Язык программирования Питон (Python)

Функции

Функции – это многократно используемые фрагменты программы.

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

Пример:

```
def printMax(a, b):      # Определение функции
    if a > b:
        print(a, 'максимально')
    elif a == b:
        print(a, 'равно', b)
    else:
        print(b, 'максимально')
```

– после того как функция определена, можно ее ***вызывать***, передавая в скобках различные аргументы:

```
printMax(3, 4)           # вызов функции (прямая передача значений)
                                    -> 4 максимально
```

Переменные в функциях

– ***переменные***, объявленные *внутри* определения функции являются *локальными*.
Область видимости всех переменных ограничена блоком, в котором они объявлены.

Пример:

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

func(x)
print('x по прежнему', x)                  ->  x по прежнему 50
```

– для *изменения* переменной, определённой *за пределами функции*, используется зарезервированное слово `global`

Пример:

```
x = 50
def func():
    global x
    print('x равно', x)                    ->  x равно 50
    x = 2
    print('Замена глобального x на', x)    ->  Замена глобального x на 2

func()
print('Значение x составляет', x)          ->  Значение x составляет 2
```

– при определении одной функции внутри другой для *изменения переменной внешней функции* из *внутренней* используется зарезервированное слово **nonlocal**

Пример:

```
x = 50
def func_outer():
    x = 2
    print('x равно', x)                          ->  x равно 2

    def func_inner():
        nonlocal x
        x = 5

    func_inner()
    print('Локальное x сменилось на', x)  ->  Локальное x сменилось на 5

func_outer()
print('Глобальное x ', x)                        ->  Глобальное x  50
```

Аргументы функции

– ***значения аргументов по умолчанию*** задаются добавлением к имени параметра оператора присваивания с последующим значением

Замечания. *– значениями по умолчанию могут быть снабжены только параметры, находящиеся в конце списка параметров;*

– значения по умолчанию должны быть неизменяемыми (константы).

Пример:

```
def say(message, times = 1):
    print(message * times)

say('Привет')       ->  Привет
say('Мир ', 5)      ->  Мир Мир Мир Мир Мир
```

– ***ключевые аргументы*** – передача аргументов при вызове функции не по порядку следования, а по именам (ключам)

Пример:

```
def func(a, b=5, c=10):
    print('a равно', a, ', b равно', b, ', a c равно', c)

func(3, 7)          ->  a равно 3 , b равно 7 , a c равно 10
func(25, c=24)      ->  a равно 25 , b равно 5 , a c равно 24
func(c=50, a=100)   ->  a равно 100 , b равно 5 , a c равно 50
```

– определение функции ***с произвольным числом параметров*** осуществляется при помощи звёздочек:

- при объявлении параметра со звёздочкой (например, **`*param`**), все позиционные аргументы начиная с этой позиции и до конца будут собраны в кортеж под именем **`param`**.
- при объявлении параметра с двумя звёздочками (**`**param`**), все ключевые аргументы начиная с этой позиции и до конца будут собраны в словарь под именем **`param`**.

Пример:

```
def total(initial, *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))    ->  166
```

Замечание. *– после параметров со звёздочкой могут идти только ключевые аргументы (можно указать одну звёздочку без имени параметра).*

Значения, возвращаемые функцией

– оператор **`return`** используется для возврата значения из функции, прекращения её работы и выхода из неё.

- значение выражения, стоящего после ключевого слова **`return`** будет возвращаться в качестве результата вызова функции;
- каждая функция содержит в неявной форме оператор **`return None`** в конце, если он не указан явно;
- **`None`** – специальный тип данных, обозначающий ничего.

Строки документации

Строка документации для функции – строка в первой логической строке этой функции

– *принято записывать* в форме многострочной строки, где
  - первая строка начинается с заглавной буквы и заканчивается точкой.
  - вторая строка оставляется пустой,
  - с третьей строки начинается подробное описание.

– *доступ* к строке документации функции можно получить с помощью атрибута этой функции __doc__.

Функция **`help()`** просто считывает атрибут __doc__ соответствующей функции
и выводит его на экран.

Строки документации *применимы* также к *модулям* и *классам*

Дополнительные замечания о функциях

– Python позволяет ***определять функцию внутри другой функции***:

Пример:

```
def outer(a, b):
    def inner(c, d):
        return c + d
    return inner(a, b)

print(outer(4, 7))    ->  11
```

– имена функций в Python являются переменными, содержащими адрес объекта типа функция, поэтому этот адрес можно присвоить другой переменной и ***вызвать функцию под другим именем***.

Пример:

```
def summa(x, y):
    return x + y

f = summa

v = f(10, 3)          # вызываем функцию с другим именем
print(v)    ->  13
```

– поскольку имя функции – обычная переменная, мы можем ***передать имя функции в качестве аргумента*** при вызове другой функции:

Пример:

```
def summa(x, y):
    return x + y

def func(f, a, b):
    return f(a, b)

v = func(summa, 10, 3)    # передаем summa() в качестве аргумента
print(v)   ->  13
```

– оператор `pass` используется в Python для обозначения ***пустого блока команд***.

Декораторы

Декоратор в Python – функция, которая принимает одну функцию в качестве аргумента и возвращает другую функцию.

Используется, когда необходимо модифицировать существующую функцию, не меняя её исходный код.

Ручное присваивание декоратора

Пример:

```
def sq_it(func):
    def new_func(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * result
    return new_func

def add_ints(a, b):
    return a + b

new_add_ints = sq_it(add_ints)    # ручное присваивание декоратора

print('add_ints(3, 6)=', add_ints(3, 6))
                                        ->  add_ints(3, 6)= 9
print('new_add_ints(3, 6)=', new_add_ints(3, 6))
                                        ->  new_add_ints(3, 6)= 81
```

Упрощённое декорирование

Пример:

```
def sq_it(func):
    def new_func(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * result
    return new_func

@sq_it                          # упрощённое декорирование
def add_ints(a, b):
    return a + b

print('add_ints(3, 6)=', add_ints(3, 6))
                                        ->  add_ints(3, 6)= 81
```

Измерение времени выполнения функции и задержка её выполнения

Пример:

```
# Декоратор, измеряющий время выполнения функции
import time
def timer(f):
    def tmp(*args, **kwargs):
        t = time.time()
        res = f(*args, **kwargs)
        print ("Время выполнения функции: %f" % (time.time()-t))
        return res
    return tmp

# Декоратор, выполняющий задержку функции на 1 с
def pause(f):
    def tmp(*args, **kwargs):
        time.sleep(1)
        return f(*args, **kwargs)
    return tmp

@timer
@pause
def func(x, y):
    return x + y

print(func(111, 222))          ->  Время выполнения функции: 1.000054
                                   333
```

Использование анонимной функции (функция lambda)

Пример:

```
# Вначале определим функцию edit_story():
def edit_story(words, func):
    for word in words:
        print(func(word))

# Создадим список
s = ['aaaaa', 'bbbbbb', 'cccccc']

# Вызов edit story(s, f) без использования анонимной функции

# Создадим функцию для обработка данного списка
def f(word):
    return word.capitalize() + '!'

edit_story(s, f)

# Вызов edit story(s, f) с использованием анонимной функции
edit_story(s, lambda word: word.capitalize() + '!')
```

```
-> Aaaaa!
   Bbbbbb!
   Cccccc!
```

Определение функции - генератора

– ранее встречались с функцией **`range()`**, которая генерирует последовательность целых чисел *без необходимости создания всей последовательности и её хранения в памяти*.

– при написании собственной функции-генератора необходимо вместо **`return`** использует инструкцию **`yield`**:

Пример:

```
def my_range(first=0, last=10, step=1):
    number = first
    while number < last:
        yield number
        number += step

ranger = my_range(1, 2, 0.25)
print(ranger)       -> <generator object my_range at 0x000001A76F293EB8>

for i in ranger:
    print(i)     -> 1
                    1.25
                    1.5
                    1.75
```