Компьютерные технологии

Лекция № 2.
Функции. Модули.

Нижний
Новгород
2022 г.

Функции

Функции – это многократно используемые фрагменты программы.

Функции определяются при помощи зарезервированного слова ***def***.

После этого слова указывается ***имя*** функции, за которым следует пара скобок, в которых можно указать имена некоторых ***переменных***, и заключительное двоеточие в конце строки.

Функции

```
def sayHello():
    print('Привет, Мир!') # блок, принадлежащий функции
sayHello() # вызов функции
sayHello() # ещё один вызов функции
```

Вывод: $ python function1.py

Привет, Мир!

Привет, Мир!

Параметры функций

Параметры указываются в скобках при объявлении функции и разделяются запятыми.

Имена, указанные в объявлении функции, называются ***параметрами***, тогда как значения, которые вы передаёте в функцию при её вызове, – ***аргументами***.

Параметры функций

```
def printMax(a, b):
    if a > b:
        print(a, 'максимально')
    elif a == b:
        print(a, 'равно', b)
    else:
        print(b, 'максимально')
printMax(3, 4) # прямая передача значений
x = 5 y = 7
printMax(x, y) # передача переменных в качестве аргументов
```

Вывод: $ python func_param.py

4 максимально

7 максимально

Локальные переменные

При объявлении переменных внутри определения функции, они никоим образом не связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются **локальными** в функции. Это называется **областью видимости переменной**. Область видимости всех переменных ограничена блоком, в котором они объявлены, начиная с точки объявления имени.

Локальные переменные

```
x = 50
def func(x):
    print('x равен', x)
    x = 2
    print('Замена локального x на', x)
func(x)
print('x по прежнему', x)
```

Вывод: $ python func_local.py

x равен 50

Замена локального x на 2

x по прежнему 50

Зарезервированное слово «global»

Чтобы присвоить некоторое значение переменной, определённой в функции необходимо указать Python, что её имя не локально, а глобально (**global**).

Без применения зарезервированного слова **global** невозможно присвоить значение переменной, определённой за пределами функции.

Зарезервированное слово «global»

```
x = 50
def func():
    global x
    print('x равно', x)
    x = 2
    print('Заменяем глобальное значение x на', x)
func()
print('Значение x составляет', x)
```

Вывод: $ python func_global.py

x равно 50

Заменяем глобальное значение x на 2

Значение x составляет 2

Значения аргументов по умолчанию

Часть параметров функций могут быть **необязательными**, и для них будут использоваться некоторые заданные значения **по умолчанию**.

Их можно указать, добавив к имени параметра в определении функции **оператор присваивания** (=) с последующим значением.

Значение по умолчанию должно быть константой.

Значения аргументов по умолчанию

```
def say(message, times = 1):
    print(message * times)
say('Привет')
say('Мир', 5)
```

Вывод: $ python func_default.py

Привет

МирМирМирМирМир

Значения аргументов по умолчанию

Важно: Значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров.

Таким образом, в списке параметров функции параметр со значением по умолчанию не может предшествовать параметру без значения по умолчанию.

Это связано с тем, что значения присваиваются параметрам в соответствии с их положением. Например,

def func(a, b=5) допустимо, а

def func(a=5, b) – не допустимо.

Ключевые аргументы

Если имеется некоторая функция с большим числом параметров, и при её вызове требуется указать только некоторые из них, значения этих параметров могут задаваться по их имени – это называется ***ключевые параметры***. В этом случае для передачи аргументов функции используется имя (ключ) вместо позиции.

- использование функции становится легче, поскольку нет необходимости отслеживать порядок аргументов;
- можно задавать значения только некоторым избранным аргументам, при условии, что остальные параметры имеют значения аргумента по умолчанию.

Ключевые аргументы

```
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a c равно', c)
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Вывод: $ python func_key.py

a равно 3, b равно 7, a c равно 10

a равно 25, b равно 5, a c равно 24

a равно 100, b равно 5, a c равно 50

Переменное число параметров

Иногда бывает нужно определить функцию, способную принимать любое число параметров. Этого можно достичь при помощи звёздочек.

