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:

formatted_name.py

```
❶ def get_formatted_name(first_name, last_name):
      """Возвращает аккуратно отформатированное полное имя."""
❷     full_name = f"{first_name} {last_name}"
❸     return full_name.title()

❹ musician = get_formatted_name('jimi', 'hendrix')
  print(musician)
```

Определение `get_formatted_name()` получает в параметрах имя и фамилию ❶. Функция объединяет эти два имени, добавляет между ними пробел и сохраняет результат в `full_name` ❷. Значение `full_name` преобразуется в формат с начальной буквой верхнего регистра, а затем возвращается в точку вызова ❸.

Вызывая функцию, которая возвращает значение, необходимо предоставить переменную, в которой должно храниться возвращаемое значение. В данном случае возвращаемое значение сохраняется в переменной `musician` ❹. Результат содержит аккуратно отформатированное полное имя, построенное из имени и фамилии:

```
Jimi Hendrix
```

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:

```
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)
```

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:

```
John Lee Hooker
```

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:

```
❶ def get_formatted_name(first_name, last_name, middle_name=''):
      """Возвращает аккуратно отформатированное полное имя."""
❷     if middle_name:
          full_name = f"{first_name} {middle_name} {last_name}"
❸     else:
          full_name = f"{first_name} {last_name}"
      return full_name.title()
```

```
  musician = get_formatted_name('jimi', 'hendrix')
  print(musician)

❹ musician = get_formatted_name('john', 'hooker', 'lee')
  print(musician)
```

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 ❶.

В теле функции мы сначала проверяем, было ли задано второе имя. Python интерпретирует непустые строки как истинное значение, и если при вызове задан аргумент второго имени, `middle_name` дает результат `True` ❷. 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` ❸. 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 ❹.

Обновленная версия этой функции подойдет как для людей, у которых задается только имя и фамилия, так и для людей со вторым именем:

```
Jimi Hendrix
John Lee Hooker
```

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 (например, список или словарь). Так, следующая функция получает части имени и возвращает словарь, представляющий человека:

person.py

```
  def build_person(first_name, last_name):
      """Возвращает словарь с информацией о человеке."""
❶     person = {'first': first_name, 'last': last_name}
❷     return person

  musician = build_person('jimi', 'hendrix')
❸ print(musician)
```

Функция `build_person()` получает имя и фамилию и сохраняет полученные значения в словаре в точке ❶. Значение `first_name` сохраняется с ключом `'first'`, а значение `last_name` — с ключом `'last'`. Весь словарь с описанием человека возвращается в точке ❷. Возвращаемое значение выводится в точке ❸ с двумя исходными фрагментами текстовой информации, теперь хранящимися в словаре:

```
{'first': 'jimi', 'last': 'hendrix'}
```

Функция получает простую текстовую информацию и помещает ее в более удобную 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:

```
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:

greeter.py

```
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:
❶     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` ❶ 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:

```
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}!")
```

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'`:

```
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
```

Передача списка

Часто при вызове функции удобно передать список — имен, чисел или более сложных объектов (например, словарей). 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:

greet_users.py

```
def greet_users(names):
    """Вывод простого приветствия для каждого пользователя в списке."""
    for name in names:
        msg = f"Hello, {name.title()}!"
        print(msg)

❶ usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

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 ❶ мы определяем список пользователей `usernames`, который затем передается `greet_users()` в вызове функции:

```
Hello, Hannah!
Hello, Ty!
Hello, Margot!
```

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:

printing_models.py

```
# 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)
```

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:

```
Printing model: dodecahedron
Printing model: robot pendant
Printing model: phone case

The following models have been printed:
dodecahedron
robot pendant
phone case
```

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:

```
❶ 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)

❷ 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 ❶ 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 ❷ 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 вызы-

вается 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 поскольку 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 в print_models.py 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).