Джейми Чан

Python

Быстрый старт >>

Jamie Chan

Learn Python in One Day

and Learn It Well

Python for Beginners with Hands-on Project

Джейми Чан

Python

Быстрый старт >>

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

ББК 32.988.02-018
УДК 004.738.5
Ч-18

Чан Джейми

Ч-18 Python: быстрый старт. — СПб.: Питер, 2021. — 224 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1800-7

Всегда хотели научиться программировать на Python, но не знаете, с чего начать? Или хотите быстро перейти с другого языка на Python? Уже перепробовали множество книг и курсов, но ничего не подходит? Серия «Быстрый старт» — отличное решение, и вот почему: сложные понятия разбиты на простые шаги — вы сможете освоить язык Python, даже если никогда раньше не занимались программированием; все фундаментальные концепции подкреплены реальными примерами; вы получите полное представление о Python: структуры управления, методы обработки ошибок, концепции объектно-ориентированного программирования и т. д.; в конце книги вас ждет интересный проект, который поможет усвоить полученные знания. Ну что, готовы? Погнали!

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

ISBN 978-1546488330 англ.
ISBN 978-5-4461-1800-7

КРАТКОЕ СОДЕРЖАНИЕ

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Эта книга написана, чтобы помочь вам БЫСТРО изучить Python — и изучить ХОРОШО. Она не требует от читателя опыта программирования. Даже стопроцентный новичок обнаружит, что в этой книге просто объясняются сложные концепции. Если вы — опытный разработчик, переходящий на Python, материал обладает достаточной глубиной, чтобы вы могли немедленно взяться за программирование.

Чтобы дать вам широкое представление о Python, не перегружая лишней информацией, я тщательно подошел к выбору тем. В частности, рассматриваются структуры управления, методы обработки ошибок и методы обработки файлов, а также многое другое. В книгу включены главы по объектно-ориентированному программированию.

Для наглядного представления каждой концепции есть примеры, которые позволят понять язык на более глубоком уровне. Приложения в конце книги можно использовать в качестве удобного справочника по некоторым из часто используемых функций в Python.

Как сказал Ричард Брэнсон, «самый лучший способ чему-либо научиться — начать это делать». В конце курса вы познакомитесь с проектом, который позволит применить на практике полученные знания.

Исходный код проекта и приложения можно загрузить по адресу: https://www.learncodingfast.com/python.

Список исправлений можно найти на https://www.learncodingfast.com/errata.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения, вопросы отправляйте по адресу: comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

В Python отступы являются частью синтаксиса, и согласно PEP8 предпочтительным способом переноса длинных строк является использование подразумеваемых продолжений строк Python внутри круглых, квадратных

и фигурных скобок. Длинные строки могут быть разбиты на несколько строк, обернутые в скобки. Это предпочтительнее использования обратной косой черты для продолжения строки.

В связи с ограничением полосы набора и в целях удобочитаемости обратный слеш добавлен лишь в некоторых местах, в остальной части книги используются круглые скобки.

Если при наборе кода у вас возникает ошибка, то сверьтесь с исходным кодом по адресу: https://www.learncodingfast.com/python.

1

ЗНАКОМСТВО С PYTHON

Добро пожаловать в захватывающий мир программирования! Очень рад, что вы приобрели эту книгу, и искренне надеюсь, что она поможет вам овладеть языком Python и испытать радость от написания кода. Прежде чем мы углубимся в основы программирования на Python, давайте сначала разберем несколько тем.

1.1. ЧТО ТАКОЕ PYTHON?

Python — широко используемый высокоуровневый язык программирования, созданный Гвидо ван Россумом в конце 1980-х годов. В языке делается упор на удобочитаемость и простоту кода, что позволяет программистам разрабатывать приложения быстро и легко.

Как и все языки высокого уровня, код Python напоминает естественный (английский) язык, который не могут понять компьютеры. Написанный нами код должен интерпретироваться специальной программой — интерпретатором Python, которую нужно установить, прежде чем мы сможем разрабатывать, тестировать и запускать программы. Установку интерпретатора Python рассмотрим в главе 2.

Также существует ряд сторонних инструментов, например Py2exe или Pyinstaller, которые позволяют упаковывать код на Python в автономные исполняемые программы для некоторых из самых популярных операционных систем — Windows и Mac OS. Это дает возможность распространять программы Python, не требуя от пользователей установки интерпретатора Python.

1.2. ЗАЧЕМ ИЗУЧАТЬ PYTHON?

Есть огромное количество языков программирования высокого уровня, например C, C ++ и Java. Хорошая новость заключается в том, что все высокоуровневые языки очень похожи друг на друга. Они различаются главным образом синтаксисом, доступными библиотеками и способом доступа к ним. Библиотека — это набор ресурсов из заранее написанного кода, которые можно использовать при написании собственных программ. Если вы хорошо выучите один язык, то легко сможете выучить новый за короткое время.

Если вы новичок в программировании, то Python — отличный вариант для старта. Одна из ключевых особенностей Python — простота, что делает его идеальным языком для изучения начинающими. Большинство программ на Python требует значительно меньше строк кода по сравнению с другими языками, например C. Таким образом, ошибок становится меньше и сокращается время разработки. Кроме того, Python поставляется с обширным набором сторонних ресурсов, расширяющих возможности

языка. Исходя из этого, Python можно использовать для множества задач, например для настольных приложений, приложений баз данных, сетевого программирования, программирования игр и даже мобильной разработки. И последнее, но не менее важное: Python — кроссплатформенный язык, а это означает, что код, написанный, например, для Windows, будет хорошо работать в Mac OS или Linux без каких-либо изменений в коде Python.

Я убедил вас, что Python — это именно тот язык, который нужно изучать? Поехали!

2

ПОДГОТОВКА К РАБОТЕ НА PYTHON

2.1. УСТАНОВКА ИНТЕРПРЕТАТОРА

Прежде чем мы напишем первую программу на Python, необходимо загрузить соответствующий интерпретатор для компьютера.

В этой книге используется Python 3, потому что, как сказано на официальном сайте Python, «Python 2 уходит в прошлое, а Python 3 — это настоящее и будущее языка». Кроме того, «Python 3 лишен странностей, которые могут сбивать с толку начинающих программистов».

Обратите внимание, что Python 2 все еще довольно широко используется. Python 2 и Python 3 схожи примерно на 90 %. Следовательно, если вы изучите Python 3, то, скорее всего, проблем с пониманием кода, написанного на Python 2, не возникнет.

Чтобы установить интерпретатор для Python 3, перейдите на https://www.python.org/downloads/. Актуальная версия должна быть указана вверху страницы. В этой книге мы используем версию 3.6.1. Нажмите «Загрузить Python 3.6.1», и начнется скачивание.

Если вы хотите установить другую версию, прокрутите страницу вниз — и увидите список доступных версий. Щелкните по нужной версии, и вас перенаправят на страницу загрузки.

Прокрутите страницу вниз до конца — и увидите таблицу со списком различных установщиков для этой версии. Выберите правильный установщик для своего компьютера. Выбор установщика зависит от двух факторов:

1) операционная система (Windows, Mac OS или Linux);

2) процессор (32-битный или 64-битный).

Например, если вы используете 64-битный компьютер с Windows, то, скорее всего, будете использовать «установщик исполняемого файла Windows x86-64». Перейдите по ссылке, чтобы скачать его. Если вы скачаете и запустите не тот установщик, не волнуйтесь: вы получите сообщение об ошибке и интерпретатор не будет установлен. Скачайте подходящий вам установщик.

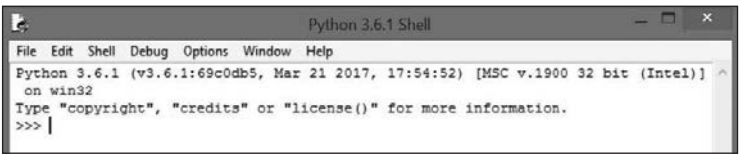
После успешной установки интерпретатора можно приступать к написанию кода на Python.

2.2. ОБОЛОЧКА PYTHON, IDLE И НАПИСАНИЕ ПЕРВОЙ ПРОГРАММЫ

Код будем писать в программе IDLE, которая поставляется вместе с интерпретатором Python.

Сначала запустим программу IDLE, как и любые другие программы. Например, в Windows 10 можно найти ее,

набрав «IDLE» в поле поиска. Затем щелкните IDLE (графический интерфейс Python), чтобы запустить ее. Появится оболочка Python (Python Shell).



Оболочка Python позволяет использовать Python в интерактивном режиме. Это означает, что можно вводить по одной команде за раз. Оболочка ожидает команды от пользователя, выполняет ее и возвращает результат выполнения. После этого ждет следующей команды.

Попробуйте ввести в оболочку следующий код. Строки, начинающиеся с `>>>`, — это команды, которые вы должны ввести, а строки после команд — это результаты:

```
>>> 2 + 3
5
>>> 3 > 2
True
>>> print ('Hello World')
Hello World
```

Когда вы вводите `2 + 3`, то даете команду вычислить значение 2 + 3. Следовательно, оболочка возвращает ответ `5`. Когда вы вводите `3 > 2`, то спрашиваете оболочку, больше ли 3, чем 2. Оболочка отвечает `True`. И наконец, команда `print` просит оболочку отобразить строку `Hello World`.

Оболочка Python — очень удобный инструмент для тестирования команд, особенно на первых этапах работы. Но если вы выйдете из оболочки и войдете в нее снова, все вводимые команды исчезнут. Кроме того, оболочку нельзя использовать для создания реальной программы. Чтобы создать настоящую программу, нужно написать код в текстовом файле и сохранить его с расширением .py. Такой файл называется сценарием Python.

Чтобы создать сценарий Python, щелкните File ▸ New File в верхнем меню оболочки Python. Появится текстовый редактор, который мы будем использовать для написания самой первой программы «Hello World». Написание программы «Hello World» — это своего рода обряд посвящения для всех начинающих программистов. С помощью этой программы мы познакомимся с программным обеспечением IDLE.

Введите следующий код в текстовый редактор (не в оболочку):

```
#Выводит слова "Hello World"
print ("Hello World")
```

Обратите внимание, что строка `#Выводит слова "Hello World"` выделена красным, слово `print` — фиолетовым, а `Hello World` — зеленым. Таким образом программа облегчает нам чтение кода. Слова `print` и `Hello World` служат разным целям в программе и поэтому отображаются разными цветами. Об этом мы поговорим подробнее в следующих главах.

Строка `#Выводит слова "Hello World"` (красная) не является частью программы. Это комментарий, который написан, чтобы сделать код понятным для других программистов. Такая строка игнорируется интерпретатором Python. Чтобы добавить комментарии, перед каждой строкой нужно написать символ `#`, например:

```
# Это комментарий
# Это тоже комментарий
# Это еще один комментарий
```

Мы также можем использовать три одинарные кавычки (или три двойные кавычки) для многострочных комментариев, например:

```
'''
Это комментарий
Это тоже комментарий
Это еще один комментарий
'''
```

Теперь нажмите File ▸ Save As..., чтобы сохранить код. Убедитесь, что выбрали расширение .py.

Готово? Вуаля! Вы только что успешно написали свою первую программу на Python.

Выберете Run ▸ Run Module, чтобы запустить программу (или нажмите F5). Вы увидите слова `Hello World` в оболочке Python.

3

ПЕРЕМЕННЫЕ И ОПЕРАТОРЫ

Итак, мы закончили с вводным материалом, давайте приступим к реальным задачам. В этой главе мы изучим переменные и операторы. Вы узнаете, что такое переменные, как их называть и объявлять, а также об общих операциях, которые можно с ними выполнять. Готовы?

Поехали!

3.1. ЧТО ТАКОЕ ПЕРЕМЕННЫЕ?

Переменные — это именованные объекты данных, которые хранятся и используются в наших программах. Допустим, ваша программа должна хранить возраст пользователя. Для этого мы можем назвать эти данные `userAge` и присвоить значение переменной `userAge`, используя следующую инструкцию:

```
userAge = 0
```

После того как вы определите переменную `userAge`, программа выделит некоторую область дискового пространства компьютера для хранения этих данных. Затем можно получить доступ к этим данным и изменить их, сославшись на них по имени `userAge`. Каждый раз, когда вы объ-

являете новую переменную, нужно присвоить ей начальное значение. В этом примере мы присвоили ей значение `0`. Значение всегда можно изменить позднее.

Также можно определить несколько переменных за раз. Для этого напишите

```
userAge, userName = 30, 'Peter'
```

Это будет равнозначно

```
userAge = 30
userName = 'Peter'
```

3.2. ИМЕНОВАНИЕ ПЕРЕМЕННЫХ

Имя переменной в Python может содержать только буквы (a–z, A–B), числа или символы подчеркивания (_). Первый символ не может быть числом. Таким образом, можно называть свои переменные `userName`, `user_name` или `userName2`, но не `2userName`.

Кроме того, есть некоторые зарезервированные слова, которые нельзя использовать в качестве имени переменной, так как они уже имеют заранее заданные значения в Python. Это `print`, `input`, `if`, `while` и т. д. Подробнее о них поговорим в следующих главах.

Наконец, имена переменных чувствительны к регистру. `username` не то же самое, что `userName`.

При именовании переменных в Python существуют два соглашения. Можно придерживаться верблюжьего ре-

гистра или использовать символы подчеркивания. Верблюжий регистр — это способ написания составных слов в смешанном регистре (например, `thisIsAVariableName`). Мы будем использовать верблюжий регистр в оставшейся части книги. Другой распространенной практикой является использование символа подчеркивания (_) для разделения слов. Таким образом, переменные можно назвать так: `this_is_a_variable_name`.

3.3. ОПЕРАТОР ПРИСВАИВАНИЯ

Обратите внимание: символ = в инструкции `userAge = 0` имеет другое значение, чем математический символ «равно». В программировании = называется оператором присваивания. Это означает, что справа от = присваивается значение переменной слева. Чтобы было легче понять инструкцию `userAge = 0` , можно думать о ней как о `userAge <- 0`.

В программировании выражения `x = y` и `y = x` означают совершенное разные вещи.

Удивлены? Приведу пример.

Напишите следующий код в редакторе IDLE и сохраните его.

```
x = 5
y = 10
x = y
print ("x = ", x)
print ("y = ", y)
```

Запустите программу. Вы увидите такой результат:

```
x = 10
y = 10
```

Хотя `x` имеет начальное значение `5` (как было объявлено в первой строке), третья строка `x = y` присваивает значение `y` переменной `x` (вспоминаем: `x <- y`), следовательно, значение `x` меняется на `10`, в то время как значение `y` остается неизменным.

Модифицируем программу, изменив ТОЛЬКО ОДИН оператор: поменяйте третью строку с `x = y` на `y = x`. С точки зрения математики `x = y` и `y = x` — это одно и то же. Однако в программировании все не так.

Запустите программу. В результате получится:

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

Здесь значение `x` остается равным `5`, а вот значение `y` изменяется на `5`. Оператор `y = x` присваивает значение `x` значению `y`.

Значение `y` становится равным `5`, а `x` остается неизменным.

3.4. ОСНОВНЫЕ ОПЕРАТОРЫ

Помимо присвоения переменной начального значения можно выполнять обычные математические операции

с переменными. Основные операторы в Python включают в себя сложение (+), вычитание (-), умножение (*), деление (/), целочисленное деление (//), остаток от деления (%) и возведение в степень (**).

Примеры:

Предположим, `x = 5, y = 2`

Сложение:

```
x + y = 7
```

Вычитание:

```
x - y = 3
```

Умножение:

```
x*y = 10
```

Деление:

```
x/y = 2.5
```

Целочисленное деление:

```
x//y = 2 (округляет результат до ближайшего целого
числа)
```

Остаток от деления:

```
x%y = 1 (дает остаток от деления 5 на 2)
```

Возведение в степень:

```
x**y = 25 (5 в степени 2)
```

3.5. ДОПОЛНИТЕЛЬНЫЕ ОПЕРАТОРЫ ПРИСВАИВАНИЯ

Помимо основного оператора = в Python (как и в большинстве языков программирования) есть еще несколько дополнительных операторов присваивания. К ним относятся такие операторы, как + =, - = и * =.

Предположим, есть переменная x с начальным значением 10. Если требуется увеличить x на 2, можно написать:

```
x = x + 2
```

Программа *сначала оценит выражение справа* (x + 2) и назначит ответ слева. Таким образом, в итоге приведенное выше выражение становится x = 12.

Вместо того чтобы писать x = x + 2, можно написать x + = 2, чтобы выразить то же значение. Обозначение + = на самом деле является сокращением, которое объединяет знак присваивания с оператором сложения. Следовательно, x + = 2 означает x = x + 2.

Точно так же, если требуется вычитание, можно написать x = x - 2 или x - = 2. Это работает для всех 7 операторов, упомянутых в разделе выше.

4

ТИПЫ ДАННЫХ В PYTHON

Перейдем к типам данных в Python. Тип данных относится к значению, которое хранит переменная.

Рассмотрим некоторые основные типы данных в Python, в частности целые числа, числа с плавающей точкой и строки. Далее поговорим о концепции приведения типов и об еще трех дополнительных типах данных в Python: списках, кортежах и словарях.

4.1. ЦЕЛЫЕ ЧИСЛА

Целые числа — это числа без десятичных частей, например –5, –4, –3, 0, 5, 7 и т. д.

Чтобы объявить целое число в Python, просто напишите
`variableName = начальное значение`

Пример:

```
userAge = 20
mobileNumber = 12398724
```

4.2. ВЕЩЕСТВЕННЫЕ ЧИСЛА (ЧИСЛА С ПЛАВАЮЩЕЙ ТОЧКОЙ)

Числа с плавающей точкой относятся к числам, имеющим десятичную точку (запятую в русской нотации. — *Примеч. ред.*), например 1.234, –0.023, 12.01.

Чтобы объявить число с плавающей точкой в Python, следует придерживаться такого шаблона: `variableName = начальное значение`

Пример:

```
userHeight = 1.82
userWeight = 67.2
```

4.3. СТРОКОВЫЙ ТИП ДАННЫХ

Строки относятся к тексту.

Чтобы объявить строку, можно использовать либо запись `variableName = 'начальное значение'` (одинарные кавычки), либо запись `variableName = "начальное значение"` (двойные кавычки).

Пример:

```
userName = 'Peter'
userSpouseName = "Janet"
userAge = '30'
```

Так как в последнем примере мы написали `userAge = '30'`, то `userAge` — это строка. Для сравнения: если вы

напишете `userAge = 30` (без кавычек), то `userAge` будет иметь целочисленное значение.

Мы можем объединить несколько подстрок, используя конкатенацию (`+`). Например, `"Peter " + "Lee"` эквивалентно строке `"Peter Lee"`.

ВСТРОЕННЫЕ ФУНКЦИИ СТРОК

Python включает ряд встроенных функций для управления строками. Функция — это блок многократно используемого кода, который выполняет определенную задачу. Более подробно о функциях мы поговорим в главе 7.

Примером функции, доступной в Python, является метод `upper()` для строк. Он делает все буквы в строке заглавными. Например, `'Peter'.upper()` вернет строку `'PETER'`. В приложении А есть дополнительные примеры и код, иллюстрирующие строковые методы.

ФОРМАТИРОВАНИЕ СТРОК С ПОМОЩЬЮ ОПЕРАТОРА %

Строки можно форматировать с помощью оператора `%`. Он дает вам больший контроль над формой отображения и сохранения строк. Синтаксис использования оператора `%` такой:

```
"форматируемая строка" %(значения или переменные
для вставки в строку, разделенные запятыми)
```

Он состоит из трех частей. Сначала в кавычках пишется строка, которую нужно отформатировать. Далее пишется символ `%`, а затем пара круглых скобок `()`, где записыва-

ются значения или переменные, которые нужно вставить в строку. Эти круглые скобки со значениями внутри называются кортежем. Это тип данных, который мы рассмотрим чуть позже в этой главе.

Введите следующий код в IDLE и запустите его:

```
brand = 'Apple'

exchangeRate = 1.235235245

message = 'The price of this %s laptop is %d USD'\
          'and the exchange rate is %4.2f USD to 1 EUR'\
           %(brand, 1299, exchangeRate)'

print (message)
```

В приведенном выше примере строка `'The price of this %s laptop is %d USD and the exchange rate is %4.2f USD to 1 EUR'` — это строка, которую нужно отформатировать.

`%s`, `%d` и `%4.2f` называются заполнителями.

Заполнители будут заменены на переменную `brand`, значение `1299` и переменную `exchangeRate` соответственно, как указано в скобках. При запуске кода получим следующий результат:

```
The price of this Apple laptop is 1299 USD and the
exchange rate is 1.24 USD to 1 EUR[1]
```

[1] Цена этого ноутбука Apple составляет 1299 долларов США, а обменный курс — 1,24 доллара США за 1 евро. — *Примеч. пер.*

Заполнитель `%s` используется для представления строки (в данном случае `'Apple'`), а заполнитель `%d` — для представления целого числа (1299). Если нужно добавить пробелы перед целым числом, то можно вставить число между `%` и `d`, чтобы указать желаемую длину строки. Например, `"%5d" %(123)` даст в результате `" 123"` (с двумя пробелами спереди и длиной 5).

Заполнитель `%f` используется для форматирования чисел с плавающей точкой (чисел с десятичными точками). Здесь мы форматируем его как `%4.2f`, где 4 обозначает общую длину, а 2 обозначает 2 десятичных знака. Если нужно добавить пробелы перед числом, можно отформатировать его как `%7.2f`, что даст `"  1.24"` (с 2 десятичными знаками, 3 пробелами спереди и длиной 7).

ФОРМАТИРОВАНИЕ СТРОК С ПОМОЩЬЮ МЕТОДА FORMAT()

Помимо использования оператора `%` для форматирования строк в Python также есть метод `format()` для тех же целей. Синтаксис такой:

```
"форматируемая строка".format(значения или переменные
для вставки в строку, разделенные запятыми)
```

При использовании метода `format()` заполнители `%s`, `%f` или `%d` не применяются. Вместо этого используются фигурные скобки `{}`, например:

```
message = 'The price of this {0:s} laptop is {1:d}
USD and the exchange rate is {2:4.2f} USD to 1 EUR'.
format('Apple', 1299, 1.235235245)
```

Внутри фигурных скобок мы пишем позицию используемого аргумента, а затем двоеточие. После двоеточия пишем заполнитель. В фигурных скобках не должно быть пробелов.

В выражении `format('Apple', 1299, 1.235235245)` методу `format()` передаются три аргумента. Аргументы — это данные, которые нужны методу для выполнения задачи. В данном случае аргументами являются: `'Apple'`, `1299` и `1.235235245`.

Аргумент `'Apple'` имеет позицию 0,

`1299` имеет позицию 1,

`1.235235245` имеет позицию 2.

Позиции всегда начинаются с НУЛЯ.

Выражение `{0: s}` дает команду интерпретатору заменить `{0: s}` аргументом в позиции 0 и указать, что это строка (поскольку заполнитель `s`).

Если мы вводим `{1: d}`, то имеем в виду аргумент в позиции 1, который является целым числом (заполнитель `d`).