```
def total(initial=5, *numbers, **keywords):
    count = initial
    for number in numbers:
        count += number
    for key in keywords:
        count += keywords[key]
    return count
print(total(10, 1, 2, 3, vegetables=50, fruits=100))
```

Вывод: $ python total.py

166

Только ключевые параметры

Если некоторые ключевые параметры должны быть доступны только по ключу, а не как позиционные аргументы, их можно объявить после параметра со звёздочкой

```
def total(initial=5, *numbers, extra_number):
    count = initial
    for number in numbers:
        count += number
        count += extra_number
        print(count)
total(10, 1, 2, 3, extra_number=50)
total(10, 1, 2, 3) # Вызовет ошибку, поскольку мы не указали значение # аргумента по умолчанию для 'extra_number‘
```

Вывод: $ python keyword_only.py

66 Traceback (most recent call last): File "keyword_only.py", line 12, in total(10, 1, 2, 3) TypeError: total() needs keyword-only argument extra_number

Оператор «return»

Оператор ***return*** используется для возврата из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также вернуть некоторое значение из функции.

```
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'Числа равны.‘
    else:
        return y
    print(“ttrrr”)
print(maximum(2, 3))
```

Вывод: $ python func_return.py
3

Оператор «return»

Оператор ***return*** без указания возвращаемого значения эквивалентен выражению ***return None***.

None – это специальный тип данных в Python, обозначающий ничего. К примеру, если значение переменной установлено в ***None***, это означает, что ей не присвоено никакого значения. Каждая функция содержит в неявной форме оператор ***return None*** в конце, если вы не указали своего собственного оператора ***return***.

В этом можно убедиться, запустив print(someFunction()), где функция someFunction – это какая-нибудь функция, не имеющая оператора return в явном виде.

Например:

```
def someFunction():
    pass
```

Модули

Для составления модулей необходимо создать файл с расширением .py, содержащий функции и переменные.

Модуль можно ***импортировать*** в другую программу, чтобы использовать функции из него.

Модули

```
import sys
print('Аргументы командной строки:')
for i in sys.argv:
    print(i)
print('\n\nПеременная PYTHONPATH содержит', sys.path, '\n')
```

Вывод: $ python using_sys.py we are arguments

Аргументы командной строки:

using_sys.py

We

are

arguments

Переменная PYTHONPATH содержит ['', 'C:\\Windows\\system32\\python30.zip', 'C:\\Python30\\DLLs', 'C:\\Python30\\lib', 'C:\\Python30\\lib\\plat-win', 'C:\\Python30', 'C:\\Python30\\lib\\site-packages']

Файлы байткода .pyc

Можно создать ***байт-компилированные*** файлы (или байткод) с расширением .pyc, которые являются некой промежуточной формой, в которую Python переводит программу.

Такой файл .pyc полезен при ***импорте*** модуля в следующий раз в другую программу – это произойдёт ***намного быстрее***, поскольку значительная часть обработки, требуемой при импорте модуля, будет ***уже проделана***.

Этот байткод также является ***платформо-независимым***.

Обычно файлы .pyc создаются в том же каталоге, где расположены и соответствующие им файлы .py. Если Python не может получить доступ для записи файлов в этот

Оператор from ... import ...

Чтобы импортировать переменную argv прямо в программу и не писать всякий раз sys. при обращении к ней, можно воспользоваться выражением “**from sys import argv**”.

Для импорта ***всех имён***, использующихся в модуле ***sys***, можно выполнить команду “from sys import *”.

Это работает ***для любых модулей***.

В общем случае следует избегать использования этого оператора и использовать вместо этого оператор ***import***, чтобы предотвратить конфликты имён и не затруднять чтение программы.

Operator from ... import ...

