Тема 9. Функции. Модули

Функция – это блок организованного, многократно используемого кода, который используется для выполнения конкретного задания. Функции обеспечивают модульность приложения и значительно повышают эффективность повторного использования кода.

9.1. Создание функции

Существуют определенные правила для создания функции:

1. Блок функции начинается с ключевого слова **def**, после которого следует название функции и обязательно круглые скобки.
2. Любые аргументы, которые принимает функция, указываются в скобках и называются **формальными**.
3. После скобок идет двоеточие. С новой строки с отступом начинается тело функции.

def *<имя функции>(<аргументы>):*
  <тело функции>

Пример:

```
def sum(a, b):
  print(a + b)
```

Но сама по себе функция, пока ее не вызовешь в программе, не имеет никакого смысла.

9.2. Вызов функции

После создания функции ее можно использовать, вызывая из другой функции или напрямую из оболочки **Python.** Для вызова функции следует ввести ее имя и добавить скобки, а в них уже указать фактические аргументы.

Пояснение: формальные и фактические (передаваемые в функцию) аргументы могут называться как одинаковыми именами, так и разными. При создании они считаются разными переменными, поэтому не влияют друг на друга.

Пример:

```
# функция вывода суммы двух чисел
def sum(a, b):
  print(a + b)

# считываем два числа и вызываем функцию вывода суммы
c = int(input('c='))
d = int(input('d='))
sum(c, d)
```

В результате на экран выведется сумма чисел *c* и *d*.

9.3. Аргументы функции

Вызывая функцию, ей можно передавать следующие типы аргументов:

- обязательные аргументы;
- аргументы – ключевые слова;
- аргументы по умолчанию;
- аргументы произвольной длины (произвольное количество аргументов).

Обязательные аргументы

Если при создании функции мы указали количество аргументов и их порядок, то и вызывать функцию мы должны с тем же количеством аргументов, заданных в том же порядке.

Пример:

```
def pmin(a, b):
  if a < b:
    print(a)
  else:
    print(b)

c = int(input('c='))
d = int(input('d='))
pmin(c, d)
```

А если вызвать функцию так:

```
...
pmin(c, d, d)
```

Получим ошибку.

Аргументы – ключевые слова

Аргументы – ключевые слова используются при вызове функции. Благодаря ключевым аргументам можно при вызове функции указывать аргументы в любом порядке.

Пример:

```
def person(name, age):
  print('Меня зовут ', name, ', мне ', age, ' лет.', sep='')

person(age=20, name='Катя')
```

```
Меня зовут Катя, мне 20 лет.
```

Обратите внимание, что в описании функции и в ее вызове порядок аргументов разный. Это работает только при явном указании названия аргумента.

Если мы напишем так:

```
...
person(20, 'Катя')
```

то в результате получим:

```
Меня зовут 20, мне Катя лет.
```

Аргументы по умолчанию

Этот способ используют, когда нужно задать функцию, которая по умолчанию работает с одним значением аргумента, а при указании иного значения – с другим. Допустим, нам нужно написать функцию увеличения аргумента на какое-либо значение. По умолчанию функция должна увеличивать его на 1.

Пример 1:

```
def inc(a, b=1):
  print(a + b)

inc(5)
```

```
6
```

Пример 2:

```
def inc(a, b=1):
  print(a + b)

inc(5, 2)
```

```
7
```

Произвольное количество аргументов

Иногда возникает ситуация, когда заранее неизвестно, какое количество аргументов необходимо будет принять функции. В этом случае используют произвольное количество аргументов. Они задаются именем, перед которым ставится знак *****.

Пример:

Напишем функцию нахождения суммы чисел.

```
def msum(*a):
  s = 0
  for i in a:
    s += i
  print(s)

msum(5, 2, 6)
```

```
13
```

Произвольное количество именованных аргументов

Функция может принимать и произвольное число именованных аргументов, тогда перед именем ставится **.

```
def tt(**a):
  return a

print(tt(a=1, b=2, c=3, d=4))
```

В результате получим словарь:

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

9.4. Ключевое слово return

В Python, как и в других языках, есть возможность создания функций, которые не возвращают результат. До этого момента мы создавали именно такие функции. Но есть возможность создания функций возвращающих результат. Для этого используется ключевое слово **return**.

В качестве примера напишем функцию для нахождения минимального из двух чисел. На самом деле такая функция в Python уже есть, но на ее примере удобно рассматривать принцип работы. Поэтому создадим такую же, но назовем ее по-другому.

```
# функция
def umin(a, b):
  if a < b:
    return a
  else:
    return b

# вызов функции
c = umin(5, 2)
print(c)
```

```
2
```

Обратите внимание, в этом примере мы присвоили функцию (результат ее работы) переменной. Это возможно только в случае использования оператора **return**.

9.5. Область видимости переменных

В Python есть две базовых области видимости переменных:

- глобальные переменные;
- локальные переменные.

Переменные, объявленные внутри тела функции, имеют **локальную** область видимости, а переменные, объявленные вне тела функции, имеют **глобальную** область видимости. То есть, доступ к локальным переменным имеют только те

функции, в которых они объявлены. К глобальным переменным доступ можно получить из любого места программы.

