпляра класса:

```
import sys
from PySide2.QtWidgets import QApplication, QWidget

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()
# Вызываем конструктор базового класса QWidget
        self.initializeUI()

    def initializeUI(self):
        self.setGeometry(100, 100, 400, 300)
# размер виджета
        self.setWindowTitle('Hello World with Qt')
        self.show()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

После запуска скрипта будет выведено главное окно приложения (рис. 8.1).

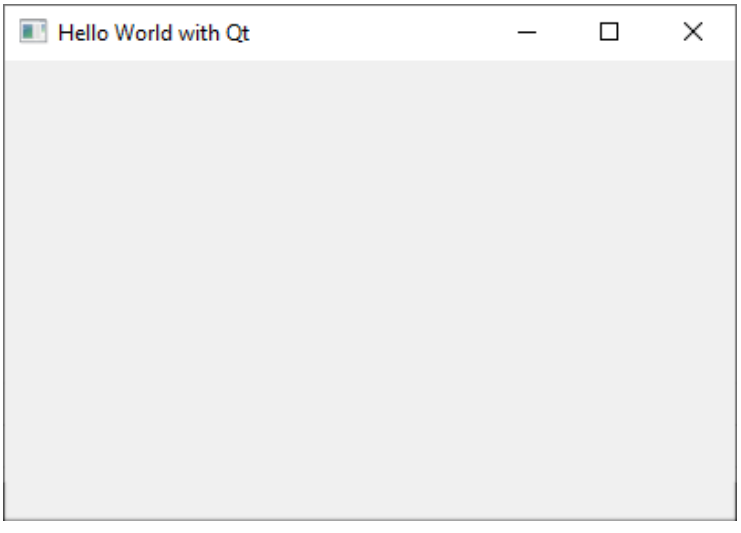

Рис. 8.1. **Пример запущенного приложения**

8.2.2. QPushButton и QLabel

Первыми из рассматриваемых нами виджетов являются одни из самых используемых в процессе разработки GUI-приложений — *QPushButton* и *QLabel*. При нажатии по кнопке на GUI приложения виджет *QPushButton* отправляет сигнал, который можно подключить к определяемому пользователем методу экземпляра класса для его последующей обработки. *QLabel* же в основном используется для информирования пользователя.

Для демонстрации работы с этими двумя элементами возьмем код предыдущего примера и для начала добавим в процесс импорта из модуля *PySide2.QtWidgets* классы **QPushButton** и **QLabel**:

```
from PySide2.QtWidgets import (QApplication,
                               QWidget, QLabel,
                               QPushButton)
```

Следующим шагом в конструкторе класса необходимо объявить переменные, посредством которых и будет строиться последующая работа с меткой и кнопкой:

```
    def __init__(self):
        super().__init__()
# Вызываем конструктор базового класса QWidget
        self.label = QLabel(self)
        self.button = QPushButton('Ok', self)
        self.initializeUI()
```

Теперь изменим код метода **initializeUI** экземпляра класса **MainWindow** следующим образом:

```
    def initializeUI(self):
        self.setGeometry(100, 100, 400, 300)
# размер виджета
        self.setWindowTitle('Hello World with Qt')
        name_label = QLabel(self)
        name_label.setText("Нажми на кнопку!")
        name_label.move(60, 30)
# позиция QLabel на виджете
        button = QPushButton(' Ok', self)
# задаем вызываемый метод экземпляра класса при нажатии кнопки
        button.clicked.connect(self.buttonClicked)
        button.move(80, 70)
# позиция QPushButton на виджете
        self.show()
```

Также в **MainWindow** добавим метод **buttonClicked**, в котором будет обрабатываться событие нажатия на кнопку:

```
    def buttonClicked(self):
        self.label.setText("Ура!")
        self.button.setText("^_^")
```

После запуска приложения пользователь увидит GUI, показанный на рис. 8.2.

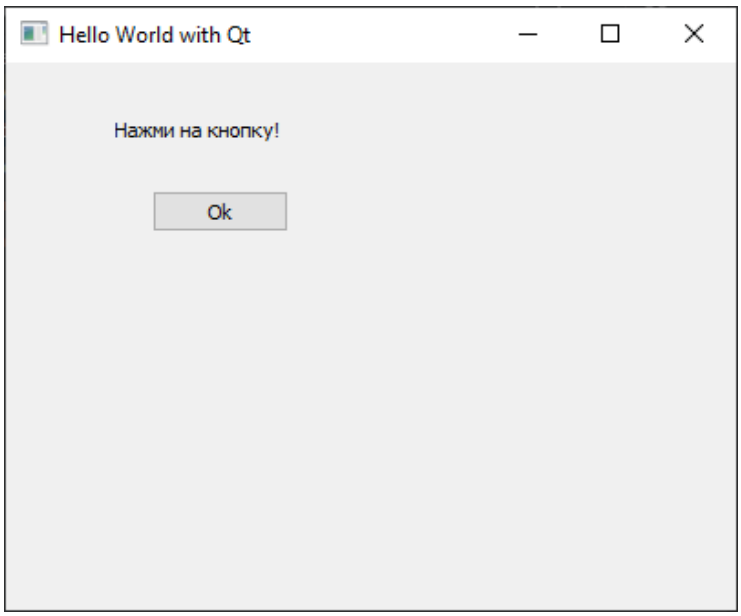

Рис. 8.2. **Начальное состояние GUI**

Давайте нажмем на кнопку и посмотрим, как изменится состояние графического пользовательского интерфейса приложения (рис. 8.3).

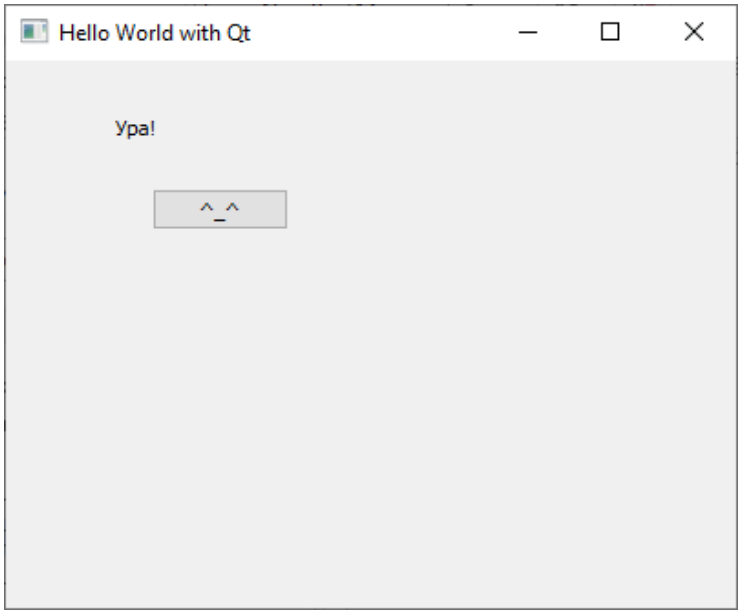

Рис. 8.3. **Состояние GUI после нажатия на кнопку**

8.2.3. QLineEdit

QLineEdit используется как один из виджетов ввода данных пользователем приложения. Он поддерживает такие функции редакти-

рования текста, как вырезание, копирование, вставка, повтор или отмена.

Виджет *QLineEdit* имеет ряд методов для добавления дополнительных функциональных возможностей графическому интерфейсу. Это может быть скрытие текста при его вводе, использование текста-заполнителя или даже установка ограничения на длину вводимого текста.

Ниже приведен пример работы с данным виджетом:

```
import sys
from PySide2.QtWidgets import (QApplication,
                        QWidget, QLabel, QLineEdit,
                        QPushButton)
from PySide2.QtCore import Qt

class MainWindow(QWidget): # Inherits QWidget
    def __init__(self):
        super().__init__()
        self.label = QLabel(self)
        self.button = QPushButton('Очистить', self)
        self.line_edit = QLineEdit(self)
        self.initializeUI()

    def initializeUI(self):
        self.setGeometry(100, 100, 400, 200)
        self.setWindowTitle('QLineEdit Test')
        QLabel("Введите login", self).move(100, 10)
        name_label = QLabel("Login:", self)
        name_label.move(70, 50)
        self.line_edit.setAlignment(Qt.AlignLeft)
# выравнивание по левому краю
        self.line_edit.move(130, 50)
        self.line_edit.resize(200, 20)
        self.button.clicked.connect(self.clearEntries)
        self.button.move(160, 110)
        self.show()

    def clearEntries(self):
# получаем ссылку на объект генерирующий сигнал
        sender = self.sender()
# Проверяем, что это кнопка с надписью "Очистить"
        if sender.text() == 'Очистить':
            self.line_edit.clear()  # очищаем текст

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

Скриншот получившегося приложения приведен на рис. 8.4.

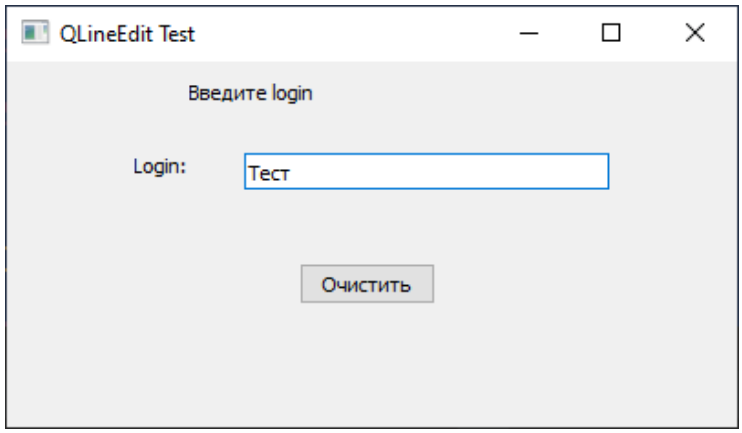

Рис. 8.4. **Пример работы с QLineEdit**

После нажатия на кнопку «Очистить» в *QLineEdit* сотрется весь введенный ранее текст.

8.2.4. QCheckBox

Виджет *QCheckBox* позволяет переключаться между двумя состояниями: активен или неактивен. Он идеально подходит для представления функций в графическом интерфейсе, которые можно включить или отключить, или для выбора из списка параметров. Также *QCheckBox* может использоваться для работы с динамическим GUI приложений, например, чтобы изменить заголовок окна или даже текст меток, когда он включен.

Ниже приводится код, демонстрирующий принципы работы с *QCheckBox*:

```
import sys
from PySide2.QtWidgets import (QApplication,
                               QWidget, QCheckBox, QLabel)
from PySide2.QtCore import Qt


class MainWindow(QWidget):

    def __init__(self):
        super().__init__()
        self.label = QLabel(self)
        self.label.setMinimumSize(200, 10)
        self.initializeUI()

    def initializeUI(self):
        self.setGeometry(100, 100, 250, 250)
        self.setWindowTitle('QCheckBox Test')
        self.label.move(30, 200)
        header_label = QLabel(self)
        header_label.setText("Во сколько "+
                             "вы просыпаетесь?")
```

```
        header_label.setWordWrap(True)
        header_label.move(10, 10)
        header_label.resize(230, 60)
        morning6 = QCheckBox("6 утра", self)

         morning6.move(20, 80)
    morning6.stateChanged.connect(
                               self.printChoose)
morning7 = QCheckBox("7 утра", self)          morning7.move(20,
100)
morning7.stateChanged.connect(
                             self.printChoose)
        never_sleep = QCheckBox("Я вообще не сплю",
                                self)
        never_sleep.move(20, 120)
        never_sleep.stateChanged.connect(
                              self.printChoose)
        self.show()

    def printChoose(self, state):
        sender = self.sender()
        if state == Qt.Checked:
            choose = f"Выбрано: {sender.text()}."
        else:
            choose = f"Отменен выбор: " +
                     f" {sender.text()}."
        self.label.setText(choose)

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

Выбор пользователя в разработанном приложении выводится на виджет *QLineEdit*, что можно наблюдать на рис. 8.5.

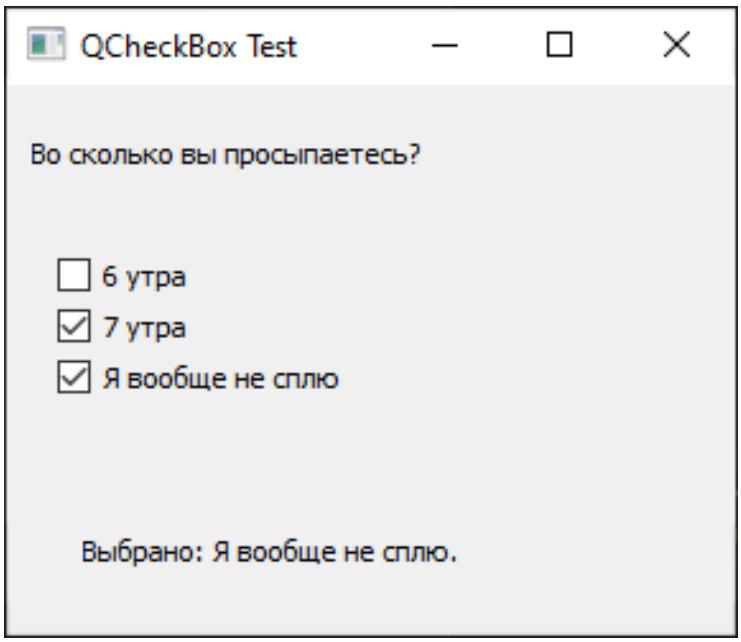

Рис. 8.5. **Пример работы с QCheckBox**

8.2.5. Компоновка элементов GUI

Вместо продолжения рассмотрения различных виджетов, коих в Qt еще огромное множество, давайте лучше познакомимся с принципами компоновки элементов GUI. Это избавит от некоторых проблем в будущем. По своей сути, компоновка макета — это способ, который использует разработчик при размещении виджетов в приложении. При этом учитываются различные факторы, такие как: размер, положение, добавление или удаление виджетов, обработка изменения размера главного окна приложения и т. д.

Менеджер компоновки (*layout manager*) в рамках Qt представляет собой набор классов, которые содержат методы для размещения виджетов внутри других виджетов или окон. Они полезны для связи между дочерними и родительскими виджетами, позволяя более эффективно использовать имеющееся пространство и настраивать политики изменения размеров.

В рамках данного учебного пособия мы рассмотрим вариант использования таких менеджеров компоновки, как классы **QVBoxLayout** и **QHBoxLayout**, являющиеся производными классами **QBoxLayout**. Данный менеджер полезен для создания простых графических интерфейсов с горизонтальным или вертикальным расположением компонентов и его функциональности достаточно для большого количества случаев компоновки элементов на GUI разрабатываемого приложения.

В качестве примера напишем код опросника, который интересуется у пользователя, как прошел его день, и предлагает выбрать один из вариантов ответа:

```
import sys
from PySide2.QtWidgets import (QApplication,
                           QWidget,
                           QLabel, QPushButton,
                           QCheckBox, QButtonGroup,
                           QHBoxLayout, QVBoxLayout)
from PySide2.QtGui import QFont
from PySide2.QtCore import Qt

class MainWindow (QWidget):
    def __init__(self):
        super().__init__()
        self.label = QLabel(self)
        self.initUI()

    def initUI(self):
        self.setGeometry(100, 100, 400, 230)
        self.setWindowTitle('Layout Test')
        self.displayWidgets()
        self.label.setAlignment(Qt.AlignHCenter)
        self.show()
```

```
    def displayWidgets(self):
        # create label and button widgets
        title = QLabel("Вечерний опрос")
        title.setFont(QFont('Arial', 17))
        question = QLabel("Насколько плодотворно"
                          " прошел Ваш день?")
        # создание горизонтального layout-а
        title_h_box = QHBoxLayout()
        title_h_box.addStretch()
        title_h_box.addWidget(title)
        title_h_box.addStretch()
        # Варианты выбора
        ratings = ["Ужасно", "Ну, такое",
                   "Нормально", "Отлично"]

        ratings_h_box = QHBoxLayout()
 # расстояние между виджетами в горизонтальном layout-е
        ratings_h_box.setSpacing(80)

        ratings_h_box.addStretch()
        for rating in ratings:
            rate_label = QLabel(rating, self)
            ratings_h_box.addWidget(rate_label)
        ratings_h_box.addStretch()

        cb_h_box = QHBoxLayout()
# расстояние между виджетами в горизонтальном layout-е
        cb_h_box.setSpacing(100)

        # Создаем контейнер для групировки QCheckBox
        scale_bg = QButtonGroup(self)

        cb_h_box.addStretch()
        for cb in range(len(ratings)):
            scale_cb = QCheckBox(str(cb), self)
            cb_h_box.addWidget(scale_cb)
            scale_bg.addButton(scale_cb)
        cb_h_box.addStretch()


# устанавливаем функцию отработки варианта выбора

        scale_bg.buttonClicked.connect(
                                self.checkboxClicked)
        close_button = QPushButton("Закрыть", self)
        close_button.clicked.connect(self.close)
# Компонуем вертикальный layout, последовательно
# добавляя виджеты and элементы h_box layout
        v_box = QVBoxLayout()
        v_box.addLayout(title_h_box)
        v_box.addWidget(question)
        v_box.addStretch(1)
        v_box.addLayout(ratings_h_box)
```