Если мы вводим `{2: 4.2f}`, то ссылаемся на аргумент в позиции 2, который является числом с плавающей точкой. Необходимо, чтобы он был отформатирован двумя десятичными знаками и общей длиной 4 (заполнитель `4.2f`).

Если мы выведем сообщение, то получим

```
The price of this Apple laptop is 1299 USD and the
exchange rate is 1.24 USD to 1 EUR
```

Примечание

Если вы не хотите форматировать строку, то можете просто написать

```
message = 'The price of this {} laptop is {}
USD and the exchange rate is {} USD to 1 EUR'.
format('Apple', 1299, 1.235235245)
```

Здесь не нужно указывать позицию аргументов. Интерпретатор заменит фигурные скобки в соответствии с порядком предоставленных аргументов. Получаем

```
The price of this Apple laptop is 1299 USD and
the exchange rate is 1.235235245 USD to 1 EUR
```

Метод `format()` может запутать новичков. Форматирование строк может быть более замысловатым, чем то, что было рассмотрено здесь. Но для большинства целей достаточно того, что мы разобрали выше. Чтобы закрепить понимание метода `format()`, посмотрите на следующую программу:

```
message1 = '{0} is easier than {1}'.format('Python',
                                           'Java')
message2 = '{1} is easier than {0}'.format('Python',
                                           'Java')
message3 = '{:10.2f} and {:d}'.format(1.234234234, 12)
message4 = '{}'.format(1.234234234)

print (message1)
#Вы получите 'Python is easier than Java'

print (message2)
#Вы получите 'Java is easier than Python'

print (message3)
```

```
# Вы получите ' 1.23 and 12'
# Позиции аргументов указывать не нужно.

print (message4)
# Вы получите '1.234234234'. Форматирование
# не выполняется.
```

Поупражняйтесь с методом `format()` в оболочке Python. Пробуйте писать разные строки и смотрите, что получится.

4.4. ПРИВЕДЕНИЕ ТИПОВ В PYTHON

Иногда в программах нужно преобразовать один тип данных в другой, например целое число в строку. Такая операция называется приведением типа.

В Python есть три встроенные функции, которые позволяют выполнять приведение типов: `int()`, `float()` и `str()`.

Функция `int()` в Python принимает значение с плавающей точкой или соответствующей строки и преобразует ее в целое число. Чтобы заменить число с плавающей точкой на целое число, можно ввести `int(5.712987)`. В результате получится 5 (все, что до десятичной точки). Чтобы преобразовать строку в целое число, напишем `int("4")` и получим 4. Однако нельзя написать `int("Hello")` или `int("4.22321")`. В обоих случаях возникнет ошибка.

Функция `float()` принимает значение целого числа или соответствующей строки и меняет ее на число с плавающей точкой. Например, если написать `float(2)` или `float("2")`, то получим 2.0. Если написать `float("2.09109")`, то полу-

чим число 2.09109, которое является числом с плавающей точкой, а не строкой, поскольку кавычек здесь нет.

Функция `str()` преобразует целое число или число с плавающей точкой в строку. Например, если написать `str(2.1)`, получим `"2.1"`.

Теперь, когда мы рассмотрели три основных типа данных в Python и их преобразование, перейдем к более сложным типам данных.

4.5. СПИСКИ

Списки относятся к коллекции данных, которые обычно связаны. Чтобы не хранить эти данные как отдельные переменные, их можно хранить в виде списка. Например, вам в программе нужно сохранить возраст пяти пользователей. Чтобы не хранить их как `user1Age`, `user2Age`, `user3Age`, `user4Age` и `user5Age`, имеет смысл хранить их в виде списка.

Чтобы объявить список, напишите `listName = [начальные значения]`. Обратите внимание, что при объявлении списка используются квадратные скобки `[]`. Значения разделяются запятой.

Пример:

```
userAge = [21, 22, 23, 24, 25]
```

Можно объявить список без присвоения начальных значений. Просто укажите `listName = []`. Теперь есть пустой

список без элементов. Чтобы добавить элементы в список, используйте метод `append()`.

Значения в списке доступны по индексам, а индексы всегда начинаются с НУЛЯ, а не с единицы. Это обычная практика почти во всех языках программирования, например C и Java. Таким образом, индекс первого значения — 0, следующего — 1 и т. д. Например, `userAge [0] = 21`, `userAge [1] = 22`.

Можно обращаться к значениям списка в обратном порядке. Последний элемент в списке имеет индекс –1, предпоследний — индекс –2 и т. д. Следовательно, `userAge [-1] = 25`, `userAge [-2] = 24`.

Список или его часть можно назначить переменной. Если написать `userAge2 = userAge`, переменная `userAge2` примет значение `[21, 22, 23, 24, 25]`.

Если написать `userAge3 = userAge [2:4]`, это назначит элементы с индексом 2 для индекса 4–1 из списка `userAge` в список `userAge3`. Другими словами, `userAge3 = [23, 24]`.

Запись `2:4` называется срезом или слайсом. Всякий раз, когда мы используем срез в Python, включается элемент начального индекса, *но всегда исключается элемент в конце*. Следовательно, обозначение 2 : 4 относится к элементам с индексом от 2 до 4 – 1 (т. е. с индексом 3), поэтому `userAge3 = [23, 24]`, а не `[23, 24, 25]`.

Запись среза может включать третье число — шаг. Если написать `userAge4 = userAge [1:5:2]`, то получится подсписок, состоящий из каждого второго числа от индекса 1

до индекса 5 – 1, потому что шаг среза равен 2. Таким образом, `userAge4 = [22, 24]`.

Кроме того, записи срезов имеют полезные значения по умолчанию. Значение по умолчанию для первого числа — ноль, а для второго — размер списка, с которого снимается срез. Например, `userAge [ :4]` дает значения от 0 до 4 – 1, тогда как `userAge [1: ]` дает значения от 1 до 5 – 1 (поскольку размер `userAge` равен 5 и `userAge` имеет 5 элементов).

Чтобы изменить элементы в списке, нужно написать `listName [индекс элемента, который нужно изменить] = новое значение`. Например, чтобы изменить второй элемент, мы пишем `userAge [1] = 5`. Список станет `userAge = [21, 5, 23, 24, 25]`.

Чтобы добавить элементы, используется функция `append()`. Например, если написать `userAge.append (99)`, это добавит значение 99 в конец списка. Список станет `userAge = [21, 5, 23, 24, 25, 99]`.

Чтобы удалить элементы, используется синтаксис `del listName [индекс удаляемого элемента]`. Например, если написать `del userAge [2]`, список станет `userAge = [21, 5, 24, 25, 99]` (третий элемент будет удален).

Чтобы понять, как работают списки, запустите следующую программу:

```
# объявление списка, элементы списка могут
# представлять разные типы данных
myList = [1, 2, 3, 4, 5, "Hello"]
```

```
# выводит весь список
print(myList)
# Получаем [1, 2, 3, 4, 5, "Hello"]

# выводит третий элемент (напоминание: индексы
# начинаются с нуля).
print(myList[2])
# You'll get 3

# выводит последний элемент
print(myList[-1])
# получаем "Hello"

# назначает myList (с индекса 1 до 4) myList2
# и выводит myList2
myList2 = myList[1:5]
print (myList2)
# получаем  [2, 3, 4, 5]

# изменяет второй элемент в myList и выводит
# обновленный список
myList[1] = 20
print(myList)
# получаем [1, 20, 3, 4, 5, 'Hello']

# добавляет новый элемент в myList и выводит
# обновленный список
myList.append("How are you")
print(myList)
# получаем [1, 20, 3, 4, 5, 'Hello', ' How are you']

# удаляет шестой элемент из myList и выводит
# обновленный список
del myList[5]
print(myList)
# получаем [1, 20, 3, 4, 5, ' How are you']
```

Со списком можно сделать еще пару вещей. Код и примеры работы со списком см. в приложении B.

4.6. КОРТЕЖИ

Кортежи похожи на списки, но в них нельзя менять значения. Начальные значения остаются до конца выполнения программы. Примером использования кортежей является ситуация, когда программе необходимо хранить названия месяцев года.

Чтобы объявить кортеж, напишите `tupleName = (начальные значения)`. Обратите внимание, что при объявлении кортежа используются круглые скобки `()`. Значения разделяются запятой.

Пример:

```
monthsOfYear = ("Jan", "Feb", "Mar", "Apr", "May",
                "Jun", "Jul", "Aug", "Sep", "Oct",
                "Nov", "Dec")
```

Как и в случае со списком, вы получаете доступ к отдельным значениям кортежа по индексам:

```
monthsOfYear[0] = "Jan", monthsOfYear[-1] = "Dec"
```

Больше примеров использования кортежей см. в приложении B.

4.7. СЛОВАРИ

Словарь позволяет хранить связанные данные, которые являются ПАРАМИ. Например, если нужно сохранить имя пользователя и возраст пяти пользователей, их можно хранить в словаре.

Чтобы объявить словарь, укажите `dictionaryName = {ключ:значение}`. Важно, чтобы ключи словаря были уникальными (в пределах одного словаря). Такой словарь объявлять не следует:

```
myDictionary = {"Peter":38, "John":51, "Peter":13}.
```

Ключ `"Peter"` используется дважды. Обратите внимание, что при объявлении словаря используются фигурные скобки `{}`. Пары разделяются запятыми.

Пример:

```
userNameAndAge = {"Peter":38, "John":51, "Alex":13,
                  "Alvin":"Not Available"}
```

Словарь можно объявить с помощью метода `dict()`. Чтобы объявить словарь `userNameAndAge` из примера выше, нужно написать

```
userNameAndAge = dict(Peter = 38, John = 51, Alex =
                      13, Alvin = "Not Available")
```

При использовании этого метода вместо фигурных скобок ставятся круглые, а кавычки для ключей словаря не указываются.

Для доступа к отдельным элементам словаря используется ключ словаря, который является первым значением в паре `{ключ словаря:значение}`. Например, чтобы узнать возраст Джона, мы пишем `userNameAndAge ["John"]`. Результатом будет значение 51.

Чтобы изменить элемент в словаре, используется синтаксис `dictionaryName [ключ элемента, который нужно изменить] = новое значение`.

Например, чтобы изменить пару `"John":51`, нужно написать `userNameAndAge["John"] = 21`. Теперь словарь поменялся на

```
userNameAndAge = {"Peter":38, "John":21, "Alex":13,
"Alvin":"Not Available"}
```

Можно объявить словарь, не присваивая никаких начальных значений. Написав `dictionaryName = {}`, мы получаем пустой словарь, в котором нет никаких элементов.

Чтобы добавить элементы в словарь, нужно написать `dictionaryName [ключ] = значение`. Например, если требуется добавить `"Joe":40`, мы пишем `userNameAndAge["Joe"] = 40`. Теперь словарь принял такой вид:

```
userNameAndAge = {"Peter":38, "John":21, "Alex":13,
                  "Alvin":"Not Available", "Joe":40}
```

Чтобы удалить элементы из словаря, следует написать `del dictionaryName [ключ]`. Например, чтобы удалить пару `"Alex":13`, напишем `del userNameAndAge["Alex"]`. Словарь поменялся на

```
userNameAndAge = {"Peter":38, "John":21, "Alvin":"Not
                  Available", "Joe":40}
```

Запустите следующую программу и посмотрите на словари в действии.

```
# объявление словаря (ключи словаря и значения могут
# иметь разные типы данных)
myDict = {"One":1.35, 2.5:"Two Point Five", 3:"+",
          7.9:2}


# выводит весь словарь
print(myDict)
# получаем {One': 1.35, 2.5: 'Two Point Five', 3:
# '+', 7.9: 2} }
# Элементы могут выводиться неупорядоченно
# Элементы в словаре не обязательно хранятся в том же
# порядке, в каком их объявили.

#выводит значение с ключом = "One".
print(myDict["One"])
#получаем 1.35


# выводит значение с ключом = 7.9.
print(myDict[7.9])
#получаем 2


# изменяет элемент с ключом = 2.5 и выводит
# обновленный словарь
myDict[2.5] = "Two and a Half"
print(myDict)
# получаем {'One': 1.35, 2.5: 'Two and a Half', 3:
'+', 7.9: 2}


# добавляет новый элемент и выводит обновленный словарь
myDict["New item"] = "I’m new"
print(myDict)
# получаем {'One': 1.35, 2.5: 'Two and a Half', 3:
# '+', 7.9: 2, 'New item': 'I’m new'}
```

```
# удаляет элемент с ключом = " One" и выводит
# обновленный словарь
del myDict["One"]
print(myDict)
# получаем {2.5: 'Two and a Half', 3: '+', 7.9: 2,
# 'New item': 'I’m new'}
```

Дополнительные примеры работы со словарем см. в приложении Г.

5

ИНТЕРАКТИВНОСТЬ

Теперь, когда мы рассмотрели базовые понятия о переменных, давайте напишем программу, которая использует их. Вернемся к программе Hello World, которую мы написали в главе 2, но на этот раз сделаем ее интерактивной. Мы не просто поприветствуем мир, а сделаем так, чтобы мир знал наши имя и возраст. Для этого программа должна подсказывать информацию и выводить ее на экран.

За нас это сделают две встроенные функции: `input()` и `print()`.

А пока наберем в IDLE следующую программу. Сохраним и запустим ее:

```
myName = input("Please enter your name: ")
myAge = input("What about your age: ")

print ("Hello World, my name is", myName, "and I am",
       myAge, "years old.")
```

Программа должна запросить ваше имя:

```
Please enter your name:
```

Допустим, вы написали «Джеймс». Нажмите Enter, и программа предложит ввести свой возраст:

```
What about your age:
```

Допустим, вы ввели 20. Теперь снова нажмите Enter. В результате появится следующее:

```
Hello World, my name is James and I am 20 years old.
```

5.1. INPUT()

В приведенном выше примере мы дважды использовали функцию `input()`, чтобы получить имя и возраст пользователя:

```
myName = input("Please enter your name: ")
```

Строка `"Please enter your name: "` — это запрос, который будет отображаться на экране, чтобы дать пользователю дальнейшие инструкции. Здесь мы использовали простую строку в качестве подсказки. Если хотите, можете использовать заполнитель `%` или метод `format()` из главы 4 для форматирования входной строки. Позже мы рассмотрим два примера.

После отображения запроса функция ожидает, пока пользователь введет соответствующую информацию. Затем эта информация сохраняется **в виде строки** в переменной `myName`. Следующий оператор ввода запрашивает у пользователя его возраст и сохраняет информацию **в виде строки** в переменной `myAge`.

Так работает функция `input ()`. Не так уж сложно, правда?

Как упоминалось выше, помимо использования простой строки в качестве приглашения можно использовать заполнитель `%` или метод `format()` для отображения пригла-

шения. Например, можно изменить вторую инструкцию ввода выше с

```
myAge = input("What about your age: ")
```

на

```
myAge = input("Hi %s, what about your age: " %(myName))
```

или

```
myAge = input("Hi {}, what about your age:
              ".format(myName))
```

Итак, получаем

```
Hi James, what about your age:
```

в качестве подсказки.

Обратите внимание, что функция `input()`в Python 2 и Python 3 несколько различается. Если вы хотите принимать вводимые пользователем данные в виде строки, то в Python 2 нужно использовать функцию `raw_input()`. Функция `raw_input()` работает аналогично функции `input()` в Python 3.

5.2. PRINT()

Теперь перейдем к функции `print()`. Она используется для отображения информации пользователям. Принимает в качестве аргументов ноль или более выражений, разделенных запятыми.

В приведенной ниже инструкции мы передали 5 аргументов функции `print()`. Можете их идентифицировать?

```
print ("Hello World, my name is", myName, "and I am",
       myAge, "years old.")
```

Первый — строка `"Hello World, меня зовут"`.

Следующей идет переменная `myName`, объявленная ранее с помощью функции `input()`.

Далее идет строка `"and I am"`, за которой следует переменная `myAge`, а затем строка `"years old."`.

Обратите внимание, что мы не используем кавычки при обращении к переменным `myName` и `myAge`. Если вы напишете кавычки, то получится вот что:

```
Hello World, my name is myName and I am myAge years
old.
```

вместо того, что нужно.

Другой способ вывести инструкцию с переменными — использовать заполнитель `%` из главы 4. Чтобы добиться того же результата, что и в первом случае, напишем

```
print ("Hello World, my name is %s and I am %s years
       old." %(myName, myAge))
```

Наконец, чтобы вывести какую-либо инструкцию с помощью метода `format()`, нужно написать

```
print ("Hello World, my name is {} and I am {} years
       old".format(myName, myAge))
```

Функция `print()` различается в Python 2 и Python 3. В Python 2 ее нужно писать без скобок, например:

```
print "Hello World, my name is " + myName + " and I
am " + myAge + " years old."
```

5.3. ТРОЙНЫЕ КАВЫЧКИ

В некоторых случаях может потребоваться отобразить длинное сообщение с помощью функции `print()`. Для этого можно использовать символ тройной кавычки (`'''` или `"""`), чтобы растянуть сообщение на несколько строк. Например,

```
print ('''Hello World.
My name is James and
I am 20 years old.''')
```

даст нам

```
Hello World.
My name is James and
I am 20 years old.
```

Этот способ повышает удобочитаемость сообщения.

5.4. ЭКРАНИРОВАННЫЕ СИМВОЛЫ

Иногда требуется вывести некоторые специальные «непечатаемые» символы, например табуляцию или новую строку. В этом случае нужно использовать символ \ (об-

ратный слеш) для экранирования символов, которые в противном случае будут иметь другое значение.

Для табуляции мы пишем слеш перед буквой t, например: `\ t`. Если символа \ не будет, то выведется только буква t. С его же помощью выводится табуляция. Следовательно, если вы напишите `print ('Hello\tWorld')`, то получите `Hello  World.`

Другие распространенные варианты использования слеша показаны ниже.

>>> отображает команду, а следующие строки — результат.

\n (вывод новой строки)

```
>>> print ('Hello\nWorld')
Hello
World
```

\\ (выводит сам символ слеша)

```
>>> print ('\\')
\
```

\" (выводит двойные кавычки, и, таким образом, они не говорят о конце строки)

```
>>> print ("I am 5'9\" tall")
I am 5'9" tall
```

\' (выводит одинарные кавычки, и, таким образом, они не говорят о конце строки)

```
>>> print ('I am 5\'9" tall')
I am 5'9" tall
```

Если вы не хотите, чтобы символы, которым предшествует обратный слеш \, интерпретировались как специальные символы, можно использовать необработанные строки, добавив `r` перед первой кавычкой. Например, если вы не хотите, чтобы `\t` интерпретировался как табуляция, следует написать `print (r'Hello\tWorld')`. В качестве вывода вы получите `Hello\tWorld`.

6

УПРАВЛЯЮЩИЕ КОМАНДЫ

Поздравляю вас! Вы дошли до самой интересной главы. Надеюсь, пока вам все нравится. В этой главе поговорим о том, как сделать вашу программу умнее — чтобы она могла сама делать выбор и принимать решения. В частности, рассмотрим оператор `if`, циклы `for` и `while`. Они известны как инструменты потока управления и контролируют ход программы. Мы также рассмотрим конструкцию `try/except`, которая определяет, что программа должна делать при возникновении ошибки.

Но, прежде чем перейти к этим инструментам управления потоком, посмотрим на операторы условий.

6.1. УСЛОВНЫЕ УТВЕРЖДЕНИЯ

Все инструменты управления потоком включают оценку условия. Программа будет вести себя по-разному в зависимости от того, выполняется ли какое-либо условие.

Самым распространенным условным оператором является оператор сравнения. Если мы хотим сравнить, совпадают ли две переменные, то используем оператор `==` (двойной `=`). Например, выражение `x == y` просит проверить программу, равно ли значение `x` значению `y`.

Если они равны, то условие выполнено и оператор вернет `True`. В противном случае выражение будет иметь значение `False`.

Другие операторы сравнения включают `! =` (не равно), `<` (меньше), `>` (больше), `<=` (меньше или равно) и `>=` (больше или равно). В списке ниже показаны примеры использования и приведены случаи, имеющие значение `True`.

Не равно:

```
5 != 2
```

Больше:

```
5 > 2
```

Меньше:

```
2 < 5
```

Больше или равно:

```
5 >= 2
5 >= 5
```

Меньше или равно:

```
2 <= 5
2 <= 2
```

Есть три логических оператора: `and`, `or`, `not`, с помощью которых можно объединить несколько условий.

Оператор `and` возвращает `True`, если все условия выполнены. В противном случае он вернет `False`. Например,

выражение `5 == 5 and 2 > 1` вернет `True`, поскольку оба условия истинны.

Оператор `or` возвращает значение `True`, если выполняется *хотя бы одно* условие. В противном случае он вернет `False`. Выражение `5 > 2 or 7 > 10 or 3 == 2` вернет `True`, поскольку первое условие `5 > 2` истинно.

Оператор `not` возвращает `True`, если условие после ключевого слова `not` ложно. В противном случае он вернет `False`. Выражение `not 2 > 5` вернет `True`, поскольку 2 меньше 5.

6.2. ИНСТРУКЦИИ IF

Оператор `if` — один из наиболее часто используемых операторов потока управления. Он позволяет программе оценить, выполнено ли определенное условие, и совершить соответствующее действие на основе результата оценки. Структура оператора `if` выглядит следующим образом:

```
if выполняется условие 1:
    выполнить A
elif выполняется условие 2:
    выполнить B
elif выполняется условие 3:
    выполнить C
elif выполняется условие 4:
    выполнить D
else:
    выполнить E
```

`elif` означает «else if», и вы можете использовать столько операторов `elif`, сколько нужно.

Если вы раньше писали код, например, на C или Java, то вы удивитесь, заметив, что в Python не нужны скобки `( )` после ключевых слов `if`, `elif` и `else`. Кроме того, Python не использует фигурные скобки `{ }` для определения начала и конца оператора `if`. В Python используются отступы. Все, что имеет отступ, рассматривается как блок кода, который будет выполнен, если условие оценивается как `True`.

Для полного понимания работы оператора `if` запустите IDLE и введите следующий код:

```
userInput = input('Enter 1 or 2: ')
if userInput == "1":
    print ("Hello World")
    print ("How are you?")
elif userInput == "2":
    print ("Python Rocks!")
    print ("I love Python")
else:
    print ("You did not enter a valid number")
```

Программа запрашивает у пользователя ввод с помощью функции `input()`. Результат сохраняется в переменной `userInput` в виде строки.

Затем оператор `if userInput == "1":` сравнивает переменную `userInput` со строкой `"1"`. Если значение, сохраненное в `userInput`, равно `"1"`, программа будет выполнять все операторы с отступом, пока отступы не закончатся. В этом примере будет выведено `"Hello World"`, а затем `" How are you?"`.

Если же значение, сохраненное в пользовательском вводе, равно `"2"`, программа выведет на экран `" Python Rocks!"`, а затем `" I love Python"`.

Для всех остальных значений появится сообщение " `You did not enter a valid number`".

Запустите программу трижды, введите 1, 2 и 3 соответственно для каждого запуска. Результат будет следующим:

```
Enter 1 or 2: 1
Hello World
How are you?

Enter 1 or 2: 2
Python Rocks!
I love Python

Enter 1 or 2: 3
You did not enter a valid number
```

6.3. ИНЛАЙНОВЫЙ IF

Встроенный оператор `if` представляет собой более простую форму инструкцию `if` и более удобен, если нужно выполнить только простую задачу. Синтаксис будет следующим:

```
do Task A if condition is True else do Task B
```

Например:

```
num1 = 12 if userInput=="1" else 13
```

Этот оператор присваивает 12 `num1` (задача А), если `userInput` равен `"1"`. В противном случае `num1` присваивается 13 (задача Б).

