# python note’s
# by Mike Zigberman
# russian edition

ОГЛАВЛЕНИЕ

- Запись в файл
- Исключения
- Сохранение данных

10. Тестирование
    - Тестирование функции
    - Тестирование класса

# Глава 1. Переменные и простые типы данных

## Переменные

*message = "Hello world!"*
*print(message)*

В программе создана *переменная* с именем message. В каждой переменной хранится *значение*, то есть данные, связанные с переменной. В нашем случае значением является текст "Hello world!".

Советы по созданию переменных:

Имена переменных могут состоять только из букв, цифр и символов подчеркивания. Они могут начинаться с буквы или символа подчеркивания, но не с цифры. Например, переменной можно присвоить имя message_1, но не 1_message.

Пробелы в именах переменных запрещены, а для разделения слов в именах переменных используются символы подчеркивания. Например, имя greeting_message допустимо, а имя greeting message вызовет ошибку.

Не используйте имена функций и ключевые слова Python в качестве имен пере- менных; иначе говоря, не используйте слова, которые зарезервированы в Python для конкретной цели, — например, слово print (см. раздел «Ключевые слова и встроенные функции Python»).

Имена переменных должны быть короткими, но содержательными. Например, имя name лучше n, имя student_name лучше s_n, а имя name_length лучше length_of_persons_name.

Будьте внимательны при использовании строчной буквы l и прописной буквы O, потому что они похожи на цифры 1 и 0.

## Строки

Строка представляет собой простую последовательность символов. Любая последовательность символов, заключенная в кавычки, в Python считается строкой; при этом строки могут быть заключены как в одиночные, так и в двойные кавычки:

*"This is a string."*

*'This is also a string.'*

Изменение регистра символов в строках

*Input:*

*name = "ada lovelace"*

*print(name.title())*

*Output:*

*Ada Lovelace*

Метод title() преобразует первый символ каждого слова в строке к верхнему регистру, тогда как все остальные символы выводятся в нижнем регистре. Например, данная возможность может

быть полезна, если в вашей программе входные значения Ada, ADA и ada должны рассматриваться как одно и то же имя и все они должны отображаться в виде Ada.

Для работы с регистром также существуют другие полезные методы. Например, все символы строки можно преобразовать к верхнему или нижнему регистру:

*Input:*

*name = "Ada Lovelace"*

*print(name.upper())*

*print(name.lower())*

*Output:*

*ADA LOVELACE*

*ada lovelace*

### Именование переменных в строках

Input:

*first_name = "ada"*

*last_name = "lovelace"*

*full_name = f"{first_name} {last_name}"*

*print(full_name)*

## Числа

### Целые числа

В Python с целыми числами можно выполнять операции сложения (+), вычита- ния (-), умножения (*) и деления(/).

Input & Output:

*>>> **2 + 3***

*5*

*>>> **3 - 2***

*1*

*>>> **2 * 3***

*6*

*>>> **3 / 2***

*1.5*

В терминальном сеансе Python просто возвращает результат операции. Для представления операции возведения в степень в Python используется сдвоенный знак умножения:

Input & Output:

*>>> **3 ** 2***

*9*

*>>> **3 ** 3***

*27*

*>>> **10 ** 6***

*1000000*

## Вещественные числа

В Python числа, имеющие дробную часть, называются *вещественными* (или «чис- лами с плавающей точкой»). Обычно разработчик может просто пользоваться дробными значениями, не особенно задумываясь об их поведении. Просто введите нужные числа, а Python, скорее всего, сделает именно то, что вы от него хотите:

Input & Output:

*>>> **0.1 + 0.1***

*0.2*

*>>> **0.2 + 0.2***

*0.4*

*>>> **2 * 0.1***

*0.2*

*>>> **2 * 0.2***

*0.4*

## Целые и вещественные числа

При делении двух любых чисел — даже если это целые числа, частным от деления которых является целое число, — вы всегда получаете вещественное число:

Input & Output:

*>>> **4/2***

*2.0*

При смешении целого и вещественного числа в любой другой операции вы также получаете вещественное число:

Input & Output:

*>>> **1 + 2.0***

*3.0*

*>>> **2 * 3.0***

*6.0*

*>>> **3.0 ** 2***

*9.0*

Python по умолчанию использует вещественный тип для результата любой операции, в которой задействовано вещественное число, даже если результат является целым числом.

### Множественное присваивание

В одной строке программы можно присвоить значения сразу нескольким перемен- ным. Этот синтаксис сократит длину программы и упростит ее чтение; чаще всего он применяется при инициализации наборов чисел.

Например, следующая строка инициализирует переменные x, y и z нулями:

Input & Output:

*>>> x, y, z = 0, 0, 0*

Имена переменных должны разделяться запятыми, точно так же должны разделяться значения. Python присваивает каждое значение переменной в соответствующей позиции. Если количество значений соответствует количеству переменных, Python правильно сопоставит их друг с другом.

## Комментарии

В языке Python признаком комментария является символ «решетка» (#). Интерпретатор Python игнорирует все символы, следующие в коде после # до конца строки. Пример:

Input & Output:

*# Say hello to everyone.*

*print("Hello Python people!")*

*>>>Hello Python people!*

# Глава 2. Списки

## Что такое список?

Список представляет собой набор элементов, следующих в определенном порядке. Так как список обычно содержит более одного элемента, рекомендуется присваивать спискам имена во множественном числе: letters, digits, names и т. д.

В языке Python список обозначается квадратными скобками [], а отдельные элементы списка разделяются запятыми. Простой пример списка с названиями моделей велосипедов:

Input & Output:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles)
```

```
>>>['trek', 'cannondale', 'redline', 'specialized']
```

## Обращение к элементам списка

Списки представляют собой упорядоченные наборы данных, поэтому для обращения к любому элементу списка следует сообщить Python позицию (*индекс*) нужного элемента. Чтобы обратиться к элементу в списке, укажите имя списка, за которым следует индекс элемента в квадратных скобках.

Например, название первого велосипеда в списке bicycles выводится следующим образом:

Input & Output:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0])
```

```
>>>trek
```

## Индексы начинаются с 0, а не с 1

Python считает, что первый элемент списка находится в позиции 0, а не в позиции 1.

Второму элементу списка соответствует индекс 1. В этой простой схеме индекс любого элемента вычисляется уменьшением на 1 его позиции в списке. Например, чтобы обратиться к четвертому элементу списка, следует запросить элемент с индексом 3.

Input & Output:

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']

print(bicycles[1])

print(bicycles[3])
```

```
>>>cannondale

>>>specialized
```

Если запросить элемент с индексом –1, Python всегда возвращает последний элемент в списке:

Input & Output:

*bicycles = ['trek', 'cannondale', 'redline', 'specialized']*

*print(bicycles[-1])*

*>>>specialized*

## Использование отдельных элементов из списка

Отдельные значения из списка используются так же, как и любые другие переменные. Например, вы можете воспользоваться f-строками для построения сообщения, содержащего значение из списка.

Попробуем извлечь название первого велосипеда из списка и составить сообщение, включающее это значение.

Input & Output:

*bicycles = ['trek', 'cannondale', 'redline', 'specialized']*

*message = f"My first bicycle was a {bicycles[0].title()}." print(message)*

*>>>My first bicycle was a Trek.*

## Изменение элементов в списке

Чтобы изменить элемент, укажите имя списка и индекс изменяемого элемента в квадратных скобках; далее задайте новое значение, которое должно быть присвоено элементу.

Input & Output:

*motorcycles = ['honda', 'yamaha', 'suzuki']*

*print(motorcycles)*

*motorcycles[0] = 'ducati'*

*print(motorcycles)*

*>>>['honda', 'yamaha', 'suzuki']*

*>>>['ducati', 'yamaha', 'suzuki']*

Из вывода видно, что первый элемент действительно изменился, тогда как остальные элементы списка сохранили прежние значения.

Изменить можно значение любого элемента в списке, не только первого.

## Добавление элементов в список

## Присоединение элементов в конец списка

Input & Output:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

print(motorcycles)

motorcycles.append('ducati')

print(motorcycles)

>>>['honda', 'yamaha', 'suzuki']

>>>['honda', 'yamaha', 'suzuki', 'ducati']
```

Метод append() присоединяет строку 'ducati' в конец списка, другие элементы в списке при этом остаются неизменными.

## Вставка элементов в список

Метод insert() позволяет добавить новый элемент в произвольную позицию списка. Для этого следует указать индекс и значение нового элемента.

Input & Output:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

motorcycles.insert(0, 'ducati')

print(motorcycles)

>>>['ducati', 'honda', 'yamaha', 'suzuki']
```

## Удаление элементов из списка

## Удаление элемента с использованием команды del

Если вам известна позиция элемента, который должен быть удален из списка, воспользуйтесь командой del.

Input & Output:

```
motorcycles = ['honda', 'yamaha', 'suzuki']

print(motorcycles)

del motorcycles[0]

print(motorcycles)

>>>['yamaha', 'suzuki']
```

В точке вызов del удаляет первый элемент, 'honda', из списка motorcycles.

## Удаление элемента с использованием метода pop()

Метод pop() удаляет последний элемент из списка, но позволяет работать с ним после удаления.

Удалим мотоцикл из списка:

Input & Output:

*motorcycles = ['honda', 'yamaha', 'suzuki']*

*print(motorcycles)*

*popped_motorcycle = motorcycles.pop()*

*print(motorcycles)*

*print(popped_motorcycle)*

*>>>['honda', 'yamaha', 'suzuki']*

*>>>['honda', 'yamaha']*

*>>>suzuki*

Сначала определяется и выводится содержимое списка motorcycles. В точке значение извлекается из списка и сохраняется в переменной с именем popped_motorcycle. Вывод измененного списка в точке показывает, что значение было удалено из списка. Затем мы выводим извлеченное значение в точке, демонстрируя, что удаленное из списка значение остается доступным в программе.

Из вывода видно, что значение 'suzuki', удаленное в конце списка, теперь хранится в переменной popped_motorcycle.

## Извлечение элементов из произвольной позиции списка

Вызов pop() может использоваться для удаления элемента в произвольной позиции списка; для этого следует указать индекс удаляемого элемента в круглых скобках.

Input & Output:

*motorcycles = ['honda', 'yamaha', 'suzuki']*

*first_owned = motorcycles.pop(0)*

*print(f"The first motorcycle I owned was a {first_owned.title()}.")*

*>>>The first motorcycle I owned was a Honda.*

Сначала первый элемент извлекается из списка в точке, а затем выводится сообщение об этом мотоцикле.

Помните, что после каждого вызова pop() элемент, с которым вы работаете, уже не находится в списке.

## Удаление элементов по значению

Иногда позиция удаляемого элемента неизвестна. Если вы знаете только значение элемента, используйте метод remove().

Из списка нужно удалить значение 'ducati':

Input & Output:

```
motorcycles = ['honda', 'yamaha', 'suzuki', 'ducati']

print(motorcycles)

motorcycles.remove('ducati')

print(motorcycles)
```

```
>>>['honda', 'yamaha', 'suzuki', 'ducati']

>>>['honda', 'yamaha', 'suzuki']
```

Код приказывает Python определить, в какой позиции списка находится значение 'ducati', и удалить этот элемент.

## Упорядочение списка

### Постоянная сортировка списка методом sort()

Метод sort() позволяет относительно легко отсортировать список. Например, имеется список машин и вы хотите переупорядочить эти элементы по алфавиту.

Input & Output:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

cars.sort()

print(cars)
```

```
>>>['audi', 'bmw', 'subaru', 'toyota']
```

Метод sort() в точке осуществляет постоянное изменение порядка элементов в списке. Названия машин располагаются в алфавитном порядке, и вернуться к исходному порядку уже не удастся.

Список также можно отсортировать в обратном алфавитном порядке; для этого методу sort() следует передать аргумент reverse=True. В следующем примере список сортируется в порядке, обратном алфавитному:

Input & Output:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

cars.sort(reverse=True)

print(cars)
```

```
>>>['toyota', 'subaru', 'bmw', 'audi']
```

### Временная сортировка списка функцией sorted()

Чтобы сохранить исходный порядок элементов списка, но временно представить их в отсортированном порядке, можно воспользоваться функцией sorted(). Функция sorted() позволяет представить список в определенном порядке, но не изменяет фактический порядок элементов в списке.

Input & Output:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Here is the original list:")

print(cars)

print("\nHere is the sorted list:")

print(sorted(cars))

print("\nHere is the original list again:")

print(cars)

>>>Here is the original list:

>>>['bmw', 'audi', 'toyota', 'subaru']

>>>Here is the sorted list:

>>>['audi', 'bmw', 'subaru', 'toyota']

>>>Here is the original list again:

>>>['bmw', 'audi', 'toyota', 'subaru']
```

Сначала список выводится в исходном порядке, а затем в алфавитном порядке. После того как список будет выведен в новом порядке, мы убеждаемся в том, что список все еще хранится в исходном порядке.

## Вывод списка в обратном порядке

Чтобы переставить элементы списка в обратном порядке, используйте метод reverse(). Скажем, если список машин первоначально хранился в хронологическом порядке даты приобретения, элементы можно легко переупорядочить в обратном хронологическом порядке:

Input & Output:

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print(cars)

cars.reverse()

print(cars)

>>>['bmw', 'audi', 'toyota', 'subaru']

>>>['subaru', 'toyota', 'audi', 'bmw']
```

## Определение длины списка

Вы можете быстро определить длину списка с помощью функции len(). Список в нашем примере состоит из четырех элементов, поэтому его длина равна 4:

Input & Output:

*cars = ['bmw', 'audi', 'toyota', 'subaru']*

*len(cars)*

*>>>4*

## Ошибки индексирования при работе со списками

Когда программист только начинает работать со списками, он часто допускает одну характерную ошибку. Допустим, имеется список с тремя элементами и программа запрашивает четвертый элемент:

Input & Output:

*motorcycles = ['honda', 'yamaha', 'suzuki']*

*print(motorcycles[3])*

В этом случае происходит *ошибка индексирования*:

*Traceback (most recent call last):*

*File "motorcycles.py", line 3, in <module>*

*print(motorcycles[3])*

*IndexError: list index out of range*

Python пытается вернуть элемент с индексом 3. Однако при поиске по списку ни один элемент motorcycles не обладает индексом 3. Из-за смещения индексов на 1 эта ошибка весьма распространена.

# Глава 3. Работа со списками

## Перебор всего списка

В ситуациях, требующих применения одного действия к каждому элементу списка, можно воспользоваться циклами for.

Допустим, имеется список с именами фокусников и вы хотите вывести каждое имя из списка.

Input & Output:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
        print(magician)

>>>alice
>>>david
>>>carolina
```

Все начинается с определения списка. Определяется цикл for. Эта строка приказывает Python взять очередное имя из списка и сохранить его в переменной magician. Затем выводится имя, только что сохраненное в переменной magician. Дальше строки повторяются для каждого имени в списке. Результат представляет собой простой перечень имен из списка.

## Подробнее о циклах

В цикле, использованном в magicians .py, Python сначала читает первую строку цикла:

*for magician in magicians:*

Эта строка означает, что нужно взять первое значение из списка magicians и сохранить его в переменной magician. Первое значение в списке — 'alice'. Затем Python читает следующую строку:

*print(magician)*

Python выводит текущее значение magician, которое все еще равно 'alice'. Так как в списке еще остались другие значения, Python возвращается к первой строке цикла:

*for magician in magicians:*

Python берет следующее значение из списка, 'david', и сохраняет его в magician.

Затем выполняется строка:

*print(magician)*

Python снова выводит текущее значение magician; теперь это строка 'david'. Весь цикл повторяется еще раз с последним значением в списке, 'carolina'. Так как других значений в списке не осталось, Python переходит к следующей строке в программе. В данном случае после цикла for ничего нет, поэтому программа просто завершается.

Помните, что все действия повторяются по одному разу для каждого элемента в списке независимо от их количества.

Также помните, что при написании собственных циклов for временной переменной для текущего значения из списка можно присвоить любое имя. Однако на практике рекомендуется выбирать осмысленное имя, описывающее отдельный элемент списка. Несколько примеров:

*for cat in cats:*
*for dog in dogs:*
*for item in list_of_items:*

Выполнение этого правила поможет вам проследить за тем, какие действия выполняются с каждым элементом в цикле for. В зависимости от того, какое число используется — одиночное или множественное, вы сможете понять, работает ли данная часть кода с отдельным элементом из списка или со всем списком.

## Более сложные действия в циклах for

В цикле for с каждым элементом списка может выполняться практически любое действие. Дополним предыдущий пример, чтобы программа выводила для каждого фокусника отдельное сообщение:

Input & Output:

```
magicians = ['alice', 'david', 'carolina']

for magician in magicians:

        print(f"{magician.title()}, that was a great trick!")
```

```
>>>Alice, that was a great trick!
>>>David, that was a great trick!
>>>Carolina, that was a great trick!
```

Тело цикла for может содержать сколько угодно строк кода.

Включим в сообщение для каждого фокусника вторую строку:

Input & Output:

```
 magicians = ['alice', 'david', 'carolina']
 for magician in magicians:
        print(f"{magician.title()}, that was a great trick!")
        print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

```
>>>Alice, that was a great trick!
>>>I can't wait to see your next trick, Alice.

>>>David, that was a great trick!
>>>I can't wait to see your next trick, David.

>>>Carolina, that was a great trick!
>>>I can't wait to see your next trick, Carolina.
```

Так как оба вызова print() снабжены отступами, каждая строка будет выполнена по одному разу для каждого фокусника в списке. Символ новой строки ("\n") во второй команде print вставляет пустую строку после каждого прохода цикла. В результате будет создан набор сообщений, аккуратно сгруппированных для каждого фокусника в списке.

## Выполнение действий после цикла for

Каждая строка кода после цикла for, не имеющая отступа, выполняется без повторения.

Input & Output:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
        print(f"{magician.title()}, that was a great trick!")
        print(f"I can't wait to see your next trick, {magician.title()}.\n")
print("Thank you, everyone. That was a great magic show!")
```

```
>>>Alice, that was a great trick!
>>>I can't wait to see your next trick, Alice.

>>>David, that was a great trick!
>>>I can't wait to see your next trick, David.

>>>Carolina, that was a great trick!
>>>I can't wait to see your next trick, Carolina.

>>>Thank you, everyone. That was a great magic show!
```

Первые две команды print повторяются по одному разу для каждого фокусника в списке, как было показано ранее. Но поскольку строка отступа не имеет, это сообщение выводится только один раз.

При обработке данных в циклах for завершающее сообщение позволяет подвести итог операции, выполненной со всем набором данных.

## Предотвращение ошибок с отступами

В Python связь одной строки кода с предшествующей строкой обозначается отступами. В приведенных примерах строки, выводившие сообщения для отдельных фокусников, были частью цикла, потому что они были снабжены отступами. Применение отступов в Python сильно упрощает чтение кода.

Несколько типичных ошибок при использовании отступов.

## Пропущенный отступ

Строка после команды for в цикле всегда должна снабжаться отступом.

Input & Output:

```
magicians = ['alice', 'david', 'carolina']

for magician in magicians:

print(magician)
```

```
>>>File "magicians.py", line 3
>>>   print(magician)
>>>    ^
>>>IndentationError: expected an indented block
```

Команда print в точке должна иметь отступ, но здесь его нет. Когда Python ожидает увидеть блок с отступом, но не находит его, появляется сообщение с указанием номера строки.

## Пропущенные отступы в других строках

Иногда цикл выполняется без ошибок, но не выдает ожидаемых результатов. Такое часто происходит, когда вы пытаетесь выполнить несколько операций в цикле, но забываете снабдить отступом некоторые из строк.

Input & Output:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
        print(magician.title() + ", that was a great trick!")
print(f"I can't wait to see your next trick, {magician.title()}.\n")
```

```
>>>Alice, that was a great trick!
>>>David, that was a great trick!
>>>Carolina, that was a great trick!
>>>I can't wait to see your next trick, Carolina.
```

Команда print должна быть снабжена отступом, но поскольку Python находит хотя бы одну строку с отступом после команды for, сообщение об ошибке не выдается. В результате первая команда print будет выполнена для каждого элемента в списке, потому что в ней есть отступ. Вторая команда print отступа не имеет, поэтому она будет выполнена только один раз после завершения цикла. Так как последним значением magician является строка 'carolina', второе сообщение будет выведено только с этим именем.

Это пример *логической ошибки*. Код имеет действительный синтаксис, но он не приводит к желаемому результату, потому что проблема кроется в его логике. Если некоторое действие должно повторяться для каждого элемента в списке, но выполняется только один раз, проверьте, не нужно ли добавить отступы в строке или нескольких строках кода.

## Лишние отступы

Если вы случайно поставите отступ в строке, в которой он не нужен, Python сообщит об этом:

Input & Output:

```
message = "Hello Python world!"

        print(message)
```

```
>>>File "hello_world.py", line 2
>>>   print(message)
>>>   ^
>>>IndentationError: unexpected indent
```

Отступ команды print в точке не нужен, потому что эта строка не подчинена предшествующей; Python сообщает об ошибке.

## Лишние отступы после цикла

Если вы случайно снабдите отступом код, который должен выполняться *после* за- вершения цикла, то этот код будет выполнен для каждого элемента. Иногда Python выводит сообщение об ошибке, но часто дело ограничивается простой логической ошибкой.

Например, что произойдет, если случайно снабдить отступом строку с выводом завершающего приветствия для группы фокусников?

Input & Output:

```
magicians = ['alice', 'david', 'carolina']
for magician in magicians:
        print(f"{magician.title()}, that was a great trick!")
        print(f"I can't wait to see your next trick, {magician.title()}.\n")
        print("Thank you everyone, that was a great magic show!")
```

```
>>>Alice, that was a great trick!
>>>I can't wait to see your next trick, Alice.
>>>Thank you everyone, that was a great magic show!

>>>David, that was a great trick!
>>>I can't wait to see your next trick, David.
>>>Thank you everyone, that was a great magic show!

>>>Carolina, that was a great trick!
>>>I can't wait to see your next trick, Carolina.
>>>Thank you everyone, that was a great magic show!
```

Так как строка имеет отступ, сообщение будет продублировано для каждого фокусника в списке.

## Пропущенное двоеточие

Двоеточие в конце команды for сообщает Python, что следующая строка является началом цикла.

Input & Output:

```
magicians = ['alice', 'david', 'carolina'] ❶

for magician in magicians

        print(magician)
```

## Создание числовых списков

Списки идеально подходят для хранения наборов чисел, а Python предоставляет специальные средства для эффективной работы с числовыми списками.

## Функция range()

Функция range() упрощает построение числовых последовательностей. Например, с ее помощью можно легко вывести серию чисел:

Input & Output:

```
for value in range(1,5):

        print(value)
```

```
>>>1 2 3 4
```

В этом примере range() выводит только числа от 1 до 4. Еще один пример явления «смещения на 1».. При выполнении функции range() Python начинает отсчет от первого переданного значения и прекращает его при достижении второго. Так как на втором значении происходит остановка, конец интервала (5 в данном случае) не встречается в выводе.

Чтобы вывести числа от 1 до 5, используйте вызов range(1,6):

Input & Output:

```
for value in range(1,6):
        print(value)
```

```
>>>1 2 3 4 5
```

Если ваша программа при использовании range() выводит не тот результат, на который вы рассчитывали, попробуйте увеличить конечное значение на 1.

При вызове range() также можно передать только один аргумент; в этом случае последовательность чисел будет начинаться с 0. Например, range(6) вернет числа от 0 до 5.

## Использование range() для создания числового списка

Если вы хотите создать числовой список, преобразуйте результаты range() в список при помощи функции list(). Если заключить вызов range() в list(), то результат будет представлять собой список с числовыми элементами.

В примере из предыдущего раздела числовая последовательность просто выводилась на экран. Тот же набор чисел можно преобразовать в список вызовом list():

Input & Output:

```
numbers = list(range(1,6))
print(numbers)
```

```
>>>[1, 2, 3, 4, 5]
```

## Простая статистика с числовыми списками

Некоторые функции Python предназначены для работы с числовыми списками. Например, вы можете легко узнать минимум, максимум и сумму числового списка:

Input & Output:

```
>>> digits = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

 >>> min(digits)

0

>>> max(digits)

9

>>> sum(digits) 45
```

## Генераторы списков

*Генератор списка* (list comprehension) позволяет сгенерировать список всего в одной строке. Генератор списка объединяет цикл for и создание новых элементов в одну строку и автоматически присоединяет к списку все новые элементы. В следующем примере список квадратов, строится с использованием генератора списка:

Input & Output:

```
squares = [value**2 for value in range(1,11)]
print(squares)
```

```
>>>[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

В данном примере это выражение value**2, которое возводит значение во вторую степень. Цикл for в данном примере — for value in range(1,11) — передает значения с 1 до 10 выражению value**2. Обратите внимание на отсутствие двоеточия в конце команды for.

## Работа с частью списка

Также можно работать с конкретным подмножеством элементов списка; в Python такие подмножества называются *сегментами* (slices).

## Создание сегмента

Чтобы создать сегмент на основе списка, следует задать индексы первого и последнего элементов, с которыми вы намереваетесь работать. Чтобы вывести первые три элемента списка, запросите индексы с 0 по 3, и вы получите элементы 0, 1 и 2.

Input & Output:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[0:3])
```

```
>>>['charles', 'martina', 'michael']
```

Подмножество может включать любую часть списка. Например, чтобы ограничить- ся вторым, третьим и четвертым элементами списка, создайте сегмент, который начинается с индекса 1 и заканчивается на индексе 4:

Input & Output:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[1:4])
```

```
>>>['martina', 'michael', 'florence']
```

Если первый индекс сегмента не указан, то Python автоматически начинает сегмент от начала списка:

Input & Output:

```
players = ['charles', 'martina', 'michael', 'florence', 'eli']

print(players[:4])
```

```
>>>['charles', 'martina', 'michael', 'florence']
```

Аналогичный синтаксис работает и для сегментов, включающих конец списка. Например, если вам нужны все элементы с третьего до последнего, начните с индекса 2 и не указывайте второй индекс:

Input & Output:

*players = ['charles', 'martina', 'michael', 'florence', 'eli']*

*print(players[2:])*

*>>>['michael', 'florence', 'eli']*

Этот синтаксис позволяет вывести все элементы от любой позиции до конца списка независимо от его длины. Отрицательный индекс возвращает элемент, находящийся на заданном расстоянии от конца списка; следовательно, вы можете получить любой сегмент от конца списка. Чтобы отобрать последних трех игроков из списка, используйте сегмент players[-3:]:

Input & Output:

*players = ['charles', 'martina', 'michael', 'florence', 'eli']*

*print(players[-3:])*

*>>>['michael', 'florence', 'eli']*

## Перебор содержимого сегмента

Если вы хотите перебрать элементы, входящие в подмножество элементов, используйте сегмент в цикле for. В следующем примере программа перебирает первых трех игроков и выводит их имена:

Input & Output:

*players = ['charles', 'martina', 'michael', 'florence', 'eli']*

*print("Here are the first three players on my team:")*

*for player in players[:3]:*

    *print(player.title())*

*>>>Here are the first three players on my team:*
*>>>Charles*
*>>>Martina*
*>>>Michael*

## Копирование списка

Чтобы скопировать список, создайте сегмент, включающий весь исходный список без указания первого и второго индекса ([:]). Эта конструкция создает сегмент, который начинается с первого элемента и завершается последним; в результате создается копия всего списка.

Input & Output:

*my_foods = ['pizza', 'falafel', 'carrot cake']*

*friend_foods = my_foods[:]*

*print("My favourite foods are:")*

*print(my_foods)*

*print("\nMy friend's favourite foods are:")*

*print(friend_foods)*

*>>>My favourite foods are:*
*>>>['pizza', 'falafel', 'carrot cake']*

*>>>My friend's favourite foods are:*
*>>>['pizza', 'falafel', 'carrot cake']*

В точке создается список блюд с именем my_foods. Затем создается другой список с именем friend_foods. Чтобы создать копию my_foods, программа запрашивает сегмент my_foods без указания индексов и сохраняет копию в friend_foods. При выводе обоих списков становится видно, что оба списка содержат одинаковые данные.

Речь в действительности идет о двух разных списках, добавим новое блюдо в каждый список:

Input & Output:

*my_foods = ['pizza', 'falafel', 'carrot cake']*

*friend_foods = my_foods[:]*

*my_foods.append('cannoli')*

*friend_foods.append('ice cream')*

*print("My favorite foods are:")*

*print(my_foods)*

*print("\nMy friend's favorite foods are:")*

*print(friend_foods)*

*>>>My favourite foods are:*

*>>>['pizza', 'falafel', 'carrot cake', 'cannoli']*

*>>>My friend's favourite foods are:*

*>>>['pizza', 'falafel', 'carrot cake', 'ice cream']*

В точке исходные элементы my_foods копируются в новый список friend_foods, как было сделано в предыдущем примере. Затем вкаждый список добавляется новый элемент: 'cannoli' в my_foods, а 'ice cream' добавляется в friend_foods.

Вывод показывает, что элемент 'cannoli' находится в списке my_foods, а элемент 'ice cream' в этот список не входит. В точке видно, что 'ice cream' входит в список friend_foods, а элемент 'cannoli' в этот список не входит. Вот что происходит при попытке копирования списка без использования сегмента:

Input & Output:

*my_foods = ['pizza', 'falafel', 'carrot cake']*

*# Не работает:*

*friend_foods = my_foods*

*my_foods.append('cannoli')*

*friend_foods.append('ice cream')*
*print("My favourite foods are:")*

*print(my_foods)*

*print("\nMy friend's favourite foods are:")*

*print(friend_foods)*

*>>>My favourite foods are:*
*>>>['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']*

*>>>My friend's favourite foods are:*
*>>>['pizza', 'falafel', 'carrot cake', 'cannoli', 'ice cream']*

Вместо того чтобы сохранять копию my_foods в friend_foods в точке , мы задаем friend_foods равным my_foods. Этот синтаксис в действительности сообщает Python, что новая переменная friend_foods должна быть связана со списком, уже хранящимся в my_foods, поэтому теперь обе переменные связаны с одним списком. В результате при добавлении элемента 'cannoli' в my_foods этот элемент также появляется в friend_foods. Аналогичным образом элемент 'ice cream' появляется в обоих списках, хотя на первый взгляд он был добавлен только в friend_foods.

Вывод показывает, что оба списка содержат одинаковые элементы, а это совсем не то, что требовалось.

В двух словах, если при работе с копией списка происходит что-то непредвиденное, убедитесь в том, что список копируется с использованием сегмента, как это делается в нашем первом примере .

## Кортежи

Списки хорошо подходят для хранения наборов элементов, которые могут из- меняться на протяжении жизненного цикла программы. Например, возможность модификации списков жизненно необходима при работе со списками пользовате- лей сайта или списками персонажей игры. Однако в некоторых ситуациях требуется создать список элементов, который не может изменяться. *Кортежи* (tuples) предоставляют именно такую возможность. В языке Python значения, которые не могут изменяться, называются *неизменяемыми* (immutable), а неизменяемый список называется *кортежем*.

## Определение кортежа

Кортеж выглядит как список, не считая того, что вместо квадратных скобок используются круглые скобки. После определения кортежа вы можете обращаться к его отдельным элементам по индексам точно так же, как это делается при работе со списком.

Например, имеется прямоугольник, который в программе всегда должен иметь строго определенные размеры. Чтобы гарантировать неизменность размеров, можно объединить размеры в кортеж:

Input & Output:

*dimensions = (200, 50)*

*print(dimensions[0])*

*print(dimensions[1])*

*>>>200*

*>>>50*

Определяется кортеж dimensions, при этом вместо квадратных скобок используются круглые.

Каждый элемент кортежа выводится по отдельности с использованием того же синтаксиса, который использовался для обращения к элементу списка.

Посмотрим, что произойдет при попытке изменения одного из элементов в кортеже:

Input & Output:

*dimensions = (200, 50)*

*dimensions[0] = 250*

*>>>Traceback (most recent call last):*
*>>>File "dimensions.py", line 3, in <module>*
*>>>  dimensions[0] = 250*
*>>>TypeError: 'tuple' object does not support item assignment*

Код пытается изменить первое значение, но Python возвращает ошибку типа. По сути, так как мы пытаемся изменить кортеж, а эта операция недопустима для объектов этого типа, Python сообщает о невозможности присваивания нового значения элементу в кортеже.