```
        v_box.addLayout(cb_h_box)
        v_box.addStretch(2)
        v_box.addWidget(self.label)
        v_box.addWidget(close_button)
        self.setLayout(v_box)

    def checkboxClicked(self, cb):
        self.label.setText(f"Выбрано: {cb.text()}")

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = MainWindow ()
    sys.exit(app.exec_())
```

Компоновка *QCheckBox* в контейнере *QButtonGroup* позволяет задавать такое поведение, что пользователь может выбрать только один вариант ответа, а не сразу несколько, как было в предыдущем примере. Внешний вид GUI написанного приложения приведен на рис. 8.6.

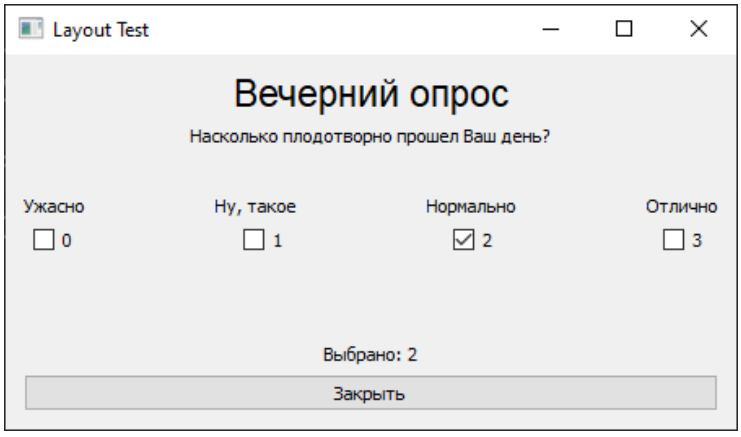

Рис. 8.6. **Пример использования GUI с менеджером компоновки**

При этом, если изменить размер главного окна, элементы GUI подстроятся под его новый размер (рис. 8.7). Без использования менеджера компоновки реализовывать такое поведение приложения было бы очень трудоемко и не продуктивно.

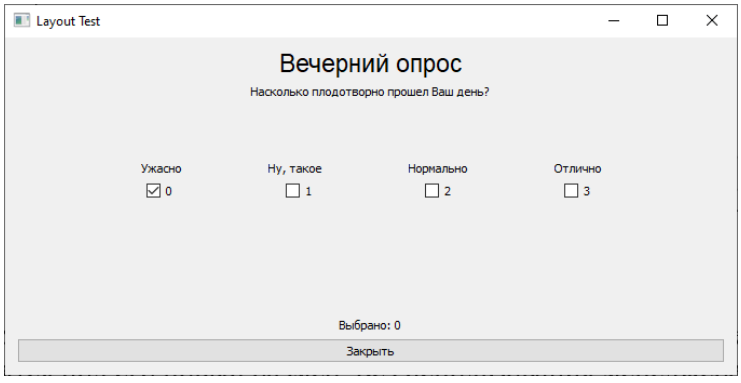

Рис. 8.7. **Поведение приложения при изменении размера главного окна**

8.3. Пользовательские виджеты и сигнал-слотовый механизм

Зачастую при разработке GUI программы приходится разбивать его на составные части и организовывать механизм передачи данных между ними. К счастью, Qt предоставляет такие средства «из коробки». Данный механизм строится на сигналах и слотах. В роли сигнала выступает объект, который оповещает все подключенные к нему слоты о каком-либо изменении внутри экземпляра класса, генерирующего этот сигнал. При объявлении сигнала может указываться различное количество атрибутов с их типами, которые впоследствии и будут выступать в качестве передаваемых данных.

Для примера давайте напишем приложение, которое позволяет вести список дел, разбив его на две части и организовав между ними взаимодействие посредством сигналов и слотов. Первая часть, собственно говоря, и будет представлять из себя виджет списка дел — *ToDoListWidget*, а вторая — главное окно (*MainWindow*), куда этот виджет помещается.

Ниже приведен код разработанного приложения:

```
import sys
from PySide2.QtWidgets import (QApplication,
                           QWidget, QLabel, QTextEdit,
                           QLineEdit, QPushButton,
                           QCheckBox, QGridLayout,
                           QVBoxLayout, QMainWindow,
                           QMessageBox)
from PySide2.QtGui import QFont
from PySide2.QtCore import Qt, Slot, Signal

class ToDoListWidget(QWidget):

    doneSignal = Signal(str, str) # объявляем сигнал
    warningSignal = Signal(str)

    def __init__(self, parent=None):
        super(ToDoListWidget, self).__init__(parent)
        self.dictToDo = {}
        self.dayMeet = []
        self.initializeUI()
        for it in self.dictToDo:
            it.clicked.connect(self.itIsDone)

    def initializeUI(self):
        main_grid = QGridLayout()
        todo_title = QLabel("Список дел на день")
        todo_title.setFont(QFont('Arial', 24))
        todo_title.setAlignment(Qt.AlignCenter)
        # создание меток
        mustdo_label = QLabel("Надо сделать")
        mustdo_label.setFont(QFont('Arial', 20))
```

```
        mustdo_label.setAlignment(Qt.AlignCenter)
        appts_label = QLabel("Запланированные " +
                             " встречи")
        appts_label.setFont(QFont('Arial', 20))
        appts_label.setAlignment(Qt.AlignCenter)
        # создание секций
        mustdo_grid = QGridLayout()
        mustdo_grid.setContentsMargins(5, 5, 5, 5)
        mustdo_grid.addWidget(mustdo_label, 0,
                              0, 1, 2)
        # создание чекбоксов с полями ввода
        for position in range(1, 15):
            checkbox = QCheckBox()
            checkbox.setObjectName(str(position))
            linedit = QLineEdit("")
            linedit.setMinimumWidth(200)
            mustdo_grid.addWidget(checkbox,
                                  position, 0)
            mustdo_grid.addWidget(linedit,
                                  position, 1)
            self.dictToDo[checkbox] = linedit

        # секция встреч
        morning_label = QLabel("Утро")
        morning_label.setFont(QFont('Arial', 16))
        noon_label = QLabel("Обед")
        noon_label.setFont(QFont('Arial', 16))
        evening_label = QLabel("Вечер")
        evening_label.setFont(QFont('Arial', 16))
        # первичная компоновка
        appt_v_box = QVBoxLayout()
        appt_v_box.setContentsMargins(5, 5, 5, 5)
        evening_entry = QTextEdit()
        noon_entry = QTextEdit()
        morning_entry = QTextEdit()
        appt_v_box.addWidget(appts_label)
        appt_v_box.addWidget(morning_label)
        appt_v_box.addWidget(morning_entry)
        appt_v_box.addWidget(noon_label)
        appt_v_box.addWidget(noon_entry)
        appt_v_box.addWidget(evening_label)
        appt_v_box.addWidget(evening_entry)
        self.dayMeet = [morning_entry, noon_entry,
                        evening_entry]
        # завершение компоновки
        main_grid.addWidget(todo_title, 0, 0, 1, 2)
        main_grid.addLayout(mustdo_grid, 1, 0)
        main_grid.addLayout(appt_v_box, 1, 1)
        self.setLayout(main_grid)

    def itIsDone(self):
        sender = self.sender()
# запрет на выполнение пустого элемента списка дел
        if not self.dictToDo[sender].text():
            sender.setChecked(False)
```

```
            self.warningSignal.emit(
                            sender.objectName())
            return
        if sender.isChecked():
            self.doneSignal.emit(
                      sender.objectName(),
                      self.dictToDo[sender].text())

    @Slot()
    def clearToDoList(self):
        for it in self.dayMeet:
            it.clear()
        for checkbox, edit in self.dictToDo.items():
            checkbox.setChecked(False)
            edit.setText("")

class MainWindow(QMainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.todo = ToDoListWidget ()
        self.todo.doneSignal.connect(self.itIsDone)
        self.todo.warningSignal.connect(
                                    self.warningDone)
        self.initializeUI()

    def initializeUI(self):
        self.setWindowTitle('Список дел')
        close_button = QPushButton("Закрыть")
        close_button.clicked.connect(self.close)
        clear_button = QPushButton("Очистить")
        clear_button.clicked.connect(
                        self.todo.clearToDoList)
        v_box = QVBoxLayout()
        v_box.addWidget(self.todo)
        v_box.addWidget(clear_button)
        v_box.addWidget(close_button)
        widget = QWidget(self)
        widget.setLayout(v_box)
        self.setCentralWidget(widget)
        self.setMaximumHeight(500)
        self.show()

    @Slot(str, str)
    def itIsDone(self, number, description):
        QMessageBox().information(self,
        "Внимание!",  f"Дело '{description}' "
        f" под №{number} выполнено!",
        QMessageBox.Ok, QMessageBox.Ok)

    @Slot(str)
    def warningDone(self, number):
        QMessageBox().warning(self, "Внимание!",
        f"Дело №{number} не заполнено!",
        QMessageBox.Ok, QMessageBox.Ok)
```