Пример:

```
print ("Это задача А" if userInput == "1" else
"Это задача Б")
```

Эта инструкция выводит: `"Это задача А"` (Task A), если `userInput` равен `"1"`. В противном случае на экран выведется `"Это задача Б"` (Task B).

6.4. ЦИКЛ FOR

Теперь рассмотрим цикл `for`. Он повторно выполняет блок кода до тех пор, пока условие в операторе `for` не станет недействительным.

ЦИКЛ ПО ИТЕРАЦИИ

В Python итерация относится ко всему, что можно зациклить, например к строке, списку, кортежу или словарю. Синтаксис цикла итерации следующий:

```
for a in iterable:
    print(a)
```

Пример:

```
pets = ['cats', 'dogs', 'rabbits', 'hamsters']

for myPets in pets:
    print(myPets)
```

В приведенной выше программе мы сначала объявляем список домашних животных и назначаем его участников

'cats', 'dogs', 'rabbits' и 'hamsters'. Затем оператор myPets in pets: проходит по списку и присваивает каждому члену в списке переменную myPets.

При первом запуске программы в цикле for она присваивает значение 'cats' переменной myPets. Затем оператор print(myPets) выводит значение 'cats'. Во второй раз, когда программа перебирает оператор for, она присваивает myPets значение 'dogs' и выводит 'dogs'. Программа продолжает перебирать список, пока не будет достигнут его конец.

При запуске программы вы увидите:

```
cats
dogs
rabbits
hamsters
```

Можно отобразить индекс элементов списка. Для этого используется функция enumerate():

```
for index, myPets in enumerate(pets):
    print(index, myPets)
```

Вывод будет следующим:

```
0 cats
1 dogs
2 rabbits
3 hamster
```

Чтобы просмотреть словарь, мы используем цикл for таким же образом.

Пример:

```
age = {'Peter': 5, 'John':7}

for i in age:
    print(i)
```

Результат будет таким:

```
Peter
John
```

Если нужно получить и ключ, и значение, сделать это можно следующим образом.

Пример:

```
age = {'Peter': 5, 'John':7}

for i in age:
    print("Name = %s, Age = %d" %(i, age[i]))
```

При первом запуске программы в цикле `for` значение `'Peter'` присваивается переменной `i`.

`age [i]` становится `age ['Peter']`, равным 5.

При запуске программы вы увидите:

```
Name = Peter, Age = 5
Name = John, Age = 7
```

В качестве альтернативы можно использовать метод `items()`. Это встроенный метод, который возвращает

каждую пару ключ — значение в виде кортежа (ключ, значение). Давайте посмотрим на примере.

Пример:

```
age = {'Peter': 5, 'John':7}

for i, j in age.items():
    print("Name = %s, Age = %d" %(i, j))
```

В результате будет:

```
Name = Peter, Age = 5
Name = John, Age = 7
```

В следующем примере показано, как цикл проходит по строке:

```
message = 'Hello'

for i in message:
    print (i)
```

Вы увидите:

```
H
e
l
l
o
```

ПЕРЕБОР ЧИСЛОВОЙ ПОСЛЕДОВАТЕЛЬНОСТИ

Для перебора последовательности чисел удобно воспользоваться встроенной функцией `range()`. Функция

`range()` генерирует список чисел и имеет следующий синтаксис:

```
range(начало, конец, шаг).
```

Если аргумент *начало* не указан, то генерируемая серия будет начинаться с нуля.

> **Примечание**
>
> Полезно запомнить, что в Python (как и в большинстве языков программирования) числовые серии начинаются с нуля, если явно не указано обратное.
>
> Например, индексы списка и кортежи начинаются с нуля.
>
> Если при использовании функции `range()` начало последовательности не указано, то генерируемые числа начинаются с нуля.

Если *шаг* не указан, то функция генерирует список последовательных чисел (т. е. *шаг* = 1). Конечное значение задается обязательно. Впрочем, у функции `range()` есть одна странность: конечное значение никогда не входит в генерируемый список.

Например:

`range(5)` генерирует список `[0, 1, 2, 3, 4]`

`range(3, 10)` генерирует список `[3, 4, 5, 6, 7, 8, 9]`

`range(4, 10, 2)` генерирует список `[4, 6, 8]`

Чтобы понять, как используется функция `range()` в командах `for`, попробуйте выполнить следующий код:

```
for i in range(5):
    print (i)
```

Результат:

```
0
1
2
3
4
```

6.5. ЦИКЛ WHILE

Следующая управляющая команда — цикл `while`. Эта разновидность цикла многократно выполняет команды в цикле, пока некоторое условие остается истинным. Структура команды `while`:

```
while условие истинно:
    команда A
```

Как правило, при использовании цикла `while` необходимо сначала объявить переменную, которая будет использоваться в качестве счетчика цикла. В следующем примере эта переменная будет называться `counter`. Условие команды `while` проверяет, что счетчик меньше (или больше) некоторого значения. Если это так, то цикл будет выполнен. Рассмотрим пример программы:

```
counter = 5


while counter > 0:
    print ("Counter = ", counter)
    counter = counter - 1
```

При выполнении этой программы вы получите следующий результат:

```
Counter = 5
Counter = 4
Counter = 3
Counter = 2
Counter = 1
```

На первый взгляд кажется, что команда `while` имеет более простой синтаксис и пользоваться ей удобнее. Однако при использовании циклов `while` необходима осторожность из-за возможного зацикливания. Вы заметили в приведенной выше программе строку `counter = counter - 1`? Это очень важная строка. Она уменьшает значение `counter` на 1 и присваивает новое значение той же переменной `counter`, заменяя исходное значение.

Переменная `counter` должна уменьшаться на 1, чтобы условие цикла `while (counter > 0)` в какой-то момент стало равно `False`. Если вы забудете это сделать, то цикл будет выполняться раз за разом; в программе возникнет бесконечный цикл. Чтобы убедиться в этом, просто удалите строку `counter = counter - 1` и попробуйте снова выполнить программу. Программа так и будет выводить `counter = 5`, пока вы каким-то образом не прервете ее

выполнение. Не лучший вариант, особенно если вы запустили большую программу и понятия не имеете, какая часть кода вызвала зацикливание.

6.6. BREAK

При работе с циклами иногда требуется прервать цикл в определенной ситуации. Для этого используется ключевое слово `break`. Выполните следующую программу и посмотрите, как она работает:

```
j = 0
for i in range(5):
    j = j + 2
    print ('i = ', i, ', j = ', j)
    if j == 6:
        break
```

Вы получите следующий результат:

```
i =  0 , j =  2
i =  1 , j =  4
i =  2 , j =  6
```

Без ключевого слова `break` программа выполнит цикл от `i = 0` до `i = 4`, так как в цикле использовалась функция `range(5)`. Однако с ключевым словом `break` программа завершится преждевременно при `i = 2`. Дело в том, что при `i = 2` переменная `j` достигает значения 6 и ключевое слово `break` вызовет завершение цикла.

В приведенном выше примере стоит заметить, что команда `if` использовалась в цикле `for`. В программировании

подобные комбинации разных управляющих команд встречаются очень часто: например, цикл `while` может использоваться внутри команды `if` или цикл `for` — внутри цикла `while`. Такие конструкции называются *вложенными* управляющими командами.

6.7. CONTINUE

Еще одно ключевое слово, которое может пригодиться в цикле, — `continue`. При выполнении команды `continue` оставшаяся часть цикла после `continue` пропускается для текущей итерации. Со следующим примером все станет понятнее:

```
j = 0
for i in range(5):
    j = j + 2
    print ('\ni = ', i, ', j = ', j)
    if j == 6:
        continue
    print ('I will be skipped over if j=6')
```

Результат выглядит так:

```
i =  0 , j =  2
I will be skipped over if j=6

i =  1 , j =  4
I will be skipped over if j=6

i =  2 , j =  6

i =  3 , j =  8
I will be skipped over if j=6
```

```
i =  4 , j =  10
I will be skipped over if j=6
```

При `j = 6` сообщение после ключевого слова `continue` не выводится. В остальном программа работает, как обычно.

6.8. TRY/EXCEPT

Последний оператор управления, который мы рассмотрим, — это оператор `try/except`. Он контролирует работу программы при возникновении ошибки. Синтаксис:

```
try:
    что-то сделать
except:
    сделать что-то другое в случае ошибки
```

Например, попробуйте выполнить следующую программу:

```
try:
    answer = 12/0
    print (answer)
except:
    print ("An error occurred")
```

При запуске программы вы получите сообщение `"An error occurred"`. Это произошло из-за того, что, когда программа пытается выполнить команду `answer = 12/0` в блоке `try`, происходит ошибка, потому что деление на нуль невозможно. Оставшаяся часть блока `try` игнорируется, и вместо нее выполняется команда из блока `except`.

Если вы хотите вывести более конкретное сообщение, зависящее от типа ошибки, укажите тип ошибки после ключевого слова `except`. Попробуйте выполнить следующую программу:

```
try:
    userInput1 = int(input("Please enter a number: "))
    userInput2 = int(input("Please enter another
                    number: "))
    answer =userInput1/userInput2
    print ("The answer is ", answer)
    myFile = open("missing.txt", 'r')
except ValueError:
    print ("Error: You did not enter a number")
except ZeroDivisionError:
    print ("Error: Cannot divide by zero")
except Exception as e:
    print ("Unknown error: ", e)
```

Ниже приведены различные варианты вывода для разных входных данных. `>>>` обозначает пользовательский ввод, а `=>` — результат выполнения:

```
>>> Please enter a number: m
=> Error: You did not enter a number
```

Причина: пользователь ввел строку, которая не может быть преобразована в целое число. Данная ошибка относится к категории `ValueError`, поэтому выводится сообщение для `ValueError` из блока `except`:

```
>>> Please enter a number: 12
>>> Please enter another number: 0
=> Error: Cannot divide by zero
```

Причина: `userInput2 = 0`. Так как деление на нуль запрещено, это ошибка `ZeroDivisionError`. Выводится сообщение для `ZeroDivisionError` из блока `except`:

```
>>> Please enter a number: 12
>>> Please enter another number: 3
=> The answer is 4.0
=> Unknown error: [Errno 2] No such file or
directory: 'missing.txt'
```

Причина: пользователь ввел допустимые значения, строка `print ("The answer is ", answer)` выполняется правильно. Однако в следующей строке происходит ошибка, потому что файл missing.txt не найден. Так как ошибка не относится ни к категории `ValueError`, ни к категории `ZeroDivisionError`, выполняется последний блок `except`.

`ValueError` и `ZeroDivisionError` — всего лишь два из многих заранее определенных типов ошибок в Python. Ошибка `ValueError` выдается в ситуации, когда встроенная операция или функция получает аргумент с правильным типом, но недопустимым значением. Ошибка `ZeroDivisionError` выдается при попытке деления на нуль в программе.

Другие типы распространенных ошибок в Python:

`IOError:`

Операция ввода/вывода (например, встроенная функция `open()`) завершилась неудачей по причине, связанной с вводом/выводом, — например, «файл не найден».

`ImportError:`

Команда `import` не находит определение модуля.

`IndexError:`

Последовательность (строка, список, кортеж) выходит за пределы диапазона.

`KeyError:`

Ключ словаря не найден.

`NameError:`

Локальное или глобальное имя не найдено.

`TypeError:`

Операция или функция применяется к объекту неподходящего типа.

***

Полный список типов ошибок в Python см. по адресу: https://docs.python.org/3/library/exceptions.html.

В Python также существуют заранее определенные описания для всех типов ошибок. Если вы хотите вывести сообщение, поставьте ключевое слово `as` после типа ошибки. Например, следующий фрагмент выводит сообщение по умолчанию для ошибки `ValueError`:

```
except ValueError as e:
    print (e)
```

`e` — имя переменной, которая связывается с ошибкой. Переменной можно присвоить и другое имя, но на практике обычно используется `e`. Последняя команда `except` в нашей программе:

```
except Exception as e:
    print ("Unknown error: ", e)
```

демонстрирует возможность использования заранее определенных сообщений об ошибках. Она становится последним рубежом для перехвата любых непредвиденных ошибок.

7

ФУНКЦИИ И МОДУЛИ

Функции и модули уже упоминались в предыдущих главах. В этой главе они будут рассмотрены более подробно. Напомню, что во все языки программирования включается встроенный код, который упрощает работу программистов. Этот код состоит из готовых классов, переменных и функций для выполнения некоторых стандартных операций; он сохраняется в файлах, называемых *модулями*. Но начнем с функций.

7.1. ЧТО ТАКОЕ ФУНКЦИИ?

Функции представляют собой готовые блоки кода для выполнения некоторой задачи. Возьмем хотя бы математические функции из MS Excel. Чтобы просуммировать несколько чисел, можно воспользоваться функцией `СУМ()` и ввести команду `СУМ(A1:A5)`, вместо того чтобы набирать длинную строку `A1+A2+A3+A4+A5`.

В зависимости от того, как написана функция, является ли она частью класса (классы — концепция объектно-ориентированного программирования, которая будет рассматриваться в следующих главах) и как она импортируется, при вызове функции можно либо просто

ввести ее имя, либо воспользоваться *точечной записью*. Некоторым функциям для выполнения своих операций необходимы дополнительные данные, которые должны передаваться им при вызове. Такие данные называются *аргументами*; чтобы передать их функции, заключите их значения, разделенные запятыми, в круглые скобки после имени функции.

Например, чтобы использовать функцию `print()` для вывода текста на экран, введите команду `print("Hello World")`; здесь `print` — имя функции, а `"Hello World"` — аргумент.

С другой стороны, чтобы использовать функцию `replace()` для обработки текста, необходимо ввести команду

```
newString = "Hello World".replace("World", "Universe")
```

где `replace` — имя функции, а `"World"` и `"Universe"` — аргументы. Строка перед точкой (например, `"Hello World"`) — та строка, с которой будет выполняться операция. Таким образом, `"Hello World"` превратится в `"Hello Universe"`.

Некоторые функции могут возвращать результат после выполнения своих задач. В данном примере функция `replace()` вернет `"Hello Universe"`, которая присваивается переменной `newString`. Если вывести значение `newString` командой

```
print(newString)
```

вы получите такой результат:

```
Hello Universe
```

7.2. ОПРЕДЕЛЕНИЕ СОБСТВЕННЫХ ФУНКЦИЙ

В языке Python можно определять собственные функции и повторно использовать их в программах. Синтаксис определения функции выглядит так:

```
def functionName(список параметров):
    код операций, выполняемых функцией
    return [выражение]
```

В определении задействованы два ключевых слова: `def` и `return`.

Ключевое слово `def` сообщает программе, что код с отступом, который начинается в следующей строке и идет далее, является частью функции. Ключевое слово `return` возвращает ответ из функции. Функция может содержать более одной команды `return`. Тем не менее после выполнения команды `return` происходит выход из функции. Если ваша функция не должна возвращать значение, команду `return` можно опустить; также можно включить команду `return` или `return None`.

Определим свою первую функцию. Допустим, вы хотите узнать, является ли заданное число простым. В следующем определении функции используется оператор вычисления остатка (`%`), который был описан в разделе 3.4, а также цикл `for` и команда `if` из главы 6:

```
def checkIfPrime (numberToCheck):
    for x in range(2, numberToCheck):
        if (numberToCheck%x == 0):
            return False
    return True
```

Эта функция использует один параметр с именем `numberToCheck`. Параметры — переменные, которые используются для хранения аргументов, передаваемых функции.

В строках 2 и 3 в цикле `for` параметр `numberToCheck` делится на все числа от 2 до `numberToCheck - 1`, после чего программа проверяет, равен ли остаток нулю. Если остаток равен нулю, то `numberToCheck` не является простым числом. Строка 4 возвращает `False`, и функция возвращает управление.

Если к последней итерации цикла ни при одном делении не будет получен нулевой остаток, функция достигает строки 5 и возвращает `True`. После этого функция возвращает управление.

Чтобы использовать эту функцию, мы вводим вызов `checkIfPrime(13)` и присваиваем ее переменной:

```
answer = checkIfPrime(13)
```

Здесь число 13 передается как аргумент и сохраняется в параметре `numberToCheck`. Затем цикл `for` проверяет, является ли число `numberToCheck` простым, и возвращает `True` или `False`. Чтобы вывести ответ, введите команду `print(answer)`. Как и следовало ожидать, выводится результат `True`.

7.3. ОБЛАСТЬ ВИДИМОСТИ ПЕРЕМЕННЫХ

При определении функций очень важно понимать концепцию *области видимости* переменных. Переменные, опре-

деленные внутри функции, отличаются от переменных, определенных вне ее. Существуют два основных различия.

Во-первых, любая переменная, объявленная внутри функции, доступна только в этой функции. Такие переменные называются *локальными*. Любая переменная, объявленная за пределами функции, называется *глобальной*; такие переменные доступны в любой точке программы.

Чтобы понять, чем глобальные переменные отличаются от локальных, попробуйте выполнить следующий код:

```
message1 = "Global Variable"

def myFunction():
    print("\nINSIDE THE FUNCTION")
    # Глобальные переменные доступны внутри функции
    print (message1)
    # Объявление локальной переменной
    message2 = "Local Variable"
    print (message2)

'''
Вызов функции
Функция myFunction() не имеет параметров,
поэтому при вызове этой функции
используется пара круглых скобок.
'''
myFunction()

print("\nOUTSIDE THE FUNCTION")

# Глобальные переменные доступны за пределами функции
print (message1)

# Локальные переменные НЕДОСТУПНЫ за пределами функции
print (message2)
```

Запустив программу, вы получите следующий результат:

```
INSIDE THE FUNCTION
Global Variable
Local Variable

OUTSIDE THE FUNCTION
Global Variable
NameError: name 'message2' is not defined
```

В приведенном фрагменте `message1` — глобальная переменная, а `message2` — локальная переменная, объявленная внутри функции `myFunction()`. Внутри функции доступны как локальные, так и глобальные переменные. За пределами функции локальная переменная `message2` становится недоступной. При попытке обращения к ней за пределами функции происходит ошибка `NameError`.

Второе, что нужно знать об области видимости переменных — если имя локальной переменной совпадает с именем глобальной переменной, весь код внутри функции будет работать с локальной переменной. Весь код за пределами функции будет обращаться к глобальной переменной. Попробуйте выполнить следующий код:

```
message1 = "Global Variable (shares same name as
            a local variable)"

def myFunction():
    message1 = "Local Variable (shares same name as
                a global variable)"
    print("\nINSIDE THE FUNCTION")
```

```
    print (message1)

# Calling the function
myFunction()

# Printing message1 OUTSIDE the function
print ("\nOUTSIDE THE FUNCTION")
print (message1)
```

Результат выглядит так:

```
INSIDE THE FUNCTION
Local Variable (shares same name as a global variable)

OUTSIDE THE FUNCTION
Global Variable (shares same name as a local variable)
```

Когда мы выводим `message1` внутри функции, выводится строка `"Local Variable (shares same name as a global variable)"`, потому что выводится локальная переменная. При выводе за пределами функции программа обращается к глобальной переменной, поэтому выводится строка `"Global Variable (shares same name as a local variable)"`.

7.4. ЗНАЧЕНИЯ ПАРАМЕТРОВ ПО УМОЛЧАНИЮ

Итак, теперь вы знаете, как работают функции и области видимости переменных. Пора рассказать о некоторых интересных возможностях, которые предоставляет Python при определении функций.

Начнем со значений по умолчанию.

Python позволяет определить значения по умолчанию для параметров функции. Если параметр имеет значение по умолчанию, передавать значение этого параметра при вызове метода не обязательно. Допустим, функция имеет 5 параметров: `a`, `b`, `c`, `d` и `e`. Определим эту функцию в следующем виде:

```
def someFunction(a, b, c=1, d=2, e=3):
    print(a, b, c, d, e)
```

Значения по умолчанию определяются для трех последних параметров. Все параметры со значениями по умолчанию должны располагаться в конце списка параметров. Иначе говоря, следующее определение функции недопустимо, потому что параметр `r` следует после параметра `q`, имеющего значения по умолчанию:

```
def someIncorrectFunction(p, q=1, r):
    print(p, q, r)
```

При вызове `someFunction()`

```
someFunction(10, 20)
```

будет получен результат

```
10, 20, 1, 2, 3
```

Передавать значения для `c`, `d` и `e` не обязательно.

Если же использовать вызов

```
someFunction(10, 20, 30, 40)
```

результат будет таким:

```
10, 20, 30, 40, 3
```

Два дополнительных аргумента, передаваемых при вызове (`30` и `40`), присваиваются параметрам со значениями по умолчанию в указанном порядке. Таким образом, `30` заменяет значение по умолчанию для `c`, а `40` заменяет `d`.

7.5. СПИСКИ АРГУМЕНТОВ ПЕРЕМЕННОЙ ДЛИНЫ

Помимо определения значений по умолчанию для параметров Python также позволяет передавать функциям переменное количество аргументов. Это очень полезно, если число аргументов функции не известно заранее. Допустим, вы пишете функцию для суммирования серий чисел, но точное количество этих чисел не известно заранее. В таких ситуациях используется знак *. Следующий пример показывает, как это делается:

```
def addNumbers(*num):
    sum = 0
    for i in num:
        sum = sum + i
    print(sum)
```

Символ * перед `num` сообщает компилятору, что в `num` хранится список аргументов переменной длины, содержащий несколько элементов.

Далее функция перебирает содержимое аргумента, чтобы вычислить сумму всех чисел и вернуть ответ.

Вызывая функцию в виде:

```
addNumbers(1, 2, 3, 4, 5)
```

вы получите результат 15. Также можно просуммировать больше чисел, используя вызов

```
addNumbers(1, 2, 3, 4, 5, 6, 7, 8)
```

Результат будет равен 36.

Как демонстрируют эти примеры, при добавлении * перед именем параметра можно передать функции произвольное количество аргументов. Это называется **списком аргументов переменной длины без ключевых слов**.

Если вы хотите передать функции список аргументов переменной длины с ключевыми словами, используйте последовательность **.

Возьмем следующий пример:

```
def printMemberAge(**age):
    for i, j in age.items():
        print("Name = %s, Age = %s" %(i, j))
```

Функция содержит параметр с именем `age`. Символы ** означают, что в параметре хранится **список аргументов переменной длины с ключевыми словами**, т. е. словарь. Затем цикл `for` перебирает аргумент и выводит значения. Для данного вызова функции:

```
printMemberAge(Peter = 5, John = 7)
```

будет получен следующий результат:

```
Name = Peter, Age = 5
Name = John, Age = 7
```

При использовании вызова

```
printMemberAge(Peter = 5, John = 7, Yvonne = 10)
```

мы получим

```
Name = Peter, Age = 5
Name = John, Age = 7
Name = Yvonne, Age = 10
```

Если функция использует обычный аргумент (также называемый *формальным аргументом*), список аргументов переменной длины без ключевых слов и список аргументов переменной длины с ключевыми словами, она должна определяться в следующем порядке:

```
def someFunction2(farg, *args, **kwargs):
```

Таким образом, на первом месте должен стоять формальный аргумент, затем следуют списки аргументов переменной длины без ключевых слов и с ключевыми словами — именно в таком порядке.