## Перебор всех значений в кортеже

Для перебора всех значений в кортеже используется цикл for, как и при работе со списками:

Input & Output:

*dimensions = (200, 50)*

*for dimension in dimensions:*

    *print(dimension)*

*>>>200*

*>>>50*

Python возвращает все элементы кортежа по аналогии с тем, как это делается со списком.

## Замена кортежа

Элементы кортежа не могут изменяться, но вы можете присвоить новое значение переменной, в которой хранится кортеж. Таким образом, для изменения размеров прямоугольника следует переопределить весь кортеж:

Input & Output:

*dimensions = (200, 50)*

*print("Original dimensions:")*

*for dimension in dimensions:*

    *print(dimension)*

```
dimensions = (400, 100)

print("\nModified dimensions:")

for dimension in dimensions:

        print(dimension)

>>>Original dimensions:
>>>200
>>>50

>>>Modified dimensions:
>>>400
>>>100
```

Блок, определяет исходный кортеж и выводит исходные размеры. В переменной dimensions сохраняется новый кортеж, после чего в точке вводятся новые размеры.

По сравнению со списками структуры данных кортежей относительно просты. Используйте их для хранения наборов значений, которые не должны изменяться на протяжении жизненного цикла программы.

## Стиль программирования

Не жалейте времени на то, чтобы ваш код читался как можно проще. Понятный код помогает следить за тем, что делает ваша программа, и упрощает изучение вашего кода другими разработчиками.

## Рекомендации по стилю

Руководство по стилю Python было написано с пониманием того факта, что код читается чаще, чем пишется. Вы пишете свой код один раз, а потом начинаете читать его, когда переходите к отладке. При расширении функциональности программы вы снова тратите время на чтение своего кода. А когда вашим кодом начинают пользоваться другие программисты, они тоже читают его.

Выбирая между написанием кода, который проще пишется, и кодом, который проще читается, программисты Python почти всегда рекомендуют второй вариант. Следующие советы помогут вам с самого начала писать чистый, понятный код.

## Отступы

PEP 8 рекомендует обозначать уровень отступа четырьмя пробелами. Использование четырех пробелов упрощает чтение программы и при этом оставляет достаточно места для нескольких уровней отступов в каждой строке.

## Длина строк

Многие программисты Python рекомендуют ограничивать длину строк 80 символами.

## Пустые строки

*Пустые строки* применяются для визуальной группировки частей программы. Используйте пустые строки для структурирования файлов, но не злоупотребляйте ими.

## Глава 4. Команды if

Команда if в языке Python позволяет проверить текущее состояние программы и выбрать дальнейшие действия в зависимости от результатов проверки.

Допустим, у вас имеется список машин и вы хотите вывести название каждой машины. Названия большинства машин должны записываться с капитализацией (первая буква в верхнем регистре, остальные в нижнем). С другой стороны, значение 'bmw' должно записываться в верхнем регистре. Следующий код перебирает список названий машин и ищет в нем значение 'bmw'. Для всех элементов, содержащих значение 'bmw', значение выводится в верхнем регистре:

Input & Output:

```
cars = ['audi', 'bmw', 'subaru', 'toyota']

for car in cars:

        if car == 'bmw':

                print(car.upper())

        else:

                print(car.title())
```

```
>>>Audi
>>>BMW
>>>Subaru
>>>Toyota
```

Цикл в этом примере сначала проверяет, содержит ли car значение 'bmw'. Если проверка дает положительный результат, то значение выводится в верхнем регистре. Если car содержит все, что угодно, кроме 'bmw', то при выводе значения применяется капитализация.

## Проверка условий

В каждой команде if центральное место занимает выражение, результатом которого является логическая истина (True) или логическая ложь (False); это выражение называется *условием*. В зависимости от результата проверки Python решает, должен ли выполняться код в команде if. Если результат условия равен True, то Python выполняет код, следующий за командой if. Если же будет получен результат False, то Python игнорирует код, следующий за командой if.

## Проверка равенства

Во многих условиях текущее значение переменной сравнивается с конкретным значением, интересующим вас. Простейшее условие проверяет, равно ли значение переменной конкретной величине:

Input & Output:

```
car = 'bmw'

car == 'bmw'
```

```
>>>True
```

В строке переменной car присваивается значение 'bmw'; Строка проверяет, равно ли значение car строке 'bmw'; для проверки используется двойной знак равенства (==). Этот оператор возвращает True, если значения слева и справа от оператора равны; если же значения не совпадают, оператор возвращает False.

Если car принимает любое другое значение вместо 'bmw', проверка возвращает False:

Input & Output:

*car = 'audi'*

*car == 'bmw'*

*>>>False*

Одиночный знак равенства выполняет операцию; код можно прочитать в форме «Присвоить car значение 'audi'». С другой стороны, двойной знак равенства, как в строке , задает вопрос: «Значение car равно 'bmw'?».

## Проверка равенства без учета регистра

В языке Python проверка равенства выполняется с учетом регистра. Например, два значения с разным регистром символов равными не считаются:

Input & Output:

*car = 'Audi'*

*car == 'audi'*

*>>>False*

Если проверка должна выполняться на уровне символов без учета регистра, преобразуйте значение переменной к нижнему регистру перед выполнением сравнения:

Input & Output:

*car = 'Audi'*

*car.lower() == 'audi'*

*>>>True*

Условие возвращает True независимо от регистра символов 'Audi', потому что проверка теперь выполняется без учета регистра. Функция lower() не изменяет значение, которое изначально хранилось в car, так что сравнение не отражается на исходной переменной:

Input & Output:

*car = 'Audi'*

*car.lower() == 'audi'*

*>>>True*

*car*

*>>>'Audi'*

Строка 'Audi' сохраняется в переменной car. Значение car приводится к нижнему регистру и сравнивается со значением строки 'audi', также записанным в нижнем регистре. Две строки совпадают, поэтому Python возвращает True. Вывод в точке показывает, что значение, хранящееся в car, не изменилось при вызове lower().

## Проверка неравенства

Если вы хотите проверить, что два значения *различны*, используйте комбинацию из восклицательного знака и знака равенства (!=). Восклицательный знак представляет отрицание.

В переменной хранится заказанный топпинг к пицце; если клиент не заказал анчоусы (anchovies), программа выводит сообщение:

Input & Output:

```
requested_topping = 'mushrooms'

if requested_topping != 'anchovies':

        print("Hold the anchovies!")

>>>Hold the anchovies!
```

Строка сравнивает значение requested_topping со значением 'anchovies'. Если эти два значения не равны, Python возвращает True и выполняет код после команды if. Если два значения равны, Python возвращает False и не выполняет код после команды if.

Так как значение requested_topping отлично от 'anchovies', команда print будет выполнена.

## Сравнения чисел

Проверка числовых значений достаточно прямолинейна. Например, следующий код проверяет, что переменная age равна 18:

Input & Output:

```
age = 18

age == 18

>>>True
```

Также можно проверить условие неравенства двух чисел.

Input & Output:

```
answer = 17

if answer != 42:

        print("That is not the correct answer. Please try again!")

>>>That is not the correct answer. Please try again!
```

Условие выполняется, потому что значение answer (17) не равно 42. Так как условие истинно, блок с отступом выполняется.

В условные команды также можно включать всевозможные математические сравнения: меньше, меньше или равно, больше, больше или равно:

Input & Output:

*age = 19*

*age < 21*

*>>>True*

*age <= 21*

*>>>True*

*age > 21*

*>>>False*

*age >= 21*

*>>>False*

Все эти математические сравнения могут использоваться в условиях if, что повышает точность формулировки интересующих вас условий.

## Проверка нескольких условий

Иногда требуется проверить несколько условий одновременно. Например, для выполнения действия бывает нужно, чтобы истинными были сразу два условия; в других случаях достаточно, чтобы истинным было хотя бы одно из двух условий. Ключевые слова and и or помогут вам в подобных ситуациях.

## Использование and для проверки нескольких условий

Чтобы проверить, что два условия истинны *одновременно*, объедините их ключевым словом and; если оба условия истинны, то и все выражение тоже истинно. Если хотя бы одно (или оба) условие ложно, то и результат всего выражения равен False.

Например, чтобы убедиться в том, что каждому из двух людей больше 21 года, используйте следующую проверку:

Input & Output:

*age_0 = 22*

*age_1 = 18*

*age_0 >= 21 and age_1 >= 21*

*>>>False*

*age_1 = 22*

*age_0 >= 21 and age_1 >= 21*

*>>>True*

Определяются две переменные, age_0 и age_1. Программа проверяет, что оба значения равны 21 и более. Левое условие выполняется, а правое нет, поэтому все условное выражение дает результат False. В переменной age_1 присваивается значение 22. Теперь значение age_1 больше 21; обе проверки проходят, а все условное выражение дает истинный результат.

Чтобы код лучше читался, отдельные условия можно заключить в круглые скобки, но это не обязательно. С круглыми скобками проверка может выглядеть так:

*(age_0 >= 21) and (age_1 >= 21)*

## Использование or для проверки нескольких условий

Ключевое слово or тоже позволяет проверить несколько условий, но результат общей проверки является истинным в том случае, когда истинно хотя бы одно или оба условия. Ложный результат достигается только в том случае, если оба условия ложны.

Вернемся к примеру с возрастом, но на этот раз проверим, что хотя бы одна из двух переменных больше 21:

Input & Output:

*age_0 = 22*

*age_1 = 18*

*age_0 >= 21 or age_1 >= 21*

*>>>True*

*age_0 = 18*

*age_0 >= 21 or age_1 >= 21*

*>>>False*

Как и в предыдущем случае, определяются две переменные. Так как условие для age_0 истинно, все выражение также дает истинный результат. Затем значение age_0 уменьшается до 18. При проверке оба условия оказываются ложными, и общий результат всего выражения тоже ложен.

## Проверка вхождения значений в список

Иногда бывает важно проверить, содержит ли список некоторое значение, пре- жде чем выполнять действие. Например, перед завершением регистрации нового пользователя на сайте можно проверить, существует ли его имя в списке имен действующих пользователей, или в картографическом проекте определить, входит ли передаваемое место в список известных мест на карте.

Чтобы узнать, присутствует ли заданное значение в списке, воспользуйтесь клю- чевым словом in. Допустим, вы пишете программу для пиццерии. Вы создали список дополнений к пицце, заказанных клиентом, и хотите проверить, входят ли некоторые дополнения в этот список.

Input & Output:

*requested_toppings = ['mushrooms', 'onions', 'pineapple']*

*'mushrooms' in requested_toppings*

*>>>True*

*'pepperoni' in requested_toppings*

*>>>False*

Ключевое слово in приказывает Python проверить, входят ли значения 'mushrooms' и 'pepperoni' в список requested_toppings. Это весьма полезно, потому что вы можете создать список значений, критичных для вашей программы, а затем легко проверить, присутствует ли проверяемое значение в списке.

## Проверка отсутствия значения в списке

В других случаях программа должна убедиться в том, что значение *не входит* в список. Для этого используется ключевое слово not. Для примера рассмотрим список пользователей, которым запрещено писать комментарии на форуме. Прежде чем разрешить пользователю отправку комментария, можно проверить, не был ли пользователь включен в черный список:

Input & Output:

*banned_users = ['andrew', 'carolina', 'david']*

*user = 'marie'*

*if user not in banned_users:*

    *print(f"{user.title()}, you can post a response if you wish.")*

*>>>Marie, you can post a response if you wish.*

Строка достаточно четко читается: если пользователь не входит в черный список banned_users, то Python возвращает True и выполняет строку с отступом.

Пользователь 'marie' в этот список не входит, поэтому программа выводит соответствующее сообщение.

## Логические выражения

Это всего лишь другое название для проверки условия. Результат логического выражения равен True или False, как и результат условного выражения после его вычисления.

Логические выражения часто используются для проверки некоторых условий.

*game_active = True*
*can_edit = False*

Логические выражения предоставляют эффективные средства для контроля состояния программы или определенного условия, играющего важную роль в вашей программе.

## Команды if

Существуют несколько разновидностей команд if, и выбор варианта зависит от количества проверяемых условий.

## Простые команды if

Простейшая форма команды if состоит из одного условия и одного действия:

*if условие:*

*действие*

В первой строке размещается условие, а в блоке с отступом — практически любое действие. Если условие истинно, то Python выполняет код в блоке после команды if, а если ложно, этот код игнорируется.

Input & Output:

*age = 19*

*if age >= 18:*

*print("You are old enough to vote!")*

>>>You are old enough to vote!

Python проверяет, что значение переменной age больше или равно 18.

В таком случае выполняется команда print в строке с отступом.

Отступы в командах if играют ту же роль, что и в циклах for. Если условие истинно, то все строки с отступом после команды if выполняются, а если ложно — весь блок с отступом игнорируется.

Блок команды if может содержать сколько угодно строк. Добавим еще одну строку для вывода дополнительного сообщения в том случае, если возраст достаточен для голосования:

Input & Output:

*age = 19*

*if age >= 18:*

*print("You are old enough to vote!")*

*print("Have you registered to vote yet?")*

*>>>You are old enough to vote!*
*>>>Have you registered to vote yet?*

Условие выполняется, а обе команды print снабжены отступом, поэтому выводятся оба сообщения.

Если значение age меньше 18, программа ничего не выводит.

## Команды if-else

Часто в программе необходимо выполнить одно действие в том случае, если условие истинно, и другое действие, если оно ложно. С синтаксисом if-else это возможно. Блок if-else в целом похож на команду if, но секция else определяет действие или набор действий, выполняемых при неудачной проверке.

В следующем примере выводится то же сообщение, которое выводилось ранее, если возраст достаточен для голосования, но на этот раз при любом другом возрасте выводится другое сообщение:

Input & Output:

```
age = 17

if age >= 18:

        print("You are old enough to vote!")

        print("Have you registered to vote yet?")

else:

        print("Sorry, you are too young to vote.")

        print("Please register to vote as soon as you turn 18!")
```

```
>>>Sorry, you are too young to vote.
>>>Please register to vote as soon as you turn 18!
```

Если условие истинно, то выполняется первый блок с командами print. Если же условие ложно, выполняется блок else. Так как значение age на этот раз меньше 18, условие оказывается ложным и выполняется код в блоке else.

Этот код работает, потому что существуют обе возможные ситуации: возраст либо достаточен для голосования, либо недостаточен. Структура if-else хорошо подходит для тех ситуаций, в которых Python всегда выполняет только одно из двух возможных действий. В подобных простых цепочках if-else всегда выполняется одно из двух возможных действий.

## Цепочки if-elif-else

Python выполняет только один блок в цепочке if-elif-else. Все условия проверяются по порядку до тех пор, пока одно из них не даст истинный результат. Далее выполняется код, следующий за этим условием, а все остальные проверки Python пропускает.

Во многих реальных ситуациях существует более двух возможных результатов.

Представьте себе парк аттракционов, который взимает разную плату за вход для разных возрастных групп:

Для посетителей младше 4 лет вход бесплатный. ‰

Для посетителей от 4 до 18 лет билет стоит $25. ‰

Для посетителей от 18 лет и старше билет стоит $40.

Следующий код определяет, к какой возрастной категории относится посетитель, и выводит сообщение со стоимостью билета:

Input & Output:

```
age = 12

if age < 4:

        print("Your admission cost is $0.")

elif age < 18:

        print("Your admission cost is $25.")
```

*else:*

*print("Your admission cost is $40.")*

*>>>Your admission cost is $25.*

Условие if в точке проверяет, что возраст посетителя меньше 4 лет. Если условие истинно, то программа выводит соответствующее сообщение и Python пропускает остальные проверки.

Строка elif в точке в действительности является еще одной проверкой if, которая выполняется только в том случае, если предыдущая проверка завершилась неудачей. В этом месте цепочки известно, что возраст посетителя не меньше 4 лет, потому что первое условие было ложным.

Если посетителю меньше 18 лет, программа выводит соответствующее сообщение и Python пропускает блок else.

Если ложны оба условия — if и elif, то Python выполняет код в блоке else.

В данном примере условиедает ложный результат, поэтому его блок не выполняется. Однако второе условие оказывается истинным (12 меньше 18), поэтому код будет выполнен. Вывод состоит из одного сообщения с ценой билета.

При любом значении возраста больше 17 первые два условия ложны. В таких ситуациях блок else будет выполнен и цена билета составит $40.

Вместо того чтобы выводить сообщение с ценой билета в блоках if-elif-else, лучше использовать другое, более компактное решение: присвоить цену в цепочке if-elif-else, а затем добавить одну команду print после выполнения цепочки:

Input & Output:

*age = 12*

*if age < 4:*

*price = 0*

*elif age < 18:*

*price = 25*

*else:*

*price = 40*

*print(f"Your admission cost is ${price}.")*

Строки присваивают значение price в зависимости от значения age, как и в предыдущем примере. После присваивания цены в цепочке if-elif-else отдельная команда print без отступа использует это значение для вывода сообщения с ценой билета.

Этот пример выводит тот же результат, что и предыдущий, но цепочка if-elif-else имеет более четкую специализацию. Вместо того чтобы определять цену и выводить сообщения, она просто определяет цену билета. Кроме повышения эффективности, у этого кода есть дополнительное преимущество: его легче модифицировать. Чтобы изменить текст выходного сообщения, достаточно будет отредактировать всего одну команду print — вместо трех разных команд.

## Серии блоков elif

Код может содержать сколько угодно блоков elif. Например, если парк аттракци- онов введет особую скидку для пожилых посетителей, вы можете добавить в свой код еще одну проверку для определения того, распространяется ли скидка на текущего посетителя. Допустим, посетители возрастом 65 и выше платят половину обычной цены билета, или $40:

Input & Output:

```
age = 12

if age < 4:
        price = 0

elif age < 18:

        price = 25

elif age < 65:

        price = 40

else:

        price = 20

print(f"Your admission cost is ${price}.")
```

Большая часть кода осталась неизменной. Второй блок elif в точке теперь убеждается в том, что посетителю меньше 65 лет, прежде чем назначить ему полную цену билета $40. Обратите внимание: значение, присвоенное в блоке else, должно быть заменено на $20, потому что до этого блока доходят только посетители с возрастом 65 и выше.

## Отсутствие блока else

Python не требует, чтобы цепочка if-elif непременно завершалась блоком else. Иногда блок else удобен; иногда бывает лучше использовать дополнительную секцию elif для обработки конкретного условия:

Input & Output:

```
age = 12

if age < 4:

        price = 0

elif age < 18:

        price = 25

elif age < 65:

        price = 40

elif age >= 65:

        price = 20

print(f"Your admission cost is ${price}.")
```

Блок elif назначает цену $20, если возраст посетителя равен 65 и выше; смысл такого кода более понятен, чем у обобщенного блока else. С таким изменением выполнение каждого блока возможно только при истинности конкретного условия.

Блок else «универсален»: он обрабатывает все условия, не подходящие ни под одну конкретную проверку if или elif, причем в эту категорию иногда могут попасть недействительные или даже вредоносные данные. Если у вас имеется завершающее конкретное условие, лучше используйте завершающий блок elif и опустите блок else. В этом случае вы можете быть уверены в том, что ваш код будет выполняться только в правильных условиях.

## Проверка нескольких условий

Однако иногда бывает важно проверить *все* условия. В таких случаях следует применять серии простых команд if без блоков elif или else. Такое решение уместно, когда истинными могут быть сразу несколько условий и вы хотите отреагировать на все истинные.

Вернемся к примеру с пиццей. Если кто-то закажет пиццу с двумя топпингами, программа должна обработать оба топпинга:

Input & Output:

```
requested_toppings = ['mushrooms', 'extra cheese']

if 'mushrooms' in requested_toppings:

        print("Adding mushrooms.")

if 'pepperoni' in requested_toppings:

        print("Adding pepperoni.")

if 'extra cheese' in requested_toppings:

        print("Adding extra cheese.")

print("\nFinished making your pizza!")

>>>Adding mushrooms.
>>>Adding extra cheese.

>>>Finished making your pizza!
```

Обработка начинается со списка, содержащего заказанные топпинги. Команды if в точке проверяют, включает ли заказ конкретные топпинги — грибы и пепперони, и если включает, выводят подтверждающее сообщение. Проверка в точке реализована простой командой if, а не elif или else, поэтому условие будет проверяться независимо от того, было ли предыдущее условие истинным или ложным. Код проверяет, была ли заказана дополнительная порция сыра, независимо от результата первых двух проверок. Эти три независимых условия проверяются при каждом выполнении программы.

Так как в этом коде проверяются все возможные варианты топпингов, в заказ будут включены два топпинга из трех.

Если бы в программе использовался блок if-elif-else, код работал бы неправильно, потому что он прерывал бы работу после обнаружения первого истинного условия.

Input & Output:

```
requested_toppings = ['mushrooms', 'extra cheese']
```

```
if 'mushrooms' in requested_toppings:
        print("Adding mushrooms.")
elif 'pepperoni' in requested_toppings:
        print("Adding pepperoni.")
elif 'extra cheese' in requested_toppings:
        print("Adding extra cheese.")
print("\nFinished making your pizza!")
```

```
>>>Adding mushrooms.
>>>Finished making your pizza!
```

Первое же проверяемое условие (для 'mushrooms') оказывается истинным. Однако значения 'extra cheese' и 'pepperoni' после этого не проверяются, потому что в цепочках if-elif-else после обнаружения первого истинного условия все остальные условия пропускаются. В результате в пиццу будет включено только первый из заказанных топпингов.

Итак, если вы хотите, чтобы в программе выполнялся только один блок кода, используйте цепочку if-elif-else. Если же выполняться должны несколько блоков, используйте серию независимых команд if.

## Использование команд if со списками

Объединение команд if со списками открывает ряд интересных возможностей. Например, вы можете отслеживать специальные значения, для которых необходима особая обработка по сравнению с другими значения в списке, или эффективно управлять изменяющимися условиями. Также объединение команд if со списками помогает продемонстрировать, что ваш код корректно работает во всех возможных ситуациях.

## Проверка специальных значений

Эта глава началась с простого примера, показывающего, как обрабатывать особые значения (такие, как 'bmw'), которые должны выводиться в другом формате по сравнению с другими значениями в списке. Теперь, когда вы лучше разбираетесь в проверках условий и командах if, давайте повнимательнее рассмотрим процесс поиска и обработки особых значений в списке.

Вернемся к примеру с пиццерией. Программа выводит сообщение каждый раз, когда пицца снабжается топпингом в процессе приготовления. Код этого действия можно записать чрезвычайно эффективно: нужно создать список топпингов, заказанных клиентом, и использовать цикл для перебора всех заказанных:

Input & Output:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']
 for requested_topping in requested_toppings:
        print(f"Adding {requested_topping}.")
 print("\nFinished making your pizza!")
```

```
>>>Adding mushrooms.
>>>Adding green peppers.
```

*>>>Adding extra cheese.*

*>>>Finished making your pizza!*

Вывод достаточно тривиален, поэтому код сводится к простому циклу for.

А если в пиццерии вдруг закончится зеленый перец? Команда if в цикле for может правильно обработать эту ситуацию:

Input & Output:

```
requested_toppings = ['mushrooms', 'green peppers', 'extra cheese']

for requested_topping in requested_toppings:

    if requested_topping == 'green peppers':

        print("Sorry, we are out of green peppers right now.")

    else:
        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")
```

```
>>>Adding mushrooms.
>>>Sorry, we are out of green peppers right now.
>>>Adding extra cheese.

>>>Finished making your pizza!
```

На этот раз программа проверяет каждый заказанный элемент перед добавлением его к пицце. Программа проверяет, заказал ли клиент зеленый перец, и если заказал, выводит сообщение о том, что этого дополнения нет. Блок else гарантирует, что все другие дополнения будут включены в заказ.

Из выходных данных видно, что все заказанные топпинги обрабатываются правильно.

## Проверка наличия элементов в списке

Перед выполнением цикла for будет полезно проверить, есть ли в списке хотя бы один элемент.

Проверим, есть ли элементы в списке заказанных топпингов, перед изготовлением пиццы. Если список пуст, программа предлагает пользователю подтвердить, что он хочет базовую пиццу без топпингов. Если список не пуст, пицца готовится так же, как в предыдущих примерах:

Input & Output:

```
requested_toppings = []

if requested_toppings:

    for requested_topping in requested_toppings:

        print(f"Adding {requested_topping}.")

print("\nFinished making your pizza!")

else:

    print("Are you sure you want a plain pizza?")
```

*>>>Are you sure you want a plain pizza?*

На этот раз мы начинаем с пустого списка заказанных топпингов. Вместо того чтобы сразу переходить к циклу for, программа выполняет проверку. Когда имя списка используется в условии if, Python возвращает True, если список содержит хотя бы один элемент; если список пуст, возвращается значение False. Если requested_toppings проходит проверку условия, выполняется тот же цикл for, который мы использовали в предыдущем примере. Если же условие ложно, то программа выводит сообщение, которое предлагает клиенту подтвердить, действительно ли он хочет получить базовую пиццу без дополнений.

В данном примере список пуст, поэтому выводится сообщение.

Если в списке есть хотя бы один элемент, то в выходные данные включается каждый заказанный топпинг.

## Множественные списки

Что, если клиент захочет положить на пиццу картофель-фри? Списки и команды if позволят вам убедиться в том, что входные данные имеют смысл, прежде чем обрабатывать их.

Давайте проверим наличие нестандартных дополнений перед тем, как готовить пиццу. В следующем примере определяются два списка. Первый список содержит перечень доступных топпингов, а второй — список топпингов, заказанных клиентом. На этот раз каждый элемент из requested_toppings проверяется по списку доступных топпингов перед добавлением в пиццу:

Input & Output:

```
available_toppings = ['mushrooms', 'olives', 'green peppers',
                      'pepperoni', 'pineapple', 'extra cheese']

requested_toppings = ['mushrooms', 'french fries', 'extra cheese']

for requested_topping in requested_toppings:

    if requested_topping in available_toppings:

        print(f"Adding {requested_topping}.")

    else:

        print(f"Sorry, we don't have {requested_topping}.")

print("\nFinished making your pizza!")
```

```
>>>Adding mushrooms.
>>>Sorry, we don't have french fries.
>>>Adding extra cheese.

>>>Finished making your pizza!
```

Определяется список доступных топпингов к пицце. Стоит заметить, что если в пиццерии используется постоянный ассортимент топпингов, этот список можно реализовать в виде кортежа. В точке создается список топпингов, заказанных клиентом. Обратите внимание на необычный заказ 'french fries'. В точке программа перебирает список заказанных топпингов. Внутри цикла программа сначала проверяет, что каждый заказанный топпинг присутствует в списке доступных топпингов. Если топпинг доступен, он добавляется в пиццу. Если заказанный топпинг не входит в список, выполняется блок else . Блок else выводит сообщение о том, что топпинг недоступен.

С этим синтаксисом программа выдает четкий, содержательный вывод.

## Оформление команд if

Во всех примерах этой главы применялись правила стилевого оформления. В PEP 8 приведена только одна рекомендация, касающаяся проверки условий: заключать операторы сравнения (такие, как ==, >=, <= и т. д.) в одиночные пробелы. Например, запись

*if age < 4:*

лучше, чем:

*if age<4:*

Пробелы не влияют на интерпретацию вашего кода Python; они только упрощают чтение кода вами и другими разработчиками.

# Глава 5. Словари

Словари — структуры данных, предназначенные для объединения взаимосвязанной информации.

Операции со словарями позволяют моделировать всевозможные реальные объекты с большей точностью. В словаре может храниться имя, возраст, место жительства, профессия и любые другие атрибуты.

## Простой словарь

В следующем простом словаре хранится информация об одном конкретном пришельце:

Input & Output:

*alien_0 = {'colour': 'green', 'points': 5}*

*print(alien_0['colour'])*

*print(alien_0['points'])*

*>>>green*

*>>>5*

В словаре alien_0 хранятся два атрибута: цвет (color) и количество очков (points). Следующие две команды print читают эту информацию из словаря и выводят ее на экран.

## Работа со словарями

*Словарь* в языке Python представляет собой совокупность пар «ключ-значение». Каждый ключ связывается с некоторым значением, и программа может получить значение, связанное с заданным ключом. Значением может быть число, строка, список и даже другой словарь.

Собственно, *любой* объект, создаваемый в программе Python, может стать значением в словаре.

В Python словарь заключается в фигурные скобки {}, в которых приводится последовательность пар «ключ-значение», как в предыдущем примере:

*alien_0 = {'colour': 'green', 'points': 5}*

*Пара «ключ-значение»* представляет данные, связанные друг с другом. Если вы укажете ключ, то Python вернет значение, связанное с этим ключом. Ключ отделяется от значения двоеточием, а отдельные пары разделяются запятыми. В словаре может храниться любое количество пар «ключ-значение».

Простейший словарь содержит ровно одну пару «ключ-значение», как в следующей измененной версии словаря alien_0:

*alien_0 = {'colour': 'green'}*

В этом словаре хранится ровно один фрагмент информации о пришельце alien_0, а именно его цвет. Строка 'colour' является ключом в словаре; с этим ключом связано значение 'green'.

## Обращение к значениям в словаре

Чтобы получить значение, связанное с ключом, укажите имя словаря, а затем ключ в квадратных скобках:

Input & Output:

*alien_0 = {'colour': 'green'}*

*print(alien_0['colour'])*

*>>>alien_0: green*

Эта конструкция возвращает значение, связанное с ключом 'colour' из словаря.

Количество пар «ключ-значение» в словаре не ограничено. Например, вот как выглядит исходный словарь alien_0 с двумя парами «ключ-значение»:

*alien_0 = {'colour': 'green', 'points': 5}*

Теперь программа может получить значение, связанное с любым из ключей в alien_0: colour или points. Если игрок сбивает корабль пришельца, то для получения количества заработанных им очков может использоваться код следующего вида:

*alien_0 = {'colour': 'green', 'points': 5}*

*new_points = alien_0['points']*

*print(f"You just earned {new_points} points!")*

*>>>You just earned 5 points!*

После того как словарь будет определен, код извлекает значение, связанное с ключом 'points', из словаря. Затем это значение сохраняется в переменной new_points. Строка преобразует целое значение в строку и выводит сообщение с количеством заработанных очков.

Если этот код будет выполняться каждый раз, когда игрок сбивает очередного пришельца, программа будет получать правильное количество очков.

## Добавление новых пар «ключ-значение»

Словари относятся к динамическим структурам данных: в словарь можно в любой момент добавлять новые пары «ключ-значение». Для этого указывается имя словаря, за которым в квадратных скобках следует новый ключ с новым значением.

Добавим в словарь alien_0 еще два атрибута: координаты x и y для вывода изображения пришельца в определенной позиции экрана. Допустим, пришелец должен отображаться у левого края экрана, в 25 пикселах от верхнего края. Так как система экранных координат обычно располагается в левом верхнем углу, для размещения пришельца у левого края координата x должна быть равна 0, а координата y — 25:

Input & Output:

*alien_0 = {'colour': 'green', 'points': 5}*

*print(alien_0)*

*alien_0['x_position'] = 0*

*alien_0['y_position'] = 25*

*print(alien_0)*

*>>>{'colour': 'green', 'points': 5}*
*>>>{'colour': 'green', 'points': 5, 'y_position': 25, 'x_position': 0}*

Программа начинается с определения того же словаря. После этого выводится «снимок» текущего состояния словаря. В точке в словарь добавляется новая пара «ключ-значение»: ключ 'x_position' и значе- ние 0. То же самое делается для ключа 'y_position' в точке . При выводе измененного словаря мы видим две дополнительные пары «ключ-значение».

Окончательная версия словаря содержит четыре пары «ключ-значение». Первые две определяют цвет и количество очков, а другие две — координаты.

## Создание пустого словаря

В некоторых ситуациях бывает удобно начать с пустого словаря, а затем добавлять в него новые элементы. Чтобы начать заполнение пустого словаря, определите словарь с пустой парой фигурных скобок, а затем добавляйте новые пары «ключ-значение» (каждая пара в отдельной строке). Например, вот как строится словарь alien_0:

Input & Output:

*alien_0 = {}*

*alien_0['colour'] = 'green'*

*alien_0['points'] = 5*

*print(alien_0)*

*>>>{'colour': 'green', 'points': 5}*

Программа определяет пустой словарь alien_0, после чего добавляет в него значения для цвета и количества очков. В результате создается словарь, который использовался в предыдущих примерах.

Обычно пустые словари используются при хранении данных, введенных пользователем, или при написании кода, автоматически генерирующего большое количество пар «ключ-значение».

## Изменение значений в словаре

Чтобы изменить значение в словаре, укажите имя словаря с ключом в квадратных скобках, а затем новое значение, которое должно быть связано с этим ключом. Допустим, в процессе игры цвет пришельца меняется с зеленого на желтый:

Input & Output:

*alien_0 = {'colour': 'green'}*
*print(f"The alien is {alien_0['colour']}.")*

*alien_0['colour'] = 'yellow'*
*print(f"The alien is now {alien_0['colour']}.")*

*>>>The alien is green.*
*>>>The alien is now yellow.*

Сначала определяется словарь alien_0, который содержит только цвет пришельца; затем значение, связанное с ключом 'colour', меняется на 'yellow'. Из выходных данных видно, что цвет пришельца действительно сменился с зеленого на желтый.

Более интересный пример: отслеживание позиции пришельца, который может двигаться с разной скоростью. Мы сохраним значение, представляющее текущую скорость пришельца, и используем его для определения величины горизонтального смещения:

Input & Output:

```
alien_0 = {'x_position': 0, 'y_position': 25, 'speed': 'medium'}

print(f"Original position: {alien_0['x_position']}")

# Пришелец перемещается вправо.
# Вычисляем величину смещения на основании текущей скорости.

if alien_0['speed'] == 'slow':

        x_increment = 1

elif alien_0['speed'] == 'medium':

        x_increment = 2

else:
        # Пришелец двигается быстро.
        x_increment = 3

# Новая позиция равна сумме старой позиции и приращения.

alien_0['x_position'] = alien_0['x_position'] + x_increment

print(f"New position: {alien_0['x_position']}")

>>>Original x-position: 0
>>>New x-position: 2
```

Сначала определяется словарь с исходной позицией (координаты *x* и *y*) и скоростью 'medium'. Значения цвета и количества очков для простоты опущены, но с ними этот пример работал бы точно так же. Аналогично выводится исходное значение x_position.

В точке цепочка if-elif-else определяет, на какое расстояние пришелец должен переместиться вправо; полученное значение сохраняется в переменной x_increment. Если пришелец двигается медленно ('slow'), то он перемещается на одну единицу вправо; при средней скорости ('medium') он перемещается на две единицы вправо; наконец, при высокой скорости ('fast') он перемещается на три единицы вправо. Вычисленное смещение прибавляется к значению x_position в , а результат сохраняется в словаре с ключом x_position.

Для пришельца со средней скоростью позиция смещается на две единицы.

Получается, что изменение одного значения в словаре меняет все поведение пришельца.

Например, чтобы превратить пришельца со средней скоростью в быстрого, добавьте следующую строку:

Input & Output:

```
alien_0['speed'] = fast
```

При следующем выполнении кода блок if-elif-else присвоит x_increment большее значение.

## Удаление пар «ключ-значение»

Когда информация, хранящаяся в словаре, перестает быть нужной, пару «ключ- значение» можно полностью удалить при помощи команды del. При вызове достаточно передать имя словаря и удаляемый ключ.

Например, в следующем примере из словаря alien_0 удаляется ключ 'points' вместе со значением:

```
alien_0 = {'colour': 'green', 'points': 5}

print(alien_0)

del alien_0['points']

print(alien_0)

>>>{'colour': 'green', 'points': 5}
>>>{'colour': 'green'}
```

Строка приказывает Python удалить ключ 'points' из словаря alien_0, а также удалить значение, связанное с этим ключом. Из вывода видно, что ключ 'points' и его значение 5 исчезли из словаря, но остальные данные остались без изменений.

## Словарь с однотипными объектами

В предыдущем примере в словаре сохранялась разнообразная информация об одном объекте. Словарь также может использоваться для хранения одного вида информации о многих объектах. Вы хотите провести опрос среди коллег и узнать их любимый язык программирования. Результаты простого опроса можно сохранить в словаре:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }
```

Пары в словаре в этой записи разбиты по строкам. Ключами являются имена участников опроса, а значениями — выбранные ими языки. Если вы знаете, что для определения словаря потребуется более одной строки, нажмите клавишу Enter после ввода открывающей фигурной скобки. Снабдите следующую строку отступом на один уровень (четыре пробела) и запишите первую пару «ключ-значение», поставив за ней запятую. После этого при нажатии Enter ваш текстовый редактор будет автоматически снабжать все последующие пары таким же отступом, как у первой.