```
if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    sys.exit(app.exec_())
```

Внешний вид GUI написанного приложения приведен на рис. 8.8.

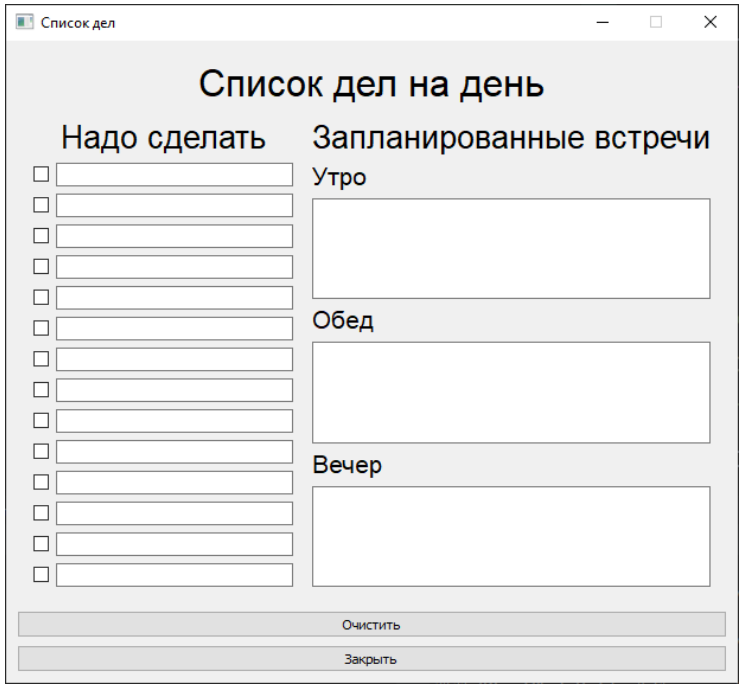

Рис. 8.8. **Внешний вид приложения «Список дел»**

Если попробуем отметить пустое дело, как выполненное, приложение выдаст предупреждение (рис. 8.9).

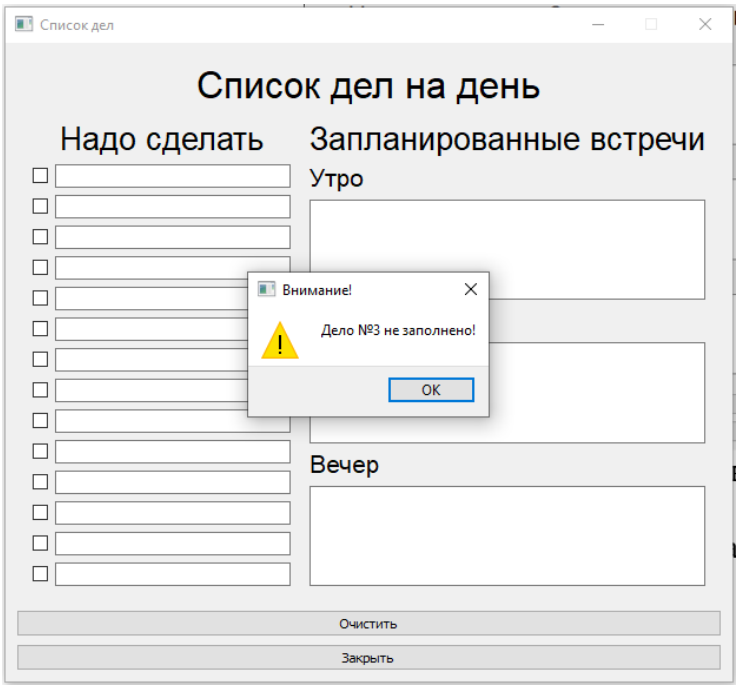

Рис. 8.9. **Пример защиты от ввода некорректных данных**

Когда дело из списка переходит в разряд выполненных, приложение извещает об этом пользователя (рис. 8.10).

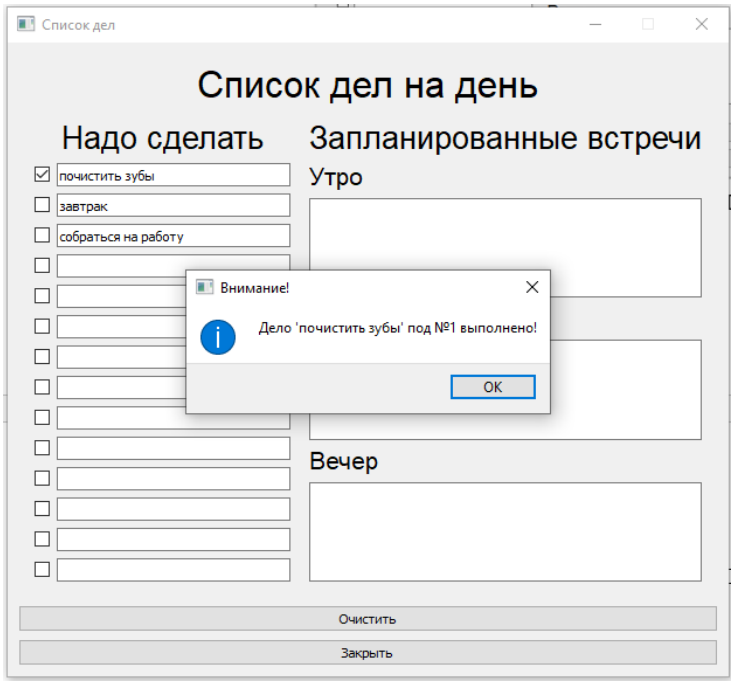

Рис. 8.10. **Оповещение пользователя о завершении дела**

При нажатии на кнопку «Очистить» сотрутся все внесенные данные, а щелчок мышкой на кнопку «Закрыть» приведет к завершению работы приложения.

На рис. 8.11 приведен внешний вид *ToDoListWidget*.

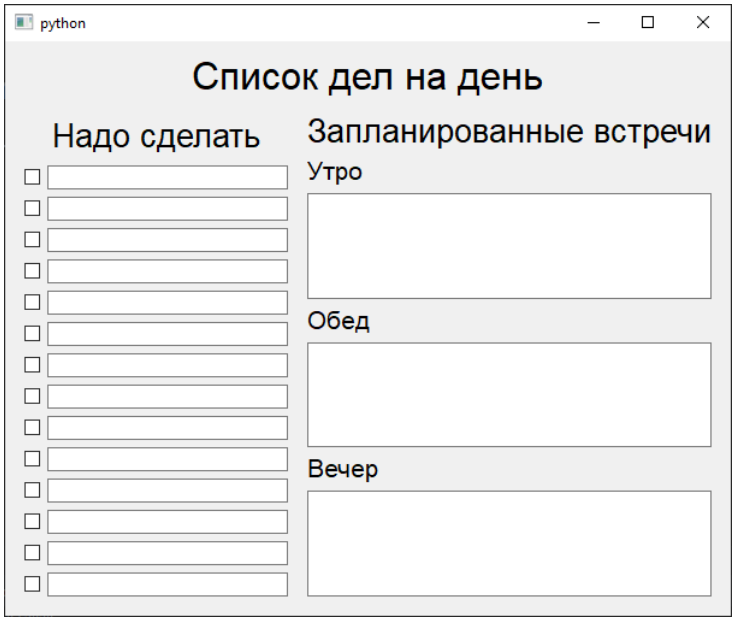

Рис. 8.11. **Внешний вид виджета *ToDoListWidget***

Как видно из рис. 8.11, виджет *ToDoListWidget* является частью приложения и не может предоставить его полную функциональ-

ность, если используется отдельно от главного окна *MainWindow*. Внешний же вид главного окна до добавления в него пользовательского виджета приведен на рис. 8.12.

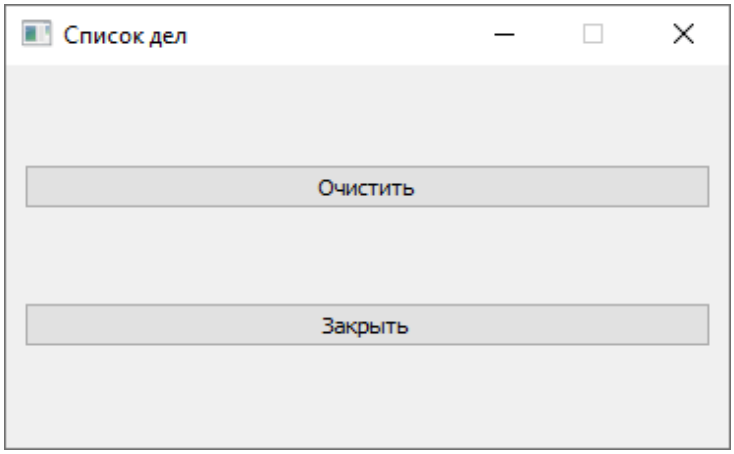

Рис. 8.12. **Внешний вид главного окна приложения «Список дел»**

8.4. Использование Qt Designer для разработки GUI

Во фреймворк Qt входит Qt Designer, упрощающий разработку графического пользовательского интерфейса приложения за счет ухода от его написания «вручную» и добавления интерактивности в процесс создания GUI. На рис. 8.13 приведен внешний вид Qt Designer.

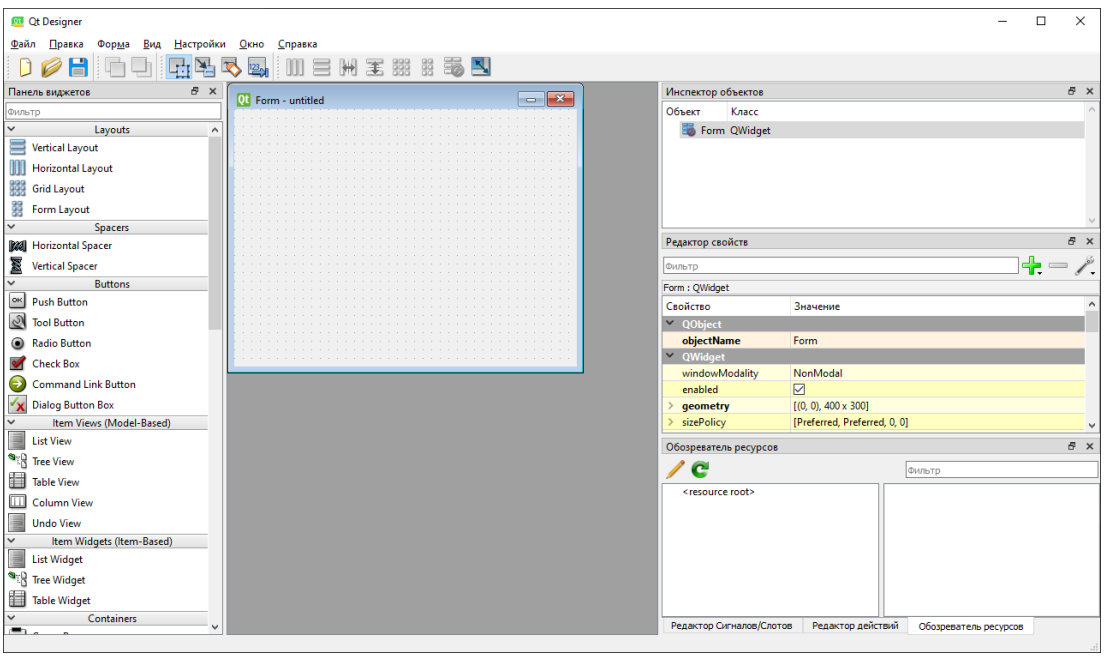

Рис. 8.13. **Внешний вид Qt Designer**

Теперь, используя предоставляемый Qt Designer инструментарий, разработаем GUI нашего будущего приложения. В качестве

примера реализуем GUI разработанного ранее опросника об оценке проведенного пользователем дня (рис. 8.14).

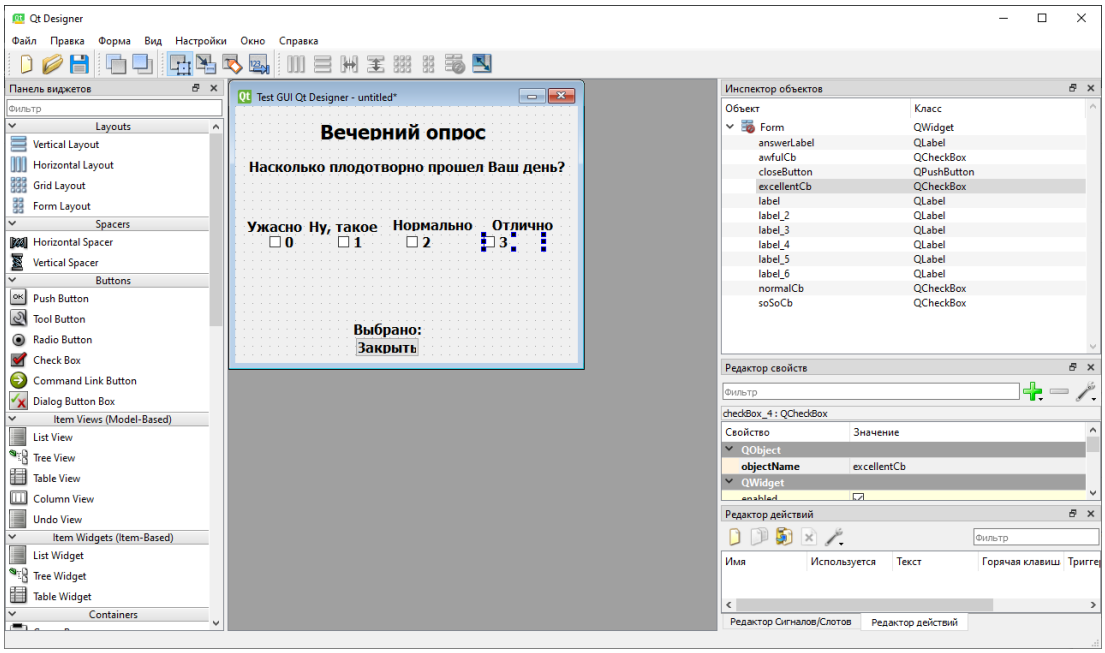

Рис. 8.14. **Внешний вид GUI разрабатываемого приложения**

Используем менеджеры компоновки (*Horizontal* и *Vertical Layout*) для связывания и расстановки элементов пользовательского интерфейса внутри разрабатываемого виджета. Чтобы некоторые элементы отображались по центру компонуемых структур, используем Horizontal и Vertical Spacer (рис. 8.15).

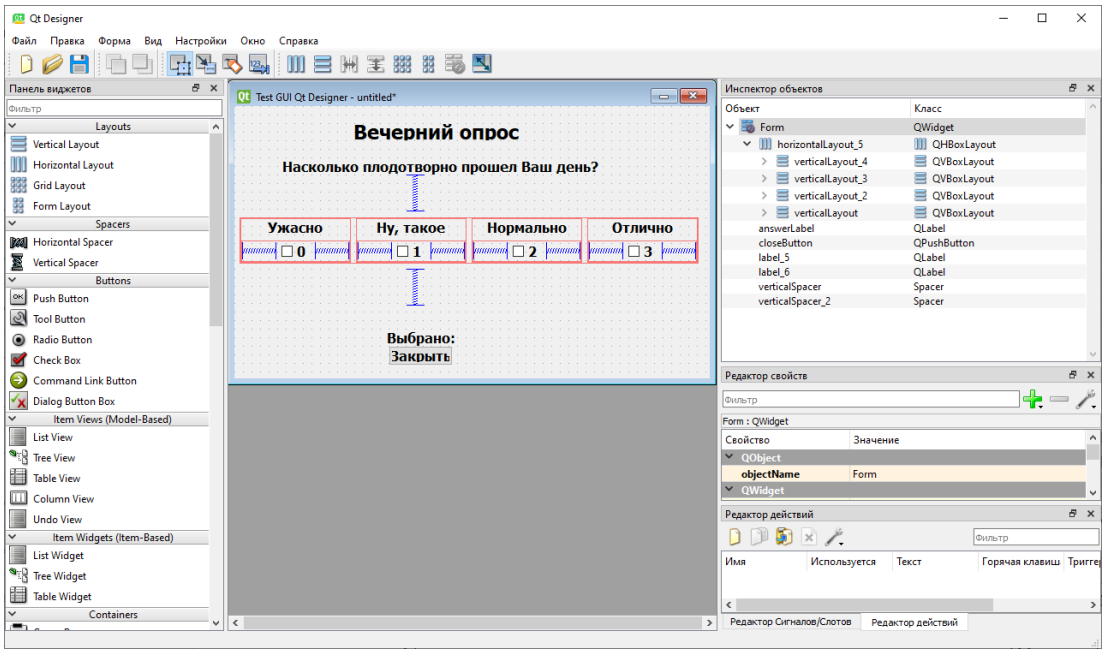

Рис. 8.15. **Внешний вид GUI разрабатываемого приложения**

Следующим шагом необходимо задать виджету его менеджер компоновки для окончательной расстановки элементов GUI. Для начала щелкните правой кнопкой мыши по виджету, затем перейдите в раздел «Компоновка» и выберите «Скомпоновать по вертикали» (рис. 8.16).

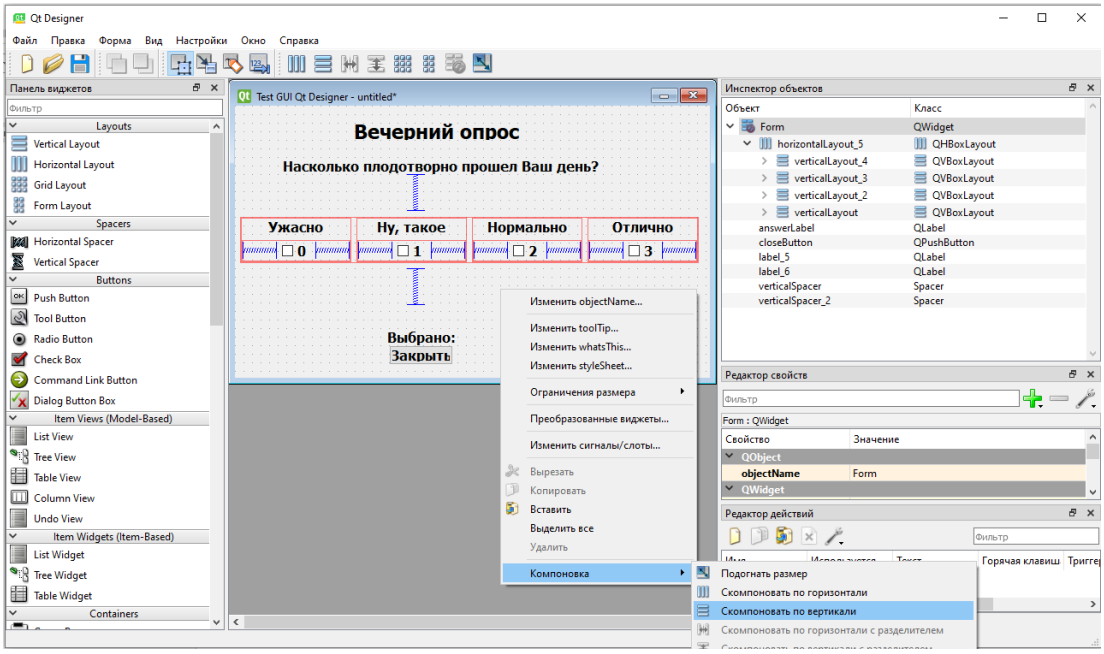

Рис. 8.16. **Задание типа компоновки элементов виджета**

Результат этого действия показан на рис. 8.17.

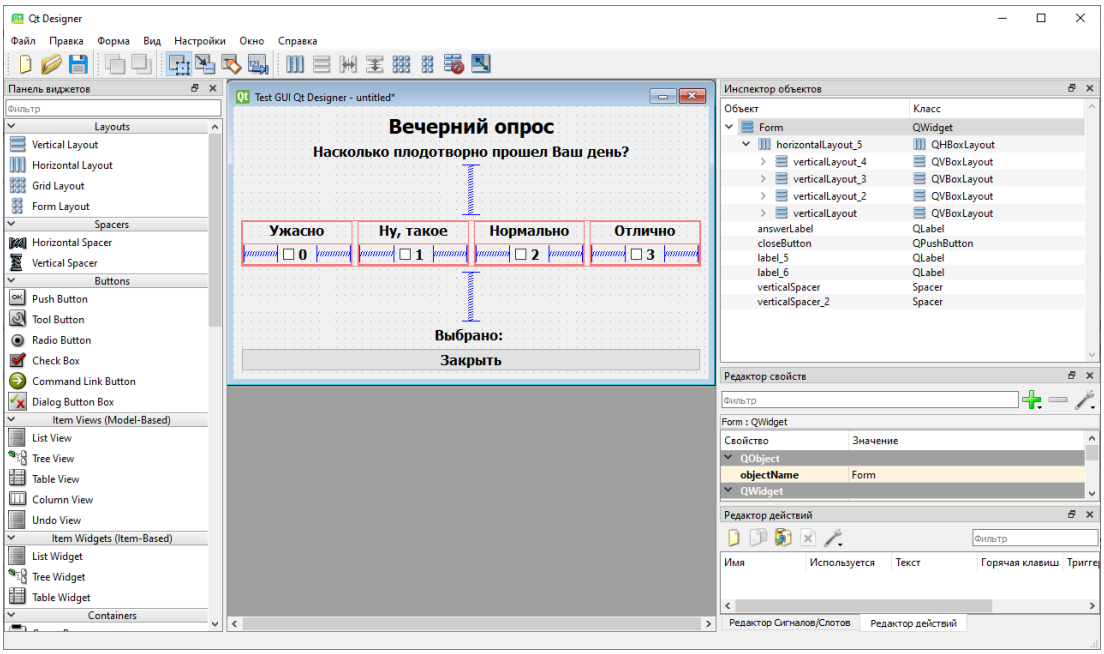

Рис. 8.17. **Внешний вид виджета разрабатываемого приложения**

Теперь в поле *objectName* виджета введем его название *QuestionnaireWidget* и сохраним, задав ему при этом имя

questionnairewidget, после чего закроем Qt Designer. Следующим шагом необходимо преобразовать получившийся файл с описанием GUI *QuestionnaireWidget.ui* в код на Python. Для этого необходимо в терминале зайти в каталог с сохраненным файлом описания пользовательского интерфейса и набрать следующую команду:

```
(venv) F:\code\python\module-GUI>pyside2-uic
questionnairewidget.ui > questionnairewidget.py
```

В результате будет сгенерирован файл *questionnairewidget.py*, который необходимо подключить к разрабатываемому приложению следующим образом:

```
import sys
from PySide2.QtWidgets import (QApplication, QButtonGroup,
QWidget)
# подключаем сгенерированный виджет
from questionnairewidget import (
                            Ui_QuestionnaireWidget)

class MainWindow(QWidget):
    ratings = {"0": "Ужасно",
               "1": "Ну, такое",
               "2": "Нормально",
               "3": "Отлично"
               }

    def __init__(self):
        super(MainWindow, self).__init__()
        self.ui = Ui_QuestionnaireWidget()
        self.ui.setupUi(self)
        scale_bg = QButtonGroup(self)
        scale_bg.addButton(self.ui.awfulCb)
        scale_bg.addButton(self.ui.soSoCb)
        scale_bg.addButton(self.ui.normalCb)
        scale_bg.addButton(self.ui.excellentCb)
 # устанавливаем метод отработки варианта выбора
        scale_bg.buttonClicked.connect(
                        self.checkboxClicked)

    def checkboxClicked(self, cb):
        self.ui.answerLabel.setText(f"Выбрано:
                   f" {self.ratings[cb.text()]} "
                   f"({cb.text()})")

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Скриншот разработанного приложения представлен на рис. 8.18.

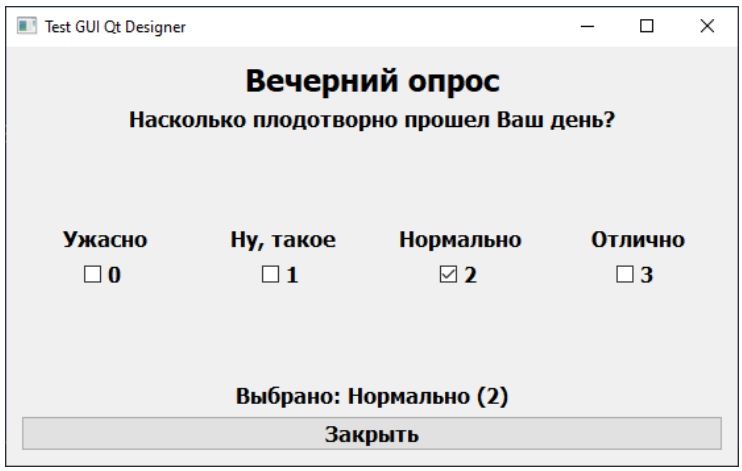

Рис. 8.18. **Пример работы разработанного приложения**

Даже в случае изменения размеров главного окна элементы GUI, за счет использования менеджера компоновки, подстраиваются под эти изменения (рис. 8.19).