7.6. ИМПОРТИРОВАНИЕ МОДУЛЕЙ

Python содержит большое количество встроенных функций. Эти функции сохраняются в файлах, называемых *модулями*. Чтобы использовать встроенный код из моду-

лей Python, необходимо сначала импортировать их в программы. Для этого используется ключевое слово `import`. Существуют три способа импортирования модулей:

1. Прежде всего можно импортировать весь модуль командой `import` *`имяМодуля`*.

   Например, для импортирования встроенного модуля `random`, входящего в поставку Python, используется команда `import random`.

   Вызов функции `randrange()` из модуля `random` выглядит так:

   ```
   random.randrange(1, 10)
   ```

2. Если вам покажется, что писать `random` при каждом использовании функции слишком утомительно, используйте команду `import random as r` (вместо `r` можно использовать любое имя на ваше усмотрение). Теперь для вызова функции `randrange()` достаточно использовать запись `r.randrange(1, 10)`.
3. В третьем варианте импортирования модулей импортируются конкретные функции модуля записью вида `from` *`имяМодуля`* `import` *`имя1`*`[, `*`имя2`*`[, ... `*`имяN`*`]]`.

Например, импортирование функции `randrange()` из модуля `random` осуществляется командой `from random import randrange`. Если вы хотите импортировать сразу несколько функций, разделите их запятыми. Чтобы импортировать функции `randrange()` и `randint()`, используйте команду `from random import randrange, randint`. Теперь для вызова функции точечная запись вообще не нужна — достаточно простого вызова `randrange(1, 10)`.

7.7. СОЗДАНИЕ СОБСТВЕННЫХ МОДУЛЕЙ

Помимо импортирования встроенных модулей вы также можете создавать собственные модули. Такая возможность чрезвычайно полезна, если у вас имеются функции, которые вы собираетесь снова использовать в других программных проектах в будущем.

Создать собственный модуль несложно. Просто сохраните файл модуля с расширением .py и разместите его в одной папке с тем файлом Python, из которого собираетесь этот модуль импортировать.

Допустим, вы хотите использовать функцию `checkIfPrime()`, определенную ранее в другом сценарии Python. Вот как это делается: сначала сохраните код функции в файле с именем prime.py на рабочем столе. Файл prime.py должен содержать следующий код:

```
def checkIfPrime (numberToCheck):
    for x in range(2, numberToCheck):
        if (numberToCheck%x == 0):
            return False
    return True
```

Затем создайте другой файл Python и присвойте ему имя usecheckifprime.py. Также сохраните его на рабочем столе. Файл usecheckifprime.py должен содержать следующий код:

```
import prime
answer = prime.checkIfPrime(13)
print (answer)
```

Теперь запустите usecheckifprime.py. Программа должна вывести результат `True`... Проще некуда.

Но допустим, вы хотите хранить файлы prime.py и usecheckifprime.py в разных каталогах. В этом случае необходимо добавить в usecheckifprime.py код, который сообщает интерпретатору Python, где искать модуль.

Допустим, для хранения prime.py на диске C был создан каталог с именем MyPythonModules. В начало файла usecheckifprime.py необходимо добавить следующий код (перед строкой `import prime`):

```
import sys

if 'C:\\MyPythonModules' not in sys.path:
    sys.path.append('C:\\MyPythonModules')
```

`sys.path` представляет *системный путь* Python. Это список каталогов, которые Python просматривает в ходе поиска модулей и файлов. Приведенный выше код присоединяет каталог C:\MyPythonModules к системному пути вашего компьютера.

Теперь файл prime.py можно разместить в каталоге C:\MyPythonModules, а файл usecheckifprime.py — в любом другом каталоге на ваше усмотрение.

8

РАБОТА С ФАЙЛАМИ

Классно! Уже восьмая глава! Здесь мы узнаем, как работать с внешними файлами.

Из главы 5 мы узнали, как получать данные от пользователей с помощью функции `input()`. Однако в некоторых случаях заставить пользователей вводить данные в нашу программу может оказаться непрактичным, особенно если программа должна работать с большими объемами данных. В таких случаях удобно подготовить нужную информацию в виде внешнего файла и дать команду программе считать информацию из файла. В этой главе мы научимся это делать.

Готовы?

8.1. ОТКРЫТИЕ И ЧТЕНИЕ ТЕКСТОВЫХ ФАЙЛОВ

Первая разновидность файлов, из которых мы будем читать данные, — простые текстовые файлы, состоящие из нескольких строк текста. Начнем с создания текстового файла, который содержит следующие строки:

```
Learn Python in One Day and Learn It Well
Python for Beginners with Hands-on Project
The only book you need to start coding in Python
immediately
https://www.learncodingfast.com/python
```

Сохраните этот текстовый файл под именем myfile.txt на своем компьютере. Запустите IDLE и создайте новый файл. Введите следующий код и сохраните файл под именем fileoperation.py на своем компьютере:

```
f = open ('myfile.txt', 'r')

firstline = f.readline()
secondline = f.readline()
print (firstline)
print (secondline)

f.close()
```

Первая строка открывает файл. Прежде чем читать данные из любого файла, его необходимо открыть (подобно тому, как вы открываете электронную книгу на своем устройстве или в приложении, чтобы читать ее). Функция `open()` открывает файл и получает два аргумента.

Первый аргумент содержит путь к файлу. Если вы не сохранили файлы fileoperation.py и myfile.txt в одном каталоге, строку `'myfile.txt'` нужно будет заменить фактическим каталогом, в котором был сохранен файл. Например, если вы сохранили его в каталоге PythonFiles на диске C, нужно будет использовать путь `'C:\\PythonFiles\\myfile.txt'` (с двойным слешем `\\`).

Второй аргумент содержит режим. Он сообщает, как будет использоваться файл. Чаще всего встречаются следующие режимы:

`'r'` – только для чтения.

`'w'` – только для записи.

Если указанный файл не существует, он будет создан.

Если указанный файл существует, то все данные в нем стираются.

`'a'` — для присоединения.

Если указанный файл не существует, он будет создан.

Если указанный файл существует, то все данные, записываемые в файл, автоматически добавляются в конец файла.

`'r+'` — для чтения и записи.

После открытия файла следующая команда `firstline = f.readline()` читает первую строку в файле и присваивает ее переменной `firstline`.

При каждом вызове функции `readline()` из файла читается новая строка. В нашей программе функция `readline()` была вызвана дважды, поэтому из файла были прочитаны две строки. При выполнении программы будет получен следующий результат:

```
Learn Python in One Day and Learn It Well

Python for Beginners with Hands-on Project
```

Обратите внимание: после каждой строки вставляется разрыв строк. Дело в том, что функция `readline()` добавляет символы `'\n'` в конец каждой строки. Если вы не хотите, чтобы каждая строка текста завершалась разрывом строки, используйте конструкцию `print(firstline, end = '')`. Эта команда удалит символы `'\n'`. Следует заметить, что `''` — две одинарные кавычки, а не одна двойная.

После чтения из файла и вывода двух строк последняя команда `f.close()` закрывает файл. Всегда закрывайте файл после завершения чтения, чтобы освободить системные ресурсы.

8.2. ЧТЕНИЕ ТЕКСТОВЫХ ФАЙЛОВ В ЦИКЛЕ FOR

Кроме метода `readline()` для чтения текстовых файлов также можно воспользоваться циклом `for`. Более того, цикл `for` предоставляет более элегантный и эффективный механизм чтения текстовых файлов. Следующий пример показывает, как это делается:

```
f = open ('myfile.txt', 'r')

for line in f:
    print (line, end = '')

f.close()
```

Цикл `for` перебирает текстовый файл строку за строкой. При его выполнении вы получите следующий результат:

```
Learn Python in One Day and Learn It Well
Python for Beginners with Hands-on Project
The only book you need to start coding in Python
immediately
https://www.learncodingfast.com/python
```

8.3. ЗАПИСЬ В ТЕКСТОВЫЙ ФАЙЛ

Итак, теперь вы умеете открывать и читать файлы; попробуем записать информацию в файл. Для этого будет использоваться режим `'a'` (присоединение). Также можно воспользоваться режимом `'w'`, но если файл существует, все хранящиеся в нем данные будут потеряны. Попробуйте выполнить следующую программу:

```
f = open ('myfile.txt', 'a')

f.write('\nThis sentence will be appended.')
f.write('\nPython is Fun!')

f.close()
```

В этом примере функция `write()` присоединяет к файлу две строки `'This sentence will be appended.'` и `'Python is Fun!'`. Каждый блок текста добавляется с новой строки, потому что в текст включаются служебные последовательности `'\n'`. Вы получите следующий результат:

```
Learn Python in One Day and Learn It Well
Python for Beginners with Hands-on Project
The only book you need to start coding in Python
immediately
https://www.learncodingfast.com/python
This sentence will be appended.
Python is Fun!
```

8.4. ОТКРЫТИЕ И ЧТЕНИЕ ТЕКСТОВЫХ ФАЙЛОВ В БУФЕР

Иногда данные из файла должны читаться блоками определенного размера, чтобы программа не занимала слишком много памяти. Для этого можно воспользоваться функцией `read()` (вместо функции `readline()`), позволяющей задать нужный размер буфера. Попробуйте выполнить следующую программу:

```
inputFile = open ( 'myfile.txt', 'r')
outputFile = open ('myoutputfile.txt', 'w')

msg = inputFile.read(10)

while len(msg):
    outputFile.write(msg)
    msg = inputFile.read(10)

inputFile.close()
outputFile.close()
```

Сначала программа открывает два файла (inputFile.txt и outputFile.txt) для чтения и записи соответственно.

Затем команда `msg = inputFile.read(10)` и цикл `while` используются для чтения файла порциями по 10 байт. Значение 10 в круглых скобках сообщает функции `read()`, что она должна прочитать только 10 байт. Условие цикла `while len(msg):` проверяет длину переменной `msg`. Пока значение `length` остается ненулевым, цикл продолжает работать.

В цикле `while` команда `outputFile.write(msg)` записывает сообщение в выходной файл. После записи сообщения

команда `msg = inputFile.read(10)` читает следующие 10 байт и продолжает делать это до тех пор, пока не будет прочитан весь файл. Когда это произойдет, программа закрывает оба файла.

При запуске программы будет создан новый файл myoutputfile.txt. Открыв этот файл, вы заметите, что он содержит те же данные, что и входной файл myfile.txt. Чтобы убедиться в том, что за один раз читаются только 10 байт, можно изменить строку `outputFile.write(msg)` на `outputFile.write(msg + '\n')`. Снова запустите программу. Теперь myoutputfile.txt состоит из строк длиной не более 10 символов. Приведу фрагмент этого файла:

```
Learn Pyth
on in One
Day and Le
arn It Wel
```

8.5. ОТКРЫТИЕ, ЧТЕНИЕ И ЗАПИСЬ ДВОИЧНЫХ ФАЙЛОВ

Двоичные файлы — любые файлы, содержащие нетекстовую информацию (например, графику или видео). Чтобы работать с двоичным файлом, достаточно указать при открытии режим `'rb'` или `'wb'`. Скопируйте файл JPEG на рабочий стол и переименуйте его в myimage.jpg. Теперь отредактируйте приведенную выше программу и измените первые две строки:

```
inputFile = open ('myfile.txt', 'r')
outputFile = open ('myoutputfile.txt', 'w')
```

Приведите их к следующему виду:

```
inputFile = open ('myimage.jpg', 'rb')
outputFile = open ('myoutputimage.jpg', 'wb')
```

Также не забудьте вернуть команду `outputFile.write(msg + '\n')` к прежнему виду `outputFile.write(msg)`.

Запустите новую программу. На рабочем столе должен появиться новый графический файл с именем myoutputimage.jpg. Если вы откроете этот файл, он должен содержать точно такое же изображение, как myimage.jpg.

8.6. УДАЛЕНИЕ И ПЕРЕИМЕНОВАНИЕ ФАЙЛОВ

Две другие полезные функции, которые необходимо освоить при работе с файлами, — `remove()` и `rename()`. Эти функции доступны в модуле `os`, и перед использованием их необходимо импортировать.

Функция `remove()` предназначена для удаления файлов. При вызове используется синтаксис `remove(`*файл*`)`. Например, для удаления файла myfile.txt используется вызов `remove('myfile.txt')`.

Функция `rename()` переименовывает файл. Синтаксис вызова — `rename(`*старое имя*, *новое имя*`)`. Для переименования файла oldfile.txt в newfile.txt используется вызов `rename('oldfile.txt', 'newfile.txt')`.

9

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, ЧАСТЬ I

Мы уже много всего рассмотрели и в следующих двух главах поговорим о важной концепции программирования на Python — объектно-ориентированном программировании (ООП).

В этой главе мы узнаем, что такое ООП, как писать собственные классы и создавать из них объекты. В следующей главе обсудим наследование и рассмотрим некоторые другие сложные темы ООП.

Что ж, приступим!

9.1. ЧТО ТАКОЕ ООП?

Простыми словами, ООП — это подход, при котором задача разбивается на взаимодействующие друг с другом объекты.

Объекты создаются из шаблонов, известных как классы. Класс — это как чертеж здания. Объект — это реальное «здание», которое мы строим на основе чертежа.

Чтобы понять, как работает ООП, напишем простой класс.

9.2 НАПИСАНИЕ СОБСТВЕННЫХ КЛАССОВ

Определение класса начинается с ключевого слова `class`, за которым указывается имя класса. Например, при создании класса `Staff` вы сначала пишете следующий заголовок:

```
class Staff:
    # содержимое класса
```

На практике при определении имен классов обычно используется схема Pascal. В ней каждое слово, включая первое, начинается с символа верхнего регистра (например, `ThisIsAClassName`). Именно эта схема будет использоваться в книге.

Класс состоит из переменных и функций. Как вы узнали из предыдущих глав, переменные предназначены для хранения данных, а функции представляют собой блоки кода, которые выполняют за нас некоторые операции. Функция, существующая внутри класса, обычно называется *методом*.

Класс можно представить как шаблон для группировки взаимосвязанных данных и методов.

Допустим, имеется класс с именем `Staff`. Этот класс может использоваться для хранения всей информации, относящейся к работнику компании. В классе можно определить две переменные для хранения имени и долж-

ности работника. Кроме того, в нем определяется метод `calculatePay()` для вычисления зарплаты работника.

Посмотрим, как это сделать.

Запустите IDLE и создайте новый файл с именем classdemo.py. Включите следующий код в classdemo.py:

```
class Staff:
    def __init__ (self, pPosition, pName, pPay):
        self.position = pPosition
        self.name = pName
        self.pay = pPay
        print('Creating Staff object')

    def __str__(self):
        return "Position = %s, Name = %s, Pay = %d"
        %(self.position, self.name, self.pay)

    def calculatePay(self):
        prompt = '\nEnter number of hours worked for
        %s: ' %(self.name)
        hours = input(prompt)
        prompt = 'Enter the hourly rate for %s: '
        %(self.name)
        hourlyRate = input(prompt)
        self.pay = int(hours)*int(hourlyRate)
        return self.pay
```

В этом фрагменте мы сначала определяем класс с именем `Staff`:

```
class Staff:
```

Затем в классе определяется специальный метод с именем `__init__`. Он называется *инициализатором* класса. Имя

этого метода всегда состоит из символов `init`, до и после которых следуют по два символа подчеркивания. В Python включено большое количество специальных методов. Имена всех специальных методов начинаются и заканчиваются двумя символами подчеркивания. Специальные методы рассматриваются в разделе 10.4.

Инициализатор вызывается каждый раз, когда в программе создается объект класса. Не огорчайтесь, если вы еще не понимаете, что это значит. О том, как создать объект класса, будет рассказано позднее.

А пока достаточно знать, что инициализатор часто используется для инициализации переменных, содержащихся в классе (т. е. присваивания им начальных значений).

В нашем классе определены три переменные — `position`, `name` и `pay`. Эти переменные называются *переменными экземпляра*, в отличие от локальных переменных (см. раздел 7.3) и переменных класса (см. раздел 9.6). Переменные экземпляров начинаются с префикса из ключевого слова `self`.

На данный момент объяснить смысл `self` довольно сложно. Проще говоря, `self` относится к конкретному экземпляру. Пока это утверждение выглядит довольно абстрактно. Смысл `self` будет рассмотрен в одном из следующих разделов. А пока достаточно знать, что когда вы хотите что-то сделать с переменной класса, перед именами этих переменных следует поставить `self`. Кроме того, `self` является первым параметром многих методов класса.

Следующие три команды присваивают три параметра метода `__init__` (`pPosition`, `pName` и `pPay`) переменным экземпляра, чтобы инициализировать их:

```
self.position = pPosition
self.name = pName
self.pay = pPay
```

После инициализации трех переменных экземпляров выводится простое сообщение `'Creating Staff object'`. И это все, что делает наш инициализатор.

Если вы не хотите инициализировать переменные экземпляра при создании объекта, инициализатор можно не определять. Переменные всегда можно инициализировать позднее.

Перейдем к следующему методу `__str__`.

`__str__` — еще один специальный метод, который часто включается при определении класса. Этот метод используется для возвращения строки, представляющей экземпляр в удобочитаемой форме.

В нашем примере метод возвращает строку со значениями трех переменных экземпляра. О том, как использовать этот метод, будет рассказано позднее.

А теперь перейдем к методу `calculatePay()`.

Метод `calculatePay()` используется для вычисления зарплаты работника. Внешне он очень похож на функцию, если не считать параметра `self`. В самом деле, метод практически не отличается от функции, если не считать

того, что он существует внутри класса, а большинство методов также получает параметр `self`.

В методе `calculatePay()` пользователю сначала предлагается ввести количество отработанных часов для работника и почасовую ставку. По этим двум значениям вычисляется зарплата, после чего метод присваивает результат переменной экземпляра `self.pay` и возвращает значение `self.pay`.

Возможно, вы заметили, что в этом методе мы не ставим `self` перед именами некоторых переменных (например, `prompt`, `hours` и `hourlyRate`). Дело в том, что эти переменные являются локальными и существуют только внутри метода `calculatePay()`. Перед локальными переменными ставить `self` не нужно.

Вот и все, что можно сказать о написанном нами классе. Напомним, что класс состоит из следующих компонентов:

Переменные экземпляра

- `position`
- `name`
- `pay`

Методы

- `__init__`
- `__str__`
- `calculatePay()`

9.3. СОЗДАНИЕ ЭКЗЕМПЛЯРА

Теперь посмотрим, как использовать этот класс в программе.

Для этого необходимо создать объект на базе класса — это называется созданием экземпляра (объекты также называются экземплярами). Хотя между объектом и экземпляром существуют некоторые различия, они имеют скорее семантическую природу, и эти два термина часто используются как синонимы.

Начнем с создания экземпляра `Staff`. Для этого мы воспользуемся оболочкой Python Shell. Прежде чем использовать класс `Staff`, необходимо выполнить соответствующий модуль. Выполните команду Run ▸ Run Module из файла classdemo.py. Команда открывает оболочку Python Shell.

А теперь можно создать экземпляр класса `Staff`.

Выполните команду

```
officeStaff1 = Staff('Basic', 'Yvonne', 0)
```

Эта команда отчасти напоминает привычное объявление переменной:

```
userAge = 10
```

В данном случае `officeStaff1` — имя переменной. Но поскольку `officeStaff1` не является целочисленной пере-

менной, ей не присваивается целое число. Вместо этого в правой части размещается выражение `Staff('Basic', 'Yvonne', 0)`. При этом мы даем команду классу `Staff` создать объект `Staff` и выполнить метод `__init__` для инициализации переменных экземпляра в классе.

Обратили внимание на три значения `'Basic'`, `'Yvonne'` и 0 в круглых скобках? Они предназначены для параметров `pPosition`, `pName` и `pPay` в написанном ранее методе `__init__`. Эти три значения используются для инициализации переменных экземпляра `position`, `name` и `pay` соответственно. Вы спросите, что произошло с первым параметром `self`? Для параметра `self` значение передавать не нужно. Это специальный параметр, и Python добавит его автоматически при вызове метода.

После того как объект будет создан, он присваивается переменной `officeStaff1`.

Попробуйте ввести следующую команду в оболочке Shell и нажать клавишу Enter:

```
officeStaff1 = Staff('Basic', 'Yvonne', 0)
```

На экране появляется сообщение

```
Creating Staff object
```

Оно означает, что инициализатор был выполнен.

После того как объект класса `Staff` будет создан, его можно будет использовать для обращения к переменным

экземпляра и методам внутри класса. Для обращения к любой переменной экземпляра или методу в классе `Staff` следует поставить оператор «точка» после имени переменной экземпляра или метода.

Например, для обращения к переменной экземпляра `name` используется запись

```
officeStaff1.name
```

При обращении к переменной в оболочке не нужно использовать функцию `print()` для вывода значения. Но если команда выполняется вне оболочки, функция `print()` обязательна. Пример такого рода будет рассмотрен в разделе 9.6.

Попробуйте ввести в Shell следующие команды и посмотрите, что произойдет.

Обращение к переменной `name`:

```
officeStaff1.name
```

Вывод:

```
'Yvonne'
```

Обращение к переменной `position`:

```
officeStaff1.position
```

Вывод:

```
'Basic'
```

Изменение переменной position с выводом нового значения:

```
#изменение переменной position
officeStaff1.position = 'Manager'

#повторный вывод position
officeStaff1.position
```

Вывод:

```
'Manager'
```

Обращение к переменной `pay`:

```
officeStaff1.pay
```

Вывод:

```
0
```

Использование метода `calculatePay()` для вычисления зарплаты:

```
officeStaff1.calculatePay()
```

Вывод:

```
Enter number of hours worked for Yvonne: 10
Enter the hourly rate for Yvonne: 15
```

Повторный вывод переменной `pay`:

```
officeStaff1.pay
```

Вывод:

```
150
```

Вывод строкового представления `officeStaff1`:

```
print(officeStaff1)
```

Вывод:

```
Position = Manager, Name = Yvonne, Pay = 150
```

Чтобы вывести строковое представление объекта, мы передаем имя этого объекта встроенной функции `print()`. При этом Python вызывает метод `__str__`, который мы написали ранее. В данном примере этот метод возвращает значения `position`, `name` и `pay` экземпляра `officeStaff1`.

9.4. СВОЙСТВА

От изучения основных концепций классов и объектов перейдем к рассмотрению свойств.

В приведенных выше примерах для обращения к переменным экземпляра использовался оператор «точка». Прямые обращения позволяют легко читать переменные и изменять их при необходимости. Тем не менее такая гибкость также порождает некоторые проблемы. Например, можно случайно изменить значение переменной `position` экземпляра `officeStaff1` и записать в нее несуществующую должность или присвоить переменной `pay` экземпляра `officeStaff1` некорректную сумму.

Для предотвращения подобных ошибок используются *свойства*. Они предоставляют возможность проверять изменения перед тем, как вносить их в экземпляры.

Чтобы показать, как работают свойства, мы определим свойство для переменной `position`. А именно добавленное свойство будет гарантировать, что переменной `position` может быть присвоено только значение `'Basic'` или `'Manager'`.

