Функции

Эта глава посвящена *функциям* — именованным блокам кода, предназначенным 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. Как вы вскоре убедитесь, использование функций упрощает чтение, написание, тестирование кода и исправление ошибок.

В этой главе также рассматриваются возможности передачи информации функциям. Вы узнаете, как писать функции, основной задачей которых является вывод информации, и другие функции, предназначенные для обработки данных и возвращения значения (или набора значений). Наконец, вы научитесь хранить функции в отдельных файлах, называемых *модулями*, для упорядочения файлов основной программы.

Определение функции

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

greeter.py

```
❶ def greet_user():
❷       """Выводит простое приветствие."""
❸       print("Hello!")

❹ greet_user()
```

В этом примере представлена простейшая структура функции. Строка ❶ 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 в точке ❷ 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. ❸. 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 ❹. Так как никакая дополнительная информация не нужна, вызов функции эквивалентен простому выполнению команды `greet_user()`. Как и ожидалось, функция выводит сообщение `Hello!`:

```
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:

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

greet_user('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:

```
Hello, Jesse!
```

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`.

ПРИМЕЧАНИЕ Иногда в литературе термины «аргумент» и «параметр» используются как синонимы. Не удивляйтесь, если переменные в определении функции вдруг будут названы аргументами, а значения, переданные при вызове функции, — параметрами.

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:

pets.py

```
❶ 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()}.")

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

Из определения ❶ 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` ❷. В теле функции эти два параметра используются для вывода информации:

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

Многократные вызовы функций

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

```
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')
```

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:

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

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

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:

```
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')
```

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:

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

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 pets.py с использованием именованных аргументов при вызове `describe_pet()`:

```
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')
```

Функция `describe_pet()` не изменилась. Однако на этот раз при вызове функции мы явно сообщаем Python, с каким параметром должен быть связан каждый аргу-

мент. 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. Следующие два вызова функции эквивалентны:

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

ПРИМЕЧАНИЕ 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:

```
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')
```

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'`:

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

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 параметром 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.

ПРИМЕЧАНИЕ 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 эквивалентных 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.

ПРИМЕЧАНИЕ На самом деле не так важно, какой стиль вызова вы используете. Если ваша функция выдает нужный результат, выберите тот стиль, который вам кажется более понятным.

Предотвращение ошибок в аргументах

Не удивляйтесь, если на первых порах вашей работы с функциями будут встречаться ошибки несоответствия аргументов. 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:

```
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()
```

Python понимает, что при вызове функции часть информации отсутствует, и мы видим это в данных трассировки:

```
  Traceback (most recent call last):
❶   File "pets.py", line 6, in <module>
❷     describe_pet()
❸ TypeError: describe_pet() missing 2 required positional arguments: 'animal_
  type' and 'pet_name'
```

В точке ❶ сообщается местонахождение проблемы, чтобы вы поняли, что с вызовом функции что-то пошло не так. В точке ❷ приводится вызов функции, приведший к ошибке. В точке ❸ 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.