Завершив определение словаря, добавьте закрывающую фигурную скобку в новой строке после последней пары «ключ-значение» и снабдите ее отступом на один уровень, чтобы она была выровнена по ключам. За последней парой также рекомендуется поставить запятую, чтобы при необходимости вы смогли легко добавить новую пару «ключ-значение» в следующей строке.

Для заданного имени участника опроса этот словарь позволяет легко определить его любимый язык:

Input & Output:

```
favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
         'phil': 'python',
         }

language = favourite_languages['sarah'].title()

print(f"Sarah's favourite language is {language}.")
```

Чтобы узнать, какой язык выбран пользователем с именем Sarah, мы запрашиваем следующее значение:

```
favourite_languages['sarah']
```

Этот синтаксис используется для получения соответствующего языка программирования из словаря и присваивания его переменной language. Создание новой переменной существенно упрощает вызов print(). В выходных данных показыва- ется значение, связанное с ключом:

```
>>>Sarah's favourite language is C.
```

Тот же синтаксис может использоваться с любым участником опроса, содержащимся в словаре.

## Обращение к значениям методом get()

Использование синтаксиса с ключом в квадратных скобках для получения интересующего вас значения из словаря имеет один потенциальный недостаток: если запрашиваемый ключ не существует, то вы получите сообщение об ошибке.

Посмотрим, что произойдет при запросе количества очков для пришельца, для которого оно не задано:

Input & Output:

```
alien_0 = {'colour': 'green', 'speed': 'slow'}

print(alien_0['points'])
```

```
>>>Traceback (most recent call last):
>>>File "alien_no_points.py", line 2, in <module>
>>>print(alien_0['points'])
>>>KeyError: 'points'
```

На экране появляется трассировка с сообщением об ошибке KeyError.

Конкретно для словарей можно воспользоваться методом get() для назначения значения по умолчанию, которое будет возвращено при отсутствии заданного ключа в словаре.

В первом аргументе метода get() передается ключ. Во втором необязательном аргументе можно передать значение, которое должно возвращаться при отсутствии ключа:

Input & Output:

```
alien_0 = {'colour': 'green', 'speed': 'slow'}

point_value = alien_0.get('points', 'No point value assigned.')
```

*print(point_value)*

*>>>No point value assigned.*

Если ключ 'points' существует в словаре, вы получите соответствующее значение; если нет — будет получено значение по умолчанию. В данном случае ключ 'points' не существует, поэтому вместо ошибки выводится понятное сообщение.

Если есть вероятность того, что запрашиваемый ключ не существует, возможно, стоит использовать метод get() вместо синтаксиса с квадратными скобками.

Если второй аргумент при вызове get() опущен, а ключ не существует, то Python вернет специальное значение None — признак того, что значение не существует. Это не ошибка, а специальное значение, указывающее на отсутствие значения.

## Перебор словаря

Словарь Python может содержать как несколько пар «ключ-значение», так и миллионы таких пар. Поскольку словарь может содержать большие объемы данных, Python предоставляет средства для перебора элементов словаря. Информация может храниться в словарях по-разному, поэтому предусмотрены разные способы перебора. Программа может перебрать все пары «ключ-значение» в словаре, только ключи или только значения.

## Перебор всех пар «ключ-значение»

Прежде чем рассматривать разные способы перебора, рассмотрим новый словарь, предназначенный для хранения информации о пользователе веб-сайта. В следующем словаре хранится имя пользователя, его имя и фамилия:

```
user_0 = {
        'username': 'efermi',
        'first': 'enrico',
        'last': 'fermi',
        }
```

Но что, если вы хотите просмотреть *все* данные из словаря этого пользователя? Для этого можно воспользоваться перебором в цикле for:

Input & Output:

```
 user_0 = {
        'username': 'efermi',
        'first': 'enrico',
        'last': 'fermi',
        }

for key, value in user_0.items():

        print(f"\nKey: {key}")

        print(f"Value: {value}")
```

```
>>>Key: last
>>>Value: fermi

>>>Key: first
>>>Value: enrico

>>>Key: username
>>>Value: efermi
```

Как мы видим, чтобы написать цикл for для словаря, необходимо создать имена для двух переменных, в которых будет храниться ключ и значение из каждой пары «ключ-значение». Этим двум переменным можно присвоить любые имена — с короткими одно буквенными именами код будет работать точно так же:

*for k, v in user_0.items()*

Вторая половина команды for в точке включает имя словаря, за которым следует вызов метода items(), возвращающий список пар «ключ-значение». Цикл for сохраняет компоненты пары в двух указанных переменных. В предыдущем примере мы используем переменные для вывода каждого ключа, за которым следует связанное значение . "\n" в первой команде print гарантирует, что перед каждой парой «ключ-значение» в выводе будет вставлена пустая строка.

Перебрав словарь favourite_languages, вы получите имя каждого человека и его любимый язык программиро- вания. Так как ключ всегда содержит имя, а значение — язык программирования, в цикле вместо имен key и value используются переменные name и language. С таким выбором имен читателю кода будет проще следить за тем, что происходит в цикле:

Input & Output:

```
favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

for name, language in favourite_languages.items():

        print(f"{name.title()}'s favourite language is {language.title()}.")
```

```
>>>Jen's favourite language is Python.
>>>Sarah's favourite language is C.
>>>Edward's favourite language is Ruby.
>>>Phil's favourite language is Python.
```

Код приказывает Python перебрать все пары «ключ-значение» в словаре. В процессе перебора пар ключ сохраняется в переменной name, а значение — в переменной language. С этими содержательными именами намного проще понять, что делает команда print.
Всего в нескольких строках кода выводится вся информация из опроса.

## Перебор всех ключей в словаре

Метод keys() удобен в тех случаях, когда вы не собираетесь работать со всеми значениями в словаре. Переберем словарь favourite_languages и выведем имена всех людей, участвовавших в опросе:

Input & Output:

```
favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

for name in favourite_languages.keys():

        print(name.title())
```

*>>>Jen*
*>>>Sarah*
*>>>Edward*
*>>>Phil*

Строка приказывает Python извлечь из словаря favourite_languages все ключи и последовательно сохранять их в переменной name. В выходных данных представлены имена всех людей, участвовавших в опросе.

На самом деле перебор ключей используется по умолчанию при переборе словаря, так что этот код будет работать точно так же, как если бы вы написали

*for name in favourite_languages:*

вместо...

*for name in favourite_languages.keys():*

Используйте явный вызов метода keys(), если вы считаете, что он упростит чтение вашего кода, или опустите его при желании.

Чтобы обратиться в цикле к значению, связанному с интересующим вас ключом, используйте текущий ключ. Для примера выведем для пары друзей сообщение о выбранном ими языке. Мы переберем имена в словаре, как это делалось ранее, но когда имя совпадает с именем одного из друзей, программа будет выводить специальное сообщение об их любимом языке:

#favourite_languages = {
#...
#}

Input & Output:

*friends = ['phil', 'sarah']*

*for name in favourite_languages.keys():*

*print(name.title())*

*if name in friends:*

*language = favourite_languages[name].title()*

*print(f"\t{name.title()}, I see you love {language}!")*

*>>>Hi Jen.*
*>>>Hi Sarah.*

*>>>Sarah, I see you love C!*
*>>>Hi Edward.*

*>>>Hi Phil.*
*>>>Phil, I see you love Python!*

Строится список друзей, для которых должно выводиться сообщение. В цикле выводится имя очередного участника опроса, а затем программа проверяет, входит ли текущее имя в список friends. Если имя входит в список, выводится специальное приветствие с упоминанием выбранного языка.

Выводятся все имена, но для наших друзей выводится специальное сообщение.

Метод keys() также может использоваться для проверки того, участвовал ли конкретный человек в опросе:

Input & Output:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

if 'erin' not in favorite_languages.keys():

        print("Erin, please take our poll!")
```

*>>>Erin, please take our poll!*

Метод keys() не ограничивается перебором: он возвращает список всех ключей, и строка просто проверяет, входит ли ключ 'erin' в список. Так как ключ в списке отсутствует, программа выводит сообщение.

## Перебор ключей словаря в определенном порядке

Один из способов получения элементов в определенном порядке основан на сор- тировке ключей, возвращаемых циклом for. Для получения упорядоченной копии ключей можно воспользоваться функцией sorted():

Input & Output:

```
favorite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

for name in sorted(favorite_languages.keys()):

        print(f"{name.title()}, thank you for taking the poll.")
```

*>>>Edward, thank you for taking the poll.*
*>>>Jen, thank you for taking the poll.*
*>>>Phil, thank you for taking the poll.*
*>>>Sarah, thank you for taking the poll.*

Эта команда for не отличается от других команд for, если не считать того, что метод dictionary.keys() заключен в вызов функции sorted(). Эта конструкция приказывает Python выдать список всех ключей в словаре и отсортировать его перед тем, как перебирать элементы. В выводе перечислены все пользователи, участвовавшие в опросе, а их имена упорядочены по алфавиту.

## Перебор всех значений в словаре

Если вас прежде всего интересуют значения, содержащиеся в словаре, используйте метод values() для получения списка значений без ключей. Допустим, вы хотите просто получить список всех языков, выбранных в опросе, и вас не интересуют имена людей, выбравших каждый язык:

Input & Output:

```
favourite_languages = {
        'jen': 'python',
        'sarah': 'c',
        'edward': 'ruby',
        'phil': 'python',
        }

print("The following languages have been mentioned:")

for language in favourite_languages.values():

        print(language.title())
```

```
>>>The following languages have been mentioned:
>>>Python
>>>C
>>>Python
>>>Ruby
```

Команда for читает каждое значение из словаря и сохраняет его в переменной language. При выводе этих значений будет получен список всех выбранных языков.

Значения извлекаются из словаря без проверки на возможные повторения. Чтобы получить список выбранных языков без повторений, можно воспользоваться *множеством* (set). Множество в целом похоже на список, но все его элементы должны быть уникальными:

```
#favourite_languages = {
#    ...
#        }
```

Input & Output:

```
print("The following languages have been mentioned:")

for language in set(favourite_languages.values()):

        print(language.title())
```

```
>>>The following languages have been mentioned:
>>>Python
>>>C
>>>Ruby
```

Когда список, содержащий дубликаты, заключается в вызов set(), Python находит уникальные элементы списка и строит множество из этих элементов. В точке set() используется для извлечения уникальных языков из favourite_languages.values().

В результате создается не содержащий дубликатов список языков программирования, упомянутых участниками опроса.

Множество можно построить прямо в фигурных скобках с разделением элементов запятыми:

Input & Output:

```
languages = {'python', 'ruby', 'python', 'c'}

languages
```

*>>>{'ruby', 'python', 'c'}*

Словари легко перепутать с множествами, потому что обе структуры заключаются в фигурные скобки. Когда вы видите фигурные скобки без пар «ключ-значение», скорее всего, перед вами множество . В отличие от списков и словарей, элементы множеств не хранятся в каком-либо определенном порядке.

## Вложение

Иногда бывает нужно сохранить множество словарей в списке или сохранить список как значение элемента словаря. Создание сложных структур такого рода называется *вложением*. Вы можете вложить множество словарей в список, список элементов в словарь или даже словарь внутрь другого словаря. Как наглядно показывают следующие примеры, вложение — чрезвычайно мощный механизм.

## Список словарей

Словарь alien_0 содержит разнообразную информацию об одном пришельце, но в нем нет места для хранения информации о втором пришельце, не говоря уже о целом экране, забитом пришельцами. Как смоделировать флот вторжения? Например, можно создать список пришельцев, в котором каждый элемент представляет собой словарь с информацией о пришельце. Например, следующий код строит список из трех пришельцев:

Input & Output:

```
alien_0 = {'colour': 'green', 'points': 5}

alien_1 = {'colour': 'yellow', 'points': 10}

alien_2 = {'colour': 'red', 'points': 15}

aliens = [alien_0, alien_1, alien_2]

for alien in aliens:

        print(alien)
```

```
>>>{'colour': 'green', 'points': 5}
>>>{'colour': 'yellow', 'points': 10}
>>>{'colour': 'red', 'points': 15}
```

Сначала создаются три словаря, каждый из которых представляет отдельного пришельца. В точке каждый словарь заносится в список с именем aliens. Наконец, программа перебирает список и выводит каждого пришельца.

В реалистичном примере будут использоваться более трех пришельцев, которые будут генерироваться автоматически. В следующем примере функция range() создает флот из 30 пришельцев:

Input & Output:

```
#Создание пустого списка для хранения пришельцев.
aliens = []

#Создание 30 зеленых пришельцев.
for alien_number in range(30):
        new_alien = {'colour': 'green', 'points': 5, 'speed': 'slow'}
        aliens.append(new_alien)
```

```
#Вывод первых 5 пришельцев:
for alien in aliens[:5]:
        print(alien)
print("...")

#Вывод количества созданных пришельцев.
print(f"Total number of aliens: {len(aliens)}")
```

```
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>...

>>>Total number of aliens: 30
```

В начале примера список для хранения всех пришельцев, которые будут созданы, пуст. Функция range() возвращает множество чисел, которое просто сообщает Python, сколько раз должен повторяться цикл. При каждом выполнении цикла создается новый пришелец, который затем добавляется в список aliens . Сегмент используется для вывода первых пяти пришельцев, а в точке выводится длина списка (для демонстрации того, что программа действительно сгенерировала весь флот из 30 пришельцев).

Все пришельцы обладают одинаковыми характеристиками, но Python рассматривает каждого пришельца как отдельный объект, что позволяет изменять атрибуты каждого владельца по отдельности.

Как работать с таким множеством? Представьте, что в этой игре некоторые пришельцы изменяют цвет и начинают двигаться быстрее. Когда приходит время смены цветов, мы можем воспользоваться циклом for и командой if для изменения цвета. Например, чтобы превратить первых трех пришельцев в желтых, двигающихся со средней скоростью и приносящих игроку по 10 очков, можно действовать так:

Input & Output:

```
#Создание пустого списка для хранения пришельцев.
aliens = []

#Создание 30 зеленых пришельцев.
for alien_number in range (0,30):

        new_alien = {'colour': 'green', 'points': 5, 'speed': 'slow'}

        aliens.append(new_alien)

for alien in aliens[0:3]:

        if alien['colour'] == 'green':
                alien['colour'] = 'yellow'
                alien['speed'] = 'medium'
                alien['points'] = 10

#Вывод первых 5 пришельцев:
for alien in aliens[0:5]:

        print(alien)

print("...")
```

```
>>>{'speed': 'medium', 'colour': 'yellow', 'points': 10}
>>>{'speed': 'medium', 'colour': 'yellow', 'points': 10}
>>>{'speed': 'medium', 'colour': 'yellow', 'points': 10}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}
>>>{'speed': 'slow', 'colour': 'green', 'points': 5}

>>>...
```

Чтобы изменить первых трех пришельцев, мы перебираем элементы сегмента, включающего только первых трех пришельцев. В данный момент все пришельцы зеленые ('green'), но так будет не всегда, поэтому мы пишем команду if, которая гарантирует, что изменяться будут только зеленые пришельцы. Если пришелец зеленый, то его цвет меняется на желтый ('yellow'), скорость — на среднюю ('medium'), а награда увеличивается до 10 очков.

Цикл можно расширить, добавив блок elif для превращения желтых пришельцев в красных — быстрых и приносящих игроку по 15 очков. Мы не станем приводить весь код, а цикл выглядит так:

Input & Output:

```
for alien in aliens[0:3]:
        if alien['colour'] == 'green':
        alien['colour'] = 'yellow'
        alien['speed'] = 'medium'
        alien['points'] = 10

elif alien['colour'] == 'yellow':
    alien['colour'] = 'red'
    alien['speed'] = 'fast'
    alien['points'] = 15
```

Решение с хранением словарей в списке достаточно часто встречается тогда, когда каждый словарь содержит разные атрибуты одного объекта. Например, вы можете создать словарь для каждого пользователя сайта, как это было сделано в програм- ме user .py на с. 114, и сохранить отдельные словари в списке с именем users. Все словари в списке должны иметь одинаковую структуру, чтобы вы могли перебрать список и выполнить с каждым объектом словаря одни и те же операции.

## Список в словаре

Вместо того чтобы помещать словарь в список, иногда бывает удобно поместить список в словарь. Представьте, как бы вы описали в программе заказанную пиццу. Если ограничиться только списком, сохранить удастся разве что список топпингов к пицце. При использовании словаря список топпингов может быть всего лишь одним аспектом описания пиццы.

В следующем примере для каждой пиццы сохраняются два вида информации: основа и список топпингов. Список топпингов представляет собой значение, связанное с ключом 'toppings'. Чтобы использовать элементы в списке, нужно указать имя словаря и ключ 'toppings', как и для любого другого значения в словаре. Вместо одного значения будет получен список топпингов:

Input & Output:

```
#Сохранение информации о заказанной пицце.

pizza = {
        'crust': 'thick',
        'toppings': ['mushrooms', 'extra cheese'],
        }
```

```
# Описание заказа.

print(f"You ordered a {pizza['crust']}-crust pizza "

        "with the following toppings:")

for topping in pizza['toppings']:

        print("\t" + topping)


>>>You ordered a thick-crust pizza with the following toppings:
>>>    mushrooms
>>>    extra cheese
```

Работа начинается со словаря с информацией о заказанной пицце. С ключом в словаре 'crust' связано строковое значение 'thick'. С другим ключом 'toppings' связано значение-список, в котором хранятся все заказанные топпинги. Выводится описание заказа перед созданием пиццы. Если вам нужно разбить длинную строку в вызове print(), выберите точку для разбиения выводимой строки и закончите строку кавычкой. Снабдите следующую строку отступом, добавьте открывающую кавычку и продолжите строку. Python автоматически объединяет все строки, обнаруженные в круглых скобках. Для вывода дополнений пишется цикл for. Чтобы вывести список топпингов, мы используем ключ 'toppings', а Python берет список топпингов из словаря.

Последующее сообщение описывает пиццу, которую мы собираемся создать.

Вложение списка в словарь может применяться каждый раз, когда с одним ключом словаря должно быть связано более одного значения. Если бы в предыдущем примере с языками программирования ответы сохранялись в списке, один участник опроса мог бы выбрать сразу несколько любимых языков. При переборе словаря значение, связанное с каждым человеком, представляло бы собой список языков (вместо одного языка). В цикле for словаря создается другой цикл для перебора списка языков, связанных с каждым участником:

Input & Output:

```
favourite_languages = {
                'jen': ['python', 'ruby'], 'sarah': ['c'],
                'edward': ['ruby', 'go'], 'phil': ['python', 'haskell'],
                }
for name, languages in favourite_languages.items():

        print(f"\n{name.title()}'s favourite languages are:")

for language in languages:

        print(f"\t{language.title()}")

>>>Jen's favourite languages are:
>>>    Python
>>>    Ruby

>>>Sarah's favourite languages are:
>>>    C

>>>Phil's favourite languages are:
>>>    Python
>>>    Haskell
```

*>>>Edward's favourite languages are:*

*>>>  Ruby*

*>>>  Go*

Вы видите, что значение, связанное с каждым именем, теперь представляет собой список. У некоторых участников один любимый язык программирования, у других таких языков несколько. При переборе словаря переменная с именем languages используется для хранения каждого значения из словаря, потому что мы знаем, что каждое значение будет представлять собой список. В основном цикле по элементам словаря другой цикл перебирает элементы списка любимых языков каждого участника. Теперь каждый участник опроса может указать сколько угодно любимых языков программирования.

## Словарь в словаре

Словарь также можно вложить в другой словарь.
Например, если на сайте есть несколько пользователей с уникальными именами, вы можете использовать имена пользователей как ключи в словаре. Информация о каждом пользователе при этом хранится в словаре, который используется как значение, связанное с именем.
В следующем примере о каждом пользователе хранится три вида информации: имя, фамилия и место жительства. Чтобы получить доступ к этой информации, переберите имена пользователей и словарь с информацией, связанной с каждым именем:

Input & Output:

```
users = {
        'aeinstein': {
                'first': 'albert',
                'last': 'einstein',
                'location': 'princeton',
        },


        'mcurie': {
                'first': 'marie',
                'last': 'curie',
                'location': 'paris',
        },

}

for username, user_info in users.items():

        print(f"\nUsername: {username}")

        full_name = f"{user_info['first']} {user_info['last']}"

        location = user_info['location']

        print(f"\tFull name: {full_name.title()}")

        print(f"\tLocation: {location.title()}")
```

```
>>>Username: aeinstein
>>>  Full name: Albert Einstein
>>>  Location: Princeton
```

*>>>Username: mcurie*
*>>>  Full name: Marie Curie*
*>>>  Location: Paris*

В программе определяется словарь с именем users, содержащий два ключа: для пользователей 'aeinstein' и 'mcurie'. Значение, связанное с каждым ключом, представляет собой словарь с именем, фамилией и местом жительства пользователя. В процессе перебора словаря users в точке Python сохраняет каждый ключ в переменной username, а словарь, связанный с каждым именем пользователя, сохраняется в переменной user_info. Внутри основного цикла в словаре выводится имя пользователя.

В точке начинается работа с внутренним словарем. Переменная user_info, содержащая словарь с информацией о пользователе, содержит три ключа: 'first', 'last' и 'location'. Каждый ключ используется для построения аккуратно отформатированных данных с полным именем и местом жительства пользователя, с последующим выводом сводки известной информации о пользователе.

Обратите внимание на идентичность структур словарей всех пользователей. Хотя Python этого и не требует, наличие единой структуры упрощает работу с вложенными словарями. Если словари разных пользователей будут содержать разные ключи, то код в цикле for заметно усложнится.

# Глава 6. Ввод данных и циклы while

## Как работает функция input()

Функция input() приостанавливает выполнение программы и ожидает, пока пользователь введет некоторый текст. Получив ввод, Python сохраняет его в переменной, чтобы вам было удобнее работать с ним.

Input & Output:

*message = input("Tell me something, and I will repeat it back to you: ")*
*print(message)*

*>>>Tell me something, and I will repeat it back to you:* ***Hello everyone!***
***>>>****Hello everyone!*

Функция input() получает один аргумент: текст подсказки, который выводится на экран, чтобы пользователь понимал, что от него требуется. В данном примере при выполнении первой строки пользователь видит подсказку с предложением ввести любой текст. Программа ожидает, пока пользователь введет ответ, и продолжает работу после нажатия Enter. Ответ сохраняется в переменной message, после чего вызов print(message) дублирует введенные данные.

## Содержательные подсказки

Каждый раз, когда в вашей программе используется функция input(), вы должны включать четкую, понятную подсказку, которая точно сообщит пользователю, какую информацию вы от него хотите получить. Подойдет любое предложение, которое объяснит пользователю, что нужно вводить. Пример:

Input & Output:

*name = input("Please enter your name: ")*
*print(f"\nHello, {name}!")*

*>>>Please enter your name:* ***Eric***
*>>>Hello, Eric!*

Добавьте пробел в конце подсказки (после двоеточия в предыдущем примере), чтобы отделить подсказку от данных, вводимых пользователем, и четко показать, где должен вводиться текст.

Иногда подсказка занимает более одной строки. Например, вы можете сообщить пользователю, для чего программа запрашивает данные. Текст подсказки можно сохранить в переменной и передать эту переменную функции input(): вы строите длинное приглашение из нескольких строк, а потом выполняете одну компактную команду input().

Input & Output:

*prompt = "If you tell us who you are, we can personalise the messages you see." prompt += "\nWhat is your first name? "*

*name = input(prompt)*

*print(f"\nHello, {name}!")*

*>>>If you tell us who you are, we can personalise the messages you see.*

*>>>What is your first name?* ***Eric***

*>>>Hello, Eric!*

В этом примере продемонстрирован один из способов построения длинных строк. Первая часть длинного сообщения сохраняется в переменной prompt. Затем оператор += объединяет текст, хранящийся в prompt, с новым фрагментом текста.
Теперь содержимое prompt занимает две строки (вопросительный знак снова отделяется от ввода пробелом для наглядности).

## Использование int() для получения числового ввода

При использовании функции input() Python интерпретирует все данные, введенные пользователем, как строку. В следующем сеансе интерпретатора программа запрашивает у пользователя возраст:

Input & Output:

*age = input("How old are you? ")*

*How old are you? **21***

*age*

*>>>'21'*

Пользователь вводит число 21, но когда мы запрашиваем у Python значение age, выводится '21' — представление введенного числа в строковом формате. Кавычки, в которые заключены данные, указывают на то, что Python интерпретирует ввод как строку. Но попытка использовать данные как число приведет к ошибке:

Input & Output:

*age = input("How old are you? ")*

*How old are you? **21***

*age >= 18*

*>>>Traceback (most recent call last):*
*>>>  File "<stdin>", line 1, in <module>*

*>>>TypeError: unorderable types: str() >= int()*

Когда вы пытаетесь сравнить введенные данные с числом, Python выдает ошибку, потому что не может сравнить строку с числом: строка '21', хранящаяся в age, не сравнивается с числовым значением 18; происходит ошибка .

Проблему можно решить при помощи функции int(), интерпретирующей строку как числовое значение. Функция int() преобразует строковое представление числа в само число:

Input & Output:

*age = input("How old are you? ")*

*How old are you? **21***

*age = int(age)*

*age >= 18*

*>>>True*

В этом примере введенный текст 21 интерпретируется как строка, но затем он преобразуется в числовое представление вызовом int(). Теперь Python может проверить условие: сравнить переменную age (которая теперь содержит числовое значение 21) с 18. Условие «значение age больше или равно 18» выполняется, и результат проверки равен True.

Как использовать функцию int() в реальной программе? Допустим, программа проверяет рост пользователя и определяет, достаточен ли он для катания на аттракционе:

Input & Output:

```
height = input("How tall are you, in inches? ")

height = int(height)

if height >= 48:

        print("\nYou're tall enough to ride!")

else:

        print("\nYou'll be able to ride when you're a little older.")
```

*>>>How tall are you, in inches?* ***71***

*>>>You're tall enough to ride!*

Программа может сравнить height с 48, потому что строка height = int(height) преобразует входное значение в число перед проведением сравнения. Если введенное число больше или равно 36, программа сообщает пользователю, что он прошел проверку.

Если пользователь вводит числовые данные, которые используются в вашей программе для вычислений и сравнений, обязательно преобразуйте введенное значение в его числовой эквивалент.

## Оператор вычисления остатка

При работе с числовыми данными может пригодиться *оператор вычисления остатка* (%), который делит одно число на другое и возвращает остаток:

Input & Output:

```
4 % 3

>>>1

5 % 3

>>>2

6 % 3

>>>0

7 % 3

>>>1
```

Оператор % не сообщает частное от целочисленного деления; он возвращает только остаток.

Когда одно число нацело делится на другое, остаток равен 0, и оператор % возвращает 0.

Например, этот факт может использоваться для проверки четности или нечетности числа:

Input & Output:

```
number = input("Enter a number, and I'll tell you if it's even or odd: ")

number = int(number)

if number % 2 == 0:

        print(f"\nThe number {number} is even.")

else:

        print(f"\nThe number {number} is odd.")
```

```
>>>Enter a number, and I'll tell you if it's even or odd: 42

>>>The number 42 is even.
```

Четные числа всегда делятся на 2. Следовательно, если остаток от деления на 2 равен 0 (number % 2 == 0), число четное, а если нет — нечетное.

## Циклы while

Цикл for получает коллекцию элементов и выполняет блок кода по одному разу для каждого элемента в коллекции. В отличие от него, цикл while продолжает выполняться, пока остается истинным некоторое условие.

## Цикл while в действии

Цикл while может использоваться для перебора числовой последовательности. Например, следующий цикл считает от 1 до 5:

Input & Output:

```
current_number = 1

        while current_number <= 5:

        print(current_number)

        current_number += 1
```

```
>>>1
>>>2
>>>3
>>>4
>>>5
```

В первой строке отсчет начинается с 1, для чего current_number присваивается значение 1. Далее запускается цикл while, который продолжает работать, пока значение current_number остается меньшим или равным 5. Код в цикле выводит значение current_number и увеличивает его на 1 командой current_number += 1. (Оператор += является сокращенной формой записи для current_number = current_number + 1.)

Цикл повторяется, пока условие current_number <= 5 остается истинным. Так как 1 меньше 5, Python выводит 1, а затем увеличивает значение на 1, отчего current_number становится равным 2. Так как 2 меньше 5, Python выводит 2 и снова прибавляет 1, и т. д. Как только значение current_number превысит 5, цикл останавливается, а программа завершается.

## Пользователь решает прервать работу программы

Программа может выполняться, пока пользователь не захочет остановить ее, — для этого большая часть кода заключается в цикл while. В программе определяется *признак завершения*, и программа работает, пока пользователь не введет нужное значение:

Input & Output:

*prompt = "\nTell me something, and I will repeat it back to you:"*

*prompt += "\nEnter 'quit' to end the program. "*

*message = ""*

*while message != 'quit':*

    *message = input(prompt)*

    *print(message)*

*>>>Tell me something, and I will repeat it back to you: Enter 'quit' to end the program.* ***Hello everyone!***

*>>>Hello everyone!*

*>>>Tell me something, and I will repeat it back to you: Enter 'quit' to end the program.* ***Hello again.***

*>>>Hello again.*

*>>>Tell me something, and I will repeat it back to you: Enter 'quit' to end the program.* ***quit***

*>>>quit*

Определяется сообщение, которое объясняет, что у пользователя есть два варианта: ввести сообщение или ввести признак завершения (в данном случае это строка 'quit'). Затем переменной message присваивается значение, введенное пользователем. В программе переменная message инициализируется пустой строкой "", чтобы значение проверялось без ошибок при первом выполнении строки while. Когда программа только запускается и выполнение достигает команды while, значение message необходимо сравнить с 'quit', но пользователь еще не вводил никакие данные. Если у Python нет данных для сравнения, продолжение выполнения становится невозможным. Чтобы решить эту проблему, необходимо предоставить message исходное значение. И хотя это всего лишь пустая строка, для Python такое значение выглядит вполне осмысленно; программа сможет выполнить сравнение, на котором основана работа цикла while. Цикл while выполняется, пока значение message не равно 'quit'.

При первом выполнении цикла message содержит пустую строку, и Python входит в цикл. При выполнении команды message = input(prompt) Python отображает подсказку и ожидает, пока пользователь введет данные. Эти данные сохраняются в message и выводятся командой print; после этого Python снова проверяет условие команды while. Пока пользователь не введёт слово 'quit', приглашение будет выводиться снова и снова, а Python будет ожидать новых данных. При вводе слова 'quit' Python перестает выполнять цикл while, а программа завершается.

Программа работает неплохо, если не считать того, что она выводит слово 'quit', словно оно является обычным сообщением. Простая проверка if решает проблему:

Input & Output:

```
prompt = "\nTell me something, and I will repeat it back to you:"

prompt += "\nEnter 'quit' to end the program. "

message = ""

while message != 'quit':

        message = input(prompt)

        if message != 'quit':

        print(message)
```

*>>>Tell me something, and I will repeat it back to you:*

*>>>Enter 'quit' to end the program.* ***Hello everyone!***

*>>>Hello everyone!*

*>>>Tell me something, and I will repeat it back to you:*

*>>>Enter 'quit' to end the program.* ***Hello again.***

*>>>Hello again.*

*>>>Tell me something, and I will repeat it back to you:*

*>>>Enter 'quit' to end the program.* ***quit***

Теперь программа проводит проверку перед выводом сообщения и выводит сообщение только в том случае, если оно не совпадает с признаком завершения.

## Флаги

В предыдущем примере программа выполняла некоторые операции, пока заданное условие оставалось истинным. А если вы пишете более сложную программу, выполнение которой может прерываться по нескольким разным условиям?

Если программа должна выполняться только при истинности нескольких условий, определите одну переменную-*флаг*. Эта переменная сообщает, должна ли программа выполняться далее. Программу можно написать так, чтобы она продолжала выполнение, если флаг находится в состоянии True, и завершалась, если любое из нескольких событий перевело флаг в состояние False. В результате в команде while достаточно проверить всего одно условие: находится ли флаг в состоянии True. Все остальные проверки (которые должны определить, произошло ли событие, переводящее флаг в состояние False) удобно организуются в остальном коде.

Добавим флаг в программу из предыдущего раздела. Этот флаг, который мы назовем active (хотя переменная может называться как угодно), управляет тем, должно ли продолжаться выполнение программы:

Input & Output:

```
prompt = "\nTell me something, and I will repeat it back to you:"

prompt += "\nEnter 'quit' to end the program. "

active = True
```

```
while active:

        message = input(prompt)

        if message == 'quit':

        active = False

        else: print(message)
```

В точке переменной active присваивается True, чтобы программа начинала работу в активном состоянии. Это присваивание упрощает команду while, потому что в самой команде while никакие сравнения не выполняются; вся логика реализуется в других частях программы. Пока переменная active остается равной True, цикл выполняется.

В команде if внутри цикла while значение message проверяется после того, как пользователь введет данные. Если пользователь ввел строку 'quit' , флаг active переходит в состояние False, а цикл while останавливается. Если пользователь ввел любой текст, кроме 'quit', то введенные им данные выводятся как сообщение.

Результаты работы этой программы ничем не отличаются от предыдущего приме- ра, в котором условная проверка выполняется прямо в команде while. Но теперь в программе имеется флаг, указывающий, находится ли она в активном состоянии, и вы сможете легко добавить новые проверки (в форме команд elif) для событий, с которыми переменная active может перейти в состояние False.

## Команда break и выход из цикла

Чтобы немедленно прервать цикл while без выполнения оставшегося кода в цикле независимо от состояния условия, используйте команду break. Команда break управляет ходом выполнения программы; она позволит вам управлять тем, какая часть кода выполняется, а какая нет.

Рассмотрим пример — программу, которая спрашивает у пользователя, в каких городах он бывал. Чтобы прервать цикл while, программа выполняет команду break, как только пользователь введет значение 'quit':

Input & Output:

```
prompt = "\nPlease enter the name of a city you have visited:"

prompt += "\n(Enter 'quit' when you are finished.) "

while True:

        city = input(prompt)

        if city == 'quit':

                break

        else:

                print(f"I'd love to go to {city.title()}!")
```

```
>>>Please enter the name of a city you have visited:
>>>(Enter 'quit' when you are finished.) New York

>>>I'd love to go to New York!

>>>Please enter the name of a city you have visited:
```

*>>>(Enter 'quit' when you are finished.)* ***San Francisco***

*>>>I'd love to go to San Francisco!*

*>>>Please enter the name of a city you have visited:*

*>>>(Enter 'quit' when you are finished.)* ***quit***

Цикл, который начинается с while True, будет выполняться бесконечно — если только в нем не будет выполнена команда break. Цикл в программе продолжает запрашивать у пользователя названия городов, пока пользователь не введет строку 'quit'. При вводе строки 'quit' выполняется команда break, по которой Python выходит из цикла.

Команда break может использоваться в любых циклах Python. Например, ее можно включить в цикл for для перебора элементов словаря .

## Команда continue и продолжение цикла

Вместо того чтобы полностью прерывать цикл без выполнения оставшейся части кода, вы можете воспользоваться командой continue для возвращения к началу цикла и проверке условия. Например, возьмем цикл, который считает от 1 до 10, но выводит только нечетные числа в этом диапазоне:

Input & Output:

```
current_number = 0

while current_number < 10:

        current_number += 1

        if current_number % 2 == 0:

        continue

        print(current_number)
```

```
>>>1
>>>3
>>>5
>>>7
>>>9
```

Сначала переменной current_number присваивается 0.
Так как значение меньше 10, Python входит в цикл while.
При входе в цикл счетчик увеличивается на 1, поэтому current_number принимает значение 1.
Затем команда if проверяет остаток от деления current_number на 2.
Если остаток равен 0 (это означает, что current_number делится на 2), команда continue приказывает Python проигнорировать оставшийся код цикла и вернуться к началу.
Если счетчик не делится на 2, то оставшаяся часть цикла выполняется и Python выводит текущее значение счетчика.

## Предотвращение зацикливания

У каждого цикла while должна быть предусмотрена возможность завершения, чтобы цикл не выполнялся бесконечно. Например, следующий цикл считает от 1 до 5:

Input & Output:

```
x= 1
```

```
while x <= 5:
    print(x)
    x += 1
```

Но если случайно пропустить строку x += 1 (см. далее), то цикл будет выполняться бесконечно:

```
# Бесконечный цикл!
x= 1
while x <= 5:
print(x)
>>>1
>>>1
>>>1
>>>1
>>>...
```

Теперь переменной x присваивается начальное значение 1, но это значение никогда не изменяется в программе. В результате проверка условия x <= 5 всегда дает результат True, и цикл while выводит бесконечную серию единиц.

Если ваша программа зациклилась, нажмите Ctrl+C или просто закройте терминальное окно с выводом программы.

Чтобы избежать зацикливания, тщательно проверьте каждый цикл while и убедитесь в том, что цикл прерывается именно тогда, когда предполагается. Если программа должна завершаться при вводе некоторого значения, запустите программу и введите это значение. Если программа не завершилась, проанализируйте обработку значения, которое должно приводить к выходу из цикла. Проверьте, что хотя бы одна часть программы может привести к тому, что условие цикла станет равно False или будет выполнена команда break.

## Использование цикла while со списками и словарями

Чтобы работать с несколькими фрагментами информации, необходимо использовать в циклах while списки и словари.

Цикл for хорошо подходит для перебора списков, но скорее всего, список не должен изменяться в цикле, потому что у Python возникнут проблемы с отслеживанием элементов. Чтобы изменять список в процессе обработки, используйте цикл while. Использование циклов while со списками и словарями позволяет собирать, хранить и упорядочивать большие объемы данных для последующего анализа и обработки.

## Перемещение элементов между списками

Возьмем список недавно зарегистрированных, но еще не проверенных пользова- телей сайта. Как переместить пользователей после проверки в отдельный список проверенных пользователей? Одно из возможных решений: используем цикл while для извлечения пользователей из списка непроверенных, проверяем их и включаем в отдельный список проверенных пользователей. Код может выглядеть так:

Input & Output:

```
#Начинаем с двух списков: пользователей для проверки
#и пустого списка для хранения проверенных пользователей.

unconfirmed_users = ['alice', 'brian', 'candace']

confirmed_users = []

#Проверяем каждого пользователя, пока остаются непроверенные
#пользователи. Каждый пользователь, прошедший проверку,
#перемещается в список проверенных.

while unconfirmed_users:

    current_user = unconfirmed_users.pop()

    print(f"Verifying user: {current_user.title()}")

    confirmed_users.append(current_user)

#Вывод всех проверенных пользователей.

print("\nThe following users have been confirmed:")

for confirmed_user in confirmed_users:

    print(confirmed_user.title())
```

```
>>>Verifying user: Candace
>>>Verifying user: Brian
>>>Verifying user: Alice

>>>The following users have been confirmed:
>>>Candace
>>>Brian
>>>Alice
```

Работа программы начинается с двух списков: непроверенных пользователей и пустого списка для проверенных пользователей. Цикл while в точке выполняется, пока в списке unconfirmed_users остаются элементы. Внутри этого списка функция pop() в точке извлекает очередного непроверенного пользователя из конца списка unconfirmed_users. В данном примере список unconfirmed_users завершается пользователем Candace; это имя первым извлекается из списка, сохраняется в current_user и добавляется в список confirmed_users в точке . Далее следуют пользователи Brian и Alice.

Программа моделирует проверку каждого пользователя выводом сообщения, после чего переносит пользователя в список проверенных. По мере сокращения списка непроверенных пользователей список проверенных пользователей растет. Когда в списке непроверенных пользователей не остается ни одного элемента, цикл останавливается и выводится список проверенных пользователей.

## Удаление всех вхождений конкретного значения из списка

В главе 2 функция remove() использовалась для удаления конкретного значения из списка. Функция remove() работала, потому что интересующее нас значение встречалось в списке только один раз. Но что, если вы захотите удалить все вхождения значения из списка?

Допустим, имеется список pets, в котором значение 'cat' встречается многократно. Чтобы удалить все экземпляры этого значения, можно выполнять цикл while до тех пор, пока в списке не останется ни одного экземпляра 'cat':

Input & Output:

```
pets = ['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']

print(pets)

while 'cat' in pets:

        pets.remove('cat')

print(pets)

>>>['dog', 'cat', 'dog', 'goldfish', 'cat', 'rabbit', 'cat']
>>>['dog', 'dog', 'goldfish', 'rabbit']
```

Программа начинает со списка, содержащего множественные экземпляры 'cat'. После вывода списка Python входит в цикл while, потому что значение 'cat' присутствует в списке хотя бы в одном экземпляре. После входа цикл Python удаляет первое вхождение 'cat', возвращается к строке while, а затем обнаруживает, что экземпляры 'cat' все еще присутствуют в списке, и проходит цикл заново. Вхождения 'cat' удаляются до тех пор, пока не окажется, что в списке значений 'cat' не осталось; в этот момент Python завершает цикл и выводит список заново.

## Заполнение словаря данными, введенными пользователем

При каждом проходе цикла while ваша программа может запрашивать любое необходимое количество данных. Напишем программу, которая при каждом проходе цикла запрашивает имя участника и его ответ. Собранные данные будут сохраняться в словаре, потому что каждый ответ должен быть связан с конкретным пользователем:

Input & Output:

```
responses = {}

#Установка флага продолжения опроса.
polling_active = True

while polling_active:

        #Запрос имени и ответа пользователя.

        name = input("\nWhat is your name? ")

        response = input("Which mountain would you like to climb someday? ")

        #Ответ сохраняется в словаре:

        responses[name] = response

        #Проверка продолжения опроса.

        repeat = input("Would you like to let another person respond? (yes/ no) ")

        if repeat == 'no':

                polling_active = False
```

```
 #Опрос завершен, вывести результаты.

print("\n--- Poll Results ---")

for name, response in responses.items():

        print(f"{name} would like to climb {response}.")
```

*>>>What is your name?* ***Eric***

*>>>Which mountain would you like to climb someday?* ***Denali***

*>>>Would you like to let another person respond? (yes/ no)* ***yes***

*>>>What is your name?* ***Lynn***

*>>>Which mountain would you like to climb someday?* ***Devil's Thumb***

*>>>Would you like to let another person respond? (yes/ no)* ***no***

*>>>--- Poll Results ---*

*>>>Lynn would like to climb Devil's Thumb.*

*>>>Eric would like to climb Denali.*

Сначала программа определяет пустой словарь (responses) и устанавливает флаг (polling_active), показывающий, что опрос продолжается. Пока polling_active содержит True, Python будет выполнять код в цикле while.

В цикле пользователю предлагается ввести имя и название горы, на которую ему хотелось бы подняться . Эта информация сохраняется в словаре responses в строке, после чего программа спрашивает у пользователя, нужно ли продолжать опрос. Если пользователь отвечает положительно, то программа снова входит в цикл while. Если же ответ отрицателен, флаг polling_active переходит в состояние False, цикл while перестает выполняться и завершающий блок кода выводит результаты опроса.

# Глава 7. Функции

*Функции* — именованные блоки кода, предназначенные для решения одной конкретной задачи.

Чтобы выполнить задачу, определенную в виде функции, вы *вызываете* функцию, отвечающую за эту задачу.
Если задача должна многократно выполняться в программе, вам не придется заново вводить весь необходимый код; просто вызовите функцию, предназначенную для решения задачи, и этот вызов прикажет Python выполнить код, содержащийся внутри функции.

## Определение функции

Вот простая функция с именем greet_user (), которая выводит приветствие:

Input & Output:

```
def greet_user():

        """Выводит простое приветствие."""

        print("Hello!")

greet_user()

>>>Hello!
```

В этом примере представлена простейшая структура функции. Строка при помощи ключевого слова def сообщает Python, что вы определяете функцию. В *определении функции* указывается имя функции и если нужно — описание информации, необходимой функции для решения ее задачи. Эта информация заключается в круглые скобки. В данном примере функции присвоено имя greet_user() и она не нуждается в дополнительной информации для решения своей задачи, поэтому круглые скобки пусты. (Впрочем, даже в этом случае они обязательны.) Наконец, определение завершается двоеточием.

Все строки с отступами, следующие за def greet_user():, образуют *тело* функции. Текст представляет собой комментарий — *строку документации* с описанием функции. Строки документации заключаются в утроенные кавычки; Python опознает их по этой последовательности символов во время генерирования документации к функциям в ваших программах.

«Настоящий» код в теле этой функции состоит всего из одной строки print("Hello!") — см. .
Таким образом, функция greet_user() решает всего одну задачу: выполнение команды print("Hello!").

Когда потребуется использовать эту функцию, вызовите ее. *Вызов функции* приказывает Python выполнить содержащийся в ней код. Чтобы вызвать функцию, укажите ее имя, за которым следует вся необходимая информация, заключенная в круглые скобки, как показано в строке. Так как никакая дополнительная информация не нужна, вызов функции эквивалентен простому выполнению команды greet_user(). Как и ожидалось, функция выводит сообщение Hello!.

## Передача информации функции

С небольшими изменениями функция greet_user() сможет не только сказать «Привет!» пользователю, но и поприветствовать его по имени. Для этого следует включить имя username в круглых скобках в определении функции def greet_user(). С добавлением username функция примет любое значение, которое будет заключено в скобки при вызове. Теперь функция ожидает, что при каждом вызове будет передаваться имя пользователя. При вызове greet_user() укажите имя (например, 'jesse') в круглых скобках:

Input & Output:

```
def greet_user(username):
    """Выводит простое приветствие."""
    print(f"Hello, {username.title()}!")

greet_user('jesse')

>>>Hello, Jesse!
```

Команда greet_user('jesse') вызывает функцию greet_user() и передает ей информацию, необходимую для выполнения команды print. Функция получает переданное имя и выводит приветствие для этого имени.

Точно так же команда greet_user('sarah') вызывает функцию greet_user() и передает ей строку 'sarah', в результате чего будет выведено сообщение Hello, Sarah! Функцию greet_user() можно вызвать сколько угодно раз и передать ей любое имя на ваше усмотрение — и вы будете получать ожидаемый результат.

## Аргументы и параметры

Функция greet_user() определена так, что для работы она должна получить значение переменной username. После того как функция будет вызвана и получит необходимую информацию (имя пользователя), она выведет правильное приветствие.

Переменная username в определении greet_user() — *параметр*, то есть условные данные, необходимые функции для выполнения ее работы. Значение 'jesse' в greet_user('jesse') — *аргумент*, то есть конкретная информация, переданная при вызове функции. Вызывая функцию, вы заключаете значение, с которым функция должна работать, в круглые скобки. В данном случае аргумент 'jesse' был передан функции greet_user(), а его значение было сохранено в переменной username.

Иногда в литературе термины «аргумент» и «параметр» используются как синонимы.

## Передача аргументов

Определение функции может иметь несколько параметров, и может оказаться, что при вызове функции должны передаваться несколько аргументов. Существуют несколько способов передачи аргументов функциям. *Позиционные аргументы* перечисляются в порядке, точно соответствующем порядку записи параметров; *именованные аргументы* состоят из имени переменной и значения; наконец, существуют списки и словари значений. Рассмотрим все эти способы.

## Позиционные аргументы

При вызове функции каждому аргументу должен быть поставлен в соответствие параметр в определении функции. Проще всего сделать это на основании порядка перечисления аргументов. Значения, связываемые с аргументами подобным образом, называются *позиционными аргументами*.

Чтобы понять, как работает эта схема, рассмотрим функцию для вывода информации о домашних животных. Функция сообщает тип животного и его имя:

Input & Output:

```
def describe_pet(animal_type, pet_name):
```

```
    """Выводит информацию о животном."""

    print(f"\nI have a {animal_type}.")

    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')

>>>I have a hamster.
>>>My hamster's name is Harry.
```

Из определения видно, что функции должен передаваться тип животного (animal_type) и его имя (pet_name). При вызове describe_pet() необходимо передать тип и имя — именно в таком порядке. В этом примере аргумент 'hamster' сохраняется в параметре animal_type, а аргумент 'harry' сохраняется в параметре pet_name. В теле функции эти два параметра используются для вывода информации.

## Многократные вызовы функций

Функция может вызываться в программе столько раз, сколько потребуется. Для вывода информации о другом животном достаточно одного вызова describe_pet():

Input & Output:

```
def describe_pet(animal_type, pet_name):

    """Выводит информацию о животном."""

    print(f"\nI have a {animal_type}.")

    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('hamster', 'harry')

describe_pet('dog', 'willie')

>>>I have a hamster.
>>>My hamster's name is Harry.

>>>I have a dog.
>>>My dog's name is Willie.
```

Во втором вызове функции describe_pet() передаются аргументы 'dog' и 'willie'. По аналогии с предыдущей парой аргументов Python сопоставляет аргумент 'dog' с параметром animal_type, а аргумент 'willie' — с параметром pet_name.

Как и в предыдущем случае, функция выполняет свою задачу, но на этот раз выводятся другие значения.

Многократный вызов функции — чрезвычайно эффективный механизм. Код вывода информации о домашнем животном пишется один раз в функции. Каждый раз, когда вам понадобится вывести информацию о новом животном, вы вызываете функцию с данными нового животного. Даже если код вывода информации разрастется до 10 строк, вы все равно сможете вывести информацию всего одной командой — для этого достаточно снова вызвать функцию.

Функция может иметь любое количество позиционных аргументов. При вызове функции Python перебирает аргументы, приведенные в вызове, и сопоставляет каждый аргумент с соответствующим параметром из определения функции.

## О важности порядка позиционных аргументов

Если нарушить порядок следования аргументов в вызове при использовании позиционных аргументов, возможны неожиданные результаты:

Input & Output:

```
def describe_pet(animal_type, pet_name):

        """Выводит информацию о животном."""

        print(f"\nI have a {animal_type}.")

        print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet('harry', 'hamster')
```

```
>>>I have a harry.
>>>My harry's name is Hamster.
```

В этом вызове функции сначала передается имя, а потом тип животного. Так как аргумент 'harry' находится в первой позиции, значение сохраняется в параметре animal_type, а аргумент 'hamster' — в pet_name. На этот раз вывод получается бессмысленным.

Если вы получили подобные странные результаты, проверьте, что порядок следования аргументов в вызове функции соответствует порядку параметров в ее определении.

## Именованные аргументы

*Именованный аргумент* представляет собой пару «имя-значение», передаваемую функции. Имя и значение связываются с аргументом напрямую, так что при передаче аргумента путаница с порядком исключается. Именованные аргументы избавляют от хлопот с порядком аргументов при вызове функции, а также проясняют роль каждого значения в вызове функции.

Перепишем программу с использованием именованных аргументов при вызове describe_pet():

Input & Output:

```
def describe_pet(animal_type, pet_name):

        """Выводит информацию о животном."""

        print(f"\nI have a {animal_type}.")

        print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(animal_type='hamster', pet_name='harry')
```

```
>>>describe_pet(animal_type='hamster', pet_name='harry')
>>>describe_pet(pet_name='harry', animal_type='hamster')
```

Функция describe_pet() не изменилась. Однако на этот раз при вызове функции мы явно сообщаем Python, с каким параметром должен быть связан каждый аргумент. При обработке вызова функции Python знает, что аргумент 'hamster' должен быть сохранен в параметре animal_type, а аргумент 'harry' — в параметре pet_name.

Порядок следования именованных аргументов в данном случае неважен, потому что Python знает, где должно храниться каждое значение.

При использовании именованных аргументов будьте внимательны — имена должны точно совпадать с именами параметров из определения функции .

## Значения по умолчанию

Для каждого параметра вашей функции можно определить значение по умолчанию. Если при вызове функции передается аргумент, соответствующий данному параметру, Python использует значение аргумента, а если нет — использует значение по умолчанию. Таким образом, если для параметра определено значение по умолчанию, вы можете опустить соответствующий аргумент, который обычно включается в вызов функции. Значения по умолчанию упрощают вызовы функций и проясняют типичные способы использования функций.

Например, если вы заметили, что большинство вызовов describe_pet() используется для описания собак, задайте animal_type значение по умолчанию 'dog'. Теперь в любом вызове describe_pet() для собаки эту информацию можно опустить:

Input & Output:

```
def describe_pet(pet_name, animal_type='dog'):


    """Выводит информацию о животном."""

    print(f"\nI have a {animal_type}.")

    print(f"My {animal_type}'s name is {pet_name.title()}.")

describe_pet(pet_name='willie')
```

```
>>>I have a dog.
>>>My dog's name is Willie.
```

Мы изменили определение describe_pet() и включили для параметра animal_type значение по умолчанию 'dog'. Если теперь функция будет вызвана без указания animal_type, Python знает, что для этого параметра следует использовать значение 'dog'.

Обратите внимание: в определении функции пришлось изменить порядок параметров. Так как благодаря значению по умолчанию указывать аргумент с типом животного не обязательно, единственным оставшимся аргументом в вызове функции остается имя домашнего животного. Python интерпретирует его как позиционный аргумент, и если функция вызывается только с именем животного, этот аргумент ставится в соответствие с первым параметром в определении функции. Именно по этой причине имя животного должно быть первым параметром.

В простейшем варианте использования этой функции при вызове передается только имя собаки:

```
describe_pet('willie')
```

Вызов функции выводит тот же результат, что и в предыдущем примере. Единственный переданный аргумент 'willie' ставится в соответствие с первым параме- тром в определении, pet_name. Так как для animal_type аргумент не указан, Python использует значение по умолчанию 'dog'.

Для вывода информации о любом другом животном, кроме собаки, используется вызов функции следующего вида:

```
describe_pet(pet_name='harry', animal_type='hamster')
```

Так как аргумент для параметра animal_type задан явно, Python игнорирует значение параметра по умолчанию.

Если вы используете значения по умолчанию, все параметры со значением по умолчанию должны следовать после параметров, у которых значений по умолчанию нет. Это необходимо для того, чтобы Python правильно интерпретировал позиционные аргументы.

## Эквивалентные вызовы функций

Так как позиционные аргументы, именованные аргументы и значения по умолчанию могут использоваться одновременно, часто существуют несколько эквива- лентных способов вызова функций. Возьмем следующий оператор describe_pets() с одним значением по умолчанию:

*def describe_pet(pet_name, animal_type='dog'):*

При таком определении аргумент для параметра pet_name должен задаваться в любом случае, но это значение может передаваться как в позиционном, так и в именованном формате. Если описываемое животное не является собакой, то аргумент animal_type тоже должен быть включен в вызов, и этот аргумент тоже может быть задан как в позиционном, так и в именованном формате.

Все следующие вызовы являются допустимыми для данной функции:

*#Пес по имени Вилли.*

*describe_pet('willie')*

*describe_pet(pet_name='willie')*

*#Хомяк по имени Гарри.*

*describe_pet('harry', 'hamster')*

*describe_pet(pet_name='harry', animal_type='hamster')*

*describe_pet(animal_type='hamster', pet_name='harry')*

Все вызовы функции выдадут такой же результат, как и в предыдущих примерах.

## Предотвращение ошибок в аргументах

Такие ошибки происходят в том случае, если вы передали меньше или больше аргументов, чем необходимо функции для выполнения ее работы. Например, вот что произойдет при попытке вызвать describe_pet() без аргументов:

Input & Output:

*def describe_pet(animal_type, pet_name):*

*"""Выводит информацию о животном."""*

*print(f"\nI have a {animal_type}.")*

*print(f"My {animal_type}'s name is {pet_name.title()}.")*

*describe_pet()*

*>>>Traceback (most recent call last):*
*>>>File "pets.py", line 6, in <module>*
*>>> describe_pet()*
*>>>TypeError: describe_pet() missing 2 required positional arguments: 'animal_type' and 'pet_name'*

Python понимает, что при вызове функции часть информации отсутствует, и мы видим это в данных трассировки.
Сообщается местонахождение проблемы, чтобы вы поняли, что с вызовом функции что-то пошло не так.
Затем приводится вызов функции, приведший к ошибке.
Python сообщает, что при вызове пропущены два аргумента, и сообщает имена этих аргументов.
Если бы функция размещалась в отдельном файле, вероятно, вы смогли бы исправить вызов и вам не пришлось бы открывать этот файл и читать код функции.
Python помогает еще и тем, что он читает код функции и сообщает имена аргументов, которые необходимо передать при вызове.
Это еще одна причина для того, чтобы присваивать переменным и функциям содержательные имена.
В этом случае сообщения об ошибках Python принесут больше пользы как вам, так и любому другому разработчику, который будет использовать ваш код.

Если при вызове будут переданы лишние аргументы, вы получите похожую трассировку, которая поможет привести вызов функции в соответствие с ее определением.

## Возвращаемое значение

Функция не обязана выводить результаты своей работы напрямую. Вместо этого она может обработать данные, а затем вернуть значение или набор сообщений. Значение, возвращаемое функцией, называется *возвращаемым значением*. Команда return передает значение из функции в точку программы, в которой эта функция была вызвана. Возвращаемые значения помогают переместить большую часть рутинной работы в вашей программе в функции, чтобы упростить основной код программы.

## Возвращение простого значения

Рассмотрим функцию, которая получает имя и фамилию и возвращает аккуратно отформатированное полное имя:

Input & Output:

```
def get_formatted_name(first_name, last_name):

        """Возвращает аккуратно отформатированное полное имя."""

        full_name = f"{first_name} {last_name}"

        return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')

print(musician)
```

*Jimi Hendrix*

Определение get_formatted_name() получает в параметрах имя и фамилию. Функция объединяет эти два имени, добавляет между ними пробел и сохраняет результат в full_name . Значение full_name преобразуется в формат с начальной буквой верхнего регистра, а затем возвращается в точку вызова.

Вызывая функцию, которая возвращает значение, необходимо предоставить переменную, в которой должно храниться возвращаемое значение. В данном случае возвращаемое значение сохраняется в переменной musician. Результат содержит аккуратно отформатированное полное имя, построенное из имени и фамилии.

Может показаться, что все эти хлопоты излишни — с таким же успехом можно было использовать команду:

*print("Jimi Hendrix")*

Но если представить, что вы пишете большую программу, в которой многочисленные имена и фамилии должны храниться по отдельности, такие функции, как get_formatted_name(), становятся чрезвычайно полезными. Вы храните имена отдельно от фамилий, а затем вызываете функцию везде, где потребуется вывести полное имя.

## Необязательные аргументы

Иногда бывает удобно сделать аргумент необязательным, чтобы разработчик, использующий функцию, мог передать дополнительную информацию только в том случае, если он этого захочет. Чтобы сделать аргумент необязательным, можно воспользоваться значением по умолчанию.

Допустим, вы захотели расширить функцию get_formatted_name(), чтобы она также работала и со вторыми именами. Первая попытка могла бы выглядеть так:

Input & Output:

```
def get_formatted_name(first_name, middle_name, last_name):

        """Возвращает аккуратно отформатированное полное имя."""

        full_name = f"{first_name} {middle_name} {last_name}"

        return full_name.title()

musician = get_formatted_name('john', 'lee', 'hooker')

print(musician)

>>>John Lee Hooker
```

Функция работает при получении имени, второго имени и фамилии. Она получает все три части имени, а затем строит из них строку. Функция добавляет пробелы там, где это уместно, и преобразует полное имя в формат с капитализацией.

Однако вторые имена нужны не всегда, а в такой записи функция не будет работать, если при вызове ей передается только имя и фамилия. Чтобы средний аргумент был необязательным, можно присвоить аргументу middle_name пустое значение по умолчанию; этот аргумент игнорируется, если пользователь не передал для него значение. Чтобы функция get_formatted_name() работала без второго имени, следует назначить для параметра middle_name пустую строку значением по умолчанию и переместить его в конец списка параметров:

Input & Output:

```
def get_formatted_name(first_name, last_name, middle_name=''):

        """Возвращает аккуратно отформатированное полное имя."""