Но сначала нужно изменить имя переменной экземпляра `position` на `_position`. Добавление одного символа подчеркивания перед именем переменной — стандартный сигнал для других программистов, который означает, что к этой переменной не следует обращаться напрямую.

Программисты Python нередко произносят фразу: «Все мы тут люди взрослые и сознательные». Предполагается, что мы все будем вести себя ответственно, как и подобает взрослым. Добавление одного подчеркивания перед именем переменной сообщает другим программистам, что вы полагаетесь на то, что они будут вести себя сознательно и не станут обращаться к переменной напрямую, если только у них нет для этого веских причин. С технической точки зрения ничто не помешает им обратиться к этой переменной. При большом желании они все равно смогут обратиться к этой переменной:

```
officeStaff1._position
```

С учетом сказанного внесем следующие изменения в файл classdemo.py, чтобы другие «сознательные взрослые» знали, что они не должны обращаться к `position` напрямую.

Приведите строку

```
self.position = pPosition
```

в методе `__init__` к следующему виду:

```
self._position = pPosition
```

А строка

```
return "Position = %s, Name = %s, Pay = %d" %(self.
position, self.name, self.pay)
```

в методе `__str__` должна выглядеть так:

```
return "Position = %s, Name = %s, Pay = %d" %(self._
position, self.name, self.pay)
```

Теперь можно переходить к добавлению свойства для переменной `_position`.

Добавьте следующие строки в класс `Staff` из файла classdemo.py:

```
@property
def position(self):
    print("Getter Method")
    return self._position

@position.setter
def position(self, value):
    if value == 'Manager' or value == 'Basic':
        self._position = value
    else:
        print('Position is invalid. No changes made.')
```

Не забудьте об отступах при добавлении строк в класс `Staff`. Если отступов не будет, эти строки не будут принадлежать классу `Staff`.

Первая строка в этом фрагменте (`@property`) называется *декоратором*. Я не стану подробно объяснять, что такое декоратор; вкратце, он позволяет изменить функциональность следующего метода. В данном случае декоратор преобразует первый метод `position()` в свойство.

Этот декоратор сообщает компилятору, что каждый раз, когда пользователь вводит выражение

```
officeStaff1.position
```

для получения запрашиваемого значения должен использоваться метод `position()`, называемый *get-методом*.

Этот метод просто выводит сообщение «Getter Method» и возвращает значение переменной `_position`.

Благодаря декоратору `@property`, который преобразует метод в свойство, использовать выражение `officeStaff1.position()` для вызова метода уже не нужно. Мы просто обращаемся к нему без круглых скобок, как к переменной.

Далее идет другой декоратор `@position.setter`, за которым следует второй метод `position()`.

Этот декоратор сообщает компилятору, что, когда пользователи пытаются обновить значение `_position` командой вида

```
officeStaff1.position = 'Manager'
```

для обновления значения должен использоваться метод `position()`, который следует после декоратора.

Второй метод `position()` называется *set-методом*. Он содержит параметр с именем `value`, значение которого присваивается `_position` при условии, что это значение равно либо `'Manager'`, либо `'Basic'`. Если значение не совпадает ни с одним из этих двух, выводится сообщение «Position is invalid. No changes made».

Сохраните файл и запустите его снова.

Введите следующую команду в Shell:

```
officeStaff1 = Staff('Basic', 'Yvonne', 0)
```

Для обращения к переменной `position` экземпляра `officeStaff1` используется запись

```
officeStaff1.position
```

На выходе будет получен следующий результат:

```
Getter Method
'Basic'
```

Ранее при выполнении выражения

```
officeStaff1.position
```

мы обращались к переменной `position` напрямую. Теперь при выполнении выражения

```
officeStaff1.position
```

мы уже не обращаемся к переменной. Вместо этого мы вызываем get-метод свойства `position`. Это доказывает тот факт, что в выводе появляется дополнительная строка («Getter Method»).

Свойству присвоено имя `property` — исходное имя переменной до того, как она была переименована в `_position`, и это вовсе не случайное совпадение.

При такой схеме назначения имен пользователь может обращаться к должности (`position`) работника точно так же, как он делал это ранее, — выражением `officeStaff1.position`. И хотя мы внесли ряд изменений во внутреннюю реализацию classdemo.py, на конечных пользователях эти изменения не отразятся (если только они не попытаются задать недопустимое значение `position`).

Попробуем изменить значение `position` для экземпляра `officeStaff1`. Введите в оболочке Shell следующую команду:

```
officeStaff1.position = 'Manager'
```

Эта команда меняет значение свойства `position` на `'Manager'`.

Чтобы убедиться в этом, введите команду

```
officeStaff1.position
```

Вы получите следующий результат:

```
Getter Method
'Manager'
```

Теперь попробуем задать `position` недопустимое значение «CEO». Введите в Shell следующую команду:

```
officeStaff1.position = 'CEO'
```

На этот раз попытка завершается неудачей, а программа выдает следующий результат:

```
Position is invalid. No changes made.
```

Этот результат демонстрирует, что set-метод не позволил изменить свойство `position` и присвоить ему недопустимое значение. Чтобы убедиться, что значение `position` осталось неизменным, снова введите команду

```
officeStaff1.position
```

Результат:

```
Getter Method
'Manager'
```

9.5. КОРРЕКТИРОВКА ИМЕН

Перейдем к обсуждению концепции корректировки (mangling) имен в Python.

Я уже говорил о том, что если мы не хотим, чтобы другие программисты обращались к некоторой переменной напрямую, то следует поставить перед именем переменной один символ подчеркивания (_). Далее пишутся свойства, управляющие доступом к этим переменным. Но даже в этом случае другие программисты при большом

желании смогут обратиться к переменной. В предыдущем примере для этого достаточно использовать выражение

```
officeStaff1._position
```

В языке Python нет возможности надежно скрыть переменную и помешать другим пользователям обратиться к ней. Но если вы хотите действительно убедительно сообщить другим программистам, что они не должны изменять некоторую переменную, поставьте перед именем переменной два символа подчеркивания (__).

Например, попробуйте ввести следующий код в оболочке Python:

```
class A:
    def __init__(self):
        self.__x = 5
        self._y = 6
```

Этот фрагмент определяет класс с именем A. Класс содержит две переменные: __x (с двумя подчеркиваниями) и _y (с одним подчеркиванием).

Теперь дважды нажмите Enter и введите следующую команду, чтобы создать экземпляр класса A:

```
varA = A()
```

Теперь попробуйте обратиться к двум переменным. Введите команду

```
varA._y
```

Вы получите результат

```
6
```

Но при попытке выполнить команду

```
varA.__x
```

произойдет ошибка.

Почему? Дело в том, что при добавлении двух символов подчеркивания перед именем переменной Python выполняет так называемую *корректировку имен*. Обнаружив переменную с двумя символами подчеркивания, компилятор Python преобразует имя, добавляя перед ним одиночный символ подчеркивания и имя класса. Другими словами, имя `__x` преобразуется в `_A__x`.

Таким образом, при добавлении двойного символа подчеркивания перед именем переменной другие программисты не смогут обратиться к переменной, просто указывая ее имя (`__x` в данном случае.) Это означает, что им будет труднее обратиться к переменной. Но при желании они все же смогут получить доступ к переменной — для этого достаточно ввести выражение

```
varA._A__x
```

Результат будет равен 5.

Иначе говоря, в Python невозможно гарантированно ограничить доступ к переменной. Символы подчеркивания перед именем усложняют обращение, но упор-

ный программист все равно сможет получить доступ к значению. Как я уже говорил, предполагается, что программисты Python — люди взрослые и сознательные. Ожидается, что мы будем вести себя ответственно и не станем изменять переменные, которые нам изменять не положено.

9.6. ЧТО ТАКОЕ SELF

Итак, теперь вы понимаете, как работают классы. Пора разобраться, что же означает ключевое слово `self`.

Чтобы объяснить смысл `self`, необходимо сначала обсудить концепцию переменных класса и переменных экземпляра.

Переменная класса принадлежит классу в целом и совместно используется всеми экземплярами этого класса. Она определяется за пределами любых методов, определяемых в классе.

С другой стороны, переменная экземпляра определяется внутри метода и принадлежит конкретному экземпляру. При обращении к ней всегда используется префикс `self`.

Рассмотрим пример. Допустим, Питер и Джон работают в компании *ProgrammingLab*. Для хранения этой информации можно создать класс `ProgStaff`. Создадим новый файл с именем selfdemo.py, который содержит следующий код:

```
class ProgStaff:
    companyName = 'ProgrammingLab'

    def __init__(self, pSalary):
        self.salary = pSalary

    def printInfo(self):
        print("Company name is", ProgStaff.companyName)
        print("Salary is", self.salary)

peter = ProgStaff(2500)
john = ProgStaff(2500)
```

В первых строках этого кода определяется класс с именем `ProgStaff`. Этот класс содержит переменную `companyName`, которая не определяется внутри какого-либо метода.

Также он содержит метод `__init__`. Внутри метода `__init__` определяется переменная с именем `salary`. При обращении к этой переменной используется префикс `self`.

Наконец, класс также содержит метод `printInfo()`, который получает `self` в числе параметров. Этот метод просто выводит значение `companyName` и `salary`.

После определения класса создаются два экземпляра класса `ProgStaff` с именами `peter` и `john`. Команды создания экземпляров не снабжаются отступами, так как они не являются частью класса `Staff`.

Чем же переменная класса отличается от переменной экземпляра?

Сейчас компания называется *ProgrammingLab*. Допустим, название компании было изменено на *ProgrammingSchool* и объект был обновлен:

```
ProgStaff.companyName = 'ProgrammingSchool'
```

Обратите внимание: теперь перед переменной `companyName` приходится ставить префикс `ProgStaff`. Изменение отразится на всех экземплярах класса `ProgStaff` (в данном случае `peter` и `john`).

Чтобы убедиться в этом, включите следующие строки в файл selfdemo.py:

```
ProgStaff.companyName = 'ProgrammingSchool'
print(peter.companyName)
print(john.companyName)
```

Обратите внимание: в этом коде команды не вводятся непосредственно в Python Shell. Чтобы вывести значение `companyName` для экземпляров `peter` и `john`, придется воспользоваться функцией `print()`.

Сохраните файл и запустите программу. Вы получите следующий результат:

```
ProgrammingSchool
ProgrammingSchool
```

Теперь предположим, что зарплата Питера увеличилась до 2700. Команда приводится к следующему виду:

```
peter.salary = 2700
```

Изменение затрагивает только экземпляр `'peter'`. Чтобы убедиться в этом, выведите значения `salary` для экземпляров `peter` и `john`. Добавьте следующие строки в файл selfdemo.py и выполните программу:

```
peter.salary = 2700
print(peter.salary)
print(john. salary)
```

Результат выглядит так:

```
2700
2500
```

Как видите, зарплата изменилась только для экземпляра `peter`.

Итак, главные различия между переменными классов и переменными экземпляров сводятся к следующему.

Переменная класса

1. Переменная класса определяется за пределами любых методов класса.
2. К переменной класса можно обращаться за пределами класса с указанием имени класса.
3. Изменение переменной класса влияет на все экземпляры класса.

Переменная экземпляра

1. Переменная экземпляра определяется внутри метода класса, а при обращении к ней используется префикс `self`.

2. К переменной экземпляра можно обращаться за пределами класса с указанием имени экземпляра.
3. Изменение переменной экземпляра влияет только на конкретный экземпляр.

В нашем примере `companyName` — переменная класса, а `salary` — переменная экземпляра.

Итак, теперь вы понимаете, что такое переменные классов и переменные экземпляров. Перейдем к обсуждению метода `printInfo()` в классе `ProgStaff`.

Этот метод называется *методом экземпляра*. У метода экземпляра один из параметров содержит `self`. Если метод имеет несколько параметров, `self` должен быть первым параметром.

Внутри метода для обращения к переменной класса `companyName` используется имя класса. С другой стороны, для обращения к переменной экземпляра `salary` используется ключевое слово `self`.

Ключевое слово `self` представляет экземпляр класса. Так как разные экземпляры имеют разные имена и на момент написания программы эти имена еще неизвестны (поскольку они еще не были созданы), для представления их внутри класса используется ключевое слово `self`.

Чтобы вызвать метод `printInfo()`, введите команду

```
john.printInfo()
```

Когда метод экземпляра вызывается подобным образом, Python неявно передает `john` как значение параметра `self`. Делать вам это вручную не нужно.

Кроме того, при желании для метода экземпляра можно использовать имя класса. Но в таком случае экземпляр `john` придется передавать самостоятельно:

```
ProgStaff.printInfo(john)
```

Оба метода выдают одинаковый результат. Попробуйте добавить две команды в selfdemo.py и запустить программу. Вы получите следующий результат:

```
Company name is ProgrammingSchool
Salary is 2500
Company name is ProgrammingSchool
Salary is 2500
```

9.7. МЕТОДЫ КЛАССА И СТАТИЧЕСКИЕ МЕТОДЫ

Итак, теперь вы понимаете смысл ключевого слова `self` и мы можем перейти к обсуждению методов класса и статических методов.

В предыдущем разделе упоминалось, что метод экземпляра представляет собой метод, получающий `self` в параметре. Это самая распространенная разновидность методов, и только такие методы использовались в примерах до настоящего момента.

Кроме методов экземпляров в Python также поддерживаются методы классов и статические методы. Эти разновидности методов используются довольно редко, поэтому здесь мы ограничимся кратким рассмотрением. Для начала создайте файл с именем methoddemo.py. Добавьте в файл следующий код:

```
class MethodDemo:

    a = 1

    @classmethod
    def classM(cls):
        print("Class Method. cls.a = ", cls.a)

    @staticmethod
    def staticM():
        print("Static method")
```

Этот класс содержит переменную класса `a` и два метода — `classM()` и `staticM()`.

Первый метод, `classM()`, является методом класса.

Чтобы определить метод класса, необходимо воспользоваться декоратором `@classmethod`. Он сообщает Python, что далее следует метод, который является методом класса.

Метод класса получает в первом параметре объект класса (вместо `self`). Для представления этого объекта класса обычно используется обозначение `cls`.

`cls` можно рассматривать как своего рода аналог `self`. Главное отличие заключается в том, что имя `self` относится к экземпляру, тогда как `cls` относится к классу. Так как

имя `cls` относится к самому классу, оно может использоваться для обращения к переменным класса. В данном примере оно используется для обращения к переменной класса `a`.

Для вызова метода класса можно использовать как имя класса, так и имя экземпляра. В обоих случаях Python автоматически передает класс в первом аргументе метода.

Например, чтобы вызвать `classM()` в приведенном примере, можно использовать следующее выражение:

```
MethodDemo.classM()
```

Также можно создать объект `MethodDemo` и использовать его для вызова метода, как показано ниже:

```
md1 = MethodDemo()
md1.classM()
```

Добавьте эти три команды в файл `methoddemo.py` и запустите программу. Вы получите следующий результат:

```
Class Method. cls.a =  1
Class Method. cls.a =  1
```

Кроме методов экземпляра и методов класса в Python также поддерживаются статические методы. При вызове *статического метода* не нужно передавать экземпляр или класс. Первый параметр статического метода не содержит `self` или `cls`. Для определения статического метода используется декоратор `@staticmethod`. Чтобы вызвать статический метод, можно указать либо имя класса, либо

имя экземпляра. Например, для вызова `staticM()` в приведенном примере используется вызов

```
md1.staticM()
```

или

```
MethodDemo.staticM()
```

В обоих случаях будет получен одинаковый результат:

```
Static method
```

Методы класса и статические методы используются не так часто. В большинстве случаев для класса Python вполне хватает методов экземпляров.

9.8. ИМПОРТИРОВАНИЕ КЛАССА

В этой главе были рассмотрены многие объектно-ориентированные концепции. Прежде чем переходить к следующей главе, посмотрим, как импортировать класс в приложение.

По аналогии с тем, что было сказано в главе 7 для модулей, класс также можно создать как отдельный файл и импортировать его в приложение. Для этого необходимо сохранить класс в отдельном файле с расширением .py. После этого остается импортировать класс по имени файла.

Например, представьте, что в файле с именем myclass.py хранится следующий код:

```
class SomeClass:
     def __init__(self):
        print('This is SomeClass')
    def someMethod(self, a):
        print('The value of a is', a)
        self.b = 5

class SomeOtherClass:
    def __init__(self):
        print('This is SomeOtherClass')
```

Файл состоит из двух классов: `SomeClass` и `SomeOtherClass`. Вы можете создать другой класс .py и импортировать два класса по имени файла (`myclass`). После этого это имя файла используется для обращения к классам из этого файла.

Создайте другой файл с именем importdemo.py. Включите в файл приведенный ниже код и запустите его:

```
import myclass

sc = myclass.SomeClass()
sc.someMethod(100)
soc = myclass.SomeOtherClass()
```

Вы получите следующий результат:

```
This is SomeClass
The value of a is 100
This is SomeOtherClass
```

В этом коде при создании экземпляров перед именем класса указывалось имя файла (например, `myclass.SomeClass()`), чтобы компилятор знал, что `SomeClass` находится в файле myclass.py.

Также можно импортировать класс следующей командой:

```
from myclass import SomeClass, SomeOtherClass
```

Если вы выберете этот вариант, то при создании экземпляра перед именем файла не нужно указывать имя файла. Например, для создания объекта `SomeClass` можно будет использовать запись

```
sc = SomeClass()
```

10

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, ЧАСТЬ II

А теперь перейдем к более сложным темам ООП. В этой главе мы узнаем об операторах наследования, полиморфизме и перегрузке операторов.

10.1. НАСЛЕДОВАНИЕ

Наследование — одна из ключевых концепций ООП. Наследование позволяет создать новый класс из существующего, чтобы можно было эффективно переиспользовать существующий код.

10.2. НАПИСАНИЕ ПРОИЗВОДНОГО КЛАССА

Чтобы понять, как работает наследование, расширим класс `Staff` из главы 9. Напомню, что класс `Staff` содержит следующие атрибуты:

Переменные экземпляра

```
_position

name
pay
```

Методы

```
__init__
__str__
calculatePay()
```

Этот класс хорошо работал, когда он использовался для вычисления зарплаты линейного сотрудника с почасовой ставкой, как показано в главе 9.

Но представьте, что кроме линейных сотрудников в компании также есть менеджер, труд которого оплачивается по другой схеме. Кроме почасовой оплаты он также получает фиксированную доплату. Как изменить класс, чтобы в нем учитывалось это обстоятельство?

Лучше всего создать *подкласс* (подклассы также называются *производными классами*). Класс, на основе которого создается производный класс, называется *базовым классом*, *суперклассом* или *родительским классом*.

Главная особенность подкласса заключается в том, что он наследует все переменные и методы от родительского класса, т. е. может использовать эти переменные и методы так, как если бы они были частью его кода, и ему не придется определять их заново. Кроме того, подкласс может содержать дополнительные переменные и методы, не существующие в родительском классе. Посмотрим, как это делается.

Мы создадим подкласс с именем `ManagementStaff` для представления менеджера. Включите следующую строку в файл classdemo.py:

```
class ManagementStaff(Staff):
```

Здесь создается новый класс с именем `ManagementStaff`. Чтобы указать, что `ManagementStaff` является подклассом по отношению к `Staff`, имя `Staff` указывается в круглых скобках после имени класса.

А теперь напишем метод `__init__` для подкласса.

Одна из главных причин для создания подклассов — упрощение повторного использования кода. В этом вам поможет встроенная функция с именем `super()`.

Посмотрим, как использовать эту функцию. В главе 9 мы уже написали метод `__init__` для класса `Staff`. Этот метод инициализирует три переменные экземпляров `_position`, `name` и `pay`. А теперь этот метод будет использован в подклассе.

Включите следующий метод в класс `ManagementStaff`:

```
def __init__ (self, pName, pPay, pAllowance, pBonus):
    super().__init__('Manager', pName, pPay)
    self.allowance = pAllowance
    self.bonus = pBonus
```

Метод `__init__` в классе `ManagementStaff` имеет пять параметров — `self`, `pName`, `pPay`, `pAllowance` и `pBonus`.

Внутри метода в первой строке функция `super()` используется для вызова метода `__init__` из базового класса. Функция `super()` — встроенная функция, которая может использоваться в подклассах для вызова метода из суперкласса.

В Python 3 при использовании функции `super()` для вызова метода из родительского класса передавать значение для параметра `self` не нужно.

В нашем примере методу `__init__` базового класса достаточно передать три значения (строка `'Manager'`, а также параметры `pName` и `pPay`). Вызывается метод базового класса, строка `'Manager'` присваивается `_position`, а `pName` и `pPay` будут присвоены `name` и `pay` соответственно.

Если вы работаете в Python 2, синтаксис использования `super()` будет несколько другим. При использовании `super()` в Python 2 необходимо передать имя подкласса и ключевое слово `self`. Другими словами, в нашем примере должна использоваться запись

```
super(ManagementStaff, self).__init__('Manager',
      pName, pPay)
```

Этот синтаксис продолжает поддерживаться в Python 3, и некоторые программисты предпочитают придерживаться его для сохранения совместимости.

Кроме функции `super()` также можно воспользоваться именем базового класса для вызова метода этого класса. Для этого используется запись

```
Staff.__init__(self, 'Manager', pName, pPay)
```

Все три способа в данном случае приводят к одному результату. У каждого способа есть свои плюсы и минусы, и споры о том, какой вариант лучше, выходят за рамки книги. В большинстве случаев выбор в основном зависит от личных предпочтений.

После вызова метода `__init__` базового класса метод `__init__` подкласса использует две следующие команды

для присваивания параметров `pAllowance` и `pBonus` переменным экземпляра `allowance` и `bonus` соответственно:

```
self.allowance = pAllowance
self.bonus = pBonus
```

Эти переменные экземпляра существуют только в подклассе; в суперклассе их нет. Собственно, это вся реальная работа, которая выполняется в методе `__init__` подкласса.

А теперь напишем метод для вычисления зарплаты менеджера. Включите следующий метод в класс `ManagementStaff`:

```
def calculatePay(self):
    basicPay = super().calculatePay()
    self.pay = basicPay + self.allowance
    return self.pay
```

Обратите внимание: этот метод снова использует функцию `super()` для вызова метода `calculatePay()` базового класса. После вызова метода базового класса результат присваивается переменной `basicPay`. Эта локальная переменная существует только внутри метода `calculatePay()`. Следовательно, перед ее именем необходимо поставить префикс `self`.

Теперь значение `basicPay` суммируется с переменной экземпляра `allowance` для вычисления общей зарплаты менеджера. Результат присваивается переменной экземпляра `pay` и возвращается в следующей команде.

Вот и все, что делает метод `calculatePay()` в подклассе.

Помните, ранее я упоминал о том, что подкласс наследует все переменные и методы из базового класса? Это означает, что писать новый метод `calculatePay()` для подкласса не обязательно; он уже существует. Но когда вы пишете новую версию метода для подкласса, новая версия заменяет версию, унаследованную подклассом от родителя. Такое замещение называется *переопределением*; именно эта возможность была использована в рассмотренном примере.