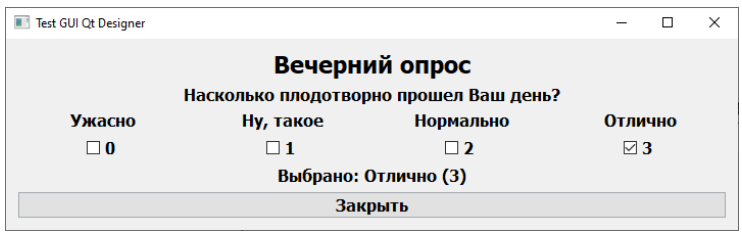

Рис. 8.19. **Изменение размеров главного окна приложения**

Резюме

В данной теме мы рассмотрели возможности библиотеки PySide2, являющейся одной из реализаций фреймворка Qt на языке программирования Python, для разработки графического пользовательского интерфейса приложений. Различия между PySide2 и аналогичной библиотекой PyQt5 — минимальны и заключаются в подключении модулей верхнего уровня с различными названиями, а также в механизме объявления сигналов и слотов.

Ввиду этого, несмотря на то, что книг по PySide2 не так много, это не является проблемой. Для более детального рассмотрения возможностей фреймворка Qt в задачах разработки GUI при его использовании средствами Python можно обратиться к следующим источникам [45, 46].

Несмотря на то, что Qt Designer предоставляет более удобный и быстрый способ разработки пользовательского интерфейса приложений, в ряде случаев бывает удобнее написать код «вручную».

Особенно это касается таких моментов, когда необходимо создать множество однотипных виджетов.

Вопросы и задания для самопроверки

1. Какие существуют библиотеки для создания приложений с графическим пользовательским интерфейсом помимо PySide2?
2. Какой командой производится установка PySide2?
3. Какую из библиотек для разработки кроссплатформенных приложений на языке программирования Python официально поддерживает The Qt Company?
4. Какой из предоставляемых Qt классов должен выступать в роли базового класса при разработке пользовательского виждета?
5. Где и зачем применяется механизм сигнал-слотовой связи, предоставляемый фреймворком Qt?
6. Для чего используется менеджер компоновки?
7. В чем различие между использованием *QVBoxLayout* и *QHBoxLayout*?
8. Для чего используется Qt Designer?
9. Какая команда позволяет преобразовать разработанный в Qt Designer виджет в код на Python?
10. Какая библиотека для разработки GUI является одним из модулей Python?

Упражнения

1. Напишите программу «Калькулятор».
2. Напишите программу «Список дел» с возможностью сохранения данных в файл и их загрузки для редактирования.
3. Напишите программу, выводящую содержимое задаваемого каталога.
4. Напишите программу для игры в «крестики-нолики».
5. Напишите программу для просмотра изображений в задаваемом каталоге.
6. Напишите программу для решения квадратных уравнений.
7. Напишите программу, позволяющую создавать новые текстовые файлы, заполнять их, сохранять, загружать и редактировать, а также осуществлять поиск по тексту.
8. Напишите программу, реализующую функциональность записной книжки с возможностью сохранения, загрузки, редактирования данных и поиска по фамилии.
9. Напишите программу, осуществляющую поиск файлов по имени или его части в задаваемом пользователем каталоге.
10. Напишите программу, позволяющую заходить в ее основное окно только авторизированным пользователям, с возможностью регистрации нового пользователя. Хеш связки «логин:пароль» пользователей хранится в отдельном файле.
11. Напишите программу, осуществляющую конвертацию валют, список которых берется из json-файла с данными об их текущем курсе.
12. Напишите программу для построения графиков по данным из файла MS Excel. Файл может содержать произвольное количество листов, каждый

из которых хранит значения для осей абсцисс и ординат (всего по 2 столбца на лист). В разработанном приложении между графиками необходимо переключаться, используя TabWidget.

13. Напишите программу с графическим пользовательским интерфейсом для составления графа расстояний между объектами и нахождения самого короткого пути между задаваемыми вершинами графа.

14. Напишите программу, позволяющую зашифровать и расшифровать файл любым выбранным алгоритмом.

15. Напишите программу, позволяющую формировать список с выбором вариантов алгоритмов для его сортировки (минимум 2 алгоритма).

16. Напишите программу, осуществляющую случайную генерацию пароля по нажатию на кнопку.

17. Напишите программу для перевода значений в различные системы счисления (десятичная, восьмеричная, двоичная, шестнадцатеричная).

18. Напишите программу для просмотра видеофайлов в задаваемом пользователем каталоге.

19. Напишите программу, реализующую базовую функциональность Paint.

20. Напишите программу-игру «Змейка».

Тема 9
СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

В результате изучения данной темы обучающиеся должны:

знать

- основные принципы разработки приложений с архитектурой «клиент-сервер»,
- типы существующих соединений и случаи их применения;

уметь

- разрабатывать собственные приложения с архитектурой «клиент-сервер»,
- пользоваться фреймворком *socketserver* для сетевых серверов;

владеть

- навыками сетевого программирования.

В настоящее время активно развиваются облачные сервисы, распределенные компьютерные системы, а жизнь без Интернета уже невозможно даже представить. Ввиду этого при проектировании систем, ориентированных на работу по сети, на первый план выходит понимание принципов обмена данными. Говоря про обмен данными, нельзя не затронуть такое определение, как протокол — своеобразный язык, используемый по договоренности двумя сторонами (компьютерными программами) для общения между собой.

Способы взаимодействия между программами по сети можно разбить на две категории: с установлением и без установления соединения. Посредством интерфейса сокетов и стека протоколов TCP/IP каждый из них может работать, используя различные транспортные механизмы. К сожалению, семиуровневая модель обмена данными по сети OSI (*Open Systems Interconnection*), представленная в стандарте ISO (*International Standards Organization*), является излишне сложной для большинства реальных применений. В связи с этим в стеке протоколов TCP/IP используется ее сокращенная, четырехуровневая версия.

В этом случае разрабатываемые приложения реализуют прикладной уровень. Он позволяет обмениваться сообщениями двум процессам, которые выполняются либо локально, либо на разных компьютерах. Далее с прикладного уровня сообщение передается

транспортному, где транспортный протокол разбивает длинные сообщения на сегменты и передает их на сетевой уровень. Здесь сегменты разбиваются на датаграммы (порции данных) и осуществляется их маршрутизация. В свою очередь, сетевой уровень использует канальный для передачи датаграмм между отдельными системами на пути от источника сообщения к пункту его назначения.

Принцип работы протоколов, которые ориентированы на установление соединения, можно описать на примере обычного телефонного разговора. Отправной точкой возьмем начало самого процесса — запрос соединения с конкретным абонентом (узлом/точкой сети). У абонента на другой стороне имеется выбор: ответить или не ответить на звонок. Если он отвечает на звонок, то разговор может состояться различным количеством способов: в виде монолога, диалога, постоянного перебивания друг друга и так далее. В этом случае вы можете быть уверены, что никакая часть передаваемой информации не теряется и перед окончанием разговора вы попрощаетесь с собеседником, после чего оба завершите звонок. Также имеется возможность понять, что в процессе разговора произошли неполадки: собеседника не слышно или резко оборвался звонок. Основной транспортный протокол, который ориентирован на установление соединения — TCP (*Transmission Control Protocol* — протокол управления передачей).

Принцип работы протоколов, которые не ориентированы на установление соединения, можно описать на примере почтовой рассылки (не по e-mail, а старой и доброй «Почтой России»). В большинстве случаев письма дойдут до адресатов, но на это нет гарантии. И если что-то пошло не так, необходимо самостоятельно найти выход из сложившейся ситуации. В этом случае основным транспортным протоколом является UDP (*User Datagram Protocol* — протокол пользовательских датаграмм).

UDP и TCP работают поверх сетевого IP-протокола. Эта комбинация протоколов, а также ряд протоколов прикладного уровня, которые выполняются поверх них, известны под общим названием «стек протоколов TCP/IP».

Существует две версии сетевого протокола IP: IPv4 и IPv6. В первом случае адреса сокетов представляют собой пары (*address*, *port*), где *address* — адрес IPv4, а *port* — номер порта в диапазоне 1—65 535. Адреса сокетов IPv6 представляют собой кортежи из четырех элементов вида (*address*, *port*, *flowinfo*, *scopeid*).

Следует запомнить, что взаимодействие по сети всегда предполагает обмен байтовыми строками (октетами), и если вам необходимо передать текст, то сначала его надо кодировать в байты, которые впоследствии декодирует получатель.

9.1. Архитектура «клиент-сервер»

В большинстве случаев сетевой обмен осуществляется с использованием клиент-серверной архитектуры, закладываемой в разрабатываемые приложения. Сервер прослушивает входящий трафик по своему сетевому адресу на определенном порту. Пока запросов от клиентов не поступает, он будет пребывать в состоянии ожидания, то есть не предпринимать каких-либо действий. В зависимости от выбранного протокола (с установлением или без установления соединения) связь сервера с клиентом будет осуществляться по-разному.

В первом случае клиент должен инициировать начальный обмен данными с сервером, который в конечном счете устанавливает соединение посредством сетевого канала связи между двумя процессами. Данное соединение будет держаться и позволять процессам обмениваться сообщениями до тех пор, пока оба не изъявят желание завершить сеанс связи. В данном случае сервер для обслуживания запросов должен реализовывать асинхронный, многопоточный или многопроцессорный подход для обработки каждого входящего соединения. Это связано с тем, что вызовы методов сокетов являются блокирующими, и сервер не сможет обрабатывать новые входящие соединения, пока не завершит работу с предыдущими.

Во втором случае запросы поступают на сервер в случайном порядке и немедленно обрабатываются. То есть ответ от сервера посылается клиенту без задержки. При этом каждое сообщение обрабатывается независимо от других, что делает UDP хорошо приспособленным для кратковременного взаимодействия без сохранения состояния (служба DNS) или процесса начальной загрузки сети.

9.1.1. Логика работы без установления соединения

В данном случае процесс работы сервера будет включать в себя следующие этапы.

1. Создание сокета типа *socket.SOCK_DGRAM*.
2. Связывание сокета с конечной точкой службы (вызов его метода **bind**).
3. Повторение перечня действий в бесконечном цикле:

— запрос у клиента датаграммы посредством вызова метода сокета **recvfrom** (блокирует вызов до получения датаграммы/ожидание сообщения от клиента);

— вычисление результата;

— отправка вычисленного результата обратно клиенту (вызов метода сокета **sendto**).

Процесс работы клиента будет состоять из следующих этапов:

1) создание сокета типа *socket.SOCK_DGRAM*;

2) ***по выбору*:** связать сокет с конкретной конечной точкой (вызов метода сокета **bind**);

3) отправка запроса/сообщения серверу (вызов метода сокета **sendto**);

4) ожидание ответа от сервера (вызов метода сокета **recvfrom**);

5) использование данных, полученных от сервера, в логике оставшейся части программы.

При использовании метода сокета **recvfrom** на стороне клиента всегда следует задавать аргумент тайм-аута. Это позволит обеспечить обработку случаев, когда сообщение не доходит до сервера, и требуется решить: повторить попытку или отказаться от этого.

9.1.2. Логика работы с установлением соединения

В данном случае процесс работы сервера будет включать в себя следующие этапы.

1. Создание сокета типа *socket. SOCK STREAM*.
2. Связывание сокета с конечной точкой службы (вызов его метода **bind**).
3. Запуск прослушивания запросов на создание соединения (вызов метода сокета **listen**).
4. Повторение перечня действий в бесконечном цикле:

а) ожидание (блокирование процесса до получения) запроса от клиента на создание соединения (вызов метода сокета **accept**). При получении такого запроса сервер создает новый объект сокета, конечной точкой для которого является клиент;

б) создание асинхронного потока управления для обработки нового соединения, путем передачи ему созданного сокета. Переход к пункту 4.а;

в) взаимодействие с клиентом в новом потоке, используя методы нового сокета **recv** и **send**. Когда со стороны сервера необходимо закрыть соединение, можно использовать метод сокета **close**.

Больше всего времени сервер проводит в п. 4.а, ожидая поступления запросов на создание соединения от клиентов.

Процесс работы клиента будет состоять из следующих этапов:

1) создание сокета типа *socket. SOCK STREAM*;

2) связывание сокета с конечной точкой службы (вызов его метода **bind**);

3) установка соединения с сервером (вызов метода сокета **connect**);

4) взаимодействие с сервером, используя методы сокета **recv** и **send;**

5) когда со стороны клиента необходимо закрыть соединение, можно использовать метод сокета **close**.

Метод **recv** блокирует процесс до тех пор, пока не будет получен ответ от сервера (или сервер не закроет соединение). Метод **send** блокирует процесс выполнения программы только в случае, если объем передаваемых данных настолько велик, что необходимо дождаться, пока транспортный уровень не освободит какую-то часть своей буферной памяти.

Такой подход к построению клиент-серверного приложения отличается большей сложностью. Отчасти это связано еще с тем, что в этом случае труднее установить момент, когда именно необходимо выполнить чтение и запись данных. Так, например, для понимания, что передача данных с другого конца сокета завершена, необходимо проанализировать принимаемые данные. Это достигается добавлением в заголовок передаваемых данных информации об их размере в байтах, а иногда и более сложными методами.

9.2. Модуль socket

В Python для обмена данными по сети используется модуль *socket*, который предоставляет разработчику интерфейс сокетов, скрывающий большинство различий между платформами.

Всего в модуле *socket* определено 4 исключения. Одно базовое — класс **socket.error,** и три производных от него:

1) *socket.herror* (*herror*) — исключение, генерируемое в случае неудачного преобразования заданного имени хоста в сетевой адрес с помощью функции **socket.gethostbyname** или неудачного определения имени хоста по заданному сетевому адресу функцией **socket.gethostbyaddr**. Сообщение, генерируемое исключением, представляет из себя кортеж (*h_errno*, *string*), где *h_errno* — код ошибки, а *string* — ее описание;

2) *socket.gaierror* (*gaierror*) — исключение, генерируемое в случае возникновения ошибок адресации в функциях **getaddrinfo** и **getnameinfo**;

3) *socket.timeout* (*timeout*) — исключение, генерируемое, если время выполнения операции превышает длительность предельного времени ожидания (может устанавливаться функцией **setdefaulttimeout** для каждого сокета).

Также модуль *socket* предоставляет разработчику большой набор констант для конфигурации создаваемых экземпляров сокетов. Наиболее значимыми из них принято считать константы семейства адресов (AF_*) и типы сокетов (SOCK_*). Их более подробное описание приведено в табл. 9.1.

Таблица 9.1

Описание части констант семейства адресов и типов сокетов модуля *socket*

Название константы	Описание
AF_INET	Создаваемый сокет будет работать с семейством адресов IPv4
AF_INET6	Создаваемый сокет будет работать с семейством адресов IPv6

Название константы	Описание
AF_CAN	Создаваемый сокет будет работать с семейством адресов Controller Area Network (CAN)
SOCK_STREAM	Создаваемый сокет будет работать по протоколу, ориентированному на установление соединения (TCP)
SOCK_DGRAM	Создаваемый сокет будет работать по протоколу, ориентированному на работу без установления соединения (UDP)
SOCK_RAW	Создаваемый сокет будет предоставлять непосредственный доступ к драйверам канального уровня (используется для реализации низкоуровневых сетевых средств)

Более подробно со всеми функциями модуля *socket* можно ознакомиться в документации [47].

9.3. Пример клиента и сервера, работающих без установления соединения

Своеобразным примером в виде всем известного «Hello World!» в сетевом программировании является разработка эхо-службы. В этом случае клиент отправляет на сервер какое-либо сообщение (допустим, текст), а сервер, принимая его, отправляет обратно клиенту.