        if middle_name:

                full_name = f"{first_name} {middle_name} {last_name}"

        else:
```

```
        full_name = f"{first_name} {last_name}"

    return full_name.title()

musician = get_formatted_name('jimi', 'hendrix')

print(musician)

musician = get_formatted_name('john', 'hooker', 'lee')

print(musician)
```

```
>>>Jimi Hendrix
>>>John Lee Hooker
```

В этом примере имя строится из трех возможных частей. Поскольку имя и фамилия указываются всегда, эти параметры стоят в начале списка в определении функции. Второе имя не обязательно, поэтому оно находится на последнем месте в определении, а его значением по умолчанию является пустая строка.
В теле функции мы сначала проверяем, было ли дано второе имя. Python интерпретирует непустые строки как истинное значение, и если при вызове задан аргумент второго имени, middle_name дает результат True. Если второе имя указано, то из имени, второго имени и фамилии строится полное имя. Затем имя преобразуется с капитализацией символов и возвращается в строку вызова функции, где оно сохраняется в переменной musician и выводится. Если второе имя не указано, то пустая строка не проходит проверку if и выполняет блок else. В этом случае полное имя строится только из имени и фамилии и отформатированное имя возвращается в строку вызова, где оно сохраняется в переменной musician и выводится.
Вызов этой функции с именем и фамилией достаточно тривиален. Но при использовании второго имени придется проследить за тем, чтобы второе имя было последним из передаваемых аргументов. Это необходимо для правильного связывания позиционных аргументов в строке.

Необязательные значения позволяют функциям работать в максимально широком спектре сценариев использования без усложнения вызовов.

## Возвращение словаря

Функция может вернуть любое значение, которое вам потребуется, в том числе и более сложную структуру данных. Так, следующая функция получает части имени и возвращает словарь, представляющий человека:

Input & Output:

```
def build_person(first_name, last_name):

    """Возвращает словарь с информацией о человеке."""

    person = {'first': first_name, 'last': last_name}

    return person

musician = build_person('jimi', 'hendrix')

print(musician)
```

```
{'first': 'jimi', 'last': 'hendrix'}
```

Функция build_person() получает имя и фамилию и сохраняет полученные значения в словаре. Значение first_name сохраняется с ключом 'first', а значение last_name — с ключом 'last'. Весь словарь с описанием человека возвращается. Возвращаемое значение выводится в точке с двумя исходными фрагментами текстовой информации, теперь хранящимися в словаре.

Функция получает простую текстовую информацию и помещает ее в более удоб- ную структуру данных, которая позволяет работать с информацией (помимо простого вывода). Строки 'jimi' и 'hendrix' теперь помечены как имя и фамилия. Функцию можно легко расширить так, чтобы она принимала дополнительные значения — второе имя, возраст, профессию или любую другую информацию о человеке, которую вы хотите сохранить. Например, следующее изменение позволяет также сохранить возраст человека:

Input & Output:

```
def build_person(first_name, last_name):
        """Возвращает словарь с информацией о человеке."""
        person = {'first': first_name, 'last': last_name}
        if age:
                person['age'] = age
        return person
musician = build_person('jimi', 'hendrix', age=27)
print(musician)
```

В определение функции добавляется новый необязательный параметр age, которому присваивается специальное значение по умолчанию None — оно используется для переменных, которым не присвоено никакое значение. При проверке условий None интерпретируется как False. Если вызов функции включает значение этого параметра, то значение сохраняется в словаре. Функция всегда сохраняет имя, но ее также можно модифицировать, чтобы она сохраняла любую необходимую информацию о человеке.

## Использование функции в цикле while

Функции могут использоваться со всеми структурами Python, уже известными вам. Например, используем функцию get_formatted_name() в цикле while, чтобы поприветствовать пользователей более официально. Первая версия программы, приветствующей пользователей по имени и фамилии, может выглядеть так:

Input & Output:

```
def get_formatted_name(first_name, last_name):
        """Возвращает аккуратно отформатированное полное имя."""
        full_name = f"{first_name} {last_name}"
        return full_name.title()


#Бесконечный цикл!
while True:
   print("\nPlease tell me your name:")
```

```
f_name = input("First name: ")

l_name = input("Last name: ")

formatted_name = get_formatted_name(f_name, l_name)

print(f"\nHello, {formatted_name}!")
```

В этом примере используется простая версия get_formatted_name(), не использующая вторые имена. В цикле while имя и фамилия пользователя запрашиваются по отдельности.

Но у этого цикла while есть один недостаток: в нем не определено условие завершения. Где следует разместить условие завершения при запросе серии данных? Пользователю нужно предоставить возможность выйти из цикла как можно раньше, так что в приглашении должен содержаться способ завершения. Команда break позволяет немедленно прервать цикл при запросе любого из компонентов:

Input & Output:

```
def get_formatted_name(first_name, last_name):

        """Возвращает аккуратно отформатированное полное имя."""

        full_name = f"{first_name} {last_name}"

        return full_name.title()

while True:

        print("\nPlease tell me your name:")

        print("(enter 'q' at any time to quit)")

        f_name = input("First name: ")

        if f_name == 'q':

            break

        l_name = input("Last name: ")

        if l_name == 'q':

            break

        formatted_name = get_formatted_name(f_name, l_name)

        print(f"\nHello, {formatted_name}!")
```

```
>>>Please tell me your name: (enter 'q' at any time to quit)

>>>First name: eric

>>>Please tell me your name: (enter 'q' at any time to quit)

>>>First name: eric

>>>Last name: matthes

>>>Hello, Eric Matthes!
```

*>>>Please tell me your name: (enter 'q' at any time to quit)*

*>>>First name:* ***q***

В программу добавляется сообщение, которое объясняет пользователю, как завершить ввод данных, и при вводе признака завершения в любом из приглашений цикл прерывается. Теперь программа будет приветствовать пользователя до тех пор, пока вместо имени или фамилии не будет введен символ 'q'.

## Передача списка

При передаче списка функция получает прямой доступ ко всему его содержимому. Мы воспользуемся функциями для того, чтобы сделать работу со списком более эффективной.

Допустим, вы хотите вывести приветствие для каждого пользователя из списка. В следующем примере список имен передается функции greet_users(), которая выводит приветствие для каждого пользователя по отдельности:

Input & Output:

```
def greet_users(names):

        """Вывод простого приветствия для каждого пользователя в списке."""

        for name in names:

                msg = f"Hello, {name.title()}!"

                print(msg)

usernames = ['hannah', 'ty', 'margot']

greet_users(usernames)
```

```
>>>Hello, Hannah!
>>>Hello, Ty!
>>>Hello, Margot!
```

В соответствии со своим определением функция greet_users() рассчитывает получить список имен, который сохраняется в параметре names. Функция перебирает полученный список и выводит приветствие для каждого пользователя. В точке мы определяем список пользователей usernames, который затем передается greet_ users() в вызове функции.

Результат выглядит именно так, как ожидалось. Каждый пользователь получает персональное сообщение, и эту функцию можно вызвать для каждого нового набора пользователей.

## Изменение списка в функции

Если вы передаете список функции, код функции сможет изменить список. Все изменения, внесенные в список в теле функции, закрепляются, что позволяет эффективно работать со списком даже при больших объемах данных.

Допустим, компания печатает на 3D-принтере модели, предоставленные пользователем. Проекты хранятся в списке, а после печати перемещаются в отдельный список. В следующем примере приведена реализация, не использующая функции:

Input & Output:

```
#Список моделей, которые необходимо напечатать.
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
```

```
completed_models = []

#Цикл последовательно печатает каждую модель до конца списка.

#После печати каждая модель перемещается в список completed_models.

while unprinted_designs:

    current_design = unprinted_designs.pop()

    print(f"Printing model: {current_design}")

    completed_models.append(current_design)

#Вывод всех готовых моделей.

print("\nThe following models have been printed:")

for completed_model in completed_models:

    print(completed_model)
```

```
>>>Printing model: dodecahedron
>>>Printing model: robot pendant
>>>Printing model: phone case

>>>The following models have been printed:
>>>dodecahedron
>>>robot pendant
>>>phone case
```

В начале программы создается список моделей и пустой список completed_models, в который каждая модель перемещается после печати. Пока в unprinted_designs остаются модели, цикл while имитирует печать каждой модели: модель удаляется из конца списка, сохраняется в current_design, а пользователь получает сообщение о том, что текущая модель была напечатана. Затем модель перемещается в список напечатанных. После завершения цикла выводится список напечатанных моделей.

Мы можем изменить структуру этого кода: для этого следует написать две функции, каждая из которых решает одну конкретную задачу. Большая часть кода останется неизменной; просто программа становится более эффективной. Первая функция занимается печатью, а вторая выводит сводку напечатанных моделей:

Input & Output:

```
def print_models(unprinted_designs, completed_models):

    #Имитирует печать моделей, пока список не станет пустым.
    #Каждая модель после печати перемещается в completed_models.

while unprinted_designs:

    current_design = unprinted_designs.pop()

    print(f"Printing model: {current_design}")

    completed_models.append(current_design)
```

```
def show_completed_models(completed_models):
    """Выводит информацию обо всех напечатанных моделях."""
    print("\nThe following models have been printed:")
    for completed_model in completed_models:
        print(completed_model)

unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

В точке определяется функция print_models() с двумя параметрами: список моделей для печати и список готовых моделей. Функция имитирует печать каждой модели, последовательно извлекая модели из первого списка и перемещая их во второй список. В точке определяется функция show_completed_models() с одним параметром: списком напечатанных моделей. Функция show_completed_models() получает этот список и выводит имена всех напечатанных моделей.

Программа выводит тот же результат, что и версия без функций, но структура кода значительно улучшилась. Код, выполняющий большую часть работы, разнесен по двум разным функциям; это упрощает чтение основной части программы. Теперь любому разработчику будет намного проще просмотреть код программы и понять, что делает программа:

```
unprinted_designs = ['phone case', 'robot pendant', 'dodecahedron']
completed_models = []
print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Программа создает список моделей для печати и пустой список для готовых моделей. Затем, поскольку обе функции уже определены, остается вызвать их и передать правильные аргументы. Мы вызываем print_models() и передаем два необходимых списка; как и ожидалось, print_models() имитирует печать моделей. Затем вызывается функция show_completed_models() и ей передается список готовых моделей, чтобы функция могла вывести информацию о напечатанных моделях. Благодаря содержательным именам функций другой разработчик сможет прочитать этот код и понять его даже без комментариев.

Вдобавок эта программа создает меньше проблем с расширением и сопровождением, чем версия без функций. Если позднее потребуется напечатать новую партию моделей, достаточно снова вызвать print_models(). Если окажется, что код печати необходимо модифицировать, изменения достаточно внести в одном месте, и они автоматически распространятся на все вызовы функции. Такой подход намного эффективнее независимой правки кода в нескольких местах программы.

Этот пример также демонстрирует принцип, в соответствии с которым каждая функция должна решать одну конкретную задачу. Первая функция печатает каждую модель, а вторая выводит информацию о готовых моделях. Такой подход предпочтительнее решения обеих задач в функции. Если вы пишете функцию и видите, что она решает слишком много разных задач, попробуйте разделить ее код на две функции. Помните, что функции всегда можно вызывать из других функций. Эта возможность может пригодиться для разбиения сложных задач на серию составляющих.

## Запрет изменения списка в функции

Иногда требуется предотвратить изменение списка в функции. Допустим, у вас имеется список моделей для печати и вы пишете функцию для перемещения их в список готовых моделей, как в предыдущем примере. Возможно, даже после печати всех моделей исходный список нужно оставить для отчетности. Но по- скольку все имена моделей были перенесены из списка unprinted_designs, остался только пустой список; исходная версия списка потеряна. Проблему можно решить передачей функции копии списка вместо оригинала. В этом случае все изменения, вносимые функцией в список, будут распространяться только на копию, а оригинал списка остается неизменным.

Чтобы передать функции копию списка, можно поступить так:

*имя_функции*(*имя_списка*[:])

Синтаксис сегмента [:] создает копию списка для передачи функции. Если удаление элементов из списка unprinted_designs нежелательно, функцию print_models() можно вызвать так:

*print_models(unprinted_designs[:], completed_models)*

Функция print_models() может выполнить свою работу, потому что она все равно получает имена всех ненапечатанных моделей. Но на этот раз она использует не сам список unprinted_designs, а его копию. Список completed_models заполняется именами напечатанных моделей, как и в предыдущем случае, но исходный список функцией не изменяется.

Несмотря на то что передача копии позволяет сохранить содержимое списка, обычно функциям следует передавать исходный список (если у вас нет веских причин для передачи копии). Работа с существующим списком более эффективна, потому что программе не приходится тратить время и память на создание отдельной копии (лишние затраты особенно заметны при работе с большими списками).

## Передача произвольного набора аргументов

В некоторых ситуациях вы не знаете заранее, сколько аргументов должно быть передано функции. Python позволяет функции получить произвольное количество аргументов из вызывающей команды.

Для примера рассмотрим функцию для создания пиццы. Функция должна получить набор топпингов к пицце, но вы не знаете заранее, сколько топпингов закажет клиент. Функция в следующем примере получает один параметр *toppings, но этот параметр объединяет все аргументы, заданные в командной строке:

Input & Output:

```
def make_pizza(*toppings):

        """Вывод списка заказанных топпингов."""

        print(toppings)

make_pizza('pepperoni')

make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

Звездочка в имени параметра *toppings приказывает Python создать пустой кортеж с именем toppings и упаковать в него все полученные значения. Результат команды print в теле функции показывает, что Python успешно справляется и с вызовом функции с одним значением, и с вызовом с тремя значениями. Разные вызовы обрабатываются похожим образом.

Обратите внимание: Python упаковывает аргументы в кортеж даже в том случае, если функция получает всего одно значение:

*>>>('pepperoni',)*
*>>>('mushrooms', 'green peppers', 'extra cheese')*

Теперь команду print можно заменить циклом, который перебирает список топпингов и выводит описание заказанной пиццы:

Input & Output:

```
def make_pizza(*toppings):

    """Выводит описание пиццы."""

    print("\nMaking a pizza with the following toppings:")

    for topping in toppings:

        print(f"- {topping}")

make_pizza('pepperoni')

make_pizza('mushrooms', 'green peppers', 'extra cheese')
```

```
>>>Making a pizza with the following toppings:
>>>- pepperoni

>>>Making a pizza with the following toppings:
>>>- mushrooms
>>>- green peppers
>>>- extra cheese
```

Функция реагирует соответственно независимо от того, сколько значений она получила — одно или три.

Этот синтаксис работает независимо от количества аргументов, переданных функции.

## Позиционные аргументы с произвольными наборами аргументов

Если вы хотите, чтобы функция могла вызываться с разными количествами аргументов, параметр для получения произвольного количества аргументов должен стоять на последнем месте в определении функции. Python сначала подбирает соответствия для позиционных и именованных аргументов, а потом объединяет все остальные аргументы в последнем параметре.

Например, если функция должна получать размер пиццы, этот параметр должен стоять в списке до параметра *toppings:

Input & Output:

```
def make_pizza(size, *toppings):

    """Выводит описание пиццы."""

    print(f"\nMaking a {size}-inch pizza with the following toppings:")

    for topping in toppings:

        print(f"- {topping}")
```

*make_pizza(16, 'pepperoni')*

*make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')*

*>>>Making a 16-inch pizza with the following toppings:*
*>>>- pepperoni*

*>>>Making a 12-inch pizza with the following toppings:*
*>>>- mushrooms*
*>>>- green peppers*
*>>>- extra cheese*

В определении функции Python сохраняет первое полученное значение в параметре size. Все остальные значения, следующие за ним, сохраняются в кортеже toppings. В вызовах функций на первом месте располагается аргумент для параметра size, а за ним следуют сколько угодно дополнений.

В итоге для каждой пиццы указывается размер и количество дополнений, и каждый фрагмент информации выводится в положенном месте: сначала размер, а потом топпинги.

В программах часто используется имя обобщенного параметра *args для хранения произвольного набора позиционных аргументов .

## Использование произвольного набора именованных аргументов

Иногда программа должна получать произвольное количество аргументов, но вы не знаете заранее, какая информация будет передаваться функции. В таких случаях можно написать функцию, получающую столько пар «ключ-значение», сколько указано в команде вызова. Один из возможных примеров — построение пользовательских профилей: вы знаете, что вы получите информацию о пользователе, но не знаете заранее, какую именно. Функция build_profile() в следующем примере всегда получает имя и фамилию, но также может получать произвольное количество именованных аргументов:

Input & Output:

```
def build_profile(first, last, **user_info):

        """Строит словарь с информацией о пользователе."""

        user_info['first_name'] = first

        user_info['last_name'] = last

        return user_info

user_profile = build_profile('albert', 'einstein',
                             location='princeton',
                             field='physics')

print(user_profile)
```

*>>>{'location': 'princeton', 'field': 'physics',*
*>>>'first_name': 'albert', 'last_name': 'einstein'}*

Определение build_profile() ожидает получить имя и фамилию пользователя, а также позволяет передать любое количество пар «имя-значение». Две звездочки перед параметром **user_info заставляют Python создать пустой словарь с име- нем user_info и упаковать в него все полученные пары «имя-значение». Внутри функции вы можете обращаться к парам «имя-значение» из user_info точно так же, как в любом словаре.

В теле build_profile() в словарь user_info добавляются имя и фамилия, потому что эти два значения всегда передаются пользователями они еще не были помещены в словарь. Затем словарь user_info возвращается в точку вызова функции.

Вызовем функцию build_profile() и передадим ей имя 'albert', фамилию 'einstein' и еще две пары «ключ-значение» — location='princeton' и field='physics'. Программа сохраняет возвращенный словарь в user_profile и выводит его содержимое.

Возвращаемый словарь содержит имя и фамилию пользователя, а в данном случае еще и местонахождение и область исследований. Функция будет работать, сколько бы дополнительных пар «ключ-значение» ни было передано при вызове функции.

При написании функций допускаются самые разнообразные комбинации позиционных, именованных и произвольных значений. Полезно знать о существовании всех этих типов аргументов, потому что они часто будут встречаться вам при чтении чужого кода. Только с практическим опытом вы научитесь правильно использовать разные типы аргументов и поймете, когда следует применять каждый тип; а пока просто используйте самый простой способ, который позволит решить задачу. С течением времени вы научитесь выбирать наиболее эффективный вариант для каждой конкретной ситуации.

В программах часто используется имя обобщенного параметра **kwargs для хранения произвольного набора ключевых аргументов.

## Хранение функций в модулях

Одно из преимуществ функций заключается в том, что они отделяют блоки кода от основной программы. Если для функций были выбраны содержательные имена, ваша программа будет намного проще читаться. Можно пойти еще дальше и сохранить функции в отдельном файле, называемом *модулем*, а затем *импортировать* модуль в свою программу. Команда import сообщает Python, что код модуля должен быть доступен в текущем выполняемом программном файле.

Хранение функций в отдельных файлах позволяет скрыть второстепенные детали кода и сосредоточиться на логике более высокого уровня. Кроме того, функции можно использовать во множестве разных программ. Функции, хранящиеся в отдельных файлах, можно передать другим программистам без распространения полного кода программы. А умение импортировать функции позволит вам использовать библиотеки функций, написанные другими программистами.

Существует несколько способов импортирования модулей; все они кратко рассматриваются ниже.

## Импортирование всего модуля

Чтобы заняться импортированием функций, сначала необходимо создать модуль. *Модуль* представляет собой файл с расширением .py, содержащий код, который вы хотите импортировать в свою программу. Давайте создадим модуль с функцией make_pizza(). Для этого из файла pizza.py следует удалить все, кроме функции make_pizza():

Input & Output:

**make_pizza.py**

```
def make_pizza(size, *toppings):

        """Выводит описание пиццы."""

        print(f"\nMaking a {size}-inch pizza with the following toppings:")

        for topping in toppings:
```

```
        print(f"- {topping}")
```

Теперь создайте отдельный файл с именем making_pizzas.py в одном каталоге с pizza.py. Файл импортирует только что созданный модуль, а затем дважды вызывает make_pizza():

Input & Output:

***making_pizzas.py***

```
import pizza

pizza.make_pizza(16, 'pepperoni')

pizza.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

В процессе обработки этого файла строка import pizza говорит Python открыть файл pizza.py и скопировать все функции из него в программу. Вы не видите, как происходит копирование, потому что Python копирует код незаметно для пользователя во время выполнения программы. Вам необходимо знать одно: что любая функция, определенная в pizza.py, будет доступна в making_pizzas.py.

Чтобы вызвать функцию из импортированного модуля, укажите имя модуля (pizza), точку и имя функции (make_pizza()), как показано в строке. Код выдает тот же результат, что и исходная программа, в которой модуль не импортировался:

```
>>>Making a 16-inch pizza with the following toppings:
>>>- pepperoni

>>>Making a 12-inch pizza with the following toppings:
>>>- mushrooms
>>>- green peppers
>>>- extra cheese
```

Первый способ импортирования, при котором записывается команда import с именем модуля, открывает доступ программе ко всем функциям из модуля. Если вы используете эту разновидность команды import для импортирования всего модуля *имя_модуля*.py, то каждая функция модуля будет доступна в следующем синтаксисе:

*имя_модуля*.*имя_функции*()

## Импортирование конкретных функций

Также возможно импортировать конкретную функцию из модуля. Общий синтаксис выглядит так:

from *имя_модуля* import *имя_функции*

Вы можете импортировать любое количество функций из модуля, разделив их имена запятыми:

from *имя_модуля* import *функция*_0, *функция*_1, *функция*_2

Если ограничиться импортированием только той функции, которую вы намереваетесь использовать, пример making_pizzas .py будет выглядеть так:

```
from pizza import make_pizza

make_pizza(16, 'pepperoni')

make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')
```

При таком синтаксисе использовать точечную запись при вызове функции не обязательно. Так как функция make_pizza() явно импортируется в команде import, при использовании ее можно вызывать прямо по имени.

## Назначение псевдонима для функции

Если имя импортируемой функции может конфликтовать с именем существующей функции или функция имеет слишком длинное имя, его можно заменить коротким уникальным *псевдонимом* (alias) — альтернативным именем для функции. Псевдоним назначается функции при импортировании.

В следующем примере функции make_pizza() назначается псевдоним mp(), для чего при импортировании используется конструкция make_pizza as mp. Ключевое слово as переименовывает функцию, используя указанный псевдоним:

*from pizza import make_pizza as mp*

*mp(16, 'pepperoni')*

*mp(12, 'mushrooms', 'green peppers', 'extra cheese')*

Команда import в этом примере назначает функции make_pizza() псевдоним mp() для этой программы. Каждый раз, когда потребуется вызвать make_pizza(), достаточно включить вызов mp() — Python выполнит код make_pizza() без конфликтов с другой функцией make_pizza(), которую вы могли включить в этот файл программы.

Общий синтаксис назначения псевдонима выглядит так:

from *имя_модуля* import *имя_функции* as *псевдоним*

## Назначение псевдонима для модуля

Псевдоним также можно назначить для всего модуля. Назначение короткого имени для модуля — скажем, p для pizza — позволит вам быстрее вызывать функции модуля. Вызов p.make_pizza() получается более компактным, чем pizza. make_pizza():

*import pizza as p*

*p.make_pizza(16, 'pepperoni')*

*p.make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')*

Модулю pizza в команде import назначается псевдоним p, но все функции модуля сохраняют свои исходные имена. Вызов функций в записи p.make_pizza() не только компактнее pizza.make_pizza(); он также отвлекает внимание от имени модуля и помогает сосредоточиться на содержательных именах функций. Эти имена функций, четко показывающие, что делает каждая функция, важнее для удобочитаемости вашего кода, чем использование полного имени модуля.

Общий синтаксис выглядит так:

import *имя_модуля* as *псевдоним*

## Импортирование всех функций модуля

Также можно приказать Python импортировать каждую функцию в модуле; для этого используется оператор *:

*from pizza import **

*make_pizza(16, 'pepperoni')*

*make_pizza(12, 'mushrooms', 'green peppers', 'extra cheese')*

Звездочка в команде import приказывает Python скопировать каждую функцию из модуля pizza в файл программы. После импортирования всех функций вы сможете вызывать каждую функцию по имени без точечной записи. Тем не менее лучше не использовать этот способ с большими модулями, написанными другими разработчиками; если модуль содержит функцию, имя которой совпадает с существующим именем из вашего проекта, возможны неожиданные результаты. Python обнаруживает несколько функций или переменных с одинаковыми именами, и вместо импортирования всех функций по отдельности происходит замена этих функций.

В таких ситуациях лучше всего импортировать только нужную функцию или функции или же импортировать весь модуль с последующим применением точечной записи. При этом создается чистый код, легко читаемый и понятный. Я включил этот раздел только для того, чтобы вы понимали команды import вроде следующей, когда вы встретите их в чужом коде:

from *имя_модуля* import *

## Стилевое оформление функций

Функции должны иметь содержательные имена, состоящие из букв нижнего регистра и символов подчеркивания. Содержательные имена помогают вам и другим разработчикам понять, что же делает ваш код. Эти соглашения следует соблюдать и в именах модулей.

Каждая функция должна быть снабжена комментарием, который кратко поясняет, что же делает эта функция. Комментарий должен следовать сразу же за определением функции в формате строк документации. Если функция хорошо документирована, другие разработчики смогут использовать ее, прочитав только описание. Конечно, для этого они должны доверять тому, что код работает в соответствии с описанием, но если знать имя функции, то, какие аргументы ей нужны и какое значение она возвращает, они смогут использовать ее в своих программах.

Если для параметра задается значение по умолчанию, слева и справа от знака равенства не должно быть пробелов:

def *имя_функции*(*параметр*_0, *параметр*_1='*значение_по_умолчанию*')

Те же соглашения должны применяться для именованных аргументов в вызовах функций:

*имя_функции*(*значение*_0, *параметр*_1='*значение*').

Многие редакторы автоматически выравнивают дополнительные строки параметров по отступам, установленным в первой строке:

def *имя_функции*(

*параметр*_0, *параметр*_1, *параметр*_2, *параметр*_3,

*параметр*_4, *параметр*_5):

*тело функции*...

Если программа или модуль состоят из нескольких функций, эти функции можно разделить двумя пустыми строками. Так вам будет проще увидеть, где кончается одна функция и начинается другая.

Все команды import следует записывать в начале файла. У этого правила есть только одно исключение: файл может начинаться с комментариев, описывающих программу в целом.

# Глава 8. Классы

Объектно-ориентированное программирование по праву считается одной из самых эффективных методологий создания программных продуктов. В объектно-ориентированном программировании вы создаете классы, описывающие реально существующие предметы и ситуации, а затем создаете *объекты* на основе этих описаний. При написании класса определяется общее поведение для целой категории объектов.

Когда вы создаете конкретные объекты на базе этих классов, каждый объект автоматически наделяется общим поведением; после этого вы можете наделить каждый объект уникальными особенностями на свой выбор.

Создание объекта на основе класса называется *созданием экземпляра*; таким образом, вы работаете с экземплярами класса.

## Создание и использование класса

Классы позволяют моделировать практически все что угодно. Начнем с написания простого класса Dog, представляющего собаку — не какую-то конкретную, а собаку вообще. Что мы знаем о собаках? У них есть кличка и возраст. Также известно, что большинство собак умеют садиться и перекатываться по команде. Эти два вида информации (кличка и возраст) и два вида поведения (сидеть и перекатываться) будут включены в класс Dog, потому что они являются общими для большинства собак. Класс сообщает Python, как создать объект, представляющий собаку. После того как класс будет написан, мы используем его для создания экземпляров, каждый из которых представляет одну конкретную собаку.

## Создание класса Dog

В каждом экземпляре, созданном на основе класса Dog, будет храниться кличка (name) и возраст (age); кроме того, в нем будут присутствовать методы sit() и roll_over():

```
class Dog():

    """Простая модель собаки."""

    def __init__(self, name, age):

        """Инициализирует атрибуты name и age."""

        self.name = name

        self.age = age

    def sit(self):

        """Собака садится по команде."""

        print(f"{self.name} is now sitting.")

    def roll_over(self):

        """Собака перекатывается по команде."""

        print(f"{self.name} rolled over!")
```

Определяется класс с именем Dog. По общепринятым соглашениям имена, начинающиеся с символа верхнего регистра, в Python обозначают классы. Круглые скобки в определении класса пусты, потому что класс создается с нуля. В точке приведена строка документации с кратким описанием класса.

## Метод __init__()

Функция, являющаяся частью класса, называется *методом*. Все, что вы узнали ранее о функциях, также относится и к методам; единственное практическое различие — способ вызова методов. Метод __init__() в точке — специальный метод, который автоматически выполняется при создании каждого нового экземпляра на базе класса Dog. Имя метода начинается и заканчивается двумя символами подчеркивания; эта схема предотвращает конфликты имен стандартных методов Python и методов ваших классов. Будьте внимательны: два символа подчеркивания должны стоять на *каждой* стороне __init__(). Если вы поставите только один символ подчеркивания с каждой стороны, то метод не будет вызываться автоматически при использовании класса, что может привести к появлению коварных ошибок.

Метод __init__() определяется с тремя параметрами: self, name и age. Параметр self обязателен в определении метода; он должен предшествовать всем остальным параметрам. Он должен быть включен в определение, потому что при будущем вызове метода __init__() (для создания экземпляра Dog) Python автоматически передает аргумент self. При каждом вызове метода, связанного с классом, автоматически передается self — ссылка на экземпляр; она предоставляет конкретному экземпляру доступ к атрибутам и методам класса. Когда вы создаете экземпляр Dog, Python вызывает метод __init__() из класса Dog. Мы передаем Dog() кличку и возраст в аргументах; значение self передается автоматически, так что его передавать не нужно. Каждый раз, когда вы захотите создать экземпляр на основе класса Dog, необходимо предоставить значения только двух последних аргументов, name и age.

Каждая из двух переменных, снабжена префиксом self. Любая переменная с префиксом self доступна для каждого метода в классе, и вы также сможете обращаться к этим переменным в каждом экземпляре, созданном на основе класса. Конструкция self.name = name берет значение, хранящееся в параметре name, и сохраняет его в переменной name, которая затем связывается с создаваемым экземпляром. Процесс также повторяется с self.age = age.

Переменные, к которым вы обращаетесь через экземпляры, также называются *атрибутами*.

В классе Dog также определяются два метода: sit() и roll_over(). Так как этим методам не нужна дополнительная информация (кличка или возраст), они определяются с единственным параметром self. Экземпляры, которые будут созданы позднее, смогут вызывать эти методы. Пока методы sit() и roll_over() ограничиваются простым выводом сообщения о том, что собака садится или перекатывается.

## Создание экземпляра класса

Считайте, что класс — это своего рода инструкция по созданию экземпляров. Соответственно класс Dog — инструкция по созданию экземпляров, представляющих конкретных собак.

Создадим экземпляр, представляющий конкретную собаку:

```
class Dog():

...

my_dog = Dog('willie', 6)

print(f"My dog's name is {my_dog.name}.")

print(f"My dog is {my_dog.age} years old.")
```

Использованный в данном случае класс Dog был написан в предыдущем примере. В точке мы приказываем Python создать экземпляр собаки с кличкой 'willie' и возрастом 6 лет. В процессе обработки этой строки Python вызывает метод __init__() класса Dog с аргументами 'willie' и 6. Метод __init__() создает экземпляр, представляющий конкретную собаку, и присваивает его атрибутам name и age переданные значения. Затем Python возвращает экземпляр, представляющий собаку. Этот экземпляр сохраняется в переменной my_dog. Здесь нелишне

вспомнить соглашения по записи имен: обычно считается, что имя, начинающееся с символа верхнего регистра (например, Dog), обозначает класс, а имя, записанное в нижнем регистре (например, my_dog), обозначает отдельный экземпляр, созданный на базе класса.

## Обращение к атрибутам

Для обращения к атрибутам экземпляра используется «точечная» запись. В строке мы обращаемся к значению атрибута name экземпляра my_dog:

```
my_dog.name
```

Точечная запись часто используется в Python. Этот синтаксис показывает, как Python ищет значения атрибутов. В данном случае Python обращается к экземпляру my_dog и ищет атрибут name, связанный с экземпляром my_dog. Это тот же атрибут, который обозначался self.name в классе Dog. В точке тот же прием используется для работы с атрибутом age.

Пример выводит известную информацию о my_dog:

```
My dog's name is Willie.