Добавим в подкласс новый метод. Допустим, если оценка эффективности менеджера равна «A», ему также полагается премия по результатам работы. Для вычисления премии в класс `ManagementStaff` добавляется новый метод:

```
def calculatePerfBonus(self):
    prompt = 'Enter performance grade for %s: '
             %(self.name)
    grade = input(prompt)
    if (grade == 'A'):
        self.bonus = 1000
    else:
        self.bonus = 0
    return self.bonus
```

Метод предлагает пользователю ввести оценку эффективности для менеджера, после чего присваивает переменной экземпляра `bonus` 1000 или 0 в зависимости от введенной оценки. Затем в следующей инструкции (`return`) возвращает значение премии.

На этом класс `ManagementStaff` можно считать завершенным. Класс состоит из следующих компонентов:

Переменные экземпляра

- Унаследованные от `Staff`:
  - `_position`
  - `name`
  - `pay`
- Объявленные в `ManagementStaff`:
  - `allowance`
  - `bonus`

Методы

- Унаследованные и непереопределенные:
  - `__str__`
- Унаследованные и переопределенные:
  - `__init__`
  - `calculatePay()`
- Объявленные в `ManagementStaff`:
  - `calculatePerfBonus()`

Прежде чем переходить к следующему разделу, мы создадим еще один подкласс, производный от `Staff`. На этот раз производный класс будет называться `BasicStaff`. Код класса приведен ниже:

```
class BasicStaff(Staff):
    def __init__(self, pName, pPay):
        super().__init__('Basic', pName, pPay)
```

Подкласс переопределяет только метод `__init__` базового класса. Метод `__init__` передает строку `'Basic'` инициализатору базового класса, чтобы значение переменной экземпляра `_position` присваивалось автоматически при создании экземпляра `BasicStaff`. В остальном подкласс наследует все переменные и методы от базового класса. Класс состоит из следующих компонентов:

Переменные экземпляра

- Унаследованные от `Staff`:
  - `_position`
  - `name`
  - `pay`

Методы

- Унаследованные и непереопределенные:
  - `__str__`
  - `calculatePay()`
- Унаследованные и переопределенные:
  - `__init__`

10.3. СОЗДАНИЕ ЭКЗЕМПЛЯРА ПРОИЗВОДНОГО КЛАССА

Разобравшись с производными классами, создадим отдельный файл .py, в котором будут использоваться эти классы.

Создайте новый файл в IDLE и присвойте ему имя inheritancedemo.py. Включите в файл следующие строки:

```
import classdemo

peter = classdemo.BasicStaff('Peter', 0)
john = classdemo.ManagementStaff('John', 0, 1000, 0)

print(peter)
print(john)

print('Peter\' s Pay = ', peter.calculatePay())

print('John\'s Pay = ', john.calculatePay())
print('John\'s Performance Bonus = ', john.
      calculatePerfBonus())
```

В этом коде сначала три класса (`Staff`, `ManagementStaff` и `BasicStaff`) импортируются командой

```
import classdemo
```

Затем создаются два экземпляра производных классов (`peter` и `john`) для двух подклассов `BasicStaff` и `ManagementStaff` соответственно.

Для экземпляра `peter` передаются значения `'Peter'` и `0` для переменных экземпляров `name` и `pay` соответственно.

Для экземпляра `john` передаются значения `'John'`, `0`, `1000` и `0` для переменных экземпляров `name`, `pay`, `allowance` и `bonus` соответственно.

Затем два объекта используются для вызова метода `__str__`.

Хотя мы не программировали метод `__str__` в двух подклассах, этот метод все равно можно использовать, потому что оба подкласса унаследовали метод от базового класса. Это упрощает повторное использование кода, так как нам не приходится переписывать метод `__str__` для обоих производных классов.

После вызова метода `__str__` программа вызывает метод `calculatePay()` для `peter` и `john` и выводит информацию на экран. Наконец, для экземпляра `john` вызывается метод `calculatePerfBonus()`. Если запустить программу и ввести запрашиваемую информацию, будет получен следующий вывод:

```
Creating Staff object
Creating Staff object
Position = Basic, Name = Peter, Pay = 0
Position = Manager, Name = John, Pay = 0
Enter number of hours worked for Peter: 120
Enter the hourly rate for Peter: 15
Peter's Pay =  1800
Enter number of hours worked for John: 150
Enter the hourly rate for John: 20
John's Pay =  4000
Enter performance grade for John: A
John's Performance Bonus =  1000
```

Так как `peter` использует метод `calculatePay()`, унаследованный от базового класса, зарплата (`pay`) для экземпляра `peter` составляет 1800 (т. е. `120*15`). С другой стороны, для `john` зарплата составляет 4000, так как для ее вычисления используется метод `calculatePay()` подкласса `ManagementStaff`. Метод `calculatePay()` из подкласса `ManagementStaff` переопределяет метод в базовом

классе. В результате зарплата составляет `150*20 + 1000` (т. е. к зарплате добавляется премия `1000`).

После вычисления зарплаты для экземпляров `peter` и `john` мы вычисляем премию для экземпляра `john`. Так как его оценка эффективности равна «A», ему начисляется премия `$1000`.

Если вы попытаетесь использовать экземпляр `peter` для вызова метода `calculatePerfBonus()`, произойдет ошибка, потому что класс `BasicStaff` не содержит метода `calculatePerfBonus()`. Попробуйте включить строку

```
print('Peter\'s Performance Bonus = ', peter.
      calculatePerfBonus())
```

в файл inheritancedemo.py и посмотрите, что получится. Вы получите сообщение об ошибке.

10.4. СПЕЦИАЛЬНЫЕ МЕТОДЫ PYTHON

От наследования и переопределения мы переходим к специальным методам.

Ранее, в главе 9, упоминалось о том, что в Python имеется множество специальных методов. Эти методы (они также называются волшебными) всегда заключаются в двойные символы подчеркивания. До настоящего момента вам уже встречались два таких метода: `__init__` и `__str__`.

Одна из особенностей специального метода состоит в том, что они не вызываются напрямую. Например, если вы

хотите вывести информацию об экземпляре `officeStaff1` из главы 9, не нужно использовать громоздкую запись

```
print(officeStaff1.__str__())
```

Вместо этого можно ограничиться выражением

```
print(officeStaff1)
```

и Python автоматически вызовет нужный специальный метод.

У специальных методов есть еще одна особенность: их можно переопределять для ваших потребностей. Метод `__str__` обычно переопределяется для создания более понятного представления класса в строковом виде.

Также часто переопределяются методы `__add__`, `__sub__`, `__mul__` и `__div__`. Эти методы позволяют изменить поведение стандартных операторов (таких, как `+`, `-`, `*` и `/`), чтобы они выполняли разные операции в зависимости от данных, к которым они применяются.

Вы уже знаете, что знак `+` может означать как сложение, так и конкатенацию. Например, в выражении

```
2+3
```

оператор `+` суммирует два числа и мы получаем результат 5.

Но если ввести выражение

```
'Hello' + ' World'
```

оператор + выполняет конкатенацию двух строк, а результатом является строка 'Hello World'.

У каждого оператора существует свой специальный метод. Вы можете переопределить соответствующий специальный метод, чтобы расширить смысл оператора. Специальные методы для операторов +, -, *, / называются __add__, __sub__, __mul__ и __div__ соответственно.

Попробуем перегрузить оператор + для класса Staff.

Добавьте следующий метод в класс classdemo.py. Мы включим его в класс Staff, чтобы этот метод мог использоваться обоими производными классами:

```
def __add__(self, other):
    return self.pay + other.pay
```

Переопределенная версия метода __add__ содержит два параметра: self и other. Параметр self ссылается на один экземпляр, параметр other — на другой. Фактически вы приказываете компилятору сложить переменные pay двух экземпляров и вернуть полученный результат.

Для вызова метода используется оператор +.

Добавьте следующие строки в файл inheritancedemo.py:

```
totalPay = john + peter
print('\nTotal Pay for Both Staff = ', totalPay)
```

Запустите программу и по запросу введите значения 120, 15, 150, 20 и A. Вы получите такой же результат,

как в предыдущем разделе, но с одной дополнительной строкой:

```
Total Pay for Both Staff =  5800
```

10.5. ВСТРОЕННЫЕ ФУНКЦИИ PYTHON ДЛЯ РАБОТЫ С ОБЪЕКТАМИ

Мы обсудили большую часть тем, относящихся к ООП. Прежде чем завершить эту главу, рассмотрим некоторые встроенные функции Python для работы с объектами. Мы исследуем эти функции на примере двух классов, приведенных ниже. Создайте новый файл в IDLE и назовите его objectfunctions.py. Скопируйте в файл следующий код:

```
class ParentClass:
    def __init__(self):
        self.a = 1
        print("Parent Class Object Created")
     def someMethod(self):
        print("Hello")

class ChildClass(ParentClass):
    def __init__(self):
        print("Child Class Object Created")

parent = ParentClass()
child = ChildClass()
```

С этими классами можно протестировать некоторые встроенные функции Python. Выполните файл, чтобы опробовать следующие команды в Python Shell.

- `isinstance()`

Функция получает два аргумента. Она проверяет, является ли первый аргумент экземпляром второго аргумента (или экземпляром подкласса второго аргумента). Вторым аргументом может быть класс или встроенный тип Python. Также им может быть кортеж, состоящий из нескольких классов или типов.

Если второй аргумент не является действительным классом или типом (или кортежем классов или типов), выдается исключение.

Попробуйте ввести в оболочке Shell следующие команды:

Пример 1

```
isinstance(parent, ParentClass)
```

Вывод:

`True`, так как `parent` является экземпляром `ParentClass`.

Пример 2

```
isinstance(5, int)
```

Вывод:

`True`, так как 5 является экземпляром `int`. (`int` — встроенный тип для представления целых чисел в Python.)

Пример 3

```
isinstance(child, ParentClass)
```

Вывод:

`True`, так как `child` является экземпляром класса `ChildClass`, в свою очередь, являющегося подклассом по отношению к `ParentClass`.

Пример 4

```
isinstance(parent, (ParentClass, int))
```

Вывод:

`True`, так как `parent` является экземпляром `ParentClass` — одного из типов, содержащихся в кортеже `(ParentClass, int)`.

Пример 5

```
isinstance(parent, ChildClass)
```

Вывод:

`False`, так как экземпляр базового класса не считается экземпляром производного класса.

Пример 6

```
isinstance(parent, MyClass)
```

Вывод:

Ошибка `NameError`: имя `'MyClass'` не определено, так как `MyClass` не является действительным классом. Для обработки таких исключений можно использовать следующий код:

```
try:
    isinstance(parent, MyClass)
except NameError:
    print("No such class")
```

Результат:

```
No such class
```

- `issubclass()`

Функция получает два аргумента — имена двух классов или типов. Она проверяет, является ли первый аргумент подклассом второго аргумента. Второй аргумент может содержать кортеж. Функция возвращает `True`, если первый аргумент является подклассом любого из классов или типов, содержащихся в кортеже. Если второй аргумент не является действительным классом или типом, выдается исключение.

Например, при выполнении фрагмента

```
issubclass(ChildClass, ParentClass)
issubclass(ParentClass, ParentClass)
issubclass(ChildClass, int)
issubclass(ChildClass, (int, ParentClass))
```

будет получен следующий результат:

```
True
True
False
True
```

Вторая команда возвращает `True`, так как класс считается подклассом самого себя:

- `hasttr()`

Функция проверяет, содержит ли экземпляр заданный атрибут. Под *атрибутом* могут пониматься как данные (переменные), так и методы. Функция получает два аргумента: имя объекта и имя атрибута. Имя атрибута задается в виде строки и как следствие должно заключаться в одинарные апострофы. Например, если добавить в Shell следующие строки:

```
hasattr(parent, 'a')
hasattr(parent, 'someMethod')
hasattr(parent, 'b')
```

будет получен следующий результат:

```
True
True
False
```

Дело в том, что `parent` содержит атрибуты `'a'` и `'someMethod'`, но не содержит атрибут `'b'`.

11

ПРОЕКТ: MATHEMATICS И BINARY

Поздравляю! Мы рассмотрели достаточно основных концепций Python (и программирования вообще), чтобы написать первую полноценную программу. В этой главе мы напишем простую консольную игру, которая состоит из двух мини-игр — Mathematics и Binary.

В начале игры пользователь выбирает одну из двух игр. Кроме того, ему предлагается выбрать количество вопросов на раунд (от 1 до 10).

Первая игра проверяет понимание правила BODMAS[1] при арифметических вычислениях. Если вы не знаете, что такое BODMAS, почитайте статью https://en.wikipedia.org/wiki/Order_of_operations.

Во второй игре нужно преобразовать число из десятичной системы (с основанием 10) в двоичную (с основанием 2). Если у вас нет опыта работы с двоичными числами, обращайтесь к статье https://ru.wikipedia.org/wiki/Двоичная_система_счисления.

---

[1] BODMAS (Brackets, Division, Multiplication, Addition, and Subtraction) — скобки, деление, умножение, сложение и вычитание. Акроним для запоминания старшинства операций. — *Примеч. ред.*

В обоих случаях игра случайно выбирает вопрос для игрока. Если игрок дал неправильный ответ, программа выводит правильный ответ и переходит к следующему вопросу. Если же игрок ответил правильно, программа обновляет его счет и переходит к следующему вопросу.

Программа ведет учет счетов разных игроков и сохраняет их во внешнем текстовом файле. После каждого раунда игрок может ввести `-1`, чтобы завершить игру, или нажать Enter для начала нового раунда.

Я разбил программу на небольшие упражнения, чтобы вы смогли работать над ней самостоятельно. Попробуйте выполнить упражнения, прежде чем переходить к ответам.

Ответы даны в приложении Д; также можно загрузить файлы Python на странице https://www.learncodingfast.com/python. Настоятельно рекомендую загрузить исходный код, так как форматирование в приложении «Документ» может привести к искажению некоторых отступов, что усложнит чтение кода.

Помните: читать описание синтаксиса Python просто, но скучно. Самое интересное происходит тогда, когда вы начинаете решать задачи. Если при работе над упражнениями у вас возникнут сложности, постарайтесь подумать еще. Именно здесь ваши усилия будут вознаграждены наиболее достойно.

Готовы? Тогда вперед!

11.1. GAMETASKS.PY

Мы напишем для программы три файла: gametasks.py, gameclasses.py и project.py. В части 1 мы сосредоточимся на написании кода для gametasks.py.

Для начала создайте новый файл в IDLE и присвойте ему имя gametasks.py.

Модуль gametasks.py содержит три функции, выполняющие определенные операции в нашем проекте. Этим функциям не нужна никакая информация о конкретном экземпляре класса, поэтому они не определяются внутри класса.

УПРАЖНЕНИЕ 1.1. ВЫВОД ИНСТРУКЦИЙ

Первая функция называется `printInstructions()`. Она получает один параметр с именем `instruction` и использует встроенную функцию `print()` для вывода значения `instruction` на экран. Попробуйте запрограммировать эту функцию самостоятельно.

УПРАЖНЕНИЕ 1.2. ПОЛУЧЕНИЕ СЧЕТА ПОЛЬЗОВАТЕЛЯ

Вторая функция называется `getUserScore()`. Она получает один параметр с именем `userName`.

Функция сначала открывает файл userScores.txt в режиме `'r'`. Содержимое файла выглядит примерно так:

```
Ann, 100
Benny, 102
Carol, 214
Darren, 129
```

Каждая строка содержит информацию об одном пользователе. Каждая строка состоит из двух полей: имени пользователя и его игрового счета.

Затем функция читает файл строку за строкой в цикле `for`.

В цикле `for` каждая строка разбивается на поля функцией `split()` (пример использования функции `split()` приведен в приложении А), а результаты функции `split()` сохраняются в списке с именем `content`. Попробуйте запрограммировать этот фрагмент самостоятельно.

Далее (все еще внутри цикла `for`) функция проверяет, содержит ли какая-либо из строк имя пользователя, совпадающее с переданным в параметре. Если имена совпадают, функция закрывает файл и возвращает значение счета из этой строки.

Если после перебора функция не находит совпадение для имени пользователя, цикл `for` завершается, функция закрывает файл и возвращает строку `-1`.

Пока понятно? Попробуйте написать эту функцию.

Получилось?

Теперь необходимо внести некоторые изменения в код. Ранее при открытии файла использовался режим `'r'`.

Этот режим предотвращает любые случайные изменения в файле. Однако если файл не существует, то при попытке открыть его в режиме `'r'` происходит ошибка `IOError`. А значит, при первом запуске программы произойдет ошибка, потому что файл userScores.txt еще не был создан. Существует пара способов предотвращения этой ошибки.

Вместо того чтобы открывать файл в режиме `'r'`, можно открыть его в режиме `'w'`. Если при попытке открытия в режиме `'w'` файл не существует, он будет создан. Правда, тогда появляется опасность того, что метод может случайно выполнить запись в файл, что приведет к стиранию всего существующего содержимого. Но, так как наша программа очень мала, можно просто тщательно проверить код и убедиться в том, что в нем не выполняются нежелательные операции записи.

Во втором варианте ошибка `IOError` обрабатывается конструкцией `try-except`. Для этого весь предшествующий код необходимо заключить в блок `try`, а затем воспользоваться блоком `except IOError:` для обработки ошибки «Файл не найден». В блоке `except` программа сообщает пользователям, что файл не найден, после чего переходит к его созданию. Для создания файла будет использоваться функция `open()` в режиме `'w'`. Этот вариант отличается от предыдущего тем, что режим `'w'` используется только в том случае, если файл не найден. Так как файл не существует изначально, исчезает риск случайной потери предыдущего содержимого. После создания файл закрывается и возвращается строка `-1`.

Попробуйте реализовать этот вариант самостоятельно. Вы можете выбрать любой из двух вариантов, описанных выше. В приведенном ответе используется второй вариант. Когда это будет сделано, переходите к упражнению 1.3.

УПРАЖНЕНИЕ 1.3. ОБНОВЛЕНИЕ СЧЕТА ПОЛЬЗОВАТЕЛЯ

В этом упражнении мы определим очередную функцию с именем `updateUserScore()`.

Для этой функции нам понадобятся две встроенные функции из модуля `os`: `remove()` и `rename()`.

Попробуйте импортировать эти две функции самостоятельно.

Получилось? Двигаемся дальше.

Функция `updateUserScore()` получает три параметра: `newUser`, `userName` и `score`. Добавьте эти параметры в определение функции.

Параметр `newUser` может принимать значения `True` или `False`.

Если `newUser` содержит `True`, то функция открывает файл userScores.txt в режиме присоединения и добавляет значения `userName` и `score` нового пользователя функцией `write()`.

После этого функция закрывает файл. Попробуйте написать этот блок `if` самостоятельно.

Если параметр `newUser` содержит `False`, то функция обновляет счет пользователя в файле. Тем не менее в Python (и в большинстве других языков программирования, если на то пошло) нет функции для обновления текстовых файлов. В файл можно записывать данные или присоединять их, но не обновлять.

А значит, необходимо создать временный файл. Это довольно стандартная практика в программировании. Назовем этот файл `userScores.tmp`. Напомню, что новый файл можно создать, открывая его в режиме `'w'`. Создайте этот файл (это должно происходить в блоке `else`).

Готово?

В том же блоке `else` откройте файл userScores.txt в режиме `'r'`, потому что мы будем только читать из него данные. После этого содержимое userScores.txt перебирается по строкам в цикле `for` и каждая строка разбивается на поля функцией `split()`. Результат `split()` присваивается списку с именем `content`.

Для каждой строки следует проверить, совпадает ли имя пользователя в строке с именем, переданным в параметре `userName`. Если имена совпадают, то счет заменяется новым счетом, содержащимся в параметре `score`, и обновленная строка записывается в файл userScores.tmp.

Если имена не совпадают, то исходная строка просто записывается во временный файл userScores.tmp.

Допустим, функции были переданы аргументы `False`, `'Benny'` и `'158'` (т. е. `updateUserScore(False, 'Benny',`

'158')). В следующей таблице показаны различия между исходным файлом userScores.txt и новым файлом userScores.tmp.

userScores.txt	userScores.tmp
Ann, 10	Ann, 100
Benny, 102	**Benny, 158**
Carol, 214	Carol, 214
Darren, 129	Darren, 129

Попробуйте написать этот фрагмент самостоятельно.

Когда все будет сделано, мы выходим из цикла `for` и закрываем оба файла, после чего файл userScores.txt удаляется. Наконец, файл userScores.tmp переименовывается в userScores.txt.

Понятно? Попробуйте написать блок `else` самостоятельно.

На этом функция `updateUserScore()` завершается, и вместе с ней завершается и файл gametasks.py.

11.2. GAMECLASSES.PY

УПРАЖНЕНИЕ 2.1. КЛАСС GAME

Переходим ко второй части проекта. В этом разделе будет создан новый файл, содержащий три класса: `Game`, `MathGame` и `BinaryGame`.

Создайте в IDLE новый файл и присвойте ему имя gameclasses.py.

Начнем с класса `Game`. Это очень простой класс, на основе которого будут созданы два других производных класса.

Класс имеет один инициализатор с двумя параметрами: `self` и `noOfQuestions`. Параметр `noOfQuestions` имеет значение по умолчанию 0.

Внутри инициализатора `noOfQuestions` присваивается переменной экземпляра с именем `_noOfQuestions`. Попробуйте определить инициализатор и добавьте команду присваивания самостоятельно.

Готово?

Кроме инициализатора класс `Game` содержит свойство для получения и присваивания значения `_noOfQuestion`. Код get-метода приведен ниже:

```
@property
def noOfQuestions(self):
    return self._noOfQuestions
```

Как видите, get-метод просто возвращает значение `_noOfQuestions`.

С другой стороны, set-метод устроен сложнее. Он получает два параметра: `self` и `value`.

Если `value` меньше 1, то set-метод присваивает `_noOfQuestions` значение 1 и выводит сообщения `"Minimum Number`

`of Questions = 1"` и `"Hence, number of questions will be set to 1"`.

Если значение превышает 10, то set-метод присваивает `_noOfQuestions` значение 10 и выводит сообщения `"Maximum Number of Questions = 10"` и `"Hence, number of questions will be set to 10"`.

Если ни одно из этих условий не выполняется, то set-метод присваивает `_noOfQuestions` значение `value`. Попробуйте написать set-метод самостоятельно.

На этом работа над классом `Game` завершена.

УПРАЖНЕНИЕ 2.2. КЛАСС BINARYGAME

Перейдем к классу `BinaryGame`. Этот класс является производным от класса `Game`. Попробуйте определить класс `BinaryGame` самостоятельно.

Класс `BinaryGame` содержит всего один метод `generateQuestions()`. Этот метод экземпляра отвечает за генерирование вопросов для игры Binary.

А именно выводит число, записанное в десятичной системе, и предлагает игроку преобразовать его в двоичную систему. Например, метод может вывести число 12. Игрок должен правильно перевести число в 1100 — двоичный аналог числа 12.

Метод `generateQuestions()` имеет следующую структуру:

```
def generateQuestions(self):
    # импортирование randint
    # объявление локальной переменной с именем score
    # генерирование вопросов и проверка ответов
    # в цикле for
    # возвращение значения score
```

1. *Импортирование функции* `randint()`

Как видно из приведенной структуры, для генерирования вопросов сначала необходимо импортировать функцию `randint()` из модуля `random`. Функция `randint()` генерирует случайное целое число из заданного диапазона. Она будет использоваться позднее для генерирования вопросов. Попробуйте импортировать функцию самостоятельно.