Ниже приведен код клиента, отправляющего серверу сообщения и получающего от него ответ:

```
import socket
import time

UDP_IP = '127.0.0.1'
UDP_PORT = 8883

def run_client(ip, port):
    time.sleep(2)
    MESSAGE = u"""Эхо-служба (€) "Hello World!""""
    sock = socket.socket(socket.AF_INET,  # IPv4
                         socket.SOCK_DGRAM)  # UDP
    server = (ip, port)
    for line in MESSAGE.split(' '):
        data = line.encode('utf-8')
        sock.sendto(data, server)
        print(f'Клиент|| На сервер {server}'
              f'  отправлено: {repr(data)}')
        response, address = sock.recvfrom(1024)
# размер буфера: 1024
        print(f"Клиент|| Получены данные:
```

```
                f"{repr(response.decode('utf-8'))}, "
                  f"от {address}")
    print("Завершение работы клиента")
```

Теперь реализуем наш эхо-сервер:

```
def run_server(ip, port):
    sock = socket.socket(socket.AF_INET,  # IPv4
                         socket.SOCK_DGRAM)  # UDP
    server = (ip, port)
    sock.bind(server)
    print(f'Запуск эхо-сервера: {server}')

    while True:
        data, address = sock.recvfrom(1024)
# размер буфера: 1024
        print(f"Сервер|| Получены данные: "
              f"{repr(data.decode('utf-8'))}, "
                 f"от {address}")
        sock.sendto(data, address)
        print(f'Сервер|| Отправлены данные: '
              f' {repr(data)}, '
              f'по адресу: {address}')
```

Для запуска на локальной машине напишем следующий код, который запускает клиента и сервер в отдельных процессах:

```
if __name__ == "__main__":
    from multiprocessing import Process

    client = Process(target=run_client,
                     args=(UDP_IP, UDP_PORT,))
    server = Process(target=run_server,
                     args=(UDP_IP, UDP_PORT,))
    server.start()
    client.start()
    client.join()
    server.terminate()
'''
Запуск эхо-сервера: ('127.0.0.1', 8883)
Клиент|| На сервер ('127.0.0.1', 8883) отправлено: b'\xd0\xad\
xd1\x85\xd0\xbe-\xd1\x81\xd0\xbb\xd1\x83\xd0\xb6\xd0\xb1\xd0\
xb0'
Сервер|| Получены данные: 'Эхо-служба', от ('127.0.0.1', 63440)
Сервер|| Отправлены данные: b'\xd0\xad\xd1\x85\xd0\xbe-\xd1\
x81\xd0\xbb\xd1\x83\ xd0\xb6\xd0\xb1\xd0\xb0', по адресу:
('127.0.0.1', 63440)
Клиент|| Получены данные: 'Эхо-служба', от ('127.0.0.1', 8883)
Клиент|| На сервер ('127.0.0.1', 8883) отправлено: b'(\xe2\x82\
xac)'
Сервер|| Получены данные: '(€)', от ('127.0.0.1', 63440)
Сервер|| Отправлены данные: b'(\xe2\x82\xac)', по адресу:
('127.0.0.1', 63440)
Клиент|| Получены данные: '(€)', от ('127.0.0.1', 8883)
Клиент|| На сервер ('127.0.0.1', 8883) отправлено: b'\xc2\
xabHello'
```

```
Сервер|| Получены данные: '"Hello', от ('127.0.0.1', 63440)
Сервер|| Отправлены данные: b'\xc2\xabHello', по адресу:
('127.0.0.1', 63440)
Клиент|| Получены данные: '"Hello', от ('127.0.0.1', 8883)
Клиент|| На сервер ('127.0.0.1', 8883) отправлено: b'World!\
xc2\xbb'
Сервер|| Получены данные: 'World!"', от ('127.0.0.1', 63440)
Сервер|| Отправлены данные: b'World!\xc2\xbb', по адресу:
('127.0.0.1', 63440)
Клиент|| Получены данные: 'World!"', от ('127.0.0.1', 8883)
Завершение работы клиента
'''
```

Клиентский процесс завершит свою работу после получения всех отправленных ранее сообщений серверу. А для завершения работы сервера необходимо будет прервать процесс путем вызова у него метода **terminate**.

9.4. Пример клиента и сервера, работающих с установлением соединения

Теперь рассмотрим упрощенную организацию работы клиента и сервера, работающих с установлением соединения. В данном случае работа сервера будет организована следующим образом:

— ожидание подключения клиента;

— получение и обратная пересылка сообщений каждому клиенту до тех пор, пока тот не закроет соединение.

Ниже приведен код клиентской части примера:

```
import socket
from concurrent import futures as cf
import time

TCP_IP = 'localhost'
TCP_PORT = 8883

def run_client(ip, port):
    time.sleep(2)
    with socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM) as sock:
        sock.connect((ip, port))
        print ('Клиент|| Соединение с '
               'сервером установлено')
        data = u"""Эхо-служба (€) "Hello World!"""
        for line in data.split(' '):
            sock.sendall(line.encode('utf-8'))
            print(f'Клиент|| Отправлено: "{line}"')
            response = sock.recv(1024)
            print(f"Клиент|| Ответ сервера: "
                f" ''{response.decode('utf-8')}''")
    print('Клиент|| Соединение с сервером закрыто')
```

Код, отвечающий за работу эхо-сервера, приведен ниже:

```
def run_server(ip, port):
    def handle(new_sock, address):
        print(f'Сервер|| Установлено соединение '
              f' с : {address} ')
        while True:
            received = new_sock.recv(1024)
            if not received: break
            s = received.decode('utf-8',
                                errors='replace')
            print(f'Сервер|| Принято сообщение: '
                  f' "{s}" от {address}')
            new_sock.sendall(received)
            print(f'Сервер|| Эхо-сообщение: "{s}" '
                  f' отправлено {address}')
        new_sock.close()
        print(f'Сервер|| Закрыто соединение '
              f' с : {address}')

    servsock = socket.socket(socket.AF_INET,
                             socket.SOCK_STREAM)
    servsock.bind((ip, port))
    servsock.listen(5)
    print(f'Запуск эхо-сервера: '
          f' {servsock.getsockname()}')
    with cf.ThreadPoolExecutor(20) as client_pool:
        try:
            while True:
                new_sock, address = servsock.accept()
                client_pool.submit(handle,
                                   new_sock,
                                   address)
        except KeyboardInterrupt:
            pass
        finally:
            servsock.close()
```

Для запуска на локальной машине повторим написанный ранее код из примера, приведенного в предыдущем разделе:

```
if __name__ == "__main__":
    from multiprocessing import Process

    client = Process(target=run_client,
                     args=(TCP_IP, TCP_PORT,))
    server = Process(target=run_server,
                     args=(TCP_IP, TCP_PORT,))
    server.start()
    client.start()
    client.join()
    server.terminate()
'''
Запуск эхо-сервера: ('127.0.0.1', 8883)
Клиент|| Соединение с сервером установлено
```

```
Клиент|| Отправлено: "Эхо-служба"
Сервер|| Установлено соединение с : ('127.0.0.1', 60782)
Сервер|| Принято сообщение: "Эхо-служба" от ('127.0.0.1',
60782)
Сервер|| Эхо-сообщение: "Эхо-служба" отправлено ('127.0.0.1',
60782)
Клиент|| Ответ сервера: ''Эхо-служба''
Клиент|| Отправлено: "(€)"
Сервер|| Принято сообщение: "(€)" от ('127.0.0.1', 60782)
Сервер|| Эхо-сообщение: "(€)" отправлено ('127.0.0.1', 60782)
Клиент|| Ответ сервера: ''(€)''
Клиент|| Отправлено: ""Hello"
Сервер|| Принято сообщение: ""Hello" от ('127.0.0.1', 60782)
Сервер|| Эхо-сообщение: ""Hello" отправлено ('127.0.0.1',
60782)
Клиент|| Ответ сервера: '"Hello''
Клиент|| Отправлено: "World!""
Сервер|| Принято сообщение: "World!"" от ('127.0.0.1', 60782)
Сервер|| Эхо-сообщение: "World!"" отправлено ('127.0.0.1',
60782)
Клиент|| Ответ сервера: ''World!"''
Клиент|| Соединение с сервером закрыто
Сервер|| Закрыто соединение с : ('127.0.0.1', 60782)
'''
```

9.5. Фреймворк для сетевых серверов — socketserver

Модуль *socketserver* упрощает задачу написания сетевых серверов. Для примера приведем реализацию TCP и UDP эхо-сервера, использующую возможности рассматриваемого модуля. Для этого необходимо определить специальный класс-обработчик, реализующих метод **handle** для обслуживания соединений с клиентами.

9.5.1. TCP эхо-сервер с использованием TCPServer

Для начала импортируем **BaseRequestHandler** и **TCPServer** из модуля *socketserver*. После чего необходимо определить производный класс от **BaseRequestHandler** и переопределить в нем метод **handle**.

Код ранее написанной клиентской части менять не будем, а просто импортируем и запустим. Ниже приведен код TCP эхо-сервера и его вызова для демонстрации работы.

```
from socketserver import (BaseRequestHandler,
                          TCPServer)
from sample_part2 import run_client

TCP_IP = 'localhost'
TCP_PORT = 8000

class EchoTCP(BaseRequestHandler):
    def handle(self):
```

```
        print(f'Сервер|| Установлено соединение с '
                  f': {self.client_address}')
        while True:
            msg = self.request.recv(1024)
            if not msg:
                break
            print(f'Сервер|| Принято сообщение: '
                  f ' "{msg}" '
                  f'от {self.client_address}')
            print(f'Сервер|| Эхо-сообщение: "{msg}"'
               f'отправлено {self.client_address}')
            self.request.send(msg)

if __name__ == "__main__":
    from multiprocessing import Process
    # Объявляем TCP-сервер
    server = TCPServer((TCP_IP, TCP_PORT), EchoTCP)
    client = Process(target=run_client,
                     args=(TCP_IP, TCP_PORT,))
    client.start()
    server.serve_forever()
```

Результат работы данного кода будет идентичен приведенному в параграфе 9.4.

9.5.2. UDP эхо-сервер с использованием UDPServer

В большинстве своем код UDP эхо-сервера, написанный с использованием класса **UDPServer** будет аналогичен тому, что приведен в разделе выше. Изменения в основном коснутся переопределенного метода **handle**:

```
from socketserver import (BaseRequestHandler,
                          UDPServer)
from sample_part1 import run_client

UDP_IP = 'localhost'
UDP_PORT = 8000
class EchoUDP(BaseRequestHandler):
    def handle(self):
        print(f'Сервер|| Установлено соединение с '
              f': {self.client_address}')
        msg, sock = self.request
        print(f'Сервер|| Принято сообщение: '
              f' "{msg.decode("utf-8")}" '
              f'от {self.client_address}')
        print(f'Сервер|| Эхо-сообщение: '
              f '"{msg.decode("utf-8")}" '
              f'отправлено {self.client_address}')
        sock.sendto(msg, self.client_address)
if __name__ == "__main__":
    from multiprocessing import Process

    server = UDPServer((UDP_IP, UDP_PORT), EchoUDP)
    client = Process(target=run_client,
```

```
                    args=(UDP_IP, UDP_PORT,))
    client.start()
    server.serve_forever()
```

Результат работы аналогичен приведенному в параграфе 9.3.

Резюме

Все больше приложений, которые устанавливают пользователи на свой компьютер, работают на основе клиент-серверной архитектуры, что уж говорить про приложения на мобильных платформах. В связи с этим знания сетевых протоколов, инструментов разработки распределенных систем и т. п. можно часто встретить как одно из требований к вакансиям при поиске компаниями новых сотрудников.

Рассмотренные нами примеры представляют собой очень простые приложения, которые приведены для описания самих принципов взаимодействия между клиентом и сервером. Зачастую приложение на серверной стороне — многопроцессорное, асинхронное или многопоточное, в зависимости от решаемой им задачи.

Для разработки серверной части распределенных систем на языке программирования Python используются такие фреймворки, как ZMQ [48, 49], в то время, как для разработки серверной части вебсайтов и приложений используют Flask [50], Django [51], FaspAPI [52] и т. д.

Вопросы и задания для самопроверки

1. Какие существуют способы взаимодействия между программами по сети?
2. В каких случаях лучше использовать TCP?
3. В каких случаях лучше использовать UDP?
4. Приведите пример логики работы серверной части, работающей без установления соединения.
5. Приведите пример логики работы клиентской части, работающей без установления соединения.
6. Приведите пример логики работы серверной части, работающей с установлением соединения.
7. Приведите пример логики работы клиентской части, работающей с установлением соединения.
8. Для чего используется модуль *socket*?
9. Какие исключения определены в модуле *socket*?
10. Для чего используется модуль *socketserver* и каков принцип его работы?

Упражнения

1. Напишите программу (клиентскую и серверную часть), позволяющую общаться пользователям внутри локальной сети. Используйте при этом протокол TCP.
2. Напишите программу (клиентскую и серверную часть), позволяющую общаться пользователям внутри локальной сети. Используйте при этом протокол UDP.
3. Напишите клиент-серверное приложение, позволяющее передавать файл по сети. Используйте при этом протокол UDP.
4. Напишите клиент-серверное приложение, позволяющее передавать файл по сети. Используйте при этом протокол TCP.
5. Напишите программу, позволяющую играть в «Крестики-нолики» по сети.
6. Напишите программу, позволяющую играть в «Морской бой» по сети.
7. Напишите сетевой калькулятор. На клиентской стороне пользователь вводит выражение, а само его вычисление производится на серверной части.
8. Напишите клиент-серверное приложение, где на клиентской стороне вход в основное окно приложения осуществляется только после его авторизации на сервере, с возможностью регистрации нового пользователя. На серверной стороне хеш связки «логин:пароль» пользователей хранится в отдельном файле.
9. Напишите программу (клиентскую и серверную часть) для перевода значений в различную систему счисления (десятичная, восьмеричная, двоичная, шестнадцатеричная).
10. Напишите программу (клиентскую и серверную часть) для решения квадратных уравнений.

Тема 10
ХРАНЕНИЕ ДАННЫХ И ОБМЕН ДАННЫМИ

Для постоянного хранения и обмена данными между приложениями или составными частями одного приложения используются базы данных (БД). В данном разделе будет рассмотрена встроенная реляционная система управдения базами данных (СУБД) SQLite [53], которая позволяет не устанавливать на компьютер такие СУБД, как MySQL, PostgreSQL и т. д.

В Python за работу с SQLite отвечает модуль *sqlite3*, реализующий к ней интерфейс доступа. Поскольку БД SQLite хранится в файловой системе в виде одного файла, модуль *sqlite3* управляет к нему доступом и предотвращает повреждение данных, связанное с одновременной записью в БД из нескольких программ.

В результате изучения данной темы обучающиеся должны:

знать

- основные особенности работы с СУБД SQLite;

уметь

- создать собственную базу данных,
- совершать запись, удаление и редактирование данных в созданной базе данных;

владеть

- навыками работы с модулем *sqlite3*,
- навыками разработки приложений, осуществляющих работу с базой данных.

10.1. Создание базы данных SQLite

База данных SQLite создается в момент первой попытки обращения (подключения) к ней в ходе работы приложения, на которое ложится вся ответственность за создание таблиц, схемы базы данных и т. д.

В приведенном ниже коде сначала выполняется проверка существования БД в файловой системе, после чего выводится сообщение о подключении к уже существующей БД или создании новой, нуждающейся в создании схемы данных:

```
import os
import sqlite3
```

```
if __name__ == "__main__":
    db_name = 'my_todolist.db'
    db_is_created = os.path.exists(db_name)
    connection = sqlite3.connect(db_name)
    if db_is_created:
        print('Подключение к уже существующей БД!')
    else:
        print('Создание новой БД! Объявите '
                    структуру схемы данных!')
```

Теперь создадим определение таблиц в БД. Так как мы проектируем БД для ведения списка дел, то она должна содержать как минимум 2 таблицы: проекта и задач.

В таблице проекта хранятся следующие данные: его имя (текст), описание (текст) и срок завершения (дата). В таблице задач представлены следующие поля:

— уникальный идентификатор (число);
— числовой приоритет (число);
— описание задачи (текст);
— статус (текст);
— запланированный срок выполнения (дата);
— реальный срок выполнения (дата);
— имя проекта для данной задачи (текст).