My dog is 6 years old.
```

## Вызов методов

После создания экземпляра на основе класса Dog можно применять точечную запись для вызова любых методов, определенных в Dog:

```
class Dog():
...

my_dog = Dog('willie', 6)

my_dog.sit()

my_dog.roll_over()
```

Чтобы вызвать метод, укажите экземпляр (в данном случае my_dog) и вызываемый метод, разделив их точкой. В ходе обработки my_dog.sit() Python ищет метод sit() в классе Dog и выполняет его код. Строка my_dog.roll_over() интерпретируется аналогичным образом.

Теперь экземпляр послушно выполняет полученные команды:

```
Willie is now sitting.

Willie rolled over!
```

Это очень полезный синтаксис. Если атрибутам и методам были присвоены содержательные имена (например, name, age, sit() и roll_over()), разработчик сможет легко понять, что делает блок кода — даже если он видит этот блок впервые.

## Создание нескольких экземпляров

На основе класса можно создать столько экземпляров, сколько вам потребуется. Создадим второй экземпляр Dog с именем your_dog:

```
class Dog():

...

my_dog = Dog('willie', 6)
```

```
your_dog = Dog('lucy', 3)

print(f"My dog's name is {my_dog.name}.")

print(f"My dog is {my_dog.age} years old.")

my_dog.sit()

print(f"\nYour dog's name is {your_dog.name}.")

print(f"Your dog is {your_dog.age} years old.")

your_dog.sit()
```

В этом примере создаются два экземпляра с именами Willie и Lucy. Каждый экземпляр обладает своим набором атрибутов и способен выполнять действия из общего набора:

```
My dog's name is Willie.
My dog is 6 years old.
Willie is now sitting.

Your dog's name is Lucy.
Your dog is 3 years old.
Lucy is now sitting.
```

Даже если второй собаке будет назначено то же имя и возраст, Python все равно создаст отдельный экземпляр класса Dog. Вы можете создать сколько угодно экземпляров одного класса при условии, что эти экземпляры хранятся в переменных с разными именами или занимают разные позиции в списке либо словаре.

## Работа с классами и экземплярами

Классы могут использоваться для моделирования многих реальных ситуаций. После того как класс будет написан, разработчик проводит большую часть времени за работой с экземплярами, созданными на основе этого класса. Одной из первых задач станет изменение атрибутов, связанных с конкретным экземпляром. Атрибуты экземпляра можно изменять напрямую или же написать методы, изменяющие атрибуты по особым правилам.

## Класс Car

Напишем класс, представляющий автомобиль. Этот класс будет содержать информацию о типе машины, а также метод для вывода краткого описания:

```
class Car():

    """Простая модель автомобиля."""

    def __init__(self, make, model, year):

        """Инициализирует атрибуты описания автомобиля."""

        self.make = make

        self.model = model

        self.year = year

    def get_descriptive_name(self):

        """Возвращает аккуратно отформатированное описание."""
```

```
        long_name = f"{self.year} {self.manufacturer} {self.model}"

        return long_name.title()

my_new_car = Car('audi', 'a4', 2019)

print(my_new_car.get_descriptive_name())
```

В точке в классе Car определяется метод __init__(); его список параметров начинается с self, как и в классе Dog. За ним следуют еще три параметра: make, model и year. Метод __init__() получает эти параметры и сохраняет их в атрибутах, которые будут связаны с экземплярами, созданными на основе класса. При создании нового экземпляра Car необходимо указать фирму-производитель, модель и год выпуска для данного экземпляра.

Затем определяется метод get_descriptive_name(), который объединяет год выпуска, фирму-производитель и модель в одну строку с описанием. Это избавит вас от необходимости выводить значение каждого атрибута по отдельности. Для работы со значениями атрибутов в этом методе используется синтаксис self.make, self.model и self.year. В точке создается экземпляр класса Car, который сохраняется в переменной my_new_car. Затем вызов метода get_descriptive_name() показывает, с какой машиной работает программа:

```
2019 Audi A4
```

Чтобы класс был более интересным, добавим атрибут, изменяющийся со временем, — в нем будет храниться пробег машины в милях.

## Назначение атрибуту значения по умолчанию

Каждый атрибут класса должен иметь исходное значение, даже если оно равно 0 или пустой строке. В некоторых случаях (например, при задании значений по умолчанию) это исходное значение есть смысл задавать в теле метода __init__(); в таком случае передавать параметр для этого атрибута при создании объекта не обязательно.

Добавим атрибут с именем odometer_reading, исходное значение которого всегда равно 0. Также в класс будет включен метод read_odometer() для чтения текущих показаний одометра:

```
class Car():

    def __init__(self, make, model, year):
        """Инициализирует атрибуты описания автомобиля."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        …

    def read_odometer(self):

    """Выводит пробег машины в милях."""

    print(f"This car has {self.odometer_reading} miles on it.")

 my_new_car = Car('audi', 'a4', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.read_odometer()
```

Когда Python вызывает метод __init__() для создания нового экземпляра, этот метод сохраняет фирму-производителя, модель и год выпуска в атрибутах, как и в предыдущем случае. Затем Python создает новый атрибут с именем odometer_ reading и присваивает ему исходное значение 0 . Также в класс добавляется новый метод read_odometer() , который упрощает чтение пробега машины в милях.

Сразу же после создания машины ее пробег равен 0:

*2019 Audi A4*
*This car has 0 miles on it.*

Далее изменим значение этого атрибута.

## Изменение значений атрибутов

Значение атрибута можно изменить одним из трех способов: изменить его прямо в экземпляре, задать значение при помощи метода или изменить его с приращением (то есть прибавлением определенной величины) при помощи метода. Рассмотрим все эти способы.

## Прямое изменение значения атрибута

Чтобы изменить значение атрибута, проще всего обратиться к нему прямо через экземпляр. В следующем примере на одометре напрямую выставляется значение 23:

```
class Car:

        ...

my_new_car = Car('audi', 'a4', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23

my_new_car.read_odometer()
```

В точке точечная запись используется для обращения к атрибуту odometer_reading экземпляра и прямого присваивания его значения. Эта строка приказывает Python взять экземпляр my_new_car, найти связанный с ним атрибут odometer_reading и задать значение атрибута равным 23:

*2019 Audi A4*
*This car has 23 miles on it.*

Иногда подобные прямые обращения к атрибутам допустимы, но чаще разработчик пишет вспомогательный метод, который изменяет значение за него.

## Изменение значения атрибута с использованием метода

В класс можно включить методы, которые изменяют некоторые атрибуты за вас. Вместо того чтобы изменять атрибут напрямую, вы передаете новое значение методу, который берет обновление атрибута на себя.

В следующем примере в класс включается метод update_odometer() для изменения показаний одометра:

```
class Car:
    ...

        def update_odometer(self, mileage):
```

```
        """Устанавливает заданное значение на одометре."""

            self.odometer_reading = mileage

my_new_car = Car('audi', 'a4', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.update_odometer(23)

my_new_car.read_odometer()
```

Класс Car почти не изменился, в нем только добавился метод update_odometer(). Этот метод получает пробег в милях и сохраняет его в self.odometer_reading. В точке мы вызываем метод update_odometer() и передаем ему значение 23 в аргументе (соответствующем параметру mileage в определении метода). Метод устанавливает на одометре значение 23, а метод read_odometer() выводит текущие показания:

```
2019 Audi A4
This car has 23 miles on it.
```

Метод update_odometer() можно расширить так, чтобы при каждом изменении показаний одометра выполнялась некоторая дополнительная работа. Добавим проверку, которая гарантирует, что никто не будет пытаться сбрасывать показания одометра:

```
 class Car():

            ...

        def update_odometer(self, mileage):
                """
                Устанавливает на одометре заданное значение.
                При попытке обратной подкрутки изменение отклоняется.
                """
                if mileage >= self.odometer_reading:

                        self.odometer_reading = mileage

                else:

                        print("You can't roll back an odometer!")
```

Теперь update_odometer() проверяет новое значение перед изменением атрибута. Если новое значение mileage больше или равно текущему, self.odometer_reading, показания одометра можно обновить новым значением. Если же новое значение меньше текущего, вы получите предупреждение о недопустимости обратной подкрутки.

## Изменение значения атрибута с приращением

Иногда значение атрибута требуется изменить с заданным приращением (вместо того, чтобы присваивать атрибуту произвольное новое значение). Допустим, вы купили подержанную машину и проехали на ней 100 миль с момента покупки. Следующий метод получает величину приращения и прибавляет ее к текущим показаниям одометра:

```
 class Car():

        ...

        def update_odometer(self, mileage):
```

```
    ...

    def increment_odometer(self, miles):

        """Увеличивает показания одометра с заданным приращением."""

        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2015)

print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23_500)

my_used_car.read_odometer()

my_used_car.increment_odometer(100)

my_used_car.read_odometer()
```

Новый метод increment_odometer() в точке получает расстояние в милях и прибавляет его к self.odometer_reading. В точке создается экземпляр my_used_car. Мы инициализируем показания его одометра значением 23 500; для этого вызывается метод update_odometer(), которому передается значение 23 500. Затем вызывается метод increment_odometer(), которому передается значение 100, чтобы увеличить показания одометра на 100 миль, пройденные с момента покупки:

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
```

При желании можно легко усовершенствовать этот метод, чтобы он отклонял отрицательные приращения; тем самым вы предотвратите обратную подкрутку одометра.

Подобные методы управляют обновлением внутренних значений экземпляров (таких, как показания одометра), однако любой пользователь, имеющий доступ к программному коду, сможет напрямую задать атрибуту любое значение. Эффективная схема безопасности должна уделять особое внимание таким подробностям, не ограничиваясь простейшими проверками.

## Наследование

Работа над новым классом не обязана начинаться с нуля. Если класс, который вы пишете, представляет собой специализированную версию ранее написанного класса, вы можете воспользоваться *наследованием*. Один класс, *наследующий* от другого, автоматически получает все атрибуты и методы первого класса. Исходный класс называется *родителем*, а новый класс — *потомком*. Класс-потомок наследует атрибуты и методы родителя, но при этом также может определять собственные атрибуты и методы.

## Метод __init__() класса-потомка

При написании нового класса на базе существующего класса часто приходится вызывать метод __init__() класса-родителя. При этом происходит инициализация любых атрибутов, определенных в методе __init__() родителя, и эти атрибуты становятся доступными для класса-потомка.

Например, попробуем построить модель электромобиля. Электромобиль представляет собой специализированную разновидность автомобиля, поэтому новый класс ElectricCar можно создать на базе класса Car, написанного ранее. Тогда нам останется добавить в него код атрибутов и поведения, относящегося только к электромобилям.

Начнем с создания простой версии класса ElectricCar, который делает все, что делает класс Car:

```
class Car():
        """Простая модель автомобиля."""
        def __init__(self, make, model, year):
                self.make = make
                self.model = model
                self.year = year
                self.odometer_reading = 0

        def get_descriptive_name(self):
                long_name = f"{self.year} {self.manufacturer} {self.model}"
                return long_name.title()

        def read_odometer(self):
                print(f"This car has {self.odometer_reading} miles on it.")

        def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
                self.odometer_reading = mileage
        else:
                print("You can't roll back an odometer!")
        def increment_odometer(self, miles):
                self.odometer_reading += miles
class ElectricCar(Car):
        """Представляет аспекты машины, специфические для электромобилей."""
        def __init__(self, make, model, year):
                """инициализирует атрибуты класса-родителя."""
                super().__init__(make, model, year)
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
```

В точке строится экземпляр Car. При создании класса-потомка класс-родитель должен быть частью текущего файла, а его определение должно предшествовать определению класса-потомка в файле. Затем определяется класс-потомок ElectricCar. В определении потомка имя класса-родителя заключается в круглые скобки. Метод __init__() в точке получает информацию, необходимую для создания экземпляра Car.

Функция super() в строке — специальная функция, которая позволяет вызвать метод родительского класса. Эта строка приказывает Python вызвать метод __init__() класса Car, в результате чего экземпляр ElectricCar получает доступ ко всем атрибутам класса-родителя. Имя super происходит из распространенной терминологии: класс-родитель называется *суперклассом*, а класс-потомок — *под-классом*.

Чтобы проверить, правильно ли сработало наследование, попробуем создать электромобиль с такой же информацией, которая передается при создании обычного экземпляра Car. В точке мы создаем экземпляр класса ElectricCar и сохраняем его в my_tesla. Эта строка вызывает метод __init__(), определенный в ElectricCar, который, в свою очередь, приказывает Python вызвать метод __init__(), определенный в классе-родителе Car. При вызове передаются аргументы 'tesla', 'model s' и 2019.

Кроме __init__(), класс еще не содержит никаких атрибутов или методов, специфических для электромобилей. Пока мы просто убеждаемся в том, что класс электромобиля содержит все поведение, присущее классу автомобиля:

*2019 Tesla Model S*

Экземпляр ElectricCar работает так же, как экземпляр Car; можно переходить к определению атрибутов и методов, специфических для электромобилей.

## Определение атрибутов и методов класса-потомка

После создания класса-потомка, наследующего от класса-родителя, можно переходить к добавлению новых атрибутов и методов, необходимых для того, чтобы потомок отличался от родителя.

Добавим атрибут, специфический для электромобилей (например, мощность аккумулятора), и метод для вывода информации об этом атрибуте:

```
 class Car():

        ...

class ElectricCar(Car):

        """Представляет аспекты машины, специфические для электромобилей."""

        def __init__(self, make, model, year):
                """
                Инициализирует атрибуты класса-родителя.
                Затем инициализирует атрибуты, специфические для электромобиля.
                """
                super().__init__(make, model, year)

                self.battery_size = 75

        def describe_battery(self):

                """Выводит информацию о мощности аккумулятора."""

                print(f"This car has a {self.battery_size}-kWh battery.")
```

```
my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())

my_tesla.describe_battery()
```

Добавляется новый атрибут self.battery_size, которому присваивается исходное значение — скажем, 75. Этот атрибут будет присутствовать во всех экземплярах, созданных на основе класса ElectricCar (но не во всяком экземпляре Car). Также добавляется метод с именем describe_battery(), который выводит информацию об аккумуляторе в точке. При вызове этого метода выводится описание, которое явно относится только к электромобилям:

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

Возможности специализации класса ElectricCar беспредельны. Вы можете добавить сколько угодно атрибутов и методов, чтобы моделировать электромобиль с любой нужной точностью. Атрибуты или методы, которые могут принадлежать любой машине (а не только электромобилю), должны добавляться в класс Car вместо ElectricCar. Тогда эта информация будет доступна всем пользователям класса Car, а класс ElectricCar будет содержать только код информации и поведения, специфических для электромобилей.

## Переопределение методов класса-родителя

Любой метод родительского класса, который в моделируемой ситуации делает не то, что нужно, можно переопределить. Для этого в классе-потомке определяется метод с тем же именем, что и у метода класса-родителя. Python игнорирует метод родителя и обращает внимание только на метод, определенный в потомке.

Допустим, в классе Car имеется метод fill_gas_tank(). Для электромобилей заправка бензином бессмысленна, поэтому этот метод логично переопределить. Например, это можно сделать так:

```
class ElectricCar(Car):

    ...

    def fill_gas_tank(self):

        """У электромобилей нет бензобака."""

        print("This car doesn't need a gas tank!")
```

И если кто-то попытается вызвать метод fill_gas_tank() для электромобиля, Python проигнорирует метод fill_gas_tank() класса Car и выполнит вместо него этот код. С применением наследования потомок сохраняет те аспекты родителя, которые вам нужны, и переопределяет все ненужное.

## Экземпляры как атрибуты

При моделировании явлений реального мира в программах классы нередко дополняются все большим количеством подробностей. Списки атрибутов и методов растут, и через какое-то время файлы становятся длинными и громоздкими. В такой ситуации часть одного класса нередко можно записать в виде отдельного класса. Большой код разбивается на меньшие классы, которые работают во взаимодействии друг с другом.

Например, при дальнейшей доработке класса ElectricCar может оказаться, что в нем появилось слишком много атрибутов и методов, относящихся к аккумулятору. В таком случае можно остановиться и переместить все эти атрибуты и методы в отдельный класс с именем Battery. Затем экземпляр Battery становится атрибутом класса ElectricCar:

```
class Car():

...

class Battery():

        """Простая модель аккумулятора электромобиля."""

        def __init__(self, battery_size=75):

                """Инициализирует атрибуты аккумулятора."""

                self.battery_size = battery_size

        def describe_battery(self):

                """Выводит информацию о мощности аккумулятора."""

                print(f"This car has a {self.battery_size}-kWh battery.")

class ElectricCar(Car):

        """Представляет аспекты машины, специфические для электромобилей."""

        def __init__(self, make, model, year):

                """
                Инициализирует атрибуты класса-родителя.
                Затем инициализирует атрибуты, специфические для электромобиля.
                """
                super().__init__(make, model, year)

                self.battery = Battery()

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())

my_tesla.battery.describe_battery()
```

В точке определяется новый класс с именем Battery, который не наследует ни один из других классов. Метод __init__() в точке получает один параметр battery_size, кроме self. Если значение не предоставлено, этот необязательный параметр задает battery_size значение 75. Метод describe_battery() также перемещен в этот класс.

Затем в класс ElectricCar добавляется атрибут с именем self.battery. Эта строка приказывает Python создать новый экземпляр Battery (со значением battery_size по умолчанию, равным 75, потому что значение не задано) и сохранить его в атрибуте self.battery. Это будет происходить при каждом вызове __init__(); теперь любой экземпляр ElectricCar будет иметь автоматически создаваемый экземпляр Battery.

Программа создает экземпляр электромобиля и сохраняет его в переменной my_ tesla. Когда потребуется вывести описание аккумулятора, необходимо обратиться к атрибуту battery:

```
my_tesla.battery.describe_battery()
```

Эта строка приказывает Python обратиться к экземпляру my_tesla, найти его атрибут battery и вызвать метод describe_battery(), связанный с экземпляром Battery из атрибута.

Результат выглядит так же, как и в предыдущей версии:

*2019 Tesla Model S*
*This car has a 75-kWh battery.*

Казалось бы, новый вариант требует большой дополнительной работы, но теперь аккумулятор можно моделировать с любой степенью детализации без загромождения класса ElectricCar. Добавим в Battery еще один метод, который выводит запас хода на основании мощности аккумулятора:

```
class Car():

        ...

class Battery():

        ...

        def get_range(self):

                """Выводит приблизительный запас хода для аккумулятора."""

                if self.battery_size == 75:

                        range = 260

                elif self.battery_size == 100:

                        range = 315

                print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):

...

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())

my_tesla.battery.describe_battery()

 my_tesla.battery.get_range()
```

Новый метод get_range() в точке проводит простой анализ. Если мощность равна 75, то get_range() устанавливает запас хода 260 миль, а при мощности 100 кВт/ч запас хода равен 315 милям. Затем программа выводит это значение. Когда вы захотите использовать этот метод, его придется вызывать через атрибут battery.

Результат сообщает запас хода машины в зависимости от мощности аккумулятора:

*2019 Tesla Model S*
*This car has a 75-kWh battery.*
*This car can go approximately 260 miles on a full charge.*

## Моделирование объектов реального мира

Занимаясь моделированием более сложных объектов, таких как электромобили, вы столкнетесь с множеством интересных вопросов. Является ли запас хода электромобиля свойством аккумулятора или машины? Если вы описываете только одну машину, вероятно,

можно связать метод get_range() с классом Battery. Но если моделируется целая линейка машин от производителя, вероятно, метод get_range() правильнее будет поместить в класс ElectricCar. Метод get_range() по-прежнему будет проверять мощность аккумулятора перед определением запаса хода, но он будет сообщать запас хода для той машины, с которой он связан. Также возможно связать метод get_range() с аккумулятором, но передавать ему параметр (например, car_model). Метод get_range() будет определять запас хода на основании мощности аккумулятора и модели автомобиля.

## Импортирование классов

С добавлением новой функциональности в классы файлы могут стать слишком длинными, даже при правильном использовании наследования. В соответствии с общей философией Python файлы не должны загромождаться лишними подробностями. Для этого Python позволяет хранить классы в модулях и импортировать нужные классы в основную программу.

## Импортирование одного класса

Начнем с создания модуля, содержащего только класс Car. При этом возникает неочевидный конфликт имен: в этой главе уже был создан файл с именем car.py, но этот модуль тоже должен называться car py, потому что в нем содержится код класса Car. Мы решим эту проблему, сохранив класс Car в модуле с именем car.py, заменяя им файл car.py, который использовался ранее. В дальнейшем любой программе, использующей этот модуль, придется присвоить более конкретное имя файла — например, my_car.py. Ниже приведен файл car.py с кодом класса Car:

***car.py***

```
"""Класс для представления автомобиля."""

class Car():

    """Простая модель автомобиля."""

    def __init__(self, make, model, year):

        """Инициализирует атрибуты описания автомобиля."""

        self.make = make

        self.model = model

        self.year = year

        self.odometer_reading = 0

    def get_descriptive_name(self):

        """Возвращает аккуратно отформатированное описание."""

         long_name = f"{self.year} {self.manufacturer} {self.model}"

        return long_name.title()

    def read_odometer(self):

        """Выводит пробег машины в милях."""

        print(f"This car has {self.odometer_reading} miles on it.")


    def update_odometer(self, mileage):
```

```
        """
        Устанавливает на одометре заданное значение.
        При попытке обратной подкрутки изменение отклоняется.
        """

        if mileage >= self.odometer_reading:

                self.odometer_reading = mileage

        else:

                print("You can't roll back an odometer!")


    def increment_odometer(self, miles):

        """Увеличивает показания одометра с заданным приращением."""

        self.odometer_reading += miles
```

Теперь мы создадим отдельный файл с именем my_car.py. Этот файл импортирует класс Car и создает экземпляр этого класса:

***my_car.py***

```
from car import Car

my_new_car = Car('audi', 'a4', 2019)

print(my_new_car.get_descriptive_name())

my_new_car.odometer_reading = 23

my_new_car.read_odometer()
```

Команда import в точке приказывает Python открыть модуль car и импортировать класс Car. Теперь мы можем использовать класс Car так, как если бы он был определен в этом файле. Результат остается тем же, что и в предыдущей версии:

```
2019 Audi A4
This car has 23 miles on it.
```

Импортирование классов повышает эффективность программирования. Представьте, каким длинным получился бы файл этой программы, если бы в него был включен весь класс Car. Перемещая класс в модуль и импортируя этот модуль, вы получаете ту же функциональность, но основной файл программы при этом остается чистым и удобочитаемым. Большая часть логики также может храниться в отдельных файлах; когда ваши классы работают так, как положено, вы можете забыть об этих файлах и сосредоточиться на высокоуровневой логике основной программы.

## Хранение нескольких классов в модуле

В одном модуле можно хранить сколько угодно классов, хотя все эти классы должны быть каким-то образом связаны друг с другом. Оба класса, Battery и ElectricCar, используются для представления автомобилей, поэтому мы добавим их в модуль car.py:

***car.py***

```
"""Классы для представления машин с бензиновым и электродвигателем."""
```

```
class Car():

...

class Battery():

        """Простая модель аккумулятора электромобиля."""

        def __init__(self, battery_size=70):

                """Инициализация атрибутов аккумулятора."""

                self.battery_size = battery_size

        def describe_battery(self):

                """Выводит информацию о мощности аккумулятора."""

                print(f"This car has a {self.battery_size}-kWh battery.")

        def get_range(self):

                """Выводит приблизительный запас хода для аккумулятора."""

                if self.battery_size == 75:

                        range = 260

                elif self.battery_size == 100:

                        range = 315

                print(f"This car can go about {range} miles on a full charge.")

class ElectricCar(Car):

        """Представляет аспекты машины, специфические для электромобилей."""

        def __init__(self, make, model, year):

                """
                Инициализирует атрибуты класса-родителя.
                Затем инициализирует атрибуты, специфические для электромобиля.
                """

                super().__init__(make, model, year)

                self.battery = Battery()
```

Теперь вы можете создать новый файл с именем my_electric_car.py, импортировать класс ElectricCar и создать новый экземпляр электромобиля:

***my_electric_car.py***

```
from car import ElectricCar

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
```

*my_tesla.battery.describe_battery()*

*my_tesla.battery.get_range()*

Программа выводит тот же результат, что и в предыдущем случае, хотя большая часть ее логики скрыта в модуле:

*2019 Tesla Model S*
*This car has a 75-kWh battery.*
*This car can go approximately 260 miles on a full charge.*

## Импортирование нескольких классов из модуля

В файл программы можно импортировать столько классов, сколько потребуется. Если вы захотите создать обычный автомобиль и электромобиль в одном файле, потребуется импортировать оба класса, Car и ElectricCar:

***my_cars.py***

*from car import Car, ElectricCar*

*my_beetle = Car('volkswagen', 'beetle', 2019)*

*print(my_beetle.get_descriptive_name())*

*my_tesla = ElectricCar('tesla', 'roadster', 2019)*

*print(my_tesla.get_descriptive_name())*

Чтобы импортировать несколько классов из модуля, разделите их имена запятыми. После того как необходимые классы будут импортированы, вы можете создать столько экземпляров каждого класса, сколько вам потребуется.

В этом примере создается обычный автомобиль Volkswagen Beetle и электромобиль Tesla Roadster:

*2019 Volkswagen Beetle*
*2019 Tesla Roadster*

## Импортирование всего модуля

Также возможно импортировать весь модуль, а потом обращаться к нужным классам с использованием точечной записи. Этот способ прост, а полученный код легко читается. Так как каждый вызов, создающий экземпляр класса, включает имя модуля, в программе не будет конфликтов с именами, используемыми в текущем файле.

***my_cars.py***

*import car*

*my_beetle = car.Car('volkswagen', 'beetle', 2019)*

*print(my_beetle.get_descriptive_name())*

*my_tesla = car.ElectricCar('tesla', 'roadster', 2019)*

*print(my_tesla.get_descriptive_name())*

Импортируется весь модуль car, после чего программа обращается к нужным классам с использованием синтаксиса *имя_модуля.имя_класса*. В точке снова создается экземпляр Volkswagen Beetle, а затем — экземпляр Tesla Roadster.

## Импортирование всех классов из модуля

Для импортирования всех классов из модуля используется следующий синтаксис:

from *имя_модуля* import *

Использовать этот способ не рекомендуется по двум причинам. Прежде всего, бывает полезно прочитать команды import в начале файла и получить четкое представление о том, какие классы используются в программе, а при таком подходе неясно, какие классы из модуля нужны программе. Также возможны конфликты с именами в файле. Если вы случайно импортируете класс с именем, уже присутствующим в файле, в программе могут возникнуть ошибки.

Итак, если вам нужно импортировать большое количество классов из модуля, лучше импортировать весь модуль и воспользоваться синтаксисом *имя_модуля*. *имя_класса*. Хотя вы не видите перечень всех используемых классов в начале файла, по крайней мере ясно видно, где модуль используется в программе. Также предотвращаются потенциальные конфликты имен, которые могут возникнуть при импортировании каждого класса в модуле.

## Импортирование модуля в модуль

Иногда классы приходится распределять по нескольким модулям, чтобы избежать чрезмерного разрастания одного файла и хранения несвязанных классов в одном модуле. При хранении классов в нескольких модулях может оказаться, что один класс из одного модуля зависит от класса из другого модуля. В таких случаях необходимый класс можно импортировать в первый модуль.

Допустим, класс Car хранится в одном модуле, а классы ElectricCar и Battery — в другом. Мы создадим новый модуль с именем electric_car.py (он заменит файл electric_car.py, созданный ранее) и копируем в него только классы Battery и ElectricCar:

***electric_car.py***

```
"""Набор классов для представления электромобилей."""

from car import Car

class Battery():

        ...

class ElectricCar(Car):

        ...
```

Классу ElectricCar необходим доступ к классу-родителю Car, поэтому класс Car импортируется прямо в модуль в точке. Если вы забудете вставить эту команду, при попытке создания экземпляра ElectricCar произойдет ошибка. Также необходимо обновить модуль Car, чтобы он содержал только класс Car:

***car.py***

```
"""Простая модель автомобиля."""

class Car():

        ...
```

Теперь вы можете импортировать классы из каждого модуля по отдельности и создать ту разновидность машины, которая вам нужна:

***my_cars.py***

```
from car import Car

from electric_car import ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2019)

print(my_beetle.get_descriptive_name())

my_tesla = ElectricCar('tesla', 'roadster', 2019)

print(my_tesla.get_descriptive_name())
```

В точке класс Car импортируется из своего модуля, а класс ElectricCar — из своего. После этого создаются экземпляры обоих разновидностей. Вывод показывает, что экземпляры были созданы правильно:

```
2019 Volkswagen Beetle
2019 Tesla Roadster
```

## Использование псевдонимов

Как было показано в главе 7, псевдонимы весьма полезны при использовании модулей для организации кода проектов. Псевдонимы также могут использоваться и при импортировании классов.

Для примера возьмем программу, которая должна создать группу экземпляров электрических машин. Многократно вводить (и читать) имя ElectricCar будет очень утомительно. Имени ElectricCar можно назначить псевдоним в команде import:

```
from electric_car import ElectricCar as EC
```

С этого момента вы сможете использовать этот псевдоним каждый раз, когда вам потребуется создать экземпляр ElectricCar:

```
my_tesla = EC('tesla', 'roadster', 2019)
```

## Выработка рабочего процесса

Как видите, Python предоставляет много возможностей структурирования кода в крупных проектах. Вы должны знать все эти возможности, чтобы найти удачные способы организации своих проектов, а также лучше понимать код других разработчиков.

## Стандартная библиотека Python

*Стандартная библиотека Python* представляет собой набор модулей, включаемых в каждую установленную копию Python.

Чтобы использовать любую функцию или класс из стандартной библиотеки, достаточно включить простую команду import в начало файла. Для примера рассмотрим модуль random, который может пригодиться для моделирования многих реальных ситуаций.

В частности, модуль random содержит интересную функцию randint(). Эта функ- ция получает два целочисленных аргумента и возвращает случайно выбранное целое число в диапазоне, определяемом этими двумя числами (включительно).

В следующем примере генерируется случайное число в диапазоне от 1 до 6:

Input & Output:

```
from random import randint

randint(1, 6)

>>>3
```

Другая полезная функция, choice(), получает список или кортеж и возвращает случайно выбранный элемент:

Input & Output:

```
from random import choice

players = ['charles', 'martina', 'michael', 'florence', 'eli']

first_up = choice(players)

first_up

>>>'florence'
```

Модуль random не должен использоваться при построении приложений, связанных с безопасностью.

Модули также можно загружать из внешних источников.

## Оформление классов

В стилевом оформлении классов есть несколько моментов, о которых стоит упомянуть отдельно, особенно с усложнением ваших программ.

Имена классов должны записываться в «верблюжьем» регистре: первая буква каждого слова записывается в верхнем регистре, слова не разделяются пробелами. Имена экземпляров и модулей записываются в нижнем регистре с разделением слов символами подчеркивания.

Каждый класс должен иметь строку документации, следующую сразу же за определением класса. Строка документации должна содержать краткое описание того, что делает класс, и в ней должны соблюдаться те же соглашения по форматированию, которые вы использовали при написании строк документации в функциях. Каждый модуль также должен содержать строку документации с описанием возможных применений классов в модуле.

Пустые строки могут использоваться для структурирования кода, но злоупотреблять ими не стоит. В классах можно разделять методы одной пустой строкой, а в модулях для разделения классов можно использовать две пустые строки.

Если вам потребуется импортировать модуль из стандартной библиотеки и модуль из библиотеки, написанной вами, начните с команды import для модуля стандартной библиотеки. Затем добавьте пустую строку и команду import для модуля, написанного вами. В программах с несколькими командами import выполнение этого соглашения поможет понять, откуда берутся разные модули, использованные в программе.

# Глава 9. Файлы и исключения

*Исключения* — специальные объекты, которые создаются для управления ошибками, возникающими во время выполнения программ Python. Также будет описан модуль json, позволяющий сохранять пользовательские данные, чтобы они не терялись по завершении работы программы.

Работа с файлами и сохранение данных упрощают использование ваших программ. Пользователь сам выбирает, какие данные и когда нужно вводить. Он может запустить вашу программу, выполнить некоторую работу, потом закрыть программу и позднее продолжить работу с того момента, на котором он прервался. Умение обрабатывать исключения поможет справиться с такими ситуациями, как отсутствие нужных файлов, а также другими проблемами, приводящими к сбою программ.