2. *Объявление переменной*

Затем необходимо объявить локальную переменную с именем `score`. Эта переменная используется для хранения счета игрока во время игры. Попробуйте объявить эту переменную и инициализировать ее нулем.

3. *Генерирование вопросов*

После завершения всей подготовки можно переходить к генерированию вопросов. Количество вопросов, генерируемых функцией, зависит от переменной экземпляра `_noOfQuestions`, которую класс `BinaryGame` наследует от `Game`. Для чтения значения этой переменной будет использоваться get-метод. Количество генерируемых вопросов будет определяться циклом `for`. Вот как будет выглядеть этот цикл `for`:

```
for i in range(self.noOfQuestions):
```

Внутри цикла `for` функция `randint()` используется для генерирования случайного числа. Функция `randint()` получает два параметра: `start` и `end`, и возвращает случайное число N из интервала `start` <= N <= `end`.

Например, вызов `randint(1, 9)` будет возвращать случайное число из набора 1, 2, 3, 4, 5, 6, 7, 8, 9.

В нашей функции вызов `randint()` генерирует число от 1 до 100, а результат присваивается локальной переменной с именем `base10`.

Затем функция `input()` используется для вывода сообщения, предлагающего пользователю преобразовать число в двоичную систему. Ответ пользователя сохраняется в локальной переменной с именем `userResult`.

После этого значение, введенное пользователем, необходимо сравнить с правильным ответом (это все еще происходит в цикле `for`). Для этого мы воспользуемся циклом `while True` — фактически это цикл, выполняемый бесконечно. Дело в том, что цикл `while True` эквивалентен записи вида `while 1==1`. Так как значение 1 всегда равно 1 (т. е. условие всегда истинно), при проверке условия никогда не будет получен результат `False` и цикл будет выполняться бесконечно. Для выхода из цикла необходимо использовать команду `break`.

Цикл `while True` в нашем упражнении будет использоваться по такой схеме:

```
while True:
    try:
        # преобразовать ответ пользователя в целое
        # число и вычислить ответ, обновить счет
        # пользователя в зависимости от ответа;
        # прервать цикл while True командой break
    except:
        # если при преобразовании возникает ошибка,
        # вывести сообщение предложить пользователю
        # снова ввести ответ
```

Помните, что функция `input()` возвращает строку со значением, введенным пользователем? Внутри цикла `while True` команда `try-except` пытается преобразовать введенное значение в число. Если попытка преобразования завершается неудачей, программа должна сообщить пользователю об ошибке и предложить ему ввести допустимое значение.

Цикл `while True` продолжается до того момента, когда блок `try` будет выполнен без ошибки и достигнет команды `break`.

А теперь займемся блоком `try`.

Для преобразования данных, введенных пользователем, в целое число в программе будет использоваться встроенная функция `int()`. Однако при этом необходимо сообщить Python, что число, которое мы пытаемся преобразовать, записано в двоичной системе счисления. Это можно сделать так:

```
answer = int(userResult, base = 2)
```

Когда при вызове функции `int()` в круглые скобки включается аргумент `base = 2`, Python знает, что значение `userResult` следует интерпретировать как двоичную запись числа.

Затем `int()` преобразует строку в целое число в десятичной записи и возвращает полученное значение. После этого оно присваивается локальной переменной с именем `answer`.

Команда `if` сравнивает `answer` с исходным десятичным числом (которое хранится в переменной `base10`).

Если два значения совпадают, программа сообщает пользователю, что ответ правилен, и увеличивает его счет на 1 очко, после чего цикл `while True` прерывается ключевым словом `break`.

Если пользователь ввел неправильный ответ, программа сообщает об этом и выводит правильный ответ. Для этого используется метод `format()`, описанный в разделе 4.3. Чтобы вывести число в двоичной записи, необходимо использовать спецификатор формата `b`:

```
print("Wrong answer. The correct answer is
      {:b}.".format(base10))
```

После вывода ответа ключевое слово `break` также используется для выхода из цикла `while True`.

На этом блок `try` завершается. Попробуйте написать блок `try` самостоятельно.

Перейдем к блоку `except`. Этот блок выполняется, если функция `int()` не может преобразовать ввод пользователя в число. В блоке `except` программа сообщает пользователю, что он не ввел двоичное число, и с помощью функции `input()` запрашивает у пользователя другое число. Полученное число используется для обновления переменной `userResult`, и на этом блок `except` завершается.

4. *Возвращение значения score*

После выхода из блока `except` программа выходит как из цикла `while True`, так и из цикла `for`. На этой стадии функция сгенерировала все необходимые вопросы, и мы просто возвращаем значение `score`.

Попробуйте написать функцию самостоятельно. После завершения работы над функцией `generateQuestions()` класс `BinaryGame` готов, и мы можем перейти к классу `MathGame`.

УПРАЖНЕНИЕ 2.3. КЛАСС MATHGAME

Класс `MathGame` очень похож на класс `BinaryGame`. Он также является производным от класса `Game`. Попробуйте определить класс самостоятельно.

Класс `MathGame` содержит всего один метод с именем `generateQuestions()`. По своей базовой структуре метод `generateQuestions()` похож на класс `BinaryGame`:

```
def generateQuestions(self):
    # импортирование randint
```

```
# объявление четырех локальных переменных score,
# numberList, symbolList и operatorDict
# генерирование вопросов и проверка ответов
# в цикле for
# возвращение значения score
```

1. *Импортирование функции* `randint()`

Как и в предыдущем случае, для генерирования вопросов необходимо импортировать функцию `randint()` из модуля `random`. Попробуйте импортировать эту функцию самостоятельно.

2. *Объявление переменных*

Затем необходимо объявить четыре локальные переменные. Первая переменная `score` используется для хранения счета игрока. Она инициализируется нулем.

Также нам понадобятся два списка. Назовем их `numberList` и `symbolList`.

В списке `numberList` должно храниться пять чисел, исходные значения которых равны 0. Список `symbolList` должен содержать четыре строки с исходными значениями `' '`.

Наконец, нам также понадобится словарь. Он состоит из четырех пар; ключами словаря являются целые числа от 1 до 3, а данными — знаки «+», «-», «*» и «**». Назовем его `operatorDict`.

Попробуйте объявить и инициализировать переменные самостоятельно.

3. *Генерирование вопросов*

Готово? Теперь можно переходить к генерированию вопросов. Количество вопросов, сгенерированных функцией, зависит от переменной экземпляра `_noOfQuestions`, унаследованной классом `MathGame` от `Game`. Попробуйте определить цикл `for` для решения этой задачи. За подсказкой обращайтесь к функции `generateQuestions()` класса `BinaryGame`.

Справились? Перейдем к циклу `for`.

В цикле `for` список `numberList` будет обновляться случайными числами. Для этого мы воспользуемся функцией `randint()` для генерирования случайного числа от 1 до 9. Так как список `numberList` содержит всего пять элементов, это можно сделать пятью отдельными командами:

```
numberList[0] = randint(1, 9)
numberList[1] = randint(1, 9)
numberList[2] = randint(1, 9)
numberList[3] = randint(1, 9)
numberList[4] = randint(1, 9)
```

При каждом вызове `randint(1, 9)` функция возвращает случайное целое число из набора 1, 2, 3, 4, 5, 6, 7, 8, 9.

Однако это не самый элегантный способ обновления `numberList`. Представьте, насколько громоздким и неудобным он будет, если список `numberList` состоит из 1000 элементов. Будет лучше воспользоваться циклом `for`.

Попробуйте воспользоваться циклом `for` для решения задачи. (Примечание: речь идет об использовании цикла `for` внутри другого цикла `for`.)

Готово? Отлично!

Числа готовы, теперь нужно сгенерировать случайные математические операторы (`+`, `-`, `*`, `**`) для вопросов. Для этого мы воспользуемся функцией `randint()` и словарем `operatorDict`.

Функция `randint()` генерирует ключ словаря, который будет связываться с оператором по словарю `operatorDict`. Например, для присваивания оператора элементу `symbolList[0]` используется команда

```
symbolList[0] = operatorDict[randint(1, 4)]
```

По аналогии с `numberList` для решения этой задачи стоит воспользоваться циклом `for`. Однако на этот раз возникает проблема, из-за которой этот цикл `for` сложнее предыдущего.

Помните, что в Python оператор ** обозначает возведение в степень (например, `2**3 = 2^3`)?

Проблема в том, что если в выражении Python встречаются два последовательных оператора возведения в степень (например, `2**3**2`), Python интерпретирует такое выражение как `2**(3**2)` вместо `(2**3)**2`. В первом случае ответ равен 2 в степени 9 (т. е. 2^9), т. е. 512. Во втором случае ответ равен 8 в степени 2 (т. е. 8^2), т. е. 64. Получается, что с вопросом вида `2**3**2` пользователь получит неправильный ответ, если интерпретирует выражение как `(2**3)**2`.

Чтобы обойти эту проблему, мы изменим код так, чтобы выражение не могло содержать два последовательных

оператора **. Иначе говоря, список `symbolList = ['+', '+', '-', '**']` допустим, а список `symbolList = ['+', '-', '**', '**']` — нет.

Это самое сложное упражнение из всех. Попробуйте предложить решение, предотвращающее два последовательных оператора ** в выражении. Когда это будет сделано, можно переходить к следующему шагу.

Подсказка: если у вас не будет собственных идей, попробуйте воспользоваться командой `if` в цикле `for`.

Наборы операторов и чисел готовы, можно переходить к генерированию математического выражения в виде строки. Это выражение строится из пяти чисел из списка `numberList` и четырех математических операторов из списка `symbolList`.

Мы объявим другую переменную с именем `questionString` и присвоим математическое выражение `questionString`. Несколько примеров `questionString`:

```
6 - 2*3 - 2**1
4 + 5 - 2*6 + 1
8 - 0*2 + 5 - 8
```

Попробуйте сгенерировать это выражение самостоятельно.

Подсказка: начните с присваивания переменной `questionString` первого элемента из `numberList`, после чего в цикле `for` выполните конкатенацию остальных элементов `numberList` и `symbolList` для построения математического

выражения. Помните, что элементы `numberList` перед конкатенацией с другими строками необходимо предварительно преобразовать в строку (с помощью встроенной функции `str()`).

Попробуйте сделать это самостоятельно.

Получилось? Хорошо! Теперь у вас есть математическое выражение в виде строки, присвоенное переменной `questionString`. Для вычисления результата выражения мы воспользуемся замечательной встроенной функцией Python — `eval()`.

Функция `eval()` интерпретирует строку как код и выполняет этот код. Например, вызов функции `eval("1+2+4")` даст результат 7.

Таким образом, чтобы вычислить результат математического выражения, мы передаем `questionString` функции `eval()` и присваиваем результат новой переменной с именем `result`.

Попробуйте сделать это самостоятельно.

Теперь нужно вывести вопрос, чтобы пользователь прочитал его. Как упоминалось ранее, в Python оператор `**` обозначает возведение в степень (т. е. `2**3 = 8`). Но многие пользователи просто не поймут последовательность `**`, и если вывести вопрос вида `2**3 + 8 - 5`, могут возникнуть недоразумения. Чтобы избежать путаницы, мы заменим все операторы `**` в `questionString` более привычным знаком `^`.

Для этого мы воспользуемся встроенной функцией `replace()`. Сделать это несложно: достаточно выполнить команду `questionString = questionString.replace ("**", "^")`.

После преобразования исходной строки к виду, более удобному для пользователя, мы с помощью функции `input()` предложим пользователю вычислить значение выражения и присвоим результат локальной переменной `userResult`.

Теперь все готово к тому, чтобы проверить ответ и обновить счет пользователя. По аналогии с тем, как это делалось в классе `BinaryGame`, мы воспользуемся циклом `while True`, чтобы получить от пользователя новый ответ, который преобразуется в целое число вызовом функции `int()`.

С другой стороны, если пользователь ввел допустимое значение, оно сравнивается с правильным ответом (который хранится в `result`). Если два значения равны, мы сообщаем пользователю, что ответ правилен, и увеличиваем значение `score` на 1. Затем ключевое слово `break` используется для прерывания цикла `while True`.

Если пользователь ввел неправильный ответ, программа сообщает об этом пользователю и выводит правильный ответ, после чего цикл `while True` прерывается ключевым словом `break`.

Попробуйте написать этот цикл `while True` самостоятельно. Он почти полностью совпадает с циклом `while True`

из класса `BinaryGame`. Обращайтесь к этой реализации, если у вас возникнут проблемы.

4. *Возвращение значения score*

Получилось? Когда цикл `while True` будет готов, можно выйти из обоих циклов `while True` и `for` и вернуть значение локальной переменной `score`. На этом завершается метод `generateQuestions()`, а вместе с ним и класс `MathGame`.

А работа над файлом gameclasses.py подходит к концу.

11.3. PROJECT.PY

Поздравляю с успешным завершением первых двух этапов проекта. Третий этап будет относительно простым, потому что в нем в основном будут вызываться функции и методы, определенные выше. Начнем с создания нового файла project.py.

УПРАЖНЕНИЕ 3.1. ИМПОРТИРОВАНИЕ КЛАССОВ И ФУНКЦИЙ

Сначала необходимо импортировать классы и функции, запрограммированные в двух предыдущих файлах. Попробуйте сделать это самостоятельно.

Затем можно переходить к написанию основной программы. Код основной программы будет заключен в конструкцию `try-except`, так как мы хотим обрабатывать все непредвиденные ошибки при выполнении основной программы. Команда `try-except` имеет следующую структуру:

```
try:
    # объявление переменных
    # выполнение программы в цикле while, пока она
    # не будет прервана пользователем
    # обновление счета пользователя после выхода
    # из программы except Exception as e:
    # оповещение пользователя об ошибке и выход
    # из программы
```

УПРАЖНЕНИЕ 3.2. БЛОК TRY

Начнем с кода блока `try`.

1. *Объявление переменных*

Объявим две локальные переменные `mathInstructions` и `binaryInstructions`. Эти переменные, как подсказывают их имена, предназначены для хранения инструкций для двух игр. Инструкция для игры Mathematics выглядит так (в переводе):

```
В этой игре вам предлагается решить простую
арифметическую задачу.
За каждый правильный ответ вам начисляется одно очко.
За ошибочные ответы очки не вычитаются.
```

Инструкция для игры Binary (в переводе):

```
В этой игре вы получаете десятичное число.
Ваша задача — преобразовать его в двоичную систему
счисления.
За каждый правильный ответ вам начисляется одно очко.
За ошибочные ответы очки не вычитаются.
```

Попробуйте объявить две переменные самостоятельно и присвоить две строки правильным переменным.

Затем мы создадим два объекта с именами `bg` и `mg` — экземпляры классов `BinaryGame` и `MathGame` соответственно.

После этого пользователю предлагается ввести имя; введенное значение присваивается переменной `userName`. Когда это будет сделано, переменная передается функции `getUserScore()`.

Функция `getUserScore()` возвращает либо счет пользователя, либо «–1» (если пользователь не найден). Этот результат преобразуется в целое число и присваивается переменной с именем `score`.

Затем необходимо задать значение другой переменной с именем `newUser`. Если `score` содержит `-1`, то переменной `newUser` присваивается `True`, а значение `score` меняется с `-1` на `0`. В любом случае, `newUser` присваивается `False`.

Попробуйте написать этот фрагмент самостоятельно.

Когда это будет сделано, на экран выводятся приветственное сообщение и счет пользователя.

2. *Выполнение программы в цикле* `while`

В следующей части программы задействован цикл `while`. А именно наша программа запрашивает у пользователя входные данные и определяет, что делать дальше — завершиться или сделать что-то еще.

Затем необходимо объявить другую переменную `userChoice` и присвоить ей исходное значение 0.

Далее, в другом цикле `while`, переменная `userChoice` сравнивается со строкой, выбранной вами, — например, `-1`. Если значение `userChoice` не равно `-1`, игра продолжает работать, пока `userChoice` не примет значение `-1`.

Структура цикла `while` выглядит так:

```
userChoice = 0

while userChoice != '-1'
    # пользователю предлагается выбрать одну из двух игр
    # пользователь вводит количество вопросов в игре
    # вывод вопросов с учетом данных, введенных
    # пользователем,
    # и обновление текущего счета пользователя
    # вывод обновленного счета
    # пользователь снова вводит свое решение
    # с обновлением userChoice
```

В цикле `while` происходит несколько событий.

Сначала пользователю предлагается выбрать одну из игр, для чего выводится следующее сообщение:

```
Math Game (1) or Binary Game (2)?
```

Значение, введенное пользователем, присваивается локальной переменной `game`. Программа в цикле `while` запрашивает у пользователя действительное значение, если пользователь не ввел `'1'` или `'2'`. Здесь цикл `while` выполняется внутри другого цикла `while`. Внутренним циклом `while` управляет переменная `game`, а внешним циклом `while` — переменная `userChoice`.

Попробуйте написать внутренний цикл `while` самостоятельно. Не забудьте обновить переменную `game` во внутреннем цикле `while`, чтобы избежать зацикливания.

После этого программа предлагает пользователю ввести количество вопросов:

```
How many questions do you want per game (1 to 10)?
```

Введенное значение присваивается переменной `numPrompt`.

Далее программа пытается преобразовать `numPrompt` в целое число. Для этого будет использоваться цикл `while True`, сходный с тем, который использовался в упражнениях 2.2 и 2.3 для метода `generateQuestions()`. Структура цикла `while True` выглядит так:

```
while True:
    try:
        # numPrompt преобразуется в целое число
        # и присваивается локальной переменной num.
        # break
    except:
        # оповещение о том, что пользователь ввел
        # недействительное число
        # пользователю предлагается снова ввести
        # количество запросов,
        # а результат присваивается numPrompt.
```

Попробуйте написать цикл `while True` самостоятельно.

Когда это будет сделано, можно переходить к выводу вопросов, которые определяются данными, введенными пользователем.

Для этой цели будет использоваться команда `if`.

Если пользователь выбрал игру Math (т. е. `game == '1'`), то программа выполняет три шага.

Сначала следующая команда присваивает значение переменной `_noOfQuestions` в классе `MathGame`:

```
mg.noOfQuestions = num
```

Здесь значение `_noOfQuestions` задается с использованием set-метода из класса `MathGame`. Тем самым предотвращается присваивание значений, больших 10 или меньших 1.

Для вывода инструкций игры Math используется функция `printInstructions()`.

Наконец, переменная `mg` используется для вызова метода `generateQuestions()` класса `MathGame`. Метод генерирует вопросы и возвращает счет пользователя в этом раунде. Значение прибавляется к существующему счету, и происходит обновление последнего.

Попробуйте реализовать эти три шага самостоятельно.

Когда все будет сделано, можно переходить к блоку `else`.

Этот блок очень похож на блок `if`, если не считать того что вместо `mg` используется переменная `bg` для вызова методов класса `BinaryGame`. Попробуйте написать блок `else` самостоятельно.

После того как все вопросы будут выведены, программа выводит текущий счет пользователя.

Затем пользователю предлагается нажать Enter, чтобы продолжить работу, или ввести «-1» для завершения игры. Результат используется для обновления переменной `userChoice`.

На этом цикл `while` завершается.

3. *Обновление файла* userScores.txt

После выхода из цикла `while` необходимо обновить файл userScores.txt. Для этого мы просто вызываем функцию `updateUserScore()` и передаем ей переменные `newUser`, `userName` и `score`. Значение `score` необходимо преобразовать в строку (функцией `str()`), так как функции `updateUserScore()` требуется строковое значение `score` для выполнения конкатенации с другими строками внутри функции.

Вот и все, что нужно знать о блоке `try`.

УПРАЖНЕНИЕ 3.3. НАПИСАНИЕ БЛОКА EXCEPT

Перейдем к блоку `except`. В блоке `except` программа сообщает пользователю о том, что произошла неизвестная ошибка, и завершается. Кроме того, также выводится сгенерированное системой сообщение об ошибке — оно предоставляет краткое описание сути ошибки.

И на этом работа над программой завершена! Вы только что завершили работу над своей первой программой на

языке Python. Здорово? Надеюсь, у вас это вызывает такой же энтузиазм, как и у меня. :)

Теперь попробуйте запустить программу project.py. Работает ли она так, как предполагалось?

Если ваша программа не работает, сравните ее с кодом ответа и попробуйте выяснить, что пошло не так. Анализируя свои ошибки, вы узнаете много полезного.

Решение прикладных задач — самое интересное в программировании, и оно же приносит наибольшую практическую пользу. Хорошо проводите время и никогда не сдавайтесь! Примерный код ответа приведен в приложении Д.

СПАСИБО!

Вы почти добрались до конца книги. Спасибо, что вы прочитали ее, — надеюсь, вам понравилось. Но что еще важнее, я искренне надеюсь, что книга поможет вам освоить азы программирования на Python.

Я знаю, что о программировании на языке Python написаны десятки книг, но вы выбрали именно эту. Еще раз спасибо, что дочитали до конца! Пробуйте упражнения и задачи, на практике вы многого сможете достичь.

И последнее, но не менее важное: весь исходный код проекта и приложения можно скачать по адресу: https://www.learncodingfast.com/python.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А.
РАБОТА СО СТРОКАМИ

Примечание: запись `[start, [end]]` означает, что аргументы `start` и `end` не являются обязательными. Если указано только одно число, то предполагается, что это значение `start`.

`#` отмечает начало комментария

`'''` отмечает начало и конец многострочного комментария

Код оформлен моноширинным шрифтом

`=>` отмечает начало вывода

`count (sub, [start, [end]])`

Возвращает количество вхождений подстроки `sub` в строке.

Поиск проводится с учетом регистра символов.

```
[Пример]
 # В следующих примерах 's' встречается в позициях
 # с индексами 3, 6 и 10 подсчет по всей строке
'This is a string'.count('s')
=> 3
# подсчет от индекса 4 до конца строки
'This is a string'.count('s', 4)
=> 2
# подсчет от индекса 4 до 10-1
'This is a string'.count('s', 4, 10 )
=> 1
```

```
# подсчет 'T'. Найдено только 1 вхождение 'T',
# так как функция различает регистр символов.
'This is a string'.count('T')
=> 1
```

`endswith (suffix, [start, [end]])`

Возвращает `True`, если строка завершается заданным суффиксом `suffix`; в противном случае возвращается `False`.

`suffix` также может содержать кортеж суффиксов.

Поиск проводится с учетом регистра символов.

```
[Пример]
# 'man' встречается в позиции с индексами от 4 до 6
# проверяется вся строка
'Postman'.endswith('man')
=> True
# проверка от индекса 3 до конца строки
'Postman'.endswith('man', 3)
=> True
# проверка от индекса 2 до 6-1
'Postman'.endswith('man', 2, 6)
=> False
# проверка от индекса 2 до 7-1
'Postman'.endswith('man', 2, 7)
=> True
# использование кортежа суффиксов (проверка от
индекса 2 до 6-1)
'Postman'.endswith(('man', 'ma'), 2, 6)
=> True
```

`find/index (sub, [start, [end]])`

Возвращает индекс позиции строки, в которой найдено первое вхождение подстроки `sub`.

