Алгоритмы и структуры данных в языке Python

Лекция 3

Понятие функции

Функция – фрагмент программы, у которого есть имя. Для выполнения этого фрагмента в любом месте программы достаточно указать его имя.

Функции объявляются с помощью инструкции def:

def *Имя_Функции* ([*Параметры*]):
[*''' Строка_Документирования '''*]
Тело_Функции
[**return** *Значение*]

Если тело функции не содержит инструкций, то внутри необходимо поместить оператор **pass**.

Примеры функций

```
def f1(x):
    '''вывод десятичного числа'''
    print("{:.2f}".format(x))

def f2(x):
    return "{:.2f}".format(x)

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

f1(1/3)
print("результат = " + f2(1/7))
z = summa(3, 0.5)
print(z)
s = summa("пример", "1")
print(s)
```

```
0.33
результат = 0.14
3.5
пример1
>>> f1(1/6)
0.17
>>> f2(1/6)
'0.17'
>>> f1.__doc__
'вывод десятичного числа'
```

Необязательные параметры

Чтобы сделать параметр необязательным, нужно в определении функции присвоить ему начальное значение.

Необязательные параметры должны следовать после обязательных.

```
import math
def L(x1=0, y1=0, x2=0, y2=1):
    return math.sqrt((x2-x1)**2+(y2-y1)**2)
```

```
>>> L()
1.0
>>> L(1,1,1,5)
4.0
>>> L(,,2,2)
SyntaxError: invalid syntax
>>> L(x2=2, y2=2)
2.8284271247461903
>>>
```

Именованные и позиционные аргументы

Передача аргументов при вызове функции возможна:

- без указания имени аргументов, при этом значения аргументов передаются строго в той последовательности, как эти аргументы были описаны в функции (позиционные аргументы);
- с явным указанием имени аргументов, при этом порядок перечисления аргументов не имеет значения (по ключу или именованные аргументы).

Если при вызове функции часть аргументов передается по ключу, а часть - позиционным способом, то в команде вызова сначала указываются позиционные аргументы, а затем те, что передаются по ключу.

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

X = summa(3, 5)
Y = summa(y=5, x=3)
Z = summa(3, y=5)
```

Функции в качестве аргументов

Инструкция def создает объект, имеющий тип function, и сохраняет ссылку на него в идентификаторе, указанном после инструкции def. Поэтому мы можем передать ссылку на функцию в качестве параметра другой функции.

```
def print_table(fun,a=0,b=1,h=0.1):
    x=a
    while x<=b:
        print("{0:5.2f}{1:7.2f}".format(x, fun(x)))
        x += h

def f3(x):
    return x+3
```

```
>>> print_table(f3, b=0.5)
 0.00   3.00
 0.10   3.10
 0.20   3.20
 0.30   3.30
 0.40   3.40
 0.50   3.50
>>>
```

Переменное число параметров в функции

Если перед параметром в определении функции указать символ * то функции можно будет передать произвольное число аргументов. Переданные параметры сохраняются в кортеже.

Если перед параметром в определении функции указать две звездочки, то функции можно будет передать произвольное число именованных аргументов. Переданные параметры сохраняются в словаре.

```
def summa ( *numb ) :
    s = 0
    for x in numb :
        s += x
    return s

X = summa(1,-2,5)
Y = summa(1,3,5,7,9,11)
```

```
def summa1 ( **d ) :
    s = 0
    for x in d :
        s += d[x]
    return s

X = summa1(a=1,b=-2,c=5)
Y = summa1(z1=1, z2=3)
```

Неизменяемые объекты в качестве аргументов

```
def f4(n, s, t):
    n = n + 5     #число
    s = s + "**" #строка
    t = t + (1,) #кортеж

def f5(n, s, t):
    n = n + 5     #число
    s = s + "**" #строка
    t = t + (1,) #кортеж
    return n, s, t
```

```
>>> a=10; b="10"; c=(3,2)
>>> a,b,c
(10, '10', (3, 2))
>>> f4(a,b,c)
>>> a,b,c
(10, '10', (3, 2))
>>> a,b,c = f5(a,b,c)
>>> a,b,c
(15, '10**', (3, 2, 1))
>>>
```

Списки в качестве аргументов