Обработка исключений повысит устойчивость ваших программ при работе с некорректными данными — появившимися как из-за случайных ошибок, так и в результате злонамеренных попыток взлома ваших программ.

## Чтение из файла

Гигантские объемы данных доступны в текстовых файлах. В них могут храниться погодные данные, социально-экономическая информация, литературные произведения и многое другое. Чтение из файла особенно актуально для приложений, предназначенных для анализа данных, но оно также может пригодиться в любой ситуации, требующей анализа или изменения информации, хранящейся в файле. Например, программа может читать содержимое текстового файла и переписывать его с форматированием, рассчитанным на отображение информации в браузере.

Работа с информацией в текстовом файле начинается с чтения данных в память.

Вы можете прочитать все содержимое файла или же читать данные по строкам.

## Чтение всего файла

Для начала нам понадобится файл с несколькими строками текста. Пусть это будет файл с числом «пи» с точностью до 30 знаков, по 10 знаков на строку:

***pi_digits.txt***

```
3.1415926535
  8979323846
  2643383279
```

Чтобы опробовать эти примеры, либо введите данные в редакторе и сохраните файл с именем pi_digits.txt.

Следующая программа открывает этот файл, читает его и выводит содержимое на экран:

***file_reader.py***

```
with open('pi_digits.txt') as file_object:
        contents = file_object.read()

print(contents)
```

Начнем с функции open(). Чтобы выполнить любые операции с файлом — даже просто вывести его содержимое, — сначала необходимо *открыть* файл. Функция open() получает один аргумент: имя открываемого файла. Python ищет файл с указанным именем в каталоге, в

котором находится файл текущей программы. В данном примере выполняется программа file_reader.py, поэтому Python ищет файл pi_digits.txt в каталоге, в котором хранится file_reader.py. Функция open() возвращает объект, представляющий файл. В данном случае open('pi_digits.txt') возвращает объект, представляющий файл pi_digits.txt. Python сохраняет этот объект в переменной file_object, с которой мы будем работать позднее в программе.

Конструкция с ключевым словом with закрывает файл после того, как надобность в нем отпадет. Обратите внимание: в этой программе есть вызов open(), но нет вызова close(). Файлы можно открывать и закрывать явными вызовами open() и close(); но если из-за ошибки в программе команда close() останется невыполненной, то файл не будет закрыт. На первый взгляд это не страшно, но некорректное закрытие файлов может привести к потере или порче данных. А если функция close() будет вызвана слишком рано, программа попытается работать с *закрытым* (то есть недоступным) файлом, что приведет к новым ошибкам. Не всегда можно заранее определить, когда нужно закрывать файл, но с приведенной конструкцией Python сделает это за вас. Вам остается лишь открыть файл и работать с ним так, как требуется, надеясь на то, что Python закроет его автоматически при завершении блока with.

После того как в программе появится объект, представляющий файл pi_digits.txt, во второй строке программы используется метод read(), который читает все содержимое файла и сохраняет его в одной длинной строке в переменной contents. При выводе значения contents на экране появляется все содержимое файла :

```
3.1415926535
  8979323846
  2643383279
```

Единственное различие между выводом и исходным файлом — лишняя пустая строка в конце вывода. Откуда она взялась? Метод read() возвращает ее при чтении, если достигнут конец файла. Если вы хотите удалить лишнюю пустую строку, включите вызов rstrip() в вызов print():

```
with open('pi_digits.txt') as file_object:

    contents = file_object.read()

    print(contents.rstrip())
```

Напомним, что метод rstrip() удаляет все пропуски в конце строки. Теперь вывод точно соответствует содержимому исходного файла:

```
3.1415926535
  8979323846
  2643383279
```

## Пути к файлам

Если передать функции open() простое имя файла, такое как pi_digits.txt, Python ищет файл в том каталоге, в котором находится файл, выполняемый в настоящий момент (то есть файл программы.py).

В некоторых случаях (в зависимости от того, как организованы ваши рабочие файлы) открываемый файл может и не находиться в одном каталоге с файлом программы. Например, файл программы может находиться в каталоге python_work; в каталоге python_work создается другой каталог с именем text_files для текстовых файлов, с которыми работает программа. И хотя папка text_files находится в python_work, простая передача open() имени файла из text_files не подойдет, потому что Python произведет поиск файла в python_work и на этом остановится; поиск не будет продолжен во вложенном каталоге text_files. Чтобы открыть файлы из каталога, отличного от того, в котором хранится файл программы, необходимо указать *путь* — то есть приказать Python искать файлы в конкретном месте файловой системы.

Так как каталог text_files находится в python_work, для открытия файла из text_files можно воспользоваться *относительным* путем. Относительный путь приказывает Python искать файлы в каталоге, который задается *относительно* каталога, в котором находится текущий файл программы. Например, это может выглядеть так:

*with open('text_files/имя_файла.txt') as file_object:*

Эта строка означает, что файл .txt следует искать в каталоге text_files; она предполагает, что каталог text_files находится в python_work.

Также можно точно определить местонахождение файла в вашей системе независимо от того, где хранится выполняемая программа. Такие пути называются *абсолютными* и используются в том случае, если относительный путь не работает. Например, если каталог text_files находится не в python_work, а в другом каталоге (скажем, в каталоге с именем other_files), то передать open() путь 'text_files/ filename.txt' не получится, потому что Python будет искать указанный каталог только внутри python_work. Чтобы объяснить Python, где следует искать файл, необходимо записать полный путь.

Абсолютные пути обычно длиннее относительных, поэтому их лучше сохранять в переменных, которые затем передаются open():

*file_path = '/home/ehmatthes/other_files/text_files/имя_файла.txt'*

*with open(file_path) as file_object:*

С абсолютными путями вы сможете читать файлы из любого каталога вашей системы. Пока будет проще хранить файлы в одном каталоге с файлами программ или в каталогах, вложенных в каталог с файлами программ (таких, как text_files из рассмотренного примера).

## Чтение по строкам

В процессе чтения файла часто бывает нужно обработать каждую строку. Возможно, вы ищете некую информацию в файле или собираетесь каким-то образом изменить текст — например, при чтении файла с метеорологическими данными вы обрабатываете каждую строку, в которой в описании погоды встречается слово «солнечно». Или, допустим, в новостях вы ищете каждую строку с тегом заголовка и заменяете ее специальными элементами форматирования.

Для последовательной обработки каждой строки в файле можно воспользоваться циклом for:

***file_reader.py***

```
filename = 'pi_digits.txt'

with open(filename) as file_object:

    for line in file_object:

        print(line)
```

Имя файла, из которого читается информация, сохраняется в переменной filename. Это стандартный прием при работе с файлами: так как переменная filename не представляет конкретный файл (это всего лишь строка, которая сообщает Python, где найти файл), вы сможете легко заменить 'pi_digits.txt' именем другого файла, с которым вы собираетесь работать. После вызова open() объект, представляющий файл и его содержимое, сохраняется в переменной file_object. Мы снова используем синтаксис with, чтобы поручить Python открывать и закрывать файл в нужный момент. Для просмотра содержимого все строки файла перебираются в цикле for по объекту файла.

На этот раз пустых строк оказывается еще больше:

*3.1415926535*
*  8979323846*
*  2643383279*

Пустые строки появляются из-за того, что каждая строка в текстовом файле завершается невидимым символом новой строки. Команда print добавляет свой символ новой строки при каждом вызове, поэтому в результате каждая строка завершается *двумя* символами новой строки: один прочитан из файла, а другой добавлен командой print. Вызов rstrip() в команде print удаляет лишние пустые строки:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:

        for line in file_object:

                print(line.rstrip())
```

Теперь вывод снова соответствует содержимому файла:

*3.1415926535*
*  8979323846*
*  2643383279*

## Создание списка строк по содержимому файла

При использовании with объект файла, возвращаемый вызовом open(), доступен только в пределах содержащего его блока with. Если вы хотите, чтобы содержимое файла оставалось доступным за пределами блока with, сохраните строки файла в списке внутри блока и в дальнейшем работайте с полученным списком. Одни части файла можно обработать немедленно, а другие отложить для обработки в будущем.

В следующем примере строки pi_digits.txt сохраняются в списке в блоке with, после чего выводятся за пределами этого блока:

```
filename = 'pi_digits.txt'

with open(filename) as file_object:

        lines = file_object.readlines()

for line in lines:

        print(line.rstrip())
```

В точке метод readlines() последовательно читает каждую строку из файла и сохраняет ее в списке. Список сохраняется в переменной lines, с которой можно продолжить работу после завершения блока with. В точке в простом цикле for выводятся все элементы списка lines. Так как каждый элемент lines соответствует ровно одной строке файла, вывод точно соответствует его содержимому.

## Работа с содержимым файла

После того как файл будет прочитан в память, вы сможете обрабатывать данные так, как считаете нужным. Для начала попробуем построить одну строку со всеми цифрами из файла без промежуточных пропусков:

***pi_string.py***

```
filename = 'pi_digits.txt'

with open(filename) as file_object:

        lines = file_object.readlines()

pi_string = ' '

for line in lines:

    pi_string += line.rstrip()

print(pi_string)

print(len(pi_string))
```

Сначала программа открывает файл и сохраняет каждую строку цифр в списке — точно так же, как это делалось в предыдущем примере. Создается переменная pi_string для хранения цифр числа «пи». Далее следует цикл, который добавляет к pi_string каждую серию цифр, из которой удаляется символ новой строки. Затем программа выводит строку и ее длину:

```
3.1415926535  8979323846  2643383279

36
```

Переменная pi_string содержит пропуски, которые присутствовали в начале каждой строки цифр. Чтобы удалить их, достаточно использовать strip() вместо rstrip():

```
…

for line in lines:

        pi_string += line.strip()

print(pi_string)

print(len(pi_string))
```

В итоге мы получаем строку, содержащую значение «пи» с точностью до 30 знаков. Длина строки равна 32 символам, потому что в нее также включается начальная цифра 3 и точка:

```
3.14159265358979323846264338327 9

32
```

Читая данные из текстового файла, Python интерпретирует весь текст в файле как строку. Если вы читаете из текстового файла число и хотите работать с ним в числовом контексте, преобразуйте его в целое число функцией int() или в вещественное число функцией float() .

## Большие файлы: миллион цифр

До настоящего момента мы ограничивались анализом текстового файла, который состоял всего из трех строк, но код этих примеров будет работать и с много большими файлами. Начиная с текстового файла, содержащего значение «пи» до 1 000 000 знаков (вместо 30), вы сможете создать одну строку, которая содержит все эти цифры. Изменять программу вообще не придется — достаточно передать ей другой файл. Также мы ограничимся выводом первых 50 цифр, чтобы не пришлось ждать, пока в терминале не прокрутится миллион знаков:

***pi_string.py***

```
filename = 'pi_million_digits.txt'

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ' '

for line in lines:
    pi_string += line.strip()

print(f"{pi_string[:52]}...")

print(len(pi_string))
```

Из выходных данных видно, что строка действительно содержит значение «пи» с точностью до 1 000 000 знаков:

*3.14159265358979323846264338327950288419716939937510...1000002*

Python не устанавливает никаких ограничений на длину данных, с которыми вы можете работать. Она ограничивается разве что объемом памяти вашей системы.

## Проверка дня рождения

Меня всегда интересовало, не встречается ли мой день рождения среди цифр числа «пи»? Воспользуемся только что созданной программой для проверки того, входит ли запись дня рождения пользователя в первый миллион цифр. Для этого можно записать день рождения в виде строки из цифр и посмотреть, присутствует ли эта строка в pi_string:

```
…

for line in lines:
    pi_string += line.strip()

birthday = input("Enter your birthday, in the form mmddyy: ")

if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")

else:
    print("Your birthday does not appear in the first million digits of pi.")
```

Программа запрашивает день рождения пользователя, а затем в точке проверяет вхождение этой строки в pi_string. Пробуем:

Enter your birthdate, in the form mmddyy: **120372**

Your birthday appears in the first million digits of pi!

Оказывается, мой день рождения встречается среди цифр «пи»! После того, как данные будут прочитаны из файла, вы сможете делать с ними все, что сочтете нужным.

## Запись в файл

Один из простейших способов сохранения данных — запись в файл. Текст, записанный в файл, останется доступным и после закрытия терминала с выводом вашей программы. Вы сможете проанализировать результаты после завершения программы или передать свои файлы другим. Вы также сможете написать программы, которые снова читают сохраненный текст в память и работают с ним.

## Запись в пустой файл

Чтобы записать текст в файл, необходимо вызвать open() со вторым аргументом, который сообщает Python, что вы собираетесь записывать данные в файл. Чтобы увидеть, как это делается, напишем простое сообщение и сохраним его в файле (вместо того, чтобы просто вывести на экран):

***write_message.py***

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:

    file_object.write("I love programming.")
```

При вызове open() в этом примере передаются два аргумента. Первый аргумент, как и прежде, содержит имя открываемого файла. Второй аргумент 'w' сообщает Python, что файл должен быть открыт в режиме *записи*. Файлы можно открывать в режиме *чтения* ('r'), *записи* ('w'), *присоединения* ('a') или в режиме, допускающем *как чтение, так и запись* в файл ('r+'). Если аргумент режима не указан, Python по умолчанию открывает файл в режиме только для чтения.

Если файл, открываемый для записи, еще не существует, функция open() автоматически создает его. Будьте внимательны, открывая файл в режиме записи ('w'): если файл существует, то Python уничтожит его данные перед возвращением объекта файла.

В точке метод write() используется с объектом файла для записи строки в файл. Программа не выводит данные на терминал, но открыв файл programming.txt, вы увидите в нем одну строку:

***programming.txt***

```
I love programming.
```

Этот файл ничем не отличается от любого другого текстового файла на вашем компьютере. Его можно открыть, записать в него новый текст, скопировать/вставить текст и т.д.

Python может записывать в текстовые файлы только строковые данные. Если вы захотите сохранить в текстовом файле числовую информацию, данные придется предварительно преобразовать в строки функцией str() .

## Многострочная запись

Функция write() не добавляет символы новой строки в записываемый текст. А это означает, что если вы записываете сразу несколько строк без включения символов новой строки, полученный файл может выглядеть не так, как вы рассчитывали:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:

    file_object.write("I love programming.")
```

```
    file_object.write("I love creating new games.")
```

Открыв файл programming.txt, вы увидите, что две строки «склеились»:

*I love programming.I love creating new games.*

Если включить символы новой строки в команды write(), текст будет состоять из двух строк:

```
filename = 'programming.txt'

with open(filename, 'w') as file_object:

    file_object.write("I love programming.\n")

    file_object.write("I love creating new games.\n")
```

Результат выглядит так:

*I love programming.*
*I love creating new games.*

Для форматирования вывода также можно использовать пробелы, символы табуляции и пустые строки по аналогии с тем, как это делалось с выводом на терминал.

## Присоединение данных к файлу

Если вы хотите добавить в файл новые данные вместо того, чтобы перезаписывать существующее содержимое, откройте файл в режиме присоединения. В этом слу- чае Python не уничтожает содержимое файла перед возвращением объекта файла. Все строки, выводимые в файл, будут добавлены в конец файла. Если файл еще не существует, то Python автоматически создаст пустой файл.

Изменим программу write_message .py и дополним существующий файл programming . txt новыми данными:

***write_message.py***

```
filename = 'programming.txt'

with open(filename, 'a') as file_object:

    file_object.write("I also love finding meaning in large datasets.\n")

    file_object.write("I love creating apps that can run in a browser.\n")
```

Аргумент 'a' используется для открытия файла в режиме присоединения (вместо перезаписи существующего файла). Затем записываются две новые строки, которые добавляются к содержимому programming.txt:

***programming.txt***

*I love programming.*
*I love creating new games.*
*I also love finding meaning in large datasets.*
*I love creating apps that can run in a browser.*

В результате к исходному содержимому файла добавляется новый текст.

## Исключения

Для управления ошибками, возникающими в ходе выполнения программы, в Python используются специальные объекты, называемые *исключениями*. Если при возникновении ошибки Python не знает, что делать дальше, создается объект исключения. Если в программу включен код обработки исключения, то выполнение программы продолжится, а если нет — программа останавливается и выводит *трассировку* с отчетом об исключении.

Исключения обрабатываются в блоках try-except. Блок try-except приказывает Python выполнить некоторые действия, но при этом также сообщает, что делать при возникновении исключения. С блоками try-except ваши программы будут работать даже в том случае, если что-то пошло не так. Вместо невразумительной трассировки выводится понятное сообщение об ошибке, которое вы определяете в программе.

Обработка исключения ZeroDivisionError

Рассмотрим простую ошибку, при которой Python инициирует исключение. Конечно, вы знаете, что деление на ноль невозможно, но мы все же прикажем Python выполнить эту операцию:

***division_calculator.py***

*print(5/0)*

*Traceback (most recent call last):*
*File "division.py", line 1, in <module>*
*print(5/0)*

*ZeroDivisionError: division by zero*

Конечно, из этого ничего не выйдет, поэтому на экран выводятся данные трассировки.

Ошибка, упоминаемая в трассировке, — ZeroDivisionError — является объектом исключения. Такие объекты создаются в том случае, если Python не может выполнить ваши распоряжения. Обычно в таких случаях Python прерывает выполнение программы и сообщает тип обнаруженного исключения. Эта информация может использоваться в программе; по сути, вы сообщаете Python, как следует поступить при возникновении исключения данного типа. В таком случае ваша программа будет подготовлена к его появлению.

## Блоки try-except

Если вы предполагаете, что в программе может произойти ошибка, напишите блок try-except для обработки возникающего исключения. Такой блок приказывает Python выполнить некоторый код, а также сообщает, что нужно делать, если при его выполнении произойдет исключение конкретного типа.

Вот как выглядит блок try-except для обработки исключений ZeroDivisionError:

*try:*

    *print(5/0)*

*except ZeroDivisionError:*

    *print("You can't divide by zero!")*

Команда print(5/0), порождающая ошибку, находится в блоке try. Если код в блоке try выполнен успешно, то Python пропускает блок except. Если код в блоке try порождает ошибку, то Python ищет блок except с соответствующей ошибкой и выпускает код в этом блоке.

В этом примере код блока try порождает ошибку ZeroDivisionError, поэтому Python ищет блок except с описанием того, как следует действовать в такой ситуации. При выполнении кода этого блока пользователь видит понятное сообщение об ошибке вместо данных трассировки:

*You can't divide by zero!*

Если за кодом try-except следует другой код, то выполнение программы продолжится, потому что мы объяснили Python, как обрабатывать эту ошибку. В следующем примере обработка ошибки позволяет программе продолжить выполнение.

## Использование исключений для предотвращения аварийного завершения программы

Правильная обработка ошибок особенно важна в том случае, если программа должна продолжить работу после возникновения ошибки. Такая ситуация часто встречается в программах, запрашивающих данные у пользователя. Если программа правильно среагировала на некорректный ввод, она может запросить новые данные после сбоя.

Создадим простой калькулятор, который выполняет только операцию деления:

***division_calculator.py***

```
print("Give me two numbers, and I'll divide them.")

print("Enter 'q' to quit.")

while True:

    first_number = input("\nFirst number: ")

    if first_number == 'q':

        break

    second_number = input("Second number: ")

    if second_number == 'q':

        break

    answer = int(first_number) / int(second_number)

    print(answer)
```

Программа запрашивает у пользователя первое число, first_number, а затем, если пользователь не ввел q для завершения работы, — второе число, second_number. Далее одно число делится на другое для получения результата answer. Программа никак не пытается обрабатывать ошибки, так что попытка деления на ноль приводит к ее аварийному завершению:

```
Give me two numbers, and I'll divide them.
Enter 'q' to quit.

First number: 5
Second number: 0
Traceback (most recent call last):
    File "division.py", line 9, in <module>
        answer = int(first_number) / int(second_number)
ZeroDivisionError: division by zero
```

Конечно, аварийное завершение — это плохо, но еще хуже, что пользователь увидит данные трассировки. Неопытного пользователя они собьют с толку, а при сознательной попытке взлома злоумышленник сможет получить из них больше информации, чем вам хотелось бы. Например, он узнает имя файла программы и увидит некорректно работающую часть кода. На основании этой информации опытный хакер иногда может определить, какие атаки следует применять против вашего кода.

## Блок else

Для повышения устойчивости программы к ошибкам можно заключить строку, выдающую ошибки, в блок try-except. Ошибка происходит в строке, выполняющей деление; следовательно, именно эту строку следует заключить в блок try-except. Данный пример также включает блок else. Любой код, зависящий от успешного выполнения блока try, размещается в блоке else:

```
...

while True:

...

        if second_number == 'q':

                break

        try:

                answer = int(first_number) / int(second_number)

        except ZeroDivisionError: print("You can't divide by 0!")

        else:

                print(answer)
```

Программа пытается выполнить операцию деления в блоке try, который включает только код, способный породить ошибку. Любой код, зависящий от успешного выполнения блока try, добавляется в блок else. В данном случае, если операция деления выполняется успешно, блок else используется для вывода результата .

Блок except сообщает Python, как следует поступать при возникновении ошибки ZeroDivisionError. Если при выполнении команды из блока try происходит ошибка, связанная с делением на 0, программа выводит понятное сообщение, которое объясняет пользователю, как избежать подобных ошибок. Выполнение программы продолжается, и пользователь не сталкивается с трассировкой:

```
Give me two numbers, and I'll divide them.

Enter 'q' to quit.

First number: 5

Second number: 0

You can't divide by 0! First number: 5

Second number: 2

2.5

First number: q
```

Блок try-except-else работает так: Python пытается выполнить код в блоке try. В блоках try следует размещать только тот код, при выполнении которого может возникнуть исключение. Иногда некоторый код должен выполняться только в том случае, если выполнение try прошло успешно; такой код размещается в блоке else. Блок except сообщает Python, что делать, если при выполнении кода try произошло определенное исключение.

## Обработка исключения FileNotFoundError

Одна из стандартных проблем при работе с файлами — отсутствие необходимых файлов. Тот файл, который вам нужен, может находиться в другом месте, в имени файла может быть допущена ошибка или файл может вообще не существовать. Все эти ситуации достаточно прямолинейно обрабатываются в блоках try-except.

Попробуем прочитать данные из несуществующего файла. Следующая программа пытается прочитать содержимое файла с текстом «Алисы в Стране чудес», но я не сохранил файл alice.txt в одном каталоге с файлом alice.py:

***alice.py***

```
filename = 'alice.txt'

with open(filename, encoding='utf-8') as f:

    contents = f.read()
```

В программе видны два изменения. Во-первых, объект файла представляется переменной с именем f — это общепринятое соглашение. Во-вторых, в программе используется аргумент encoding. Он необходим в тех случаях, когда кодировка вашей системы по умолчанию не совпадает с кодировкой читаемого файла.

Прочитать данные из несуществующего файла нельзя, поэтому Python выдает исключение:

```
Traceback (most recent call last):
File "alice.py", line 3, in <module>
    with open(filename, encoding='utf-8') as f:
    FileNotFoundError: [Errno 2] No such file or directory: 'alice.txt'
```

В последней строке трассировки упоминается FileNotFoundError: это исключение выдается в том случае, если Python не может найти открываемый файл. В данном примере функция open() порождает ошибку, и чтобы обработать ее, блок try начинается перед строкой с вызовом open():

```
filename = 'alice.txt'

try:

    with open(filename, encoding='utf-8') as f:

        contents = f.read()

except FileNotFoundError:

    print(f"Sorry, the file {filename} does not exist.")
```

В этом примере код блока try выдает исключение FileNotFoundError, поэтому Python ищет блок except для этой ошибки. Затем выполняется код этого блока, в результате чего вместо трассировки выдается доступное сообщение об ошибке:

```
Sorry, the file alice.txt does not exist.
```

Если файл не существует, программе больше делать нечего, поэтому код обработки ошибок почти ничего не добавляет в эту программу. Доведем до ума этот пример и посмотрим, как обработка исключений помогает при работе с несколькими файлами.

## Анализ текста

Программа может анализировать текстовые файлы, содержащие целые книги. Многие классические произведения, ставшие общественным достоянием, доступны в виде простых текстовых файлов.

Прочитаем текст «Алисы в Стране чудес» и попробуем подсчитать количество слов в тексте. Мы воспользуемся методом split(), предназначенным для построения списка слов на основе строки. Вот как метод split() работает со строкой, содержащей только название книги:

```
title = "Alice in Wonderland"

title.split()

['Alice', 'in', 'Wonderland']
```

Метод split() разделяет строку на части по всем позициям, в которых обнаружит пробел, и сохраняет все части строки в элементах списка. В результате создается список слов, входящих в строку (впрочем, вместе с некоторыми словами могут храниться знаки препинания.) Для подсчета слов в книге мы вызовем split() для всего текста, а затем подсчитаем элементы списка, чтобы получить примерное количество слов в тексте:

```
filename = 'alice.txt'

try:
        with open(filename, encoding='utf-8') as f:
                contents = f.read()
except FileNotFoundError:
        print(f"Sorry, the file {filename} does not exist.")
else:
        #Подсчет приблизительного количества строк в файле.
        words = contents.split()
        num_words = len(words)
        print(f"The file {filename} has about {num_words} words.")
```

Затем файл alice.txt нужно переместить в правильный каталог, чтобы код в блоке try был выполнен без ошибок. Программа загружает текст в переменную contents, которая теперь содержит весь текст в виде одной длинной строки и использует метод split() для получения списка всех слов в книге. Запрашивая длину этого списка при помощи функции len(), мы получаем неплохое приближенное значение количества слов в исходной строке. Выводится сообщение с количеством слов, найденных в файле. Этот код помещен в блок else, потому что он должен выводиться только в случае успешного выполнения блока try. Выходные данные программы сообщают, сколько слов содержит файл alice.txt:

```
The file alice.txt has about 29465 words.
```

Количество слов немного завышено, потому что в нем учитывается дополнительная информация, включенная в текстовый файл издателем, но в целом оно довольно точно оценивает объем «Алисы в Стране чудес».

## Работа с несколькими файлами

Добавим еще несколько файлов с книгами для анализа. Но для начала переместим основной код программы в функцию с именем count_words(). Это упростит проведение анализа для нескольких книг:

***word_count.py***

```
def count_words(filename):
        """Подсчет приблизительного количества строк в файле."""
        try:
                with open(filename,encoding='utf-8') as f:
                        contents = f.read()
        except FileNotFoundError:
                print(f"Sorry, the file {filename} does not exist.")
        else:
                words = contents.split()
                num_words = len(words)
                print(f"The file {filename} has about {num_words} words.")
filename = 'alice.txt'
count_words(filename)
```

Большая часть кода не изменилась. Мы просто снабдили код отступом и переместили его в тело count_words(). При внесении изменений в программу желательно обновлять комментарии, поэтому мы преобразовали комментарий в строку документации и слегка переформулировали его.

Теперь мы можем написать простой цикл для подсчета слов в любом тексте, который нужно проанализировать. Для этого имена анализируемых файлов сохраняются в списке, после чего для каждого файла в списке вызывается функция count_words(). Мы попробуем подсчитать слова в «Алисе в Стране чудес», «Сидд- хартхе», «Моби Дике» и «Маленьких женщинах».

```
def count_words(filename):
...
filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt',
             'little_women.txt']
for filename in filenames:
        count_words(filename)
```

Отсутствие файла siddhartha .txt не влияет на выполнение программы:

*The file alice.txt has about 29465 words.*
*Sorry, the file siddhartha.txt does not exist.*
*The file moby_dick.txt has about 215830 words.*
*The file little_women.txt has about 189079 words.*

Использование блока try-except в данном примере предоставляет два важных преимущества: программа ограждает пользователя от вывода трассировки и продолжает выполнение, анализируя тексты, которые ей удается найти. Если бы в программе не перехватывалось исключение FileNotFoundError, инициированное из-за отсутствия siddhartha.txt, то пользователь увидел бы полную трассировку, а работа программы прервалась бы после попытки подсчитать слова в тексте «Сиддхартхи»; до анализа «Моби Дика» или «Маленьких женщин» дело не дошло бы.

## Ошибки без уведомления пользователя

В предыдущем примере мы сообщили пользователю о том, что один из файлов оказался недоступен. Тем не менее вы не обязаны сообщать о каждом обнаруженном исключении. Иногда при возникновении исключения программа должна просто проигнорировать сбой и продолжать работу, словно ничего не произошло. Для этого блок try пишется так же, как обычно, но в блоке except вы явно приказываете Python не предпринимать никаких особых действий в случае ошибки. В языке Python существует команда pass, с которой блок ничего не делает:

```
def count_words(filename):
        """Подсчет приблизительного количества строк в файле."""
        try: ...
        except FileNotFoundError:
                pass
        else: ...

filenames = ['alice.txt', 'siddhartha.txt', 'moby_dick.txt', 'little_women.txt']

for filename in filenames:
        count_words(filename)
```

Единственное отличие этого листинга от предыдущего — команда pass в точке. Теперь при возникновении ошибки FileNotFoundError выполняется код в блоке except, но при этом ничего не происходит. Программа не выдает данных трассировки и вообще никаких результатов, указывающих на возникновение ошибки.

Пользователи получают данные о количестве слов во всех существующих файлах, однако ничто не сообщает о том, что какой-то файл не был найден:

*The file alice.txt has about 29465 words.*
*The file moby_dick.txt has about 215830 words.*
*The file little_women.txt has about 189079 words.*

Команда pass также может служить временным заполнителем. Она напоминает, что в этот конкретный момент выполнения вашей программы вы решили ничего не предпринимать, хотя, возможно, решите сделать что-то позднее. Например, эта программа может записать все имена отсутствующих файлов в файл с именем missing_files.txt. Пользователи этот файл не увидят, но создатель программы сможет прочитать его и разобраться с отсутствующими текстами.

## О каких ошибках нужно сообщать?

Как определить, в каком случае следует сообщить об ошибке пользователю, а когда можно просто проигнорировать ее незаметно для пользователя? Если пользователь знает, с какими текстами должна работать программа, вероятно, он предпочтет получить сообщение, объясняющее, почему некоторые тексты были пропущены при анализе. Пользователь ожидает увидеть какие-то результаты, но не знает, какие книги должны быть проанализированы. Возможно, ему и не нужно знать о недоступности каких-то файлов. Лишняя информация только сделает вашу программу менее удобной для пользователя. Средства обработки ошибок Python позволяют достаточно точно управлять тем, какой объем информации следует предоставить пользователю.

Хорошо написанный, правильно протестированный код редко содержит внутренние ошибки (например, синтаксические или логические). Но в любой ситуации, в которой ваша программа зависит от внешних факторов (пользовательского ввода, существования файла, доступности сетевого подключения), существует риск возникновения исключения. С накоплением практического опыта вы начнете видеть, в каких местах программы следует разместить блоки обработки исключений и сколько информации предоставлять пользователям о возникающих ошибках.

## Сохранение данных

Многие ваши программы будут запрашивать у пользователя информацию. Например, пользователь может вводить настройки для компьютерной игры или данные для визуального представления. Чем бы ни занималась ваша программа, информация, введенная пользователем, будет сохраняться в структурах данных (таких, как списки или словари). Когда пользователь закрывает программу, введенную им информацию почти всегда следует сохранять на будущее. Простейший способ сохранения данных основан на использовании модуля json.

Модуль json обеспечивает запись простых структур данных Python в файл и загрузку данных из файла при следующем запуске программы. Модуль json также может использоваться для обмена данными между программами Python. Более того, формат данных JSON не привязан к Python, поэтому данные в этом формате можно передавать программам, написанным на многих других языках программирования. Это полезный и универсальный формат, который к тому же легко изучается.

Формат JSON (JavaScript Object Notation) был изначально разработан для JavaScript. Впрочем, с того времени он стал использоваться во многих языках, включая Python .

## Функции json .dump() и json .load()

Напишем короткую программу для сохранения набора чисел и другую программу, которая будет читать эти числа обратно в память. Первая программа использует функцию json.dump(), а вторая — функцию json.load().

Функция json.dump() получает два аргумента: сохраняемые данные и объект файла, используемый для сохранения. В следующем примере json.dump() используется для сохранения списка чисел:

***number_writer.py***