Для описания структуры приведенных таблиц используется язык описания данных (*Data Definition Language*, DDL). Описание структуры таблиц можно сохранить в отдельном файле и подгружать в процессе создания новой базы данных:

```
# my_todolist_definition.sql
create table my_project(
    name primary key,
    description text,
    deadline date
);

create table my_task(
    id integer primary key autoincrement not null,
    priority integer default 0,
    description text,
    status text,
    deadline date
    real_done_date date,
    project text not null references my_project(name)
);
```

Теперь перепишем приведенный ранее код таким образом, чтобы при создании новой БД сразу создавалась структура таблиц, заполняемых набором значений:

```
import os
import sqlite3
```

```
if __name__ == "__main__":
    db_name = 'my_todolist.db'
    db_schema_name = 'my_todolist_definition.sql'
    db_is_created = os.path.exists(db_name)
    connection = sqlite3.connect(db_name)
    if db_is_created:
        print('Подключение к уже существующей БД!')
    else:
        print('Создание новой БД!')
    with open(db_schema_name, 'rt') as schema_file:
        my_schema = schema_file.read()
        # выполнение кода создания таблиц,
        # загруженного из my_todolist_definition.sql
    connection.executescript(my_schema)

    print('Добавление записей в БД!')
    connection.executescript('''
        insert into my_project(name, description,
                               deadline)
        values ('MagicMonth', 'Изучить Python за 21
                 день', '2020-09-01')
    ''')
    connection.executescript('''
        insert into my_task(description, deadline,
                            status, project)
        values ('Синтаксис и структуры данных',
                '2020-08-16',
                'done', 'MagicMonth')
    ''')
    connection.executescript('''
        insert into my_task(description, deadline,
                 status,  project)
        values ('Функции, классы, модули',
                '2020-08-25', 'wait', 'MagicMonth')
    ''')
    connection.executescript('''
        insert into my_task(description, deadline,
                            status,  project)
        values ('Поплакать, что не так всё просто',
                '2020-08-20',
                'wait', 'MagicMonth')
    ''')
# Создание новой БД!
# Добавление записей в БД!
```

За извлечение данных из таблиц БД отвечает объект *Cursor*, обеспечивающий согласованное представление обрабатываемых записей для различных типов данных:

```
import sqlite3

db_name = 'my_todolist.db'
with sqlite3.connect(db_name) as connection:
    cursor = connection.cursor()
```

```
    # сообщаем БД, какие данные должны быть выбраны
    cursor.execute("""
        select id, priority, description,
        status, deadline from my_task
        where project = 'MagicMonth'
        """)
    # извлекаем данные
    for it_row in cursor.fetchall():
        id, priority, description, status,
                                  deadline = row
        print(f'{id}, {priority}, {description}, '
              f' {status}, {deadline}')
'''
1, 0, Синтаксис и структуры данных, done, 2020-08-16
2, 0, Функции, классы, модули, wait, 2020-08-25
3, 0, Поплакать, что не так всё просто, wait, 2020-08-20
'''
```

Из приведенного выше кода видно, что данные извлекаются в форме кортежа и их правильная распаковка ложится на плечи разработчика. Когда же количество извлекаемых данных слишком большое, лучше осуществлять работу с ними, обращаясь по именам столбцов. За эту функциональность в модуле *sqlite3* отвечает класс **Row**. Для его использования необходимо у экземпляра класса соединения с БД изменить значение атрибута *row_factory*, отвечающего за управление типом объектов результирующего набора запроса:

```
import sqlite3

if __name__ == "__main__":
    db_name = 'my_todolist.db'
    with sqlite3.connect(db_name) as connection:
        connection.row_factory = sqlite3.Row
        cursor = connection.cursor()
# сообщаем БД, какие данные должны быть выбраны
# и отсортированы в порядке дедлайнов
        cursor.execute('''
            select id, priority, description,
                   status, deadline from my_task
            where project = 'MagicMonth'
            ''')
# извлекаем данные
        for it_row in cursor.fetchall():
            print(f"{it_row['id']}, "
                  f"{it_row['priority']}, "
                  f"{it_row['description']}, "
                  f"{it_row['status']}",
                  f"{it_row['deadline']}")
'''
1, 0, Синтаксис и структуры данных, done,  2020-08-16
2, 0, Функции, классы, модули, wait,  2020-08-25
3, 0, Поплакать, что не так всё просто, wait,  2020-08-20
'''
```

10.2. Использование переменных в запросах

Приведенные выше запросы лишены гибкости. Так, например, при добавлении в базу данных другого проекта необходимо будет сформировать для него перечень запросов, что отрицательно скажется на удобстве повторного использования ранее написанного кода. Чтобы этого избежать, можно при формировании запросов использовать переменные, уже существующие в рамках разрабатываемого приложения. Формировать такие SQL-инструкции необходимо с особой осторожностью, так как неправильное экранирование данных может привести к тому, что злоумышленники могут прибегнуть к атакам путем внедрения SQL-кода (SQL-инъекции) и ничем хорошим это не закончится.

SQLite поддерживает две формы инструкций, которые обеспечивают замену подстановочных символов в запросах: позиционную и именованную.

10.2.1. Запросы с позиционными параметрами

Позиционные аргументы в запросе к БД задаются посредством символа «?». Сами значения передаются запросу в кортеже, являясь одним из аргументов метода **execute**:

```
import sqlite3

if __name__ == "__main__":
    db_name = 'my_todolist.db'
    with sqlite3.connect(db_name) as connection:
        connection.row_factory = sqlite3.Row
        cursor = connection.cursor()
     # сообщаем БД, какие данные должны быть выбраны
     # и отсортированы в порядке дедлайнов
        query = """select id, priority, description,
                   status, deadline
                   from my_task where project = ?
                   order by deadline
            """
        cursor.execute(query, ('MagicMonth',))
        # извлекаем данные
        for it_row in cursor.fetchall():
            print(f"{it_row['id']}, "
                  f"{it_row['priority']}, "
                  f"{it_row['description']}, "
                  f"{it_row['status']}",
                  f"{it_row['deadline']}")

'''
1, 0, Синтаксис и структуры данных, done 2020-08-16
3, 0, Поплакать, что не так всё просто, wait 2020-08-20
2, 0, Функции, классы, модули, wait 2020-08-25
'''
```

10.2.2. Запросы с именованными параметрами

Данный вид запросов используется в случае большого количества параметров, а также если параметры могут повторяться несколько раз. В этом случае при формировании запроса используется обозначение вида «:arg_name»:

```
import sqlite3

if __name__ == "__main__":
    db_name = 'my_todolist.db'
    with sqlite3.connect(db_name) as connection:
        connection.row_factory = sqlite3.Row
        cursor = connection.cursor()
# сообщаем БД, какие данные должны быть выбраны
# и отсортированы в порядке дедлайнов и приоритетов

        query = """select id, priority, description,
                   status, deadline
                   from my_task where
                      project = :my_proj_name
                      order by deadline, priority"""
        cursor.execute(query,
                    {'my_proj_name': 'MagicMonth'})
        # извлекаем данные
        for it_row in cursor.fetchall():
            print(f"{it_row['id']}, "
                  f"{it_row['priority']}, "
                  f"{it_row['description']}, "
                  f"{it_row['status']}",
                  f"{it_row['deadline']}")
# 1, 0, Синтаксис и структуры данных, done 2020-08-16
# 3,0, Поплакать, что не так всё просто, wait 2020-08-20
# 2, 0, Функции, классы, модули, wait 2020-08-25
```

Параметрические запросы можно использовать с такими операторами, как *select*, *insert* и *update* в любом месте запроса, где допустимо использование литерального значения.

10.3. Транзакции

Реляционные базы данных предоставляют механизм управления транзакциями, осуществляющий поддержку согласованного внутреннего состояния базы данных. Он позволяет внести изменения посредством одного подключения, при этом не влияя на работу сторонних пользователей, пока вносимые изменения не будут зафиксированы (записаны в БД).

Для этого все изменения, которые могут вноситься в БД с использованием инструкций *insert* и *update*, должны фиксироваться посредством вызова метода **commit**. Такой способ внесения измене-

ний позволяет за раз внести различное их количество. Изменяемые (вносимые) данные сохранятся атомарно, а не по очереди.

В коде ниже приводится пример внесения изменений в базу данных посредством механизма транзакций:

```
import sqlite3

def all_projects_show(connection):
    cursor = connection.cursor()
    cursor.execute('select name, description,
                    deadline  from my_project')
    print("Текущие проекты в списке дел: ")
    for name, description, deadline  \
                          in cursor.fetchall():
        print("----------------------------------")
        print(f'  Имя проекта:"{name}"  \n'
              f'  Описание: "{description}" \n'
              f'  Предельный срок выполнения: '
              f'{deadline}')
    print("-----------------------------------")

if __name__ == "__main__":
    db_name = 'my_todolist.db'
    with sqlite3.connect(db_name) as first_connection:
        first_cursor = first_connection.cursor()
        print("Начальное состояние БД")
        all_projects_show(first_connection)
# вывести текущие проекты
        first_cursor.execute('''
                insert into my_project(name,
                             description, deadline)
                values ('SuperProgrammer',
                        'после изучения Python за 21
                         день устроиться
                        на должность Senior Python
                        Developer',
                        '2020-09-16')
            ''')
        print("Состояние БД, после "
              " внесения изменений")
        all_projects_show(first_connection)

    print("Состояние БД, до вызова метода commit")
    with sqlite3.connect(db_name) as second_connection:
        all_projects_show(second_connection)
# вывести текущие проекты

    first_connection.commit()
    print("Состояние БД, после вызова "
          " метода commit")
    with sqlite3.connect(db_name) as third_connection:
        all_projects_show(third_connection)
# вывести текущие проекты
'''
```

```
Начальное состояние БД
Текущие проекты в списке дел:
----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
----------------------------------------
Состояние БД, после внесения изменений
Текущие проекты в списке дел:
----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
----------------------------------------
  Имя проекта:"SuperProgrammer"
  Описание: "после изучения Python за 21 день устроиться
                        на должность Senior Python Developer"
  Предельный срок выполнения: 2020-09-16
----------------------------------------
Состояние БД, до вызова метода commit
Текущие проекты в списке дел:
----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
----------------------------------------
  Имя проекта:"SuperProgrammer"
  Описание: "после изучения Python за 21 день устроиться
                        на должность Senior Python Developer"
  Предельный срок выполнения: 2020-09-16
----------------------------------------
Состояние БД, после вызова метода commit
Текущие проекты в списке дел:
----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
----------------------------------------
  Имя проекта:"SuperProgrammer"
  Описание: "после изучения Python за 21 день устроиться
                        на должность Senior Python Developer"
  Предельный срок выполнения: 2020-09-16
----------------------------------------
'''
```

До тех пор, пока внесенные изменения в базу данных не зафиксированы, от них можно отказаться посредством вызова метода **rollback**. Для того чтобы приложение корректно обеспечивало внесение изменений и их отмену, методы rollback и commit вызываются из разных частей блока *try*:

```
import sqlite3

def all_projects_show(connection):
    cursor = connection.cursor()
```

```
    cursor.execute('select name, description,
                    deadline  from my_project')
    print("Текущие проекты в списке дел: ")
    for name, description, deadline \
                          in cursor.fetchall():
        print("----------------------------------")
        print(f'  Имя проекта:"{name}"  \n'
              f'  Описание: "{description}" \n'
              f'  Предельный срок выполнения: '
              f' {deadline}')
    print("-----------------------------------")

if __name__ == "__main__":
    db_name = 'my_todolist.db'
    with sqlite3.connect(db_name) as connection:
        print("Начальное состояние БД")
        all_projects_show(connection)
# вывести текущие проекты

        try:
            cursor = connection.cursor()
            cursor.execute("""delete from
                    my_project where name =
                    'SuperProgrammer'
            """)
            print("Состояние БД, после " +
                   "удаления проекта")
            all_projects_show(connection)

            raise RuntimeError("ошибка при"  +
                               "удалении")

        except Exception as my_simulated_error:
            print("Откат изменений")
            connection.rollback()
        else:
            connection.commit()

        print("Состояние БД, после вызова "+
              "метода rollback")
        all_projects_show(connection)
'''
Начальное состояние БД
Текущие проекты в списке дел:
----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
----------------------------------------
  Имя проекта:"SuperProgrammer"
      Описание: "после изучения Python за 21 день
      устроиться на должность Senior Python Developer"
  Предельный срок выполнения: 2020-09-16
----------------------------------------
```

```
Состояние БД, после удаления проекта
Текущие проекты в списке дел:
-----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
-----------------------------------------
Откат изменений
Состояние БД, после вызова метода rollback
Текущие проекты в списке дел:
-----------------------------------------
  Имя проекта:"MagicMonth"
  Описание: "Изучить Python за 21 день"
  Предельный срок выполнения: 2020-09-01
-----------------------------------------
  Имя проекта:"SuperProgrammer"
  Описание: "после изучения Python за 21 день
  устроиться на должность Senior Python Developer"
  Предельный срок выполнения: 2020-09-16
-----------------------------------------
'''
```

10.4. Уровни изоляции (доступа)

Уровни изоляции используются для предотвращения внесения изменений, приводящих к несовместимости состояний БД при использовании различных соединений (как, например, в многопоточном приложении). Всего существует три уровня изоляции, передаваемые в качестве аргумента *isolation_level* методу **connect** при соединении с базой данных:

1) *DEFERRED* — используется как уровень изоляции по умолчанию. Блокирует БД с момента попытки ее изменения;

2) *IMMEDIATE* — блокирует БД при попытке ее изменения до тех пор, пока транзакция не завершится. При этом другие экземпляры объекта *Cursor* лишаются возможности вносить изменения в БД;

3) *EXCLUSIVE* — блокирует БД для всех объектов чтения и записи до момента завершения транзакции.

Если атрибут *isolation_level* установлен в *None*, включится режим автоматической фиксации транзакций, что повлечет к фиксации изменений после каждого успешного вызова метода **execute** и исключает необходимость вызова **commit**.

Приведем пример кода, который демонстрирует влияние различных уровней изоляции на очередность событий в потоках, использующих отдельные соединения для подключения к одной и той же базе данных. Для синхронизации потоков используется объект *Event* из модуля *threading*:

```
import sqlite3
import logging
```

```
from threading import Thread, Event
import time

logging.basicConfig(
      level=logging.DEBUG,
      format='%(asctime)s (%(threadName)-10s) '
             '%(message)s', )

db_name = 'my_todolist.db'

def writer_to_db(sync_event, level):
    with sqlite3.connect(db_name,
              isolation_level=level) as connection:
        cursor = connection.cursor()
        cursor.execute('update my_task set priority
                        = priority+1')
        logging.debug("Ожидание синхронизации")
        sync_event.wait()
        logging.debug("Пауза в работе")
        time.sleep(1)
        connection.commit()
        logging.debug("Фиксация изменений")

def reader_to_db(sync_event, level):
    with sqlite3.connect(db_name,
             isolation_level=level) as connection:
        cursor = connection.cursor()
        logging.debug("Ожидание синхронизации")
        sync_event.wait()
        logging.debug("Ожидание")
        cursor.execute('select * from my_task')
        logging.debug("Выборка из таблицы " +
                      "выполнена")
        cursor.fetchall()
        logging.debug("Получение результата " +
                      " выборки")

if __name__ == "__main__":
    isolation_level = 'DEFERRED'
    sync_event = Event()
    my_threads = [
        Thread(name='А.С. Пушкин',
               target=writer_to_db,
               args=(sync_event,isolation_level,)),
        Thread(name='Л.Н. Толстой',
               target=writer_to_db,
               args=(sync_event,isolation_level,)),
        Thread(name='Иванов И. И.',
               target=reader_to_db,
               args=(sync_event,isolation_level,)),
        Thread(name='Сидоров С. С.',
               target=reader_to_db,
               args=(sync_event,isolation_level,)),
    ]
```

```
    [it.start() for it in my_threads]

    time.sleep(2)
    logging.debug("Подготовительные работы " +
                  " завершены")
    sync_event.set()

    [it.join() for it in my_threads]
```

Ниже приведен пример вывода состояний программы в процессе ее работы с установленным уровнем изоляции *DEFERRED*:

```
'''
2020-08-05 17:56:17,509 (Иванов И. И.) Ожидание синхронизации
2020-08-05 17:56:17,509 (Сидоров С. С.) Ожидание синхронизации
2020-08-05 17:56:17,510 (А.С. Пушкин) Ожидание синхронизации
2020-08-05 17:56:19,509 (MainThread) Подго-ые работы завершены
2020-08-05 17:56:19,509 (Иванов И. И.) Ожидание
2020-08-05 17:56:19,509 (А.С. Пушкин) Пауза в работе
2020-08-05 17:56:19,509 (Сидоров С. С.) Ожидание
2020-08-05 17:56:19,510 (Иванов И. И.) Выборка из таблицы
выполнена
2020-08-05 17:56:19,510 (Иванов И. И.) Получение результата
выборки
2020-08-05 17:56:19,511 (Сидоров С. С.) Выборка из таблицы
выполнена
2020-08-05 17:56:19,511 (Сидоров С. С.) Получение результата
выборки
2020-08-05 17:56:20,518 (А.С. Пушкин) Фиксация изменений
2020-08-05 17:56:20,569 (Л.Н. Толстой) Ожидание синхронизации
2020-08-05 17:56:20,570 (Л.Н. Толстой) Пауза в работе
2020-08-05 17:56:21,578 (Л.Н. Толстой) Фиксация изменений
'''
```

В случае работы с установленным уровнем изоляции *IMMEDIATE* вывод состояния примет следующий вид:

```
'''
2020-08-05 17:58:34,819 (Иванов И. И.) Ожидание синхронизации
2020-08-05 17:58:34,819 (Сидоров С. С.) Ожидание синхронизации
2020-08-05 17:58:34,820 (А.С. Пушкин) Ожидание синхронизации
2020-08-05 17:58:36,819 (MainThread) Подго-ые работы завершены
2020-08-05 17:58:36,819 (А.С. Пушкин) Пауза в работе
2020-08-05 17:58:36,819 (Сидоров С. С.) Ожидание
2020-08-05 17:58:36,819 (Иванов И. И.) Ожидание
2020-08-05 17:58:36,820 (Сидоров С. С.) Выборка из таблицы
выполнена
2020-08-05 17:58:36,821 (Иванов И. И.) Выборка из таблицы
выполнена
2020-08-05 17:58:36,821 (Сидоров С. С.) Получение результата
выборки
2020-08-05 17:58:36,821 (Иванов И. И.) Получение результата
выборки
2020-08-05 17:58:37,826 (А.С. Пушкин) Фиксация изменений
```

```
2020-08-05 17:58:37,883 (Л.Н. Толстой) Ожидание синхронизации
2020-08-05 17:58:37,883 (Л.Н. Толстой) Пауза в работе
2020-08-05 17:58:38,891 (Л.Н. Толстой) Фиксация изменений
'''
```

Для уровня изоляции *EXCLUSIVE* актуален следующий способ работы:

```
'''
2020-08-05 17:59:41,929 (Иванов И. И.) Ожидание синхронизации
2020-08-05 17:59:41,929 (Сидоров С. С.) Ожидание синхронизации
2020-08-05 17:59:41,930 (Л.Н. Толстой) Ожидание синхронизации
2020-08-05 17:59:43,928 (MainThread) Подго-ые работы завершены
2020-08-05 17:59:43,928 (Сидоров С. С.) Ожидание
2020-08-05 17:59:43,928 (Л.Н. Толстой) Пауза в работе
2020-08-05 17:59:43,928 (Иванов И. И.) Ожидание
2020-08-05 17:59:44,933 (Л.Н. Толстой) Фиксация изменений
2020-08-05 17:59:44,934 (Сидоров С. С.) Выборка из таблицы
выполнена
2020-08-05 17:59:44,934 (Сидоров С. С.) Получение результата
выборки
2020-08-05 17:59:44,936 (Иванов И. И.) Выборка из таблицы
выполнена
2020-08-05 17:59:44,936 (Иванов И. И.) Получение результата
выборки
2020-08-05 17:59:45,025 (А.С. Пушкин) Ожидание синхронизации
2020-08-05 17:59:45,025 (А.С. Пушкин) Пауза в работе
2020-08-05 17:59:46,033 (А.С. Пушкин) Фиксация изменений
'''
```

Резюме

В данном разделе мы рассмотрели возможности постоянного хранения и обмена данными между приложениями или составными частями одного приложения на примере работы с встроенной реляционной базой данных SQLite и модуля *sqlite3*.