```
from math import *
n = input(" Enter the range:- ")
p = [2, 3]
count = 2
a = 5
while (count < n):
    b=0
    for i in range(2,a):
        if ( i <= sqrt(a)):
            if (a % i == 0):
                print("a not a Prime number ",a)
                b = 1
            else: pass
    if (b != 1):
        print("a Prime number ",a)
        p = p + [a]
    count = count + 1
    a = a + 2
print p
```

Имя модуля – __name__

У каждого модуля есть ***имя***, и ***команды*** в модуле могут узнать имя их модуля.

Можно заставить модуль ***вести себя по-разному*** в зависимости от того, используется ли он ***сам по себе*** или ***импортируется*** в другую программа.

Этого можно достичь с применением атрибута модуля под названием ***__name__.***

Имя модуля – __name__

```
if __name__ == '__main__':
    print('Эта программа запущена сама по себе.')
else:
    print('Меня импортировали в другой модуль.')
```

Вывод:

```
$ python3 using_name.py
Эта программа запущена сама по себе.

$ python3 >>> import using_name
Меня импортировали в другой модуль.
>>>
```

Создание собственных модулей

Каждая программа на Python также является и модулем. Необходимо лишь убедиться, что у неё установлено расширение .py.
Для использования модуля в других программах, необходимо, чтобы модуль находился либо в том же каталоге, что и программа, в которую мы импортируем его, либо в одном из каталогов, указанных в sys.path.

Создание собственных модулей

Пример: mymodule.py

```
def sayhi():
    print('Привет! Это говорит мой модуль.')
__version__ = '0.1‘
```

```
import mymodule
mymodule.sayhi()
print ('Версия', mymodule.__version__)
```

Вывод: $ python mymodule_demo.py
Привет! Это говорит мой модуль.
Версия 0.1

Создание собственных модулей

Версия, использующая синтаксис ***from..import***

```
from mymodule import sayhi, __version__
sayhi()
print('Версия', __version__)
```

Можно также использовать:

```
from mymodule import *
```

Это импортирует все публичные имена, такие как sayhi, но не импортирует __version__, потому что оно начинается с двойного подчёркивания

Функция dir

Можно использовать встроенную функцию dir, чтобы получить **список идентификаторов**, которые объект определяет.
Так в число идентификаторов модуля входят **функции**, **классы** и **переменные**, определённые в этом модуле.

Если передать функции dir() имя модуля, она возвращает **список имён**, определённых в этом модуле.

Если никакого аргумента не передавать, она вернёт список имён, **определённых в текущем модуле**.

```
$ python3
>>> import sys # получим список атрибутов модуля 'sys‘
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__s tderr__', '__stdin__', '__stdout__', '_clear_type_cache', '_compact_freelists',
'_current_frames', …]
```

Функция dir

```
>>> dir() # получим список атрибутов текущего модуля
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>> a = 5 # создадим новую переменную 'a‘
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a', 'sys']

>>> del a # удалим имя 'a‘

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']

>>>
```

Пакеты

Иерархия в организации программ: переменные обычно находятся в функциях. Функции и глобальные переменные обычно находятся в модулях. Для организации модулей используются пакеты.

Пакеты – это просто каталоги с модулями и специальным файлом __init__.py, который показывает Python, что этот каталог особый, так как содержит модули Python.

Пакеты – это удобный способ иерархически организовать модули.

Пакеты

Необходимо создать пакет под названием «world» с субпакетами «asia», «africa» и т.д., которые, в свою очередь, будут содержать модули «india», «madagascar» и т.д.
Для этого следовало бы создать следующую структуру каталогов:

```
| - <некоторый каталог из sys.path>/
| |---- world/
|       |---- __init__.py
|       |---- asia/
|       |     |---- __init__.py
|       |     |---- india/
|       |           |---- __init__.py
|       |           |---- foo.py
|       |---- africa/
|             |---- __init__.py
|             |---- madagascar/
|                   |---- __init__.py
|                   |---- bar.py
```