Пример:

```
a = 5

def tt_local():
    a = 7
    print(a, end=' ')

def tt():
    print(a, end=' ')

tt_local()
tt()
```

```
7 5
```

Как видно из примера, 2 переменные с одним и тем же именем, но одна локальная, а другая глобальная, обрабатываются не зависимо друг от друга в своей области видимости.

Если локальная переменная и глобальная имеют одно и то же имя, то функция работает с локальной переменной.

Важно понимать, что для того, чтобы получить доступ к глобальной переменной, достаточно лишь указать ее имя. В примере выше мы просто выводим значение переменной, поэтому обращаемся к ней напрямую.

Однако если перед нами стоит задача изменить глобальную переменную внутри функции, необходимо использовать ключевое слово **global**.

```
a = 5
def tt():
    global a
    a += 1

tt()
print(a)
```

```
6
```

Как видно из примера, мы обратились к глобальной переменной из функции и изменили ее значение.

А если сделать так:

```
a = 5
def tt():
    a += 1
tt()
print(a)
```

то получим ошибку:

```
builtins.UnboundLocalError: local variable 'a' referenced before assignment
```

Измерение параметров внутри функции

Если при вызове функции мы подставим в качестве значения аргумента переменную, а в теле функции происходит изменение аргумента, то изменится значение переданной в качестве аргумента переменной или нет зависит от того, с каким типом данных она связана. Если переменная связана со значением неизменяемого типа, например **int**, **str**, **tuple**, то изменение не произойдет. А если переменная связана со списком, словарем или классом, то значение связанного с переменной объекта изменится.

Пример 1 (значение не изменится):

```
def a(b):
    b = 2

c = 3
a(c)
print(c)
```

```
3
```

Пример 2 (значение изменится):

```
def a(b):
    b[0] = 2

c = [3]
a(c)
print(c)
```

```
[2]
```

9.6. Рекурсия

Рекурсией в программировании называется ситуация, в которой функция вызывает саму себя. Классическим примером рекурсии может послужить функция вычисления факториала числа.

```
# функция вычисления факториала числа
def fact(num):
    if num == 0:
        return 1
    else:
        return num * fact(num - 1)

# вызов функции
print(fact(3))
```

```
6
```

Однако следует помнить, что использование рекурсии часто может быть неоправданным. Дело в том, что в момент вызова функции в оперативной памяти компьютера резервируется определенное количество памяти, соответственно, чем больше функций одновременно мы запускаем, тем больше памяти потребуется, что может привести к переполнению стека (**stack overflow**) и программа завершится аварийно, не так как предполагалось. Учитывая это, там, где это возможно, вместо рекурсии лучше применять циклы.

9.7. Подключение модулей

При написании программ очень часто приходится писать один и тот же код. Особенно это касается создания проектов, состоящих из нескольких файлов исходного кода. В таких случаях логично вынести одинаковые функции в отдельный файл, называемый **модулем**, а затем по необходимости подключать его. Это позволит уменьшить объем кода и сократить время работы по написанию программы.

Модулем в Python называется любой файл с программой. Каждая программа может импортировать модуль и получить доступ к его классам, функциям и объектам. Нужно заметить, что модуль может быть написан не только на Python, а, например, на C или C++.

Для Python существует большое количество модулей. Изначально много модулей устанавливается вместе с Python. Список установленных модулей можно посмотреть в Python командой:

```
help('modules')
```

После запуска увидим список всех модулей.

Мы уже работали с модулями **math** и **random.**

Готовые модули можно подключать двумя способами:

- полное импортирование модуля **import**;
- частичное импортирование модуля **from** … **import**.

Команда import

Как было сказано выше, команда **import** подключает модуль Python к программе, импортируя его полностью. Синтаксис команды выглядит следующим образом:

import *<модуль1> [, <модуль2>, ..., <модуль N>]*

Модуль загружается только один раз, независимо от того, сколько раз его импортировали в коде.

Обращение к функции из модуля происходит следующим образом:

<имя модуля>.<функция>*(<параметры>)*

Пример:

Импортируем модуль **math** и используем функцию извлечения корня

```
import math
print(math.sqrt(25))
```

Для просмотра всех функций модуля **math** можно использовать команду **help**

```
help('math')
```

Команда from…import

Команда **from…import** позволяет импортировать не весь модуль целиком, а только необходимое для программы содержимое.

Синтаксис выглядит следующим образом:

from *<имя модуля>* ***import*** *<функция1> [, <функция2>, … , <функция N>]*

Допустим, нам из модуля **math** нужна только функция **sqrt**, тогда мы можем импортировать ее отдельно без всего модуля следующим образом:

```
from math import sqrt
print(sqrt(25))
```

Заметьте, в этом примере перед именем функции для ее вызова уже не нужно дописывать название модуля.

При помощи этой конструкции можно также импортировать все функции модуля сразу. Для этого нужно использовать символ ***.**

from *<имя модуля>* ***import ****

После выполнения инструкции, ко всем функциям уже нужно обращаться напрямую, без указания имени модуля. Но этот способ может создать проблемы в случае, если в нескольких импортируемых модулях есть функции с одинаковыми именами.