Использование непараметрических запросов (позиционных или именованных) лишает ваш код гибкости, а также может привести к неправильному экранированию данных, чем с удовольствием воспользуются злоумышленники, прибегнув к атакам путем внедрения SQL-кода (SQL-инъекции).

Для поддержки согласованного внутреннего состояния БД используйте механизм управления транзакциями и правильно настраивайте уровни изоляции в зависимости от решаемой задачи.

Чтобы более подробно ознакомиться с различными возможностями SQLite, воспользуйтесь руководством [54].

Вопросы для самопроверки

1. Какой модуль в Python отвечает за работу с SQLite?
2. Какая сторона должна обеспечить создание схемы базы данных, таблиц и т. д., если база данных SQLite создается впервые?
3. За что отвечает метод connect и какие атрибуты у него существуют?
4. Что выполняет метод executerscript?
5. Что выполняет метод execute?
6. Какие виды параметрических запросов существуют и в чем их различие?
7. Зачем использовать запросы с параметрами?
8. За что отвечает механизм транзакций в SQLite?
9. Как можно отменить еще не зафиксированные в базе данных изменения?
10. Что такое уровни изоляции и зачем они нужны?
11. Что будет, если isolation_level установлен в None?

Упражнения

1. Спроектируйте базу данных и напишите программу «Список дел» с возможностью сохранения, удаления и редактирования данных. Используйте SQLite.
2. Спроектируйте базу данных и напишите программу «Электронный журнал класса» с возможностью сохранения, удаления и редактирования данных.
3. Спроектируйте базу данных и напишите программу «Электронный каталог библиотеки» с возможностью сохранения, удаления и редактирования данных.
4. Спроектируйте базу данных и напишите программу «Электронный каталог продуктов магазина» с возможностью сохранения, удаления и редактирования данных.
5. Спроектируйте базу данных и напишите программу «Электронная фильмотека» с возможностью сохранения, удаления и редактирования данных.
6. Спроектируйте базу данных и напишите программу «Таксопарк» с возможностью сохранения, удаления и редактирования данных.
7. Спроектируйте базу данных и напишите программу «Электронный отдел кадров (любой фирмы)» с возможностью сохранения, удаления и редактирования данных.
8. Спроектируйте базу данных и напишите программу «Электронная поваренная книга» с возможностью сохранения, удаления и редактирования данных.
9. Спроектируйте базу данных и напишите программу «Электронная телефонная книга» с возможностью сохранения, удаления и редактирования данных.
10. Спроектируйте базу данных и напишите программу «Электронная регистратура (поликлиники/библиотеки/и т. д.)» с возможностью сохранения, удаления и редактирования данных.

Тема 11
ТЕСТИРОВАНИЕ

В результате изучения данной темы обучающиеся должны:

знать

- для чего необходимо тестировать разрабатываемые приложения,
- как организовать структуру тестового окружения проекта;

уметь

- тестировать написанный код с использованием библиотеки *pytest*,
- тестировать написанный код с использованием встроенного модуля *unittest*;

владеть

- навыками написания тестов к разрабатываемым приложениям.

В современных реалиях уже трудно представить серьезный проект, в котором не использовалось бы тестирование (хотя, признаюсь, приходилось лицезреть и такие). Несмотря на то, что многие разработчики рассматривают тестирование как достаточно нудный и трудоемкий процесс, оно позволяет: сократить время отладки кода в проекте, повысить качество работы разрабатываемого приложения и т. д.

Конечно, можно привести много доводов за и против тестирования, но необходимо понимать одно — покрытие проекта тестами уже многие годы де-факто является стандартом в IT-индустрии.

Чтобы сделать тестирование более управляемым, рекомендуется придерживаться следующего подхода. В каталоге проекта необходимо создать папку с именем «test», куда и будут помещаться тесты модулей проекта. Так, например, для файла *file_name.py*, расположенного по адресу *./myProject/*, тест должен находиться по следующему пути: *./myProject/test/test_file_name.py*. Обратите внимание на само именование файла с некоторым кодом проекта и теста, который будет проверять его функциональность. Они практически идентичны, за тем исключением, что название тестового файла начинается с префикса «test».

Для тестирования будем использовать библиотеку *pytest* [55], которую необходимо установить, написав в терминале виртуальной среды (venv) проекта следующую команду:

```
pip install pytest
```

Этой библиотеки достаточно для небольших проектов. Она предоставляет команду *pytest*, которая загружает каждый файл (если не задавать имя тестового файла явно), имя которого начинается с префикса «test_» (или постфикса «_test.py»), после чего выполняет все функции внутри каждого из файлов, если они тоже начинаются с «test».

11.1. Тестирование с использованием библиотеки PyTest

11.1.1. Простые тесты

Для демонстрации работы библиотеки *pytest* создадим две директории: *src* — для кода приложения и *test* — для тестирования модулей, хранящихся в *src*. В оба каталога добавим по пустому файлу *__init__.py*, после чего в первом каталоге создадим файл с именем *calculation.py*, а во втором — *test_calculation.py* для его тестирования. Добавим в эти файлы следующий код:

```
#calculation.py
class Calculation:

    def sum(self, a, b):
        return a + b

    def sub(self, a, b):
        return a-b

    def mul(self, a, b):
        return a*b

    def div(self, a, b):
        return a//b

    def pow2(self, a):
        return a*a

# test_calculation.py
from src.calculation import Calculation

def test_sum():
    assert Calculation().sum(3, 2) == 5

def test_sub():
    assert Calculation().sub(10, 3) == 6

def test_mul():
    assert Calculation().mul(5, 5) == 25

def test_div():
    assert Calculation().div(10, 3) == 3
```

```
def test_pow2():
    assert Calculation().pow2(10) == 99

def tes_invisible():
    assert 23 == 1
```

При запуске команды *pytest* будет сгенерирован следующий вывод:

```
==test session starts =============
collected 5 items

test\test_calculation.py .F..F   [100%]

==FAILURES ==================
___test_sub _________________________
    def test_sub():
>       assert 10-3 == 6
E       assert (10 - 3) == 6

test\test_calculation.py:5: AssertionError
___test_pow2 ________________________
    def test_pow2():
>       assert 10*10 == 99
E       assert (10 * 10) == 99

test\test_calculation.py:14: AssertionError
==short test summary info================
FAILED test/test_calculation.py::test_sub - assert (10 - 3) ==
6
FAILED test/test_calculation.py::test_pow2 - assert (10 * 10)
== 99
==2 failed, 3 passed in 0.09s ============
```

Несмотря на то, что в процессе тестирования мы нашли функции, которые его не прошли, хотелось бы иметь более читаемый вывод о ходе тестирования. Для этого повторно запустим команду *pytest* с ключом *–v*:

```
 pytest  -v
==test session starts =================
cachedir: .pytest_cache
rootdir: F:\code\python\module-11
collected 5 items

test/test_calculation.py::test_sum PASSED    [ 20%]
test/test_calculation.py::test_sub FAILED    [ 40%]
test/test_calculation.py::test_mul PASSED    [ 60%]
test/test_calculation.py::test_div PASSED    [ 80%]
test/test_calculation.py::test_pow2 FAILED   [100%]

== FAILURES ============================
___ test_sub ___________________________________
    def test_sub():
>       assert 10-3 == 6
```

```
E       assert 7 == 6
E         +7
E         -6

test\test_calculation.py:5: AssertionError
__test_pow2 ____________________________________
    def test_pow2():
>       assert 10*10 == 99
E       assert 100 == 99
E         +100
E         -99

test\test_calculation.py:14: AssertionError
==short test summary info ======================
FAILED test/test_calculation.py::test_sub - assert 7 == 6
FAILED test/test_calculation.py::test_pow2 - assert 100 == 99
==2 failed, 3 passed in 0.08s ==================
```

11.1.2. Запуск одного или нескольких тестов

Теперь проверим, как *pytest* запускает сразу несколько тестовых файлов. Для этого создадим следующие файлы: my_pow.p» и test_pow.py, в которые добавим приведенный ниже код:

```
#my_pow.py
import math

def my_pow2(a):
    return a*a

def my_pow2_math(a):
    return math.pow(a, 2)

def my_pow3_math(a):
    return math.pow(a, 3)

#test_pow.py
from src.my_pow import *

def test_pow2():
    assert my_pow2(10) == 100

def test_pow2_math():
    assert my_pow2_math(10) == 100

def test_math_pow3():
    assert my_math_pow3(2) == 8

def tes_notpow():
    assert 23 == 1
```

После очередного запуска команды *pytest* с ключом *–v* получим следующий вывод:

```
test/test_calculation.py::test_sum PASSED      [ 20%]
test/test_calculation.py::test_sub FAILED      [ 40%]
```

```
test/test_calculation.py::test_mul PASSED     [ 60%]
test/test_calculation.py::test_div PASSED     [ 80%]
test/test_calculation.py::test_pow2 FAILED    [100%]
test/test_pow.py::test_pow2 PASSED            [ 75%]
test/test_pow.py::test_pow2_math PASSED      [ 87%]
test/test_pow.py::test_math_pow3 PASSED      [100%]
```

Если необходимо запустить проверку тестов только из одного файла, то его имя надо явно передать при вызове команды *pytest*:

```
pytest test/test_pow.py -v
==test session starts ==================
cachedir: .pytest_cache
rootdir: F:\code\python\module-11
collected 3 items

test/test_pow.py::test_pow2 PASSED             [ 33%]
test/test_pow.py::test_pow2_math PASSED        [ 66%]
test/test_pow.py::test_math_pow3 PASSED       [100%]
==3 passed in 0.01s ==================
```

11.1.3. Запуск подмножества тестов

Для запуска некоторого подмножества тестов из всех, которые имеются в проекте, *pytest* предоставляет несколько механизмов:

— запуск тестов, в названии которых встречается задаваемая подстрока;

— запуск тестов, основанный на их маркировке.

В первом случае при запуске команды *pytest* посредством использования ключа *–k* задается подстрока для поиска в именах тестов. Для примера запустим все тесты, в которых содержится подстрока «pow»:

```
pytest  -k  pow  -v
==test session starts ====================
cachedir: .pytest_cache
rootdir: F:\code\python\module-11
collected 8 items / 4 deselected / 4 selected

test/test_calculation.py::test_pow2 FAILED     [ 25%]
test/test_pow.py::test_pow2 PASSED             [ 50%]
test/test_pow.py::test_pow2_math PASSED       [ 75%]
test/test_pow.py::test_math_pow3 PASSED       [100%]

==FAILURES ========================
__test_pow2 _____________________________
    def test_pow2():
>       assert 10*10 == 99
E       assert 100 == 99
E         +100
E         -99

test\test_calculation.py:14: AssertionError
==short test summary info ================
```

```
FAILED test/test_calculation.py::test_pow2 - assert 100 == 99
==1 failed, 3 passed, 4 deselected in 0.07s =====
```

Во втором случае необходимо импортировать модуль *pytest* в тестовый файл и произвести маркировку тестов, используя декоратор следующего вида:

```
@pytest.mark.<marker_name>
```

У *pytest* имеется встроенное множество собственных маркеров, отвечающих за пропуск теста, его параметризацию и т. д.

В качестве примера скопируем рассматриваемые ранее тестовые файлы, изменив названия следующим образом: *test_pow_mark.py* и *test_calculation_mark.py*, а следом декорируем их различными маркерами:

```
# test_pow_mark.py
import pytest
from src.my_pow import *

@pytest.mark.pow2
def test_pow2():
    assert my_pow2(10) == 100

@pytest.mark.pow2
def test_pow2_math():
    assert my_pow2_math(10) == 100

@pytest.mark.pow3
def test_math_pow3():
    assert my_pow3_math(2) == 8

@pytest.mark.skip("Не трогать!")
def tes_notpow():
    assert 23 == 1
# test_calculation_mark.py
import pytest
from src.calculation import Calculation

@pytest.mark.calc
def test_sum():
    assert Calculation().sum(3, 2) == 5

@pytest.mark.calc
def test_sub():
    assert Calculation().sub(10, 3) == 6

@pytest.mark.calc
def test_mul():
    assert Calculation().mul(5, 5) == 25

@pytest.mark.calc
def test_div():
    assert Calculation().div(10, 3) == 3
```

```
@pytest.mark.pow2
def test_pow2():
    assert Calculation().pow2(10) == 99

@pytest.mark.skip("Не трогать!")
def tes_invisible():
    assert 23 == 1
```

Теперь запустим только те тесты, которые соответствуют маркеру «calc». Для этого в качестве одного из атрибутов команды *pytest* используем ключ *–m*, за которым следует имя маркера тестов:

```
pytest  -m  calc  -v
==test session starts ====================
cachedir: .pytest_cache
rootdir: F:\code\python\module-11
collected 16 items / 12 deselected / 4 selected

test/test_calculation_mark.py::test_sum PASSED [ 25%]
test/test_calculation_mark.py::test_sub FAILED [ 50%]
test/test_calculation_mark.py::test_mul PASSED [75%]
test/test_calculation_mark.py::test_div PASSED [100%]

==FAILURES ========================
__test_sub _______________________________
    @pytest.mark.calc
    def test_sub():
>       assert 10-3 == 6
E       assert 7 == 6
E         +7
E         -6

test\test_calculation_mark.py:9: AssertionError

==short test summary info ========================
FAILED test/test_calculation_mark.py::test_sub - assert 7 == 6
==1 failed, 3 passed, 12 deselected, 8 warnings in 0.09s ===
```

Если необходимо пропустить тест, то декорируйте его следующим образом:

```
@pytest.mark.skip("Не запускать!")
```

11.1.4. Использование фикстур

В ходе тестирования может понадобиться выполнять до и (или) после запуска теста какой-либо стандартный набор операций, задействующих различные компоненты (в рамках *pytest* такие компоненты называются фикстурами). Так, например, это может быть объект, который отвечает за конфигурацию приложения. Возможно, в ходе тестирования понадобится создавать файл, а после — удалять. При использовании приведенных ранее механизмов тестирования вам бы потребовалось каждый раз перед очередным тестом

проводить его инициализацию, а потом сбрасывать его состояние до начальных значений после выполнения.

Чтобы объявить обычную или генераторную функцию в качестве фикстуры, ее необходимо декорировать следующим образом:

```
@pytest.fixture
def input_value():
   value = 374 # так делать не обязательно
   return value
```

Давайте создадим файл *test_new_calculation.py*, в который поместим заданную фикстуру и адаптируем код файла *test_calculation.py* под ее использование:

```
#test_new_calculation.py
import pytest
from src.calculation import Calculation

@pytest.fixture
def input_value():
    return 374

def test_sum(input_value):
    assert Calculation().sum(input_value, 2) == 376

def test_sub(input_value):
    assert Calculation().sum(input_value, 3) == 6
def test_mul(input_value):
    assert Calculation().sum(input_value, 1) == 25
```

Результатом запуска данного кода будет следующий вывод *pytest*:

```
test/test_new_calculation.py::test_sum PASSED [ 33%]
test/test_new_calculation.py::test_sub FAILED [ 66%]
test/test_new_calculation.py::test_mul FAILED [100%]
```

Однако объявленная таким образом фикстура будет доступна только в текущем файле с тестами. Чтобы обойти данное ограничение, создайте в тестовой директории файл с именем *conftest.py* и поместите в него необходимые для тестирования фикстуры.