```
import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'

with open(filename, 'w') as f:
```

```
    json.dump(numbers, f)
```

Программа импортирует модуль json и создает список чисел для работы. В точке выбирается имя файла для хранения списка. Обычно для таких файлов принято использовать расширение .json, указывающее, что данные в файле хранятся в формате JSON. Затем файл открывается в режиме записи, чтобы модуль json мог записать в него данные. В точке функция json.dump() используется для сохранения списка numbers в файле numbers.json.

Программа ничего не выводит, но давайте откроем файл numbers .json и посмотрим на его содержимое. Данные хранятся в формате, очень похожем на код Python:

```
[2, 3, 5, 7, 11, 13]
```

А теперь напишем следующую программу, которая использует json.load() для чтения списка обратно в память:

***number_reader.py***

```
import json

filename = 'numbers.json'

with open(filename) as f:

    numbers = json.load(f)

 print(numbers)
```

Для чтения данных используется тот же файл, в который эти данные были записаны. На этот раз файл открывается в режиме чтения, потому что Python нужно только прочитать данные из файла. В точке функция json.load() используется для загрузки информации из numbers.json; эта информация сохраняется в переменной numbers. Наконец, программа выводит прочитанный список. Как видите, это тот же список, который был создан в программе number_writer.py:

```
[2, 3, 5, 7, 11, 13]
```

Модуль json позволяет организовать простейший обмен данными между программами.

## Сохранение и чтение данных, сгенерированных пользователем

Сохранение с использованием модуля json особенно полезно при работе с данными, сгенерированными пользователем, потому что без сохранения эта информация будет потеряна при остановке программы. В следующем примере программа запрашивает у пользователя имя при первом запуске программы и «вспоминает» его при повторных запусках.

Начнем с сохранения имени пользователя:

***remember_me.py***

```
import json

username = input("What is your name? ")

filename = 'username.json'

with open(filename, 'w') as f:

    json.dump(username, f)

    print(f"We'll remember you when you come back, {username}!")
```

Программа запрашивает имя пользователя для сохранения. Затем вызывается функция json.dump(), которой передается имя пользователя и объект файла; функция сохраняет имя пользователя в файле. Далее выводится сообщение о том, что имя пользователя было сохранено:

*What is your name?* ***Eric***

*We'll remember you when you come back, Eric!*

А теперь напишем другую программу, которая приветствует пользователя по ранее сохраненному имени:

***greet_user.py***

```
import json

filename = 'username.json'

with open(filename) as f:

        username = json.load(f)

        print(f"Welcome back, {username}!")
```

В точке вызов json.load() читает информацию из файла username .json в переменную username. После того как данные будут успешно прочитаны, мы можем поприветствовать пользователя по имени:

*Welcome back, Eric!*

Теперь эти две программы необходимо объединить в один файл. Когда пользователь запускает remember_me.py, программа должна взять имя пользователя из памяти, если это возможно; соответственно, программа начинается с блока try, который пытается прочитать имя пользователя. Если файл username.json не существует, блок except запросит имя пользователя и сохранит его в username.json на будущее:

***remember_me.py***

```
import json

#Программа загружает имя пользователя, если оно было сохранено ранее.
# В противном случае она запрашивает имя пользователя и сохраняет его.

filename = 'username.json'

try:

        with open(filename) as f:

                username = json.load(f)

except FileNotFoundError:

        username = input("What is your name? ")

        with open(filename, 'w') as f:

                json.dump(username, f)

                print(f"We'll remember you when you come back, {username}!")
```

```
else:
    print(f"Welcome back, {username}!")
```

Никакого нового кода здесь нет; просто блоки кода из двух предыдущих примеров были объединены в один файл. Программа пытается открыть файл username.json. Если файл существует, программа читает имя пользователя в память и выводит сообщение, приветствующее пользователя, в блоке else. Если программа запускается впервые, то файл username.json не существует и происходит исключение FileNotFoundError. Python переходит к блоку except, в котором пользователю предлагается ввести имя. Затем программа вызывает json.dump() для сохранения имени пользователя и выводит приветствие.

Какой бы блок ни выполнялся, результатом является имя пользователя и соответствующее сообщение. При первом запуске программы результат выглядит так:

*What is your name?* ***Eric***

*We'll remember you when you come back, Eric!*

Если же программа уже была выполнена хотя бы один раз, то результат будет таким:

*Welcome back, Eric!*

## Рефакторинг

Код работает, но вы понимаете, что его структуру можно усовершенствовать, разбив его на функции, каждая из которых решает свою конкретную задачу. Этот процесс называется *рефакторингом* (или переработкой). Рефакторинг делает ваш код более чистым, понятным и простым в расширении.

В процессе рефакторинга remember_me.py мы можем переместить основную часть логики в одну или несколько функций. Основной задачей remember_me.py является вывод приветствия для пользователя, поэтому весь существующий код будет пере- мещен в функцию greet_user():

***remember_me.py***

```
import json

def greet_user():
    """Приветствует пользователя по имени."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        username = input("What is your name? ")
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")
```

```
    else:
        print(f"Welcome back, {username}!")
greet_user()
```

С переходом на функцию комментарии дополняются строкой документации, которая описывает работу кода в текущей версии. Код становится немного чище, но функция greet_user() не только приветствует пользователя — она также загружает хранимое имя пользователя, если оно существует, и запрашивает новое имя, если оно не было сохранено ранее.

Переработаем функцию greet_user(), чтобы она не решала столько разных задач. Начнем с перемещения кода загрузки хранимого имени пользователя в отдельную функцию:

```
import json
def get_stored_username():
    """Получает хранимое имя пользователя, если оно существует."""
    filename = 'username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        return None
    else:
        return username
def greet_user():
    """Приветствует пользователя по имени."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = input("What is your name? ")
        filename = 'username.json'
        with open(filename, 'w') as f:
            json.dump(username, f)
            print(f"We'll remember you when you come back, {username}!")
greet_user()
```

Новая функция get_stored_username() имеет четкое предназначение, изложенное в строке документации. Эта функция читает и возвращает сохраненное имя пользователя, если его удается найти. Если файл username.json не существует, то функция возвращает None . И это правильно: функция должна возвращать либо ожидаемое значение, либо None. Это позволяет провести простую проверку возвращаемого значения функции. Программа выводит приветствие для пользователя, если попытка получения имени пользователя была успешной; в противном случае программа запрашивает новое имя пользователя.

Из функции greet_user() стоит вынести еще один блок кода. Если имя пользователя не существует, то код запроса нового имени должен размещаться в функции, специализирующейся на решении этой задачи:

```
import json

def get_stored_username():
    """Получает хранимое имя пользователя, если оно существует."""
    ...
def get_new_username():
    """Запрашивает новое имя пользователя."""
    username = input("What is your name? ")
    filename = 'username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
    return username
def greet_user():
    """Приветствует пользователя по имени."""
    username = get_stored_username()
    if username:
        print(f"Welcome back, {username}!")
    else:
        username = get_new_username()
        print(f"We'll remember you when you come back, {username}!")
greet_user()
```

Каждая функция в окончательной версии remember_me.py имеет четкое, конкретное предназначение. Мы вызываем greet_user(), и эта функция выводит нужное приветствие: либо для уже знакомого, либо для нового пользователя. Для этого функция вызывает функцию get_stored_username(), которая отвечает только за чтение хранимого имени пользователя (если оно есть). Наконец, функция greet_user() при необходимости вызывает функцию get_new_username(), которая отвечает только за получение нового имени пользователя и его сохранение. Такое «разделение обязанностей» является важнейшим аспектом написания чистого кода, простого в сопровождении и расширении.

# Глава 10. Тестирование

Вместе с функциями и классами вы также можете написать тесты для своего кода. Тестирование доказывает, что код работает так, как положено, для любых разновидностей входных данных, которые он может получать. Тесты позволят вам быть уверенными в том, что код будет работать правильно и тогда, когда вашими программами начнут пользоваться другие люди. Тестирование при добавлении нового кода гарантирует, что внесенные изменения не изменят текущее поведение программы. Все программисты допускают ошибки, поэтому каждый программист должен часто тестировать свой код и выявлять ошибки до того, как с ними столкнутся другие пользователи.

В этой главе вы научитесь тестировать код средствами модуля Python unittest. Вы узнаете, как построить тестовые сценарии, как проверить, что для конкретных входных данных программа выдает ожидаемый результат. Вы поймете, как выглядят успешно проходящие или сбойные тесты, и узнаете, как сбойный тест помогает усовершенствовать код. Также вы научитесь тестировать функции и классы и оценивать примерное количество необходимых тестов для проекта.

## Тестирование функции

Чтобы потренироваться в тестировании, нам понадобится код. Ниже приведена простая функция, которая получает имя и фамилию и возвращает отформатированное полное имя:

***name_function.py***

```
def get_formatted_name(first, last):
    """Строит отформатированное полное имя."""
    full_name = f"{first} {last}"
    return full_name.title()
```

Функция get_formatted_name() строит полное имя из имени и фамилии, разделив их пробелом, преобразует первый символ каждого слова к верхнему регистру и возвращает полученный результат. Чтобы убедиться в том, что функция get_formatted_name() работает правильно, мы напишем программу, использующую эту функцию. Программа names.py запрашивает у пользователя имя и фамилию и выдает отформатированное полное имя:

***names.py***

```
from name_function import get_formatted_name

print("Enter 'q' at any time to quit.")

while True:
    first = input("\nPlease give me a first name: ")
    if first == 'q':
      break
    last = input("Please give me a last name: ")
    if last == 'q':
      break
```

```
    formatted_name = get_formatted_name(first, last)

    print(f"\tNeatly formatted name: {formatted_name}.")
```

Программа импортирует функцию get_formatted_name() из модуля name_function.py. Пользователь вводит последовательность имен и фамилий и видит, что программа сгенерировала отформатированные полные имена:

*Enter 'q' at any time to quit.*

*Please give me a first name:* ***janis***

*Please give me a last name:* ***joplin***

*Neatly formatted name: Janis Joplin.*

*Please give me a first name:* ***bob***

*Please give me a last name:* ***dylan***

*Neatly formatted name: Bob Dylan.*

*Please give me a first name:* ***q***

Как видно из листинга, имена сгенерированы правильно. Но допустим, вы решили изменить функцию get_formatted_name(), чтобы она также работала со вторыми именами. При этом необходимо проследить за тем, чтобы функция не перестала правильно работать для имен, состоящих только из имени и фамилии. Чтобы протестировать код, можно запустить names.py и для проверки вводить имя из двух компонентов (скажем, Janis Joplin) при каждом изменении get_formatted_name(), но это довольно утомительно. К счастью, Python предоставляет эффективный механизм автоматизации тестирования вывода функций. При автоматизации тестирования get_formatted_name() вы будете уверены в том, что функция успешно работает для всех видов имен, для которых написаны тесты.

## Модульные тесты и тестовые сценарии

Модуль unittest из стандартной библиотеки Python предоставляет функциональность для тестирования вашего кода. *Модульный тест* проверяет правильность работы одного конкретного аспекта поведения функции. *Тестовый сценарий* представляет собой совокупность модульных тестов, которые совместно доказывают, что функция ведет себя так, как положено, во всем диапазоне ситуаций, которые она должна обрабатывать. Хороший тестовый сценарий учитывает все возможные виды ввода, которые может получать функция, и включает тесты для представления всех таких ситуаций. Тестовый сценарий *с полным покрытием* включает полный спектр модульных тестов, покрывающих все возможные варианты использования функции. Обеспечение полного покрытия для крупного проекта может быть весьма непростой задачей. Часто бывает достаточно написать модульные тесты для критичных аспектов поведения вашего кода, а затем стремиться к полному покрытию только в том случае, если проект перейдет в фазу масштабного использования.

## Прохождение теста

Чтобы написать тестовый сценарий для функции, импортируйте модуль unittest и функцию, которую необходимо протестировать. Затем создайте класс, наследующий от unittest. TestCase, и напишите серию методов для тестирования различных аспектов поведения своей функции.

Ниже приведен тестовый сценарий с одним методом, который проверяет, что функция get_formatted_name() правильно работает при передаче имени и фамилии:

***test_name_function.py***

```
import unittest

from name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):

        """Тесты для 'name_function.py'."""

        def test_first_last_name(self):

        """Имена вида 'Janis Joplin' работают правильно?"""

        formatted_name = get_formatted_name('janis', 'joplin')

        self.assertEqual(formatted_name, 'Janis Joplin')

if __name__ == '__main__':

        unittest.main()
```

Сначала мы импортируем unittest и тестируемую функцию get_formatted_name(). В точке создается класс NamesTestCase, который содержит серию модульных тестов для get_formatted_name(). Имя класса выбирается произвольно, но лучше выбрать имя, связанное с функцией, которую вы собираетесь тестировать, и включить в имя класса слово Test. Этот класс должен наследовать от класса unittest. TestCase, чтобы Python знал, как запустить написанные вами тесты.

Класс NamesTestCase содержит один метод, который тестирует всего один аспект get_formatted_name() — правильность форматирования имен, состоящих только из имени и фамилии. Мы назвали этот метод test_first_last_name(). Любой метод, имя которого начинается с test_, будет выполняться автоматически при запуске test_name_function .py. В тестовом методе вызывается тестируемая функция и сохраняется возвращаемое значение, которое необходимо проверить. В данном примере вызывается функция get_formatted_name() с аргументами 'janis' и 'joplin', а результат сохраняется в переменной formatted_name .

В точке используется одна из самых полезных особенностей unittest: метод assert. Методы assert проверяют, что полученный результат соответствует тому результату, который вы рассчитывали получить. В данном случае известно, что функция get_formatted_name() должна вернуть полное имя с пробелами и капитализацией слов, поэтому переменная formatted_name должна содержать текст «Janis Joplin». Чтобы убедиться в этом, мы используем метод assertEqual() из модуля unittest и передаем ему переменную formatted_name и строку 'Janis Joplin'. Вызов

```
self.assertEqual(formatted_name, 'Janis Joplin')
```

означает: «Сравни значение formatted_name со строкой 'Janis Joplin'. Если они равны, как и ожидалось, — хорошо. Но если они не равны, обязательно сообщи мне!»

Мы запустим этот файл напрямую, но важно заметить, что многие тестовые фреймворки импортируют ваши тестовые файлы перед их выполнением. При импортировании файла интерпретатор выполняет файл в процессе импортирования. Блок if в точке проверяет специальную переменную __name__, значение которой задается при выполнении программы. Если файл выполняется как главная программа, то переменной __name__ будет присвоено значение '__main__'. В этом случае вызывается метод unittest.main(), который выполняет тестовый сценарий. Если файл импортируется тестовым сценарием, то переменная __name__ будет содержать значение '__main__', и этот блок выполняться не будет.

При запуске test_name_function .py будет получен следующий результат:

*.*

*----------------------------------------------------------------------*

*Ran 1 test in 0.000s*

*OK*

Точка в первой строке вывода сообщает, что один тест прошел успешно. Следующая строка говорит, что для выполнения одного теста Python потребовалось менее 0,001 секунды. Наконец, завершающее сообщение OK говорит о том, что прошли все модульные тесты в тестовом сценарии.

Этот результат показывает, что функция get_formatted_name() успешно работает для полных имен, состоящих из имени и фамилии, если только функция не была изменена. В случае внесения изменений в get_formatted_name() тест можно запустить снова. И если тестовый сценарий снова пройдет, мы будем знать, что функция продолжает успешно работать с полными именами из двух компонентов.

## Сбой теста

Как выглядит сбойный тест? Попробуем изменить функцию get_formatted_name(), чтобы она работала со вторыми именами, но сделаем это так, чтобы она перестала работать с полными именами из двух компонентов.

Новая версия get_formatted_name() с дополнительным аргументом второго имени выглядит так:

***name_function.py***

```
def get_formatted_name(first, middle, last):

    """Строит отформатированное полное имя."""

    full_name = f"{first} {middle} {last}"

    return full_name.title()
```

Эта версия должна работать для полных имен из трех компонентов, но тестирова- ние показывает, что она перестала работать для полных имен из двух компонентов. На этот раз файл test_name_function .py выдает следующий результат:

```
E

======================================================================

ERROR: test_first_last_name (__main__.NamesTestCase)

----------------------------------------------------------------------

Traceback (most recent call last):

    File "test_name_function.py", line 8, in test_first_last_name

        formatted_name = get_formatted_name('janis', 'joplin')

TypeError: get_formatted_name() missing 1 required positional argument:
```

*'last'*

*----------------------------------------------------------------------*

*Ran 1 test in 0.000s*

*FAILED (errors=1)*

Теперь информации гораздо больше, потому что при сбое теста разработчик должен знать, почему это произошло. Вывод начинается с одной буквы E, которая сообщает, что один модульный тест в тестовом сценарии привел к ошибке. Затем мы видим, что ошибка произошла в тесте test_first_last_name() в NamesTestCase. Конкретная информация о сбойном тесте особенно важна в том случае, если тестовый сценарий состоит из нескольких модульных тестов. В точке — стандартная трассировка, из которой видно, что вызов функции get_formatted_name('janis', 'joplin') перестал работать из-за необходимого позиционного аргумента.

Также из вывода следует, что был выполнен один модульный тест. Наконец, дополнительное сообщение информирует, что тестовый сценарий в целом не прошел и в ходе выполнения произошла одна ошибка при выполнении тестового сценария. Эта информация размещается в конце вывода, чтобы она была видна сразу; разработчику не придется прокручивать длинный протокол, чтобы узнать количество сбойных тестов.

## Реакция на сбойный тест

Что делать, если тест не проходит? Если предположить, что проверяются правильные условия, прохождение теста означает, что функция работает правильно, а сбой — что в новом коде добавилась ошибка. Итак, если тест не прошел, изменять нужно не тест, а код, который привел к сбою теста. Проанализируйте изменения, внесенные в функцию, и разберитесь, как они привели к нарушению ожидаемого поведения.

В данном случае у функции get_formatted_name() было всего два обязательных параметра: имя и фамилия. Теперь она требует три обязательных параметра: имя, второе имя и фамилию. Добавление обязательного параметра для второго имени нарушило ожидаемое поведение get_formatted_name(). В таком случае лучше всего сделать параметр второго имени необязательным. После этого тесты для имен с двумя компонентами снова будут проходить, и программа сможет получать также вторые имена. Изменим функцию get_formatted_name(), чтобы параметр второго имени перестал быть обязательным, и снова выполним тестовый сценарий. Если он пройдет, можно переходить к проверке правильности обработки вторых имен.

Чтобы сделать второе имя необязательным, нужно переместить параметр middle в конец списка параметров в определении функции и задать ему пустое значение по умолчанию. Также будет добавлена проверка if, которая правильно строит полное имя в зависимости от того, передается второе имя или нет:

***name_function.py***

```
def get_formatted_name(first, last, middle=''):
    """Строит отформатированное полное имя."""
    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"
```

```
    return full_name.title()
```

В новой версии get_formatted_name() параметр middle необязателен. Если второе имя передается функции, то полное имя будет содержать имя, второе имя и фамилию. В противном случае полное имя состоит только из имени и фамилии. Теперь функция должна работать для обеих разновидностей имен. Чтобы узнать, работает ли функция для имен из двух компонентов, снова запустите test_name_function.py:

```
----------------------------------------------------------------------

Ran 1 test in 0.000s

OK
```

Теперь тестовый сценарий проходит. Такой исход идеален; он означает, что функция снова работает для имен из двух компонентов и нам не придется тестировать функцию вручную. Исправить ошибку было несложно, потому что сбойный тест помог выявить новый код, нарушивший существующее поведение.

## Добавление новых тестов

Теперь мы знаем, что get_formatted_name() работает для простых имен, и можем написать второй тест для имен из трех компонентов. Для этого в класс NamesTestCase добавляется еще один метод:

***test_name_function.py***

```
…

class NamesTestCase(unittest.TestCase):

    """Тесты для 'name_function.py'."""

    def test_first_last_name(self):

    …

    def test_first_last_middle_name(self):

        """Работают ли такие имена, как 'Wolfgang Amadeus Mozart'?"""

        formatted_name = get_formatted_name(

                'wolfgang', 'mozart', 'amadeus')

        self.assertEqual(formatted_name, 'Wolfgang Amadeus Mozart')

if __name__ == '__main__':

    unittest.main()
```

Новому методу присваивается имя test_first_last_middle_name(). Имя метода должно начинаться с test_, чтобы этот метод выполнялся автоматически при запуске test_name_function .py. В остальном имя выбирается так, чтобы оно четко показывало, какое именно поведение get_formatted_name() мы тестируем. В результате при сбое теста вы сразу видите, к каким именам он относится. Не нужно опасаться длинных имен методов в классах TestCase: имена должны быть содержательными, чтобы донести информацию до разработчика в случае сбоя, а поскольку Python вызывает их автоматически, вам никогда не придется вручную вводить эти имена при вызове.

Чтобы протестировать функцию, мы вызываем get_formatted_name() с тремя компонентами, после чего используем assertEqual() для проверки того, что возвращенное полное имя совпадает с ожидаемым. При повторном запуске test_ name_function.py оба теста проходят успешно:

*…*

*----------------------------------------------------------------------*

*Ran 2 tests in 0.000s*

*OK*

Отлично! Теперь мы знаем, что функция по-прежнему работает с именами из двух компонентов, как Janis Joplin, но можем быть уверены в том, что она сработает и для имен с тремя компонентами, такими как Wolfgang Amadeus Mozart.

## Тестирование класса

В первой части этой главы мы писали тесты для отдельной функции. Сейчас мы займемся написанием тестов для класса. Классы будут использоваться во многих ваших программах, поэтому возможность доказать, что ваши классы работают правильно, будет безусловно полезной. Если тесты для класса, над которым вы работаете, проходят успешно, вы можете быть уверены в том, что дальнейшая доработка класса не приведет к случайному нарушению его текущего поведения.

## Разные методы assert

Класс unittest. TestCase содержит целое семейство проверочных методов assert. Как упоминалось ранее, эти методы проверяют, выполняется ли условие, которое должно выполняться в определенной точке вашего кода. Если условие истинно, как и предполагалось, то ваши ожидания относительно поведения части вашей программы подтверждаются; вы можете быть уверены в отсутствии ошибок. Если же условие, которое должно быть истинным, окажется ложным, то Python выдает исключение.

В таблице перечислены шесть часто используемых методов assert. С их помощью можно проверить, что возвращаемые значения равны или не равны ожидаемым, что значения равны True или False или что значения входят или не входят в заданный список. Эти методы могут использоваться только в классах, наследующих от unittest. TestCase; рассмотрим пример использования такого метода в контексте тестирования реального класса.

**Таблица** Методы assert, предоставляемые модулем unittest

| **Метод** | **Использование** |
|---|---|
| **assertEqual(a, b)** | **Проверяет, что a == b** |
| **assertNotEqual(a, b)** | **Проверяет, что a != b** |
| **assertTrue(x)** | **Проверяет, что значение x истинно** |
| **assertFalse(x)** | **Проверяет, что значение x ложно** |
| **assertIn(элемент, список)** | **Проверяет, что элемент входит в список** |
| **assertNotIn(элемент, список)** | **Проверяет, что элемент не входит в список** |

## Класс для тестирования

Тестирование класса имеет много общего с тестированием функции — значительная часть работы направлена на тестирование поведения методов класса. Впрочем, существуют и различия, поэтому мы напишем отдельный класс для тестирования. Возьмем класс для управления проведением анонимных опросов:

***survey.py***

```
class AnonymousSurvey():

    """Сбор анонимных ответов на опросы."""

    def __init__(self, question):

        """Сохраняет вопрос и готовится к сохранению ответов."""

        self.question = question

        self.responses = []

    def show_question(self):

        """Выводит вопрос."""

        print(self.question)

    def store_response(self, new_response):

        """Сохраняет один ответ на опрос."""

        self.responses.append(new_response)

    def show_results(self):

        """Выводит все полученные ответы."""

        print("Survey results:")

        for response in self.responses:

            print(f"- {response}")
```

Класс начинается с вопроса, который вы предоставили, и включает пустой список для хранения ответов. Класс содержит методы для вывода вопроса, добавления нового ответа в список ответов и вывода всех ответов, хранящихся в списке. Чтобы создать экземпляр на основе этого класса, необходимо предоставить вопрос. После того как будет создан экземпляр, представляющий конкретный опрос, программа выводит вопрос методом show_question(), сохраняет ответ методом store_response() и выводит результаты вызовом show_results().

Чтобы продемонстрировать, что класс AnonymousSurvey работает, напишем программу, которая использует этот класс:

***language_survey.py***

```
from survey import AnonymousSurvey

#Определение вопроса с созданием экземпляра AnonymousSurvey.
```

```
question = "What language did you first learn to speak?"

my_survey = AnonymousSurvey(question)

#Вывод вопроса и сохранение ответов.

my_survey.show_question()

print("Enter 'q' at any time to quit.\n")

while True:

        response = input("Language: ")

        if response == 'q':

            break

        my_survey.store_response(response)

#Вывод результатов опроса.

print("\nThank you to everyone who participated in the survey!")

my_survey.show_results()
```

Программа определяет вопрос и создает объект AnonymousSurvey на базе этого вопроса. Программа вызывает метод show_question() для вывода вопроса, после чего переходит к получению ответов. Каждый ответ сохраняется сразу же при получении. Когда ввод ответов был завершен (пользователь ввел q), метод show_results() выводит результаты опроса:

```
What language did you first learn to speak?

Enter 'q' at any time to quit.

Language: English

Language: Spanish

Language: English

Language: Mandarin

Language: q

Thank you to everyone who participated in the survey!

Survey results:

- English

- Spanish

- English

- Mandarin
```

Этот класс работает для простого анонимного опроса. Но допустим, вы решили усовершенствовать класс AnonymousSurvey и модуль survey, в котором он находится.

Например, каждому пользователю будет разрешено ввести несколько ответов. Или вы напишете метод, который будет выводить только уникальные ответы и сообщать, сколько раз был дан тот или иной ответ. Или вы напишете другой класс для проведения неанонимных опросов.

Реализация таких изменений грозит повлиять на текущее поведение класса AnonymousSurvey. Например, может оказаться, что поддержка ввода нескольких ответов случайно повлияет на процесс обработки одиночных ответов. Чтобы гарантировать, что доработка модуля не нарушит существующее поведение, для класса нужно написать тесты.

## Тестирование класса AnonymousSurvey

Напишем тест, проверяющий всего один аспект поведения AnonymousSurvey. Этот тест будет проверять, что один ответ на опрос сохраняется правильно. После того как метод будет сохранен, метод assertIn() проверяет, что он действительно находится в списке ответов:

***test_survey.py***

```
import unittest

from survey import AnonymousSurvey

class TestAnonmyousSurvey(unittest.TestCase):

    """Тесты для класса AnonymousSurvey"""

    def test_store_single_response(self):

        """Проверяет, что один ответ сохранен правильно."""

        question = "What language did you first learn to speak?"

        my_survey = AnonymousSurvey(question)

        my_survey.store_response('English')

        self.assertIn('English', my_survey.responses)

if __name__ == '__main__':

unittest.main()
```

Программа начинается с импортирования модуля unittest и тестируемого класса AnonymousSurvey. Тестовый сценарий TestAnonymousSurvey, как и в предыдущих случаях, наследует от unittest.TestCase. Первый тестовый метод проверяет, что сохраненный ответ действительно попадает в список ответов опроса. Этому методу присваивается хорошее содержательное имя test_store_single_response() . Если тест не проходит, имя метода в выходных данных сбойного теста ясно показывает, что проблема связана с сохранением отдельного ответа на опрос.

Чтобы протестировать поведение класса, необходимо создать экземпляр класса. В точке создается экземпляр с именем my_survey для вопроса "What language did you first learn to speak?". Один ответ (English) сохраняется с использованием метода store_response(). Затем программа убеждается в том, что ответ был сохранен правильно; для этого она проверяет, что значение English присутствует в списке my_survey.responses.

При запуске программы test_survey.py тест проходит успешно:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Неплохо, но опрос с одним ответом вряд ли можно назвать полезным. Убедимся в том, что три ответа сохраняются правильно. Для этого в TestAnonymousSurvey добавляется еще один метод:

```
import unittest

from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):

    """Тесты для класса AnonymousSurvey"""

    def test_store_single_response(self):

        …

    def test_store_three_responses(self):

        """Проверяет, что три ответа были сохранены правильно."""

        question = "What language did you first learn to speak?"

        my_survey = AnonymousSurvey(question)

        responses = ['English', 'Spanish', 'Mandarin']

        for response in responses:

            my_survey.store_response(response)

        for response in responses:

            self.assertIn(response, my_survey.responses)

if __name__ == '__main__':

unittest.main()
```

Новому методу присваивается имя test_store_three_responses(). Мы создаем объект опроса по аналогии с тем, как это делалось в test_store_single_response(). Затем определяется список, содержащий три разных ответа, и для каждого из этих ответов вызывается метод store_response(). После того как ответы будут сохранены, следующий цикл проверяет, что каждый ответ теперь присутствует в my_survey.responses .

Если снова запустить test_survey.py, оба теста (для одного ответа и для трех ответов) проходят успешно

```
..
----------------------------------------------------------------------
```

*Ran 2 tests in 0.000s*

*OK*

Все прекрасно работает. Тем не менее тесты выглядят немного однообразно, поэтому мы воспользуемся еще одной возможностью unittest для повышения их эффективности.

## Метод setUp()

В программе test_survey.py в каждом тестовом методе создавался новый экземпляр AnonymousSurvey, а также новые ответы. Класс unittest.TestCase содержит метод setUp(), который позволяет создать эти объекты один раз, а затем использовать их в каждом из тестовых методов. Если в класс TestCase включается метод setUp(), Python выполняет метод setUp() перед запуском каждого метода, имя которого начинается с test_. Все объекты, созданные методом setUp(), становятся доступными во всех написанных вами тестовых методах.

Применим метод setUp() для создания экземпляра AnonymousSurvey и набора ответов, которые могут использоваться в test_store_single_response() и test_store_ three_responses():

```
import unittest

from survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):

    """Тесты для класса AnonymousSurvey."""

    def setUp(self):

        """

        Создание опроса и набора ответов для всех тестовых методов.

        """

        question = "What language did you first learn to speak?"

        self.my_survey = AnonymousSurvey(question)

        self.responses = ['English', 'Spanish', 'Mandarin']

    def test_store_single_response(self):

        """Проверяет, что один ответ сохранен правильно."""

        self.my_survey.store_response(self.responses[0])

        self.assertIn(self.responses[0], self.my_survey.responses)

    def test_store_three_responses(self):

        """Проверяет, что три ответа были сохранены правильно."""

        for response in self.responses:

            self.my_survey.store_response(response)

        for response in self.responses:
```

```
        self.assertIn(response, self.my_survey.responses)

if __name__ == '__main__':

    unittest.main()
```

Метод setUp() решает две задачи: он создает экземпляр опроса и список ответов. Каждый из этих атрибутов снабжается префиксом self, поэтому он может использоваться где угодно в классе. Это обстоятельство упрощает два тестовых метода, потому что им уже не нужно создавать экземпляр опроса или ответы. Метод test_store_single_response() убеждается в том, что первый ответ в self.responses — self.responses[0] — сохранен правильно, а метод test_store_single_response() убеждается в том, что правильно были сохранены все три ответа в self.responses.

При повторном запуске test_survey.py оба теста по-прежнему проходят. Эти тесты будут особенно полезными при расширении AnonymousSurvey с поддержкой нескольких ответов для каждого участника. После внесения изменений вы можете повторить тесты и убедиться в том, что изменения не повлияли на возможность сохранения отдельного ответа или серии ответов.

При тестировании классов, написанных вами, метод setUp() упрощает написание тестовых методов. Вы создаете один набор экземпляров и атрибутов в setUp(), а затем используете эти экземпляры во всех тестовых методах. Это намного проще и удобнее, чем создавать новый набор экземпляров и атрибутов в каждом тестовом методе.

Во время работы тестового сценария Python выводит один символ для каждого модульного теста после его завершения. Для прошедшего теста выводится точка; если при выполнении произошла ошибка, выводится символ E, а если не прошла проверка условия assert, выводится символ F . Вот почему вы увидите другое количество точек и символов в первой строке вывода при выполнении ваших тестовых сценариев . Если выполнение тестового сценария занимает слишком много времени, потому что сценарий содержит слишком много тестов, эти символы дадут некоторое представление о количестве прошедших тестов.