```
def f6(L):
    for i in range(len(L)):
        L[i] = L[i] + 1

def f7(L):
    L = [0]*5

def f8(L):
    L = L[:]
    L.sort()
    print(L)
```

```
>>> A=[5,1,2]; A
[5, 1, 2]
>>> f6(A); A
[6, 2, 3]
>>> f7(A); A
[6, 2, 3]
>>> f8(A)
[2, 3, 6]
>>> A
[6, 2, 3]
>>>
```

Словари в качестве аргументов

```
def f9(D):
    D['f'] = 1
    D['a'] = 25

def f10(D):
    D = {}

def f11(D):
    D.clear()

def f12(D):
    D = D.copy()
    #D = copy.deepcopy(D)
    del D['a']
    print("D=", D)
```

```
>>> D={'a':0, 'b':2}
>>> f9(D); D
{'b': 2, 'a': 25, 'f': 1}
>>> f10(D); D
{'b': 2, 'a': 25, 'f': 1}
>>> f11(D); D
{}
>>> D={'a':0, 'b':2}
>>> f12(D); D
D= {'b': 2}
{'b': 2, 'a': 0}
>>>
```

Аргументы функции содержатся в кортеже, списке или словаре

```
def f13(a,b,c):
    return (a+b+c)/3
```

```
>>> f13(2,3,4)
3.0
>>> v=[3,4,5]
>>> f13(*v)
4.0
>>> max(*v)
5
>>>
```

```
>>> t=(3,4,5)
>>> f13(*t)
4.0
>>> max(*t)
5
>>> d={'a':3,'b':4,'c':5}
>>> f13(**d)
4.0
```

Глобальные и локальные переменные

Глобальные переменные – переменные, созданные в программе вне функции.

Локальные переменные – переменные, которым внутри функции присваивается значение.

Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции выполняются с локальной переменной, а значение глобальной переменной не изменяется.

Чтобы изменять значение глобальной переменной внутри функции, нужно объявить ее внутри функции с помощью служебного слова **global**.

Пример использования глобальной и локальной переменной

```
def f14():
    print("f14: ", value)

def f15():
    value = 2
    print("f15: ", value)
    x = 5

def f16():
    global value
    value = 20

value = 1
f14()
f15()
print(value)
f16()
print(value)
```

```
f14:  1
f15:  2
1
20
>>>
```

Анонимные функции

Анонимная функция имеет вид:

lambda [*Параметры*] : *Возвращаемое_Значение*

В качестве значения лямбда-функции возвращают ссылку на объект-функцию, которую можно сохранить в переменной или передать в другую функцию в качестве параметра.

```
f1 = lambda x: (x+2)/5
f2 = lambda x, y=1: y/(x*x+1)

print(f1(3))
print(f2(2))
print(f2(2, 2))

print_table(lambda x: x+3, h=0.2)
```

Сортировка с использованием параметра key

```
L=['aaaa','cbb','ba','c']
L.sort()
print(*L)
L.sort(key = len)
print(*L)
print(L)
```

```
aaaa ba c cbb
c ba cbb aaaa
['c', 'ba', 'cbb', 'aaaa']
```

Использование модулей

Модулем в языке Python называется файл с программой.

Чтобы воспользоваться модулем, нужно выполнить инструкцию import.

```
import math
x = math.sqrt(2)
```

Или так:

```
import math as mm  #указываем псевдоним
x = mm.sqrt(2)
```

Или так:

```
from math import sqrt, exp, log, sin   #только нужные имена
x=sqrt(2)
```

Получить информацию о содержимом модуля:

```
>>> dir(math)
```

Использование собственных модулей

Содержимое модуля ***mod_test.py***:

```
def f1():
    print("функция f1")

x = 123
```

Содержимое модуля, в котором используется модуль mod_test:

```
import mod_test as mm

mm.f1()
print(mm.x)
```

Оба файла размещаем в одной папке.

После подключения ***mod_test.py*** внутри папки будет создана папка **__pycache__** с файлом ***mod_test.cpython-35.pyc***. Этот файл содержит скомпилированный байт-код исходного модуля.

Если его переименовать в ***mod_test.pyc***, то полученный файл можно использовать вместо файла ***mod_test.py***, например, при распространении программ.

Пути поиска модулей

Поиск подключаемого модуля выполняется в папках, указанных в списке sys.path модуля sys. Чтобы увидеть этот список, выполните

```
>>> import sys
>>> sys.path
```

При поиске список просматривается слева направо. Поиск прекращается после первого найденного