Сначала *pytest* ищет встречающуюся фикстуру в файле с тестом. Если ее нет, поиск осуществляется в файле *conftest.py*. Такой подход позволяет использовать одну и ту же фикстуру в различных тестовых файлах.

В качестве примера переместим написанную ранее фикстуру в файл *conftest.py* и снова запустим тест:

```
(venv) F:\code\python\module-11>pytest test/test_new_
calculation.py -v
test/test_new_calculation.py::test_sum PASSED [ 33%]
test/test_new_calculation.py::test_sub FAILED [ 66%]
test/test_new_calculation.py::test_mul FAILED[100%]
```

Для тестов, которые работают с базой данных (или тестов, создающих некоторые файлы и удаляющих их по завершении теста), можно использовать следующий формат фикстур:

```
import pytest

@pytest.fixture()
def database():
    db = # установка соединения
    yield db
    db.close()
```

Так как в этом случае используется генераторная функция, то закрытие соединения с базой данных произойдет только после завершения теста. Но если у нас не один, а некоторое количество тестов, требующих доступ к базе данных? Тогда использование приведенной выше фикстуры приведет к неоправданным вычислительным затратам в процессе тестирования.

Чтобы избежать такого сценария, можно через именованный аргумент фикстуры *scope* указать ее область видимости. Например, при передаче в фикстуру *scope* = «*module*» она будет инициализирована единожды для всего модуля, что сделает открытое соединение доступным всем тестовым функциям, где оно требуется:

```
@pytest.fixture(scope="module")
def database():
    db = # установка соединения
    yield db
    db.close()
```

Фикстуры могут настраиваться различными способами и поддерживают механизм вложения, позволяющий одним фикстурам включать в свой состав другие. Также имеется возможность параметризации тестов и фикстур, позволяющая запустить тест за один прогон несколько раз с рядом различных параметров:

```
@pytest.fixture(params = range(10, 41, 10))
def value():
    return params

@pytest.fixture()
def scale_value():
    return = 3

@pytest.fixture()
def new_value(scale_value, listbase):
    return scale_value* listbase

def test_value_in_list(new_value):
    # какая-то работа с объектом и
    # формирование у него атрибута списка
    atr_list = [30, 60, 90, 119]
    assert testedlist in atr_list
```

11.1.5. Параллельный запуск тестов

По умолчанию *pytest* запускает тесты последовательно, что сказывается на времени тестирования крупных проектов. Ускорить этот процесс можно путем распараллеливания тестов. Для этого установите плагин *pytest-xdist* (*pip install pytest-xdist*), который расширяет командную строку *pytest* аргументом *--numprocesses* (сокращенно -n), принимающим в качестве аргумента количество используемых ядер:

```
pytest -n 2
```

Поскольку на различных компьютерах количество ядер будет различаться, у плагина *pytest-xdist* предусмотрено ключевое слово *auto* в качестве значения ключа *–n*, которое возвращает доступное количество ядер процессора:

```
pytest -n auto
```

11.2. Тестирование с использованием unittest

11.2.1. Пример простого теста

Вместо *pytest* можно использовать уже встроенный в Python модуль тестирования — *unittest* [56https://docs.python.org/3/library/unittest.html]. Для примера напишем файл *unittest_calculation.py* с тестами методов класса **Calculation** из *calculation.py*:

```
import unittest
from src.calculation import Calculation

class TestCalculation(unittest.TestCase):
    def setUp(self) -> None:
        #Метод запускается перед каждым тестом
        self.calculator = Calculation()

    def tearDown(self) -> None:
        #Метод запускается после каждого теста
        ...

    def test_sum(self):
        self.assertEqual(self.calculator.sum(5,
                                             3), 8)

    def test_sub(self):
        self.assertEqual(self.calculator.sub(10,
                                             3), 6)

    def test_div(self):
        self.assertEqual(self.calculator.div(10,
                                             3), 3)
```

```
    def test_mul(self):
        self.assertEqual(self.calculator.mul(5,
                                             3), 14)

    def test_pow(self):
        self.assertEqual(self.calculator.pow2(5),
                                               25)

if __name__ == "main":
    unittest.main()
```

При запуске данного файла с тестами *unittest* сообщит о несоответствии ожидаемых данных в процессе тестирования и тех, которые возвращают тестируемые методы класса **Calculation**. К таким тестам относятся: *test_mul* и *test_sub*. Эти ошибки связаны с тем, что в тесте прописаны некорректные ожидаемые данные.

11.2.2. Использование объектов-подделок

Представим, что приведенный в предыдущем разделе тест выполняется порядка нескольких минут. А если такого кода очень много? Тогда тестирование проекта может занимать значительное количество времени, а при внесении небольших изменений запуск всех тестов проекта превратится в «ад».

Для борьбы с этой проблемой используются объекты-подделки (*mock*), которые позволяют при тестировании не вызывать каждый раз метод тестируемого объекта, API веб-сервиса и т. д., а позволяют «подделать» реализацию части вашего приложения в процессе тестирования.

Для примера перепишем приведенный выше тест таким образом, чтобы не вызывать каждый раз методы тестируемого класса при выполнении теста:

```
# unittest_calculation_mock.py
import unittest
from unittest.mock import patch

class TestCalculation(unittest.TestCase):

    @patch('src.calculation.Calculation.sum',
           return_value=9)
    def test_sum(self, sum): # тест не пройдет
        self.assertEqual(sum(5, 3), 8)

    @patch('src.calculation.Calculation.sub',
           return_value=7)
    def test_sub(self, sub):
        self.assertEqual(sub(10, 3), 7)

    @patch('src.calculation.Calculation.div',
           return_value=3)
    def test_div(self, div):
        self.assertEqual(div(10, 3), 3)
```

```
    @patch('src.calculation.Calculation.mul',
           return_value=15)
    def test_mul(self, mul):
        self.assertEqual(mul(5, 3), 15)

    @patch('src.calculation.Calculation.pow2',
           return_value=25)
    def test_pow(self, pow2):
        self.assertEqual(pow2(5), 25)

if __name__ == "main":
    unittest.main()
```

Резюме

В данной теме рассмотрены способы и инструменты для тестирования разрабатываемых приложений на языке программирования Python. Конечно, была затронута лишь вершина айсберга, но и ее достаточно, чтобы увидеть те преимущества, которые тестирование привносит в разрабатываемый проект.

При этом можно спокойно комбинировать *pytest* и *unittest*, тем самым разрабатывая более обширное тестовое окружение к проекту и значительно сокращая время для поиска и отладки «бага» программистом.

Более подробно с возможностями рассматриваемого в разделе инструментария для тестирования можно ознакомиться в [57].

Вопросы и задания для самопроверки

1. Для чего используется тестирование?
2. Как лучше хранить код тестового окружения проекта?
3. Как запустить тесты посредством pytest?
4. Что такое маркер в рамках написания тестов на pytest?
5. Для чего используются фикстуры?
6. Как вывести более подробный отчет тестирования в pytest?
7. Нужен ли параллельный запуск тестов и как его организовать в pytest?
8. Приведите пример простого теста с использованием pytest и unittest.
9. Как посредством pytest запустить только те тесты, в названии которых встречается задаваемая подстрока?
10. Что такое объекты-подделки (пустышки/mock) для чего они используются?

Упражнения

Напишите тесты к программам и классам, реализованным в качестве упражнений в темах 5, 8, 9 и 10.

Список используемых источников

1. PEP 373 -- Python 2.7 Release Schedule. — URL: https://legacy.python.org/dev/peps/pep-0373/ (дата обращения: 20.03.2020 г.).
2. TIOBE Programming Community Index Definition. — URL: https://www.tiobe.com/tiobe-index/programming-languages-definition/ (дата обращения: 20.03.2020 г.).
3. TIOBE Index for April 2020. — URL: https://www.tiobe.com/tiobe-index/ (дата обращения: 20.03.2020 г.).
4. The RedMonk Programming Language Rankings: January 2020. — URL: https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/ (дата обращения: 20.03.2020 г.).
5. Почему Python развивается так быстро: исследование Stack Overflow. — URL: http://amazinghiring.ru/blog/2017/10/05/почему-python-развивается-так-быстро-иссле/ (дата обращения: 21.03.2020 г.).
6. *Лутц, Марк.* Изучаем Python. Т. 1 : Перевод с английского / Марк Лутц. — 5-е изд. — Санкт-Петербург : Диалектика, 2019. — 832 с.
7. *Бейдер, Д.* Чистый Python. Тонкости программирования для профи / Д. Бейдер. — Санкт-Петербург : Питер, 2018. — 288 с.
8. Pointers in Python: What's the Point? — URL: https://realpython.com/pointers-in-python/ (дата обращения: 23.03.2020 г.).
9. What is the Python Global Interpreter Lock (GIL)? — URL: https://realpython.com/python-gil/ (дата обращения: 28.03.2020 г.).
10. PEP 554 — Multiple Interpreters in the Stdlib. — URL: https://www.python.org/dev/peps/pep-0554/ (дата обращения: 2.04.2020 г.).
11. Memory Management in Python. — URL: https://realpython.com/python-memory-management/ (дата обращения: 3.04.2020 г.).
12. Все, что нужно знать о сборщике мусора в Python. — URL: https://habr.com/ru/post/417215/ (дата обращения: 9.04.2020 г.).
13. Basics of Memory Management in Python. — URL: https://stackabuse.com/basics-of-memory-management-in-python/ (дата обращения: 9.04.2020 г.).
14. *Рамальо, Л.* Python. К вершинам мастерства / Л. Рамальо ; Перевод с английского Слинкина А. А. — Москва : ДМК Пресс, 2016. — 768 с.
15. Python. Урок 3. Типы и модель данных. — URL: https://devpractice.ru/python-lesson-3-data-model/ (дата обращения: 14.04.2020 г.).

16. MyPy. — URL: http://mypy-lang.org/index.html (дата обращения: 14.04.2020 г.)

17. Built-in Types. — URL: https://docs.python.org/3/library/stdtypes.html (дата обращения: 14.04.2020 г.).

18. *Харрисон, М.* Как устроен Python. Гид для разработчиков, программистов и интересующихся / Харрисон, М. — Санкт-Петербург : Питер, 2019. — 272 с.

19. Файлы в python, ввод-вывод. — URL: https://pythonru.com/osnovy/fajly-v-python-vvod-vyvod (дата обращения: 26.04.2020 г.).

20. pickle — Python object serialization. — URL: https://docs.python.org/3/library/pickle.html (дата обращения: 27.04.2020 г.).

21. Введение в JSON. — URL: https://www.json.org/json-ru.html (дата обращения: 29.04.2020 г.).

22. json — JSON encoder and decoder. — URL: https://docs.python.org/3/library/json.html#module-json (дата обращения: 27.04.2020 г.).

23. Data Structures. — URL: https://docs.python.org/3/tutorial/datastructures.html (дата обращения: 05.05.2020 г.).

24. Множества в Python. — URL: https://foxford.ru/wiki/informatika/mnozhestva-v-python (дата обращения: 05.05.2020 г.).

25. PEP8 — стиль кода в языке Python. — URL: https://pep8.ru/doc/pep8/ (дата обращения: 07.05.2020 г.).

26. PEP 0 -- Index of Python Enhancement Proposals (PEPs). — URL: https://www.python.org/dev/peps/ (дата обращения: 07.05.2020 г.).

27. PEP 7 -- Style Guide for C Code. — URL: https://www.python.org/dev/ peps/pep-0007/ (дата обращения: 07.05.2020 г.).

28. PEP 20 -- The Zen of Python. — URL: https://www.python.org/dev/peps/pep-0020/ (дата обращения: 07.05.2020 г.).

29. PEP 257 -- Docstring Conventions. — URL: https://www.python.org/dev/ peps/pep-0257/ (дата обращения: 08.05.2020 г.).

30. PEP 3131 -- Supporting Non-ASCII Identifiers. — URL: https://www.python.org/dev/peps/pep-3131/ (дата обращения:08.05.2020 г.).

31. PEP 484 -- Type Hints. — URL: https://www.python.org/dev/peps/pep-0484/ (дата обращения: 09.05.2020 г.).

32. PEP 207 -- Rich Comparisons. — URL: https://www.python.org/dev/peps/pep-0207/ (дата обращения: 10.05.2020 г.).

33. PEP 3151 -- Reworking the OS and IO exception hierarchy. — URL: https://www.python.org/dev/peps/pep-3151/(дата обращения:10.05.2020 г.).

34. PEP 526 -- Syntax for Variable Annotations. — URL: https://www.python.org/dev/peps/pep-0526/ (дата обращения: 15.05.2020 г.).

35. Итераторы и генераторы в Python. — URL: https://shepetko.com/ru/blog/python-iterable-iterators-generators (дата обращения: 01.06.2020 г.).

36. Когда использовать List Comprehension в Python. — URL: https://webdevblog.ru/kogda-ispolzovat-list-comprehension-v-python/ (дата обращения: 03.06.2020 г.).

37. Are list-comprehensions and functional functions faster than “for loops”? — URL: https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops/22108640 #22108640 (дата обращения: 07.06.2020 г.).

38. Python's documentation, tutorials, and guides are constantly evolving. — URL: https://www.python.org/doc/ (дата обращения: 09.06.2020 г.).

39. *Лутц, Марк.* Изучаем Python. Т. 5 : Перевод с английского / Марк Лутц. — 5-е изд. — Санкт-Петербург : Диалектика, 2019. —720 с.

40. enum — Support for enumerations. — URL: https://docs.python.org/3/library/enum.html(дата обращения: 23.06.2020 г.).

41. *Мартелли, А.* Python. Справочник. Полное описание языка : Перевод с английского / А. Мартелли, А. Рейвенскрофт, С. Холден. —3-е изд. — Санкт-Петербург : Диалектика, 2019. — 896 с.

42. *Хеллман, Д.* Стандартная библиотека Python 3: справочник с примерами : Перевод с английского / Д. Хеллман. — 2-е изд. — Санкт-Петербург : Диалектика, 2019. — 1376 с.

43. Qt. — URL: https://www.qt.io (дата обращения: 01.08.2020 г.).

44. The Python Package Index (PyPI) is a repository of software for the Python programming language. URL: https://pypi.org (дата обращения: 03.08.2020 г.)

45. *Прохоренок, Н. А.* Python 3 и PyQt 5. Разработка приложений. / Н. А. Прохоренок, В. А. Дронов. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2018. — 832 с.: ил. — (Профессиональное программирование)

46. Joshua M. Willman Beginning PyQt: A Hands-on Approach to GUI Programming / Apress, 2020. - p. 449, DOI https://doi.org/10.1007/978-1-4842-5857-6_1

47. Socket — Low-level networking interface. — URL: https://docs.python.org/3/library/socket.html (дата обращения: 07.08.2020 г.)

48. ZeroMQ. An open-source universal messaging library. URL: https://zeromq.org (дата обращения: 09.04.2021 г.).

49. ØMQ - The Guide. — URL: http://zguide.zeromq.org (дата обращения: 10.08.2020 г.).

50. Flask. — URL: https://flask.palletsprojects.com/en/1.1.x/ (дата обращения: 11.08.2020 г.).

51. Django. The web framework for perfectionists with deadlines. — URL: https://www.djangoproject.com (дата обращения: 11.08.2020 г.).

52. FastAPI framework, high performance, easy to learn, fast to code, ready for production. — URL: https://fastapi.tiangolo.com/ (дата обращения: 11.08.2020 г.).

53. SQLite. — URL: https://www.sqlite.org/index.html (дата обращения: 15.08.2020 г.).

54. SQLite Tutorial. — URL: https://www.sqlitetutorial.net (дата обращения: 15.08.2020 г.).

55. Full pytest documentation. — URL: https://docs.pytest.org/en/stable/contents.html (дата обращения: 20.08.2020 г.).

56. unittest — Unit testing framework. URL: https://docs.python.org/3/ library/unittest.html (дата обращения: 23.08.2020 г.).

57. Okken, Brian. Python Testing with pytest / Brian Okken. — The Pragmatic Programmers, LLC, 2017, p. 240.

Новинки по дисциплине «Программирование на языке Python» и смежным дисциплинам

1. Гниденко, И. Г. Технологии и методы программирования : учебное пособие для вузов / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. — Москва : Издательство Юрайт, 2021. — 235 с. — (Высшее образование).

2. Илюшечкин, В. М. Основы использования и проектирования баз данных : учебник для вузов / В. М. Илюшечкин. — Москва : Издательство Юрайт, 2020. — 213 с.

3. Сети и телекоммуникации : учебник и практикум для вузов / К. Е. Самуйлов [и др.] ; под редакцией К. Е. Самуйлова, И. А. Шалимова, Д. С. Кулябова. — Москва : Издательство Юрайт, 2020. — 363 с.

4. Федоров, Д. Ю. Программирование на языке высокого уровня Python : учебное пособие для вузов / Д. Ю. Федоров. — 2-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2021. — 161 с. — (Высшее образование).