9.8. Псевдонимы

Если название модуля является слишком длинным и его неудобно каждый раз указывать для доступа к идентификаторам внутри модуля, то можно создать псевдоним. Псевдоним задается после ключевого слова **as**. Создадим псевдоним для модуля **math**:

```
import math as m
print(m.pi)
```

Теперь доступ к атрибутам модуля **math** может осуществляться только с помощью идентификатора **m**. Идентификатор **math** в этом случае использовать уже нельзя. Все идентификаторы внутри импортированного модуля доступны только через идентификатор, указанный в инструкции **import**.

Аналогично можно использовать псевдонимы и в **from … import**:

```
from math import pi as p
print(p)
```

9.9. Создание собственных модулей

Собственный модуль создаем точно так же, как и обычную программу, и сохраняем с тем же расширением. Чтобы программу можно было использовать как модуль, нужно написать функции, которые мы затем сможем использовать в других программах.

Файл модуля лучше разместить в папке с создаваемой программой, так как Python при подключении модуля в первую очередь ищет его в текущем каталоге.

Допустим, нам нужен модуль для выполнения геометрических расчетов. В нем у нас будет 2 функции. Одна для вычисления площади треугольника, другая для вычисления площади круга.

Создадим файл и сохраним его под именем **mygeometry.py**

В нем реализуем 2 функции:

- площадь треугольника по формуле Герона;
- площадь круга.

Для упрощения примера мы не будем осуществлять проверку существования треугольника по введенным сторонам.

```
import math

def striangle(a, b, c):
    p = (a + b + c) / 2
    s = math.sqrt(p * (p - a) * (p - b) * (p - c))
    return s

def scircle(r):
    s = math.pi * r * r
    return s
```

Заметьте, что в данном модуле мы вначале подключаем модуль **math**, нам он нужен для извлечения квадратного корня и получения числа π.

Теперь создадим второй файл, в котором будем использовать наш созданный модуль, и напишем в нем следующий код:

```
import mygeometry
help('mygeometry')
```

Программа выдаст всю информацию о модуле: где он находится, и какие функции в нем реализованы.

Попробуем его использовать для расчета площади треугольника.

Отредактируем содержимое второго созданного файла следующим образом:

```
import mygeometry

a = int(input('a='))
b = int(input('b='))
c = int(input('c='))
print('s=', mygeometry.striangle(a, b, c))
```

В результате программа выведет площадь треугольника с заданными нами сторонами. Как мы видим, создание модуля не составляет никаких сложностей, но при этом является мощным инструментом в программировании.

Если необходимо поместить модуль в другом месте, то нужно указать к нему путь в **sys.path**. Например, мы создали модуль **mymodule.py** и поместили его в папку C**:\\modules,** а программу, использующую этот модуль, сохранили в другом каталоге. Тогда модуль нужно подключать следующим образом:

```
import sys

sys.path.append("C:\\modules")
import mymodule
```

9.10. Разделение кода внутри программы (модуля)

Как было указано выше, модулем в Python может быть любой файл исходного кода на этом языке. Каждый файл Python может быть запущен как отдельная программа, а может быть подключен в качестве модуля к другой запускаемой программе. Если подключаемый файл создавался изначально как «классический» модуль, т. е. содержащий только набор различных функций, то функции срабатывают только во время их вызова. А если подключаемый файл является программой, содержащей как обычный набор инструкций, так и функции, то в том месте, где подключается модуль, выполнение программы остановится и не возобновится до тех пор, пока не выполнится код из подключаемого файла. В связи с этим возникает необходимость разделять код, который должен выполниться при непосредственном запуске файла как отдельной программы от того, который должен выполняться при использовании этого файла в качестве модуля.

У каждого файла в Python есть специальный атрибут **__name__**, значение которого устанавливается равным «**__main__**», если файл запускается как основная программа, а не подключается в качестве модуля. Проверяя равенство атрибута **__name__** значению «**__main__**» можно узнать, является файл основной запущенной программой или подключенным модулем.

Рассмотрим два простых файла.

Первый файл: *m1.py*

```
# m1.py
print('Запущен модуль 1')
print('Подключаем модуль 2')
import m2
print('Последняя инструкция модуля 1')
```

Второй файл: *m2.py*

```
# m2.py
print('Запущен модуль 2')
if __name__ == '__main__':
    print('Модуль 2 запущен как основная программа')
print('Последняя инструкция модуля 2')
```

Если запустить файл *m2.py*, то выведется следующий результат:

```
Запущен модуль 2
Модуль 2 запущен как основная программа
Последняя инструкция модуля 2
```

Если запустить файл *m1.py*, то выведется следующий результат:

```
Запущен модуль 1
Подключение модуля 2
Запущен модуль 2
Последняя инструкция модуля 2
Последняя инструкция модуля 1
```

Обратите внимание, что во втором случае не вывелась строка «*Модуль 2 запущен как основная программа*». Это говорит о том, что во втором случае файл *m2.py* запущен как модуль внутри файла *m1.py*.