# ГЛАВА 6
# Функции и циклы

Функции являются строительными блоками практически любой программы Python. Обычно именно в них происходят основные события!

Вы уже видели, как использовать некоторые функции, например `print()`, `len()` и `round()`. Все эти функции называются **встроенными**, потому что они реализованы непосредственно в языке Python. Вы также можете создавать **пользовательские** функции для решения конкретных задач.

Функции разбивают код на меньшие блоки. Их используют для определения действий, которые должны неоднократно выполняться в программном коде. Вам не придется писать один и тот же код каждый раз, когда программе потребуется выполнить эту задачу, достаточно вызвать функцию!

Но иногда некоторую часть кода требуется повторить несколько раз подряд. Для этого используются **циклы**.

**В этой главе вы:**

- создадите пользовательские функции;
- научитесь работать с циклами `for` и `while`;
- узнаете, что такое область видимости и почему она важна.

Итак, за дело!

## 6.1. ЧТО ЖЕ ТАКОЕ ФУНКЦИЯ?

В предыдущих главах мы использовали функции `print()` и `len()` для вывода текста и определения длины строки. Давайте разберемся с функциями более подробно.

В этом разделе на примере `len()` я покажу, что такое функция и как она выполняется.

## Функции как значения

Одна из самых важных особенностей функций в языке Python состоит в том, что функции — это значения, которые могут присваиваться переменным.

Проверим имя `len` в интерактивном окне IDLE. Для этого введите его после приглашения:

```
>>> len
<built-in function len>
```

Python сообщает, что `len` является встроенной функцией. По аналогии с тем, как целочисленные значения имеют тип `int`, а строки — тип `str`, значения-функции также имеют тип:

```
>>> type(len)
<class 'builtin_function_or_method'>
```

Однако при желании с именем `len` можно связать другое значение:

```
>>> len = "I'm not the len you're looking for."
>>> len
"I'm not the len you're looking for."
```

Теперь имя `len` связано со строковым значением. Вы можете убедиться в том, что оно относится к типу `str`, при помощи функции `type()`:

```
>>> type(len)
<class 'str'>
```

И хотя значение, связанное с именем `len`, можно изменить, делать так обычно не рекомендуется. Изменение значения `len` только усложнит чтение вашего кода, потому что новое имя `len` легко перепутать со встроенной функцией. Сказанное относится к любой встроенной функции.

**ВАЖНО!**

После выполнения этих примеров кода встроенная функция len станет недоступной в IDLE. Чтобы вернуть ее, введите команду:

```
>>> del len
```

Ключевое слово `del` используется для отмены присваивания переменной. Это сокращение от delete (удалить), но значение не удаляется. Вместо этого связь имени со значением разрывается и удаляется имя.

Обычно после выполнения `del` при попытке использования имени удаленной переменной происходит ошибка `NameError`. Однако в данном случае имя `len` не удаляется:

```
>>> len
<built-in function len>
```

Так как `len` является именем встроенной функции, оно снова связывается с исходным значением-функцией.

Какой же вывод из этого можно сделать? У функций есть имена, но эти имена не имеют жесткой связи с функцией, и им можно присваивать другие значения.

Когда вы пишете собственные функции, будьте внимательны и не присваивайте им значения, используемые встроенными функциями.

## Как Python работает с функциями

Давайте более детально разберемся в том, как Python выполняет функции.

Прежде всего следует заметить, что функцию невозможно выполнить, просто указав ее имя. Необходимо **вызвать** функцию, чтобы сообщить Python, как она должна выполняться.

Посмотрим, как работает механизм вызова на примере `len()`:

```
>>> # Печать имени функции не приводит к ее исполнению.
>>> # IDLE проверяет переменную как обычно.
>>> len
<built-in function len>

>>> # Используем скобки для вызова функции.
>>> len()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    len()
TypeError: len() takes exactly one argument (0 given)
```

В этом примере Python выдает ошибку `TypeError` при вызове `len()`, потому что функция `len()` ожидает получить аргумент.

**Аргумент** представляет собой значение, которое передается функции. Некоторые функции вызываются без аргументов, другие могут получать сколько угодно аргументов. Функция `len()` должна получать только один аргумент. Завершив выполнение, функция **возвращает** значение. Возвращаемое значение обычно — хотя и не всегда — зависит от значений других аргументов, передаваемых функции.

Функция выполняется в три этапа.