`find()` возвращает `-1`, если подстрока `sub` не найдена.

`index()` возвращает `ValueError`, если подстрока `sub` не найдена.

Поиск проводится с учетом регистра символов.

```
[Пример]
# проверка всей строки
'This is a string'.find('s')
=> 3
# проверка от индекса 4 до конца строки
'This is a string'.find('s', 4)
=> 6
# проверка от индекса 7 до 11-1
'This is a string'.find('s', 7,11 )
=> 10
# подстрока sub не найдена
'This is a string'.find('p')
=> -1
'This is a string'.index('p')
=> ValueError
```

`isalnum()`

Возвращает `True`, если все символы строки являются алфавитно-цифровыми и строка содержит хотя бы один символ; в противном случае возвращается `False`.

Пропуски (whitespace) не относятся к категории алфавитно-цифровых символов.

```
[Пример]
'abcd1234'.isalnum()
=> True
'a b c d 1 2 3 4'.isalnum()
=> False
'abcd'.isalnum()
=> True
'1234'.isalnum()
=> True
```

isalpha()

Возвращает `True`, если все символы строки являются алфавитными и строка содержит хотя бы один символ; в противном случае возвращается `False`.

```
[Пример]
'abcd'.isalpha()
=> True
'abcd1234'.isalpha()
=> False
'1234'.isalpha()
=> False
'a b c'.isalpha()
=> False
```

isdigit()

Возвращает `True`, если все символы строки являются цифрами и строка содержит хотя бы один символ; в противном случае возвращается `False`.

```
[Пример]
'1234'.isdigit()
=> True
'abcd1234'.isdigit()
```

```
=> False
'abcd'.isdigit()
=> False
'1 2 3 4'.isdigit()
=> False
```

`islower()`

Возвращает `True`, если все регистровые символы имеют нижний регистр и строка содержит хотя бы один регистровый символ; в противном случае возвращается `False`.

```
[Пример]
'abcd'.islower()
=> True
'Abcd'.islower()
=> False
'ABCD'.islower()
=> False
```

`isspace()`

Возвращает `True`, если строка содержит только символы-пропуски и хотя бы один символ; в противном случае возвращается `False`.

```
[Пример]
' '.isspace()
=> True
'a b'.isspace()
=> False
```

istitle()

Возвращает `True`, если строка имеет титульный регистр (все слова начинаются с символа верхнего регистра, остальные символы относятся к нижнему регистру) и содержит хотя бы один символ.

```
[Пример]
'This Is A String'.istitle()
=> True
'This is a string'.istitle()
=> False
```

isupper()

Возвращает `True`, если все регистровые символы имеют верхний регистр и строка содержит хотя бы один регистровый символ; в противном случае возвращается `False`.

```
[Пример]
'ABCD'.isupper()
=> True
'Abcd'.isupper()
=> False
'abcd'.isupper()
=> False
```

join()

Возвращает строку, в которой компоненты переданного аргумента соединяются через разделитель.

```
[Пример]
sep = '-'
myTuple = ('a', 'b', 'c')
```

```
myList = ['d', 'e', 'f']
myString = "Hello World"
sep.join(myTuple)
=> 'a-b-c'
sep.join(myList)
=> 'd-e-f'
sep.join(myString)
=> 'H-e-l-l-o- -W-o-r-l-d''
```

`lower()`

Возвращает копию строки, преобразованную к нижнему регистру.

```
[Пример]
'Hello Python'.lower()
=> 'hello python'
```

`replace(old, new[, count])`

Возвращает копию строки, в которой все вхождения подстроки `old` заменяются на `new`.

Аргумент `count` не является обязательным. Если он задан, то заменяются только первые `count` вхождений.

Поиск проводится с учетом регистра символов.

```
[Пример]
# замена всех вхождений
'This is a string'.replace('s', 'p')
=> 'Thip ip a ptring'
# замена первых 2 вхождений
'This is a string'.replace('s', 'p', 2)
=> 'Thip ip a string'
```

`split([sep [,maxsplit]])`

Возвращает список слов в строке, разбитой по разделителю `sep`.

Аргументы `sep` и `maxsplit` не являются обязательными.

Если аргумент `sep` не задан, то в качестве разделителя используется пробел.

Если аргумент `maxsplit` задан, выполняется максимум `maxsplit` разбиений.

Поиск проводится с учетом регистра символов.

```
[Пример]
# разбиение с использованием пробелов как разделителей
'This is a string'.split()
=> ['This', 'is', 'a', 'string']
# разделителем является запятая, за которой следует
# пробел
'This, is, a, string'.split(', ')
=> ['This', 'is', 'a', 'string']
# разделителем является запятая, за которой следует
# пробел
# выполняются только 2 разбиения
'This, is, a, string'.split(', ', 2)
=> ['This', 'is', 'a, string']
```

`splitlines ([keepends])`

Возвращает список из строки, разбитой на логические строки; разбиение происходит по границам логических строк.

Разрывы строк не включаются в полученный список, если только аргумент не задан и не равен `True`.

```
[Пример]
# строки разбиваются по \n
'This is the first line.\nThis is the second line'.
splitlines()
=> ['This is the first line.', 'This is the second line.']
# разбиение строки, состоящей из нескольких
# логических строк (например, строки, определяемой
# маркерами ''')
'''This is the first line.
This is the second line.'''.splitlines()
=> ['This is the first line.', 'This is the second line.']
# разбиение с сохранением разрывов строк
'This is the first line.\nThis is the second
line.'.splitlines(True)
=> ['This is the first line.\n', 'This is the second
    line.']
'''This is the first line.
This is the second line.'''.splitlines(True)
=> ['This is the first line.\n', 'This is the second
    line.']
```

`startswith (prefix[, start[, end]])`

Возвращает `True`, если строка начинается с заданного префикса `prefix`; в противном случае возвращается `False`.

`prefix` также может содержать кортеж префиксов.

Поиск проводится с учетом регистра символов.

```
[Пример]
# 'Post' встречается в позиции с индексами от 0 до 3
# проверка всей строки
'Postman'.startswith('Post')
```

```
=> True
# проверка от индекса 3 до конца строки
'Postman'.startswith('Post', 3)
=> False
# проверка от индекса 2 до 6-1
'Postman'.startswith('Post', 2, 6)
=> False
# проверка от индекса 2 до 6-1
'Postman'.startswith('stm', 2, 6)
=> True
# использование кортежа префиксов (проверка
# от индекса 3 до конца строки)
'Postman'.startswith(('Post', 'tma'), 3)
=> True
```

`strip ([chars])`

Возвращает копию строки, из которой удаляются начальные и конечные символы, входящие в `chars`.

Если аргумент `chars` не задан, то из строки удаляются пропуски.

Поиск проводится с учетом регистра символов.

```
[Пример]
# удаление пропусков
'   This is a string   '.strip()
=> 'This is a string'
# удаление 's'. Никакие символы не удаляются, потому
# что 's' не находится в начале или конце строки
'This is a string'.strip('s')
=> 'This is a string'
# удаление 'g'.
'This is a string'.strip('g')
=> 'This is a strin'
```

`upper()`

Возвращает копию строки, преобразованную к верхнему регистру.

```
[Пример]
'Hello Python'.upper()
=> 'HELLO PYTHON'
```

ПРИЛОЖЕНИЕ Б.
РАБОТА СО СПИСКАМИ

=> отмечает начало вывода

append()

Добавляет элемент в конец списка.

```
[Пример]
myList = ['a', 'b', 'c', 'd']
myList.append('e')
print (myList)
=> ['a', 'b', 'c', 'd', 'e']
```

del

Удаляет элементы из списка.

```
[Пример]
myList = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
          'i', 'j', 'k', 'l']
# удаление третьего элемента (индекс = 2)
del myList[2]
 print (myList)
=> ['a', 'b', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
    'l']
# удаление элементов от индекса 1 до 5-1
del myList[1:5]
print (myList)
=> ['a', 'g', 'h', 'i', 'j', 'k', 'l']
# удаление элементов от индекса 0 до 3-1
del myList [ :3]
print (myList)
=> ['i', 'j', 'k', 'l']
```

```
# удаление элементов от индекса 2 до конца
del myList [2:]
print (myList)
=> ['i', 'j']
```

`extend( )`

Объединяет два списка.

```
[Пример]
myList = ['a', 'b', 'c', 'd', 'e']
myList2 = [1, 2, 3, 4]
myList.extend(myList2)
print (myList)
=> ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4]
```

`in`

Проверяет, присутствует ли элемент в списке.

```
[Пример]
myList = ['a', 'b', 'c', 'd']
'c' in myList
=> True
'e' in myList
=> False
```

`insert()`

Добавляет элемент в список в заданной позиции.

```
[Пример]
myList = ['a', 'b', 'c', 'd', 'e']
myList.insert(1, 'Hi')
print (myList)
=> ['a', 'Hi', 'b', 'c', 'd', 'e']
```

`len()`

Определяет количество элементов в списке.

```
[Пример]
myList = ['a', 'b', 'c', 'd']
print (len(myList))
=> 4
```

`pop()`

Получает значение элемента и удаляет его из списка.

В аргументе должен передаваться индекс элемента.

```
[Пример]
myList = ['a', 'b', 'c', 'd', 'e']
# удаление третьего элемента
member = myList.pop(2)
print (member)
=> c
print (myList)
=> ['a', 'b', 'd', 'e']
# удаление последнего элемента
member = myList.pop( )
print (member)
=> e
print (myList)
=> ['a', 'b', 'd']
```

`remove()`

Удаляет элемент из списка. В аргументе должен передаваться список.

```
[Пример]
myList = ['a', 'b', 'c', 'd', 'e']
# удаление элемента 'c'
myList.remove('c')
print (myList)
=> ['a', 'b', 'd', 'e']
```

`reverse()`

Переставляет элементы списка в обратном порядке.

```
[Пример]
myList = [1, 2, 3, 4]
myList.reverse()
print (myList)
=> [4, 3, 2, 1]
```

`sort()`

Сортирует список в алфавитном или числовом порядке.

```
[Пример]
myList = [3, 0, -1, 4, 6]
myList.sort()
print(myList)
=> [-1, 0, 3, 4, 6]
```

`sorted()`

Возвращает новый отсортированный список без изменения исходного списка.

В аргументе должен передаваться список.

```
[Пример]
myList = [3, 0, -1, 4, 6]
myList2 = sorted(myList)
# исходный список не отсортирован
print (myList)
=> [3, 0, -1, 4, 6]
# новый список отсортирован
print (myList2)
=> [-1, 0, 3, 4, 6]
```

`Оператор сложения: +`

Выполняет слияние списков.

```
[Пример]
myList = ['a', 'b', 'c', 'd']
print (myList + ['e', 'f'])
=> ['a', 'b', 'c', 'd', 'e', 'f']
print (myList)
=> ['a', 'b', 'c', 'd']
```

`Оператор умножения: *`

Дублирует список и присоединяет копию к концу списка.

```
[Пример]
myList = ['a', 'b', 'c', 'd']
print (myList*3)
=> ['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b',
'c', 'd']
```

```
print (myList)
=> ['a', 'b', 'c', 'd']
```

Примечание

Операторы + и * не изменяют исходного списка. В обоих примерах список после выполнения операции содержит те же элементы `['a', 'b', 'c', 'd']`.

ПРИЛОЖЕНИЕ В. РАБОТА С КОРТЕЖАМИ

=> отмечает начало вывода

del

Удаляет кортеж.

```
[Пример]
myTuple = ('a', 'b', 'c', 'd')
del myTuple
print (myTuple)
=> NameError: name 'myTuple' is not defined
```

in

Проверяет, присутствует ли элемент в кортеже.

```
[Пример]
myTuple = ('a', 'b', 'c', 'd')
'c' in myTuple
=> True
'e' in myTuple

=> False
```

len( )

Определяет количество элементов в кортеже.

```
[Пример]
myTuple = ('a', 'b', 'c', 'd')
print (len(myTuple))
=> 4
```

`Оператор сложения: +`

Выполняет слияние кортежей.

```
[Пример]
myTuple = ('a', 'b', 'c', 'd')
print (myTuple + ('e', 'f'))
=> ('a', 'b', 'c', 'd', 'e', 'f')
print (myTuple)
=> ('a', 'b', 'c', 'd')
```

`Оператор умножения: *`

Дублирует кортеж и присоединяет копию к концу кортежа.

```
[Пример]
myTuple = ('a', 'b', 'c', 'd')
print(myTuple*3)
=> ('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b',
    'c', 'd')
print (myTuple)
=> ('a', 'b', 'c', 'd')
```

Примечание

Операторы `+` и `*` не изменяют исходного кортежа. В обоих примерах кортеж после выполнения операции содержит те же элементы `['a', 'b', 'c', 'd']`.

ПРИЛОЖЕНИЕ Г.
РАБОТА СО СЛОВАРЯМИ

=> отмечает начало вывода

clear()

Удаляет из словаря все элементы и возвращает пустой словарь.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
print (dic1)
=> {1: 'one', 2: 'two'}
dic1.clear()
print (dic1)
=> { }
```

del

Удаляет весь словарь.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
del dic1
print (dic1)
=> NameError: name 'dic1' is not defined
```

get()

Возвращает значение для заданного ключа.

Если ключ не найден, возвращается ключевое слово None.

Также можно задать значение, которое должно возвращаться для отсутствующего ключа.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
dic1.get(1)
=> 'one'
dic1.get(5)
=> None
dic1.get(5, "Not Found")
=> 'Not Found'
```

in

Проверяет, что элемент присутствует в словаре.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
# по ключу
1 in dic1
=> True
3 in dic1
=> False
# по значению
'one' in dic1.values()
=> True
'three' in dic1.values()
=> False
```

items()

Возвращает список пар словаря в виде кортежей.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
dic1.items()
=> dict_items([(1, 'one'), (2, 'two')])
```

`keys()`

Возвращает список ключей словаря.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
dic1.keys()
=> dict_keys([1, 2])
```

`len()`

Определяет количество элементов в словаре.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
print (len(dic1))
=> 2
```

`update()`

Добавляет пары «ключ — значение» из одного словаря в другой. Дубликаты удаляются.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
dic2 = {1: 'one', 3: 'three'}
dic1.update(dic2)
print (dic1)
=> {1: 'one', 2: 'two', 3: 'three'}
print (dic2)  #no change
=> {1: 'one', 3: 'three'}
```

`values()`

Возвращает список значений словаря.

```
[Пример]
dic1 = {1: 'one', 2: 'two'}
dic1.values()
=> dict_values(['one', 'two'])
```

ПРИЛОЖЕНИЕ Д.
ОТВЕТЫ К УПРАЖНЕНИЯМ

УПРАЖНЕНИЕ 1.1

```
def printInstructions(instruction):
  print(instruction)
```

УПРАЖНЕНИЕ 1.2

```
def getUserScore(userName):
  try:
    input = open('userScores.txt', 'r')
    for line in input:
      content = line.split(', ')
      if content[0] == userName:
        input.close()
        return content[ 1]
    input.close()
    return '-1'
  except IOError:
    print("File not found. A new file will be
          created.")
    input = open('userScores.txt', 'w')
    input.close()
    return '-1'
```

УПРАЖНЕНИЕ 1.3

```
def updateUserScore(newUser, userName, score):
  from os import remove, rename

  if newUser == True:
    input = open('userScores.txt', 'a')
    input.write(userName + ', ' + score + '\n')
    input.close()
  else:
```

```
    temp = open('userScores.tmp', 'w')
    input = open('userScores.txt', 'r')
    for line in input:
      content = line.split(', ')
      if content[0] == userName:
        temp.write(userName + ', ' + score + '\n')
      else:
        temp.write(line)

    input.close()
    temp.close()

    remove('userScores.txt')
    rename('userScores.tmp', 'userScores.txt')
```

УПРАЖНЕНИЕ 2.1

```
class Game:
  def __init__(self, noOfQuestions = 0):
    self._noOfQuestions = noOfQuestions

  @property
  def noOfQuestions(self):
    return self._noOfQuestions

  @noOfQuestions.setter
  def noOfQuestions(self, value):
    if value < 1:
      self._noOfQuestions = 1
      print("\nMinimum Number of Questions = 1")
      print("Hence, number of questions will be set
            to 1")
    elif value > 10:
      self._noOfQuestions = 10
      print("\nMaximum Number of Questions = 10")
      print("Hence, number of questions will be set
            to 10")
    else:
      self._noOfQuestions = value
```

УПРАЖНЕНИЕ 2.2

```
class BinaryGame(Game):
  def generateQuestions(self):
    from random import randint
    score = 0

    for i in range(self.noOfQuestions):
      base10 = randint(1, 100)
      userResult = input("\nPlease convert %d to
                         binary: " %(base10))
      while True:
        try:
          answer = int(userResult, base = 2)
          if answer == base10:
            print("Correct Answer!")
            score = score + 1
            break
          else:
            print("Wrong answer. The correct answer
                  is {:b}.".format(base10))
            break
        except:
          print("You did not enter a binary number.
                Please try again.")
          userResult = input("\nPlease convert %d to
                             binary: " %(base10))

    return score
```

УПРАЖНЕНИЕ 2.3

```
class MathGame(Game):
  def generateQuestions(self):
    from random import randint
    score = 0
    numberList = [0, 0, 0, 0, 0]
    symbolList = ['', '', '', '']
    operatorDict = {1:' + ', 2:' - ', 3:'*', 4:'**'}
```

```
for i in range(self.noOfQuestions):
  for index in range(0, 5):
    numberList[index] = randint(1, 9)
  # См. объяснение ниже
  for index in range(0, 4):
    if index > 0 and symbolList[index - 1] == '**':
      symbolList[index] = operatorDict[randint(1,
                                        3)]
    else:
      symbolList[index] = operatorDict[randint(1,
                                        4)]

  questionString = str(numberList[0])

  for index in range(0, 4):
    questionString = questionString +
  symbolList[index] + str(numberList[index+1])

 result = eval(questionString)
 questionString = questionString.replace("**", "^")
 userResult = input("\nPlease evaluate %s:
                    "%(questionString))

 while True:
   try:
     answer = int(userResult)
     if answer == result:
       print("Correct Answer!")
       score = score + 1
        break
      else:
        print("Wrong answer. The correct answer
              is {:d}.".format(result))
        break
    except:
      print("You did not enter a valid number.
            Please try again.")
```

```
            userResult = input("\nPlease evaluate %s:
                               "%(questionString))

        return score

    '''

    Начиная со второго элемента (т. е. index = 1)
    в symbolList, строка
    if index > 0 and symbolList[index-1] == '**':
    проверяет, содержит ли
    предыдущий элемент в symbolList оператор **.
    Если содержит, то команда symbolList[index] =
    operatorDict[randint(1, 3)]
    выполняется. В этом случае функции randint передается
    диапазон от 1 до 3.
    Таким образом, оператор **, имеющий ключ 4
    в operatorDict, НЕ БУДЕТ
    присвоен symbolList[index].
    С другой стороны, если предыдущий элемент не содержит
    оператор **, команда
    symbolList[index] = operatorDict[randint(1, 4)] будет
    выполнена. Так как
    функции randint передается диапазон от 1 до 4, будут
    сгенерированы числа
    из набора 1, 2, 3 и 4. А значит, symbolList[index]
    будут присваиваться
    операторы +, -, * и **.

    '''
```

УПРАЖНЕНИЕ 3.1

```
from gametasks import printInstructions,
getUserScore, updateUserScore
from gameclasses import Game, MathGame, BinaryGame
```

УПРАЖНЕНИЕ 3.2

```
try:
  mathInstructions = '''
В этой игре вам предлагается решить простую
арифметическую задачу.
За каждый правильный ответ вам начисляется одно очко.
За ошибочные ответы очки не отнимаются.
'''

  binaryInstructions = '''
В этой игре вы получаете десятичное число.
Ваша задача — преобразовать его в двоичную систему
счисления.
За каждый правильный ответ вам начисляется одно очко.
За ошибочные ответы очки не отнимаются.
'''
  mg = MathGame()
  bg = BinaryGame()

  userName = input("\nPlease enter your username: ")
  score = int(getUserScore(userName))
  if score == -1:
    newUser = True
    score = 0
  else:
    newUser = False

  print("\nHello %s, welcome to the game." %(userName))
  print("Your current score is %d." %(score))

  userChoice = 0

  while userChoice != '-1':
    game = input("\nMath Game (1) or Binary Game
                (2)?: ")
    while game != '1' and game != '2':
      print("You did not enter a valid choice. Please
            try again.")
```

```
    game = input("\nMath Game (1) or Binary Game
                 (2)?: ")

  numPrompt = input("\nHow many questions do you
                    want per game (1 to 10)?: ")
  while True:
    try:
      num = int(numPrompt)
      break
    except:
      print("You did not enter a valid number.
            Please try again.")
      numPrompt = input("\nHow many questions do
                        you want per game (1 to
                        10)?: ")

  if game == '1':
    mg.noOfQuestions = num
    printInstructions(mathInstructions)
    score = score + mg.generateQuestions()
  else:
    bg.noOfQuestions = num
    printInstructions(binaryInstructions)
    score = score + bg.generateQuestions()

  print("\nYour current score is %d." %(score))

  userChoice = input("\nPress Enter to continue or
                     -1 to end: ")

updateUserScore(newUser, userName, str(score))
```

УПРАЖНЕНИЕ 3.3

```
except Exception as e:
  print("An unknown error occurred. Program will
        exit.")
  print("Error: ", e)
```

И НАПОСЛЕДОК...

Если моя книга помогла вам, я буду очень признателен, если вы расскажете о ней своим друзьям.

Программирование для меня — это искусство и наука. Это в высшей степени увлекательное и приятное занятие. Я надеюсь заразить своим энтузиазмом как можно больше людей.

Кроме того, надеюсь, что ваше обучение на этом не закончится. Если вас интересуют другие задачи по программированию, посетите сайт https://projecteuler.net/. Желаю удачи!

Джейми Чан

Python: быстрый старт

Перевели с английского
Е. Матвеев, А. Попова

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*Н. Римицан*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Корректоры	*Н. Петрова, М. Одинокова*
Верстка	*Е. Неволайнен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 02.2021. Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 01.02.21. Формат 60x90/16. Бумага офсетная.
Усл. п. л. 14,000. Тираж 1000. Заказ

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР»

предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а; тел. (812) 703-73-73, доб. 6282; e-mail: dudina@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж; тел./факс (495) 234-38-15; e-mail: reception@piter.com

Воронеж: тел. +7 951 861-72-70; e-mail: hitsenko@piter.com

Нижний Новгород: тел. +7 930 712-75-13; e-mail: yashny@yandex.ru

Ростов-на-Дону: тел. +7 908 509-35-24; e-mail: rostov1@piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223; тел./факс (846) 277-89-79, 8-960-818-14-66; e-mail: pitvolga@mail.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс +37 517 348-60-01, 374-43-25; e-mail: pugacheva@piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: тел./факс (812) 703-73-73, доб. 6282; e-mail: sales@piter.com

---

Заказ книг для вузов и библиотек:
тел./факс (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг в интернет-магазине: на сайте www.piter.com;
тел. (812) 703-73-74, доб. 6216; e-mail: books@piter.com

Вопросы по продаже электронных книг: тел. (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com