1. Функция **вызывается**, и все аргументы передаются ей в качестве входных значений.
2. Функция **выполняется**, и с ее аргументами выполняются некоторые действия.
3. Функция **возвращает** управление, а исходный вызов функции заменяется возвращаемым значением.

Давайте рассмотрим конкретный пример и разберемся, как Python выполняет следующую строку кода:

```
>>> num_letters = len("four")
```

Сначала `len()` вызывается с аргументом `"four"`. Вычисляется длина строки `"four"`, которая равна 4. Затем `len()` возвращает число 4 и заменяет вызов функции полученным значением.

После выполнения функции строка кода будет выглядеть так:

```
>>> num_letters = 4
```

Затем Python присваивает значение 4 переменной `num_letters` и продолжает выполнение остальных строк кода в программе.

## Функции могут обладать побочными эффектами

Вы узнали, как вызывать функции и что функции возвращают значение при завершении выполнения. Тем не менее иногда функции не ограничиваются простым возвращением значения.

Если функция изменяет что-то в программе за пределами своего собственного кода, говорят, что она имеет **побочный эффект**. Вы уже видели одну функцию с побочным эффектом: `print()`.

Когда вы вызываете `print()` со строковым аргументом, строка выводится в оболочке Python в текстовом виде. Однако `print()` не возвращает текстового значения.

Чтобы понять, что возвращает `print()`, присвойте возвращаемое значение `print()` переменной:

```
>>> return_value = print("What do I return?")
What do I return?
>>> return_value
>>>
```

Когда вы присваиваете `print("What do I return?")` переменной `return_value`, выводится строка `"What do I return?"`. Но при проверке значения `return_value` ничего не выводится.

Функция `print()` возвращает специальное значение `None`, которое указывает на отсутствие данных; `None` относится к типу, который называется `NoneType`:

```
>>> type(return_value)
<class 'NoneType'>
>>> print(return_value)
None
```

При вызове `print()` выводимый текст не является возвращаемым значением, это побочный эффект `print()`.

## 6.2. НАПИСАНИЕ ВАШИХ СОБСТВЕННЫХ ФУНКЦИЙ

При написании более сложных программ может оказаться, что вам нужно многократно выполнять несколько строк кода — например, вычислять одну и ту же формулу для разных входных значений.

Возможно, у вас появится искушение скопировать код в другие части программы и изменить его по мере надобности, но так поступать не стоит! Если вы обнаружите ошибку в скопированном коде, исправление придется вносить во все копии. А это огромная работа!

В этом разделе вы научитесь создавать собственные функции, чтобы избежать дублирования кода при его многократном использовании.

### Анатомия функции

Каждая функция состоит из двух частей.

1. Сигнатура функции определяет имя функции и все входные данные, которые она ожидает получить.
2. Тело функции содержит код, который выполняется при каждом использовании функции.

Напишем функцию, которая получает два числа на входе и возвращает их произведение. Вот как может выглядеть функция (сигнатура и тело обозначены в комментариях):

```
def multiply(x, y): # Сигнатура функции
  # Тело функции
  product = x * y
  return product
```

Создание функции для чего-то настолько простого, как оператор `*`, выглядит странно. Пожалуй, `multiply()` — не та функция, которую вы будете использовать в реальной программе. Но это хороший первый пример, чтобы понять, как создаются функции!

**ВАЖНО!**

Когда вы определяете функцию в интерактивном окне IDLE, необходимо нажать Enter дважды после строки, содержащей команду return, чтобы функция была зарегистрирована Python:

```
>>> def multiply(x, y):
... product = x * y
... return product
... # <--- Здесь нужно нажать Enter во второй раз.
>>>
```

Разобьем функцию на части и посмотрим, что в какой момент происходит.

## Сигнатура функции

Первая строка кода функции называется **сигнатурой функции**. Она всегда начинается с ключевого слова `def` (сокращение от define — определить).

Рассмотрим повнимательнее сигнатуру `multiply()`:

```
def multiply(x, y):
```

Сигнатура функции состоит из четырех частей.

1. Ключевое слово `def`.
2. Имя функции `multiply`.
3. Список параметров `(x, y)`
4. Двоеточие (:) в конце строки.

Когда Python читает строку, начинающуюся с ключевого слова `def`, он создает новую функцию. Эта функция присваивается переменной, имя которой совпадает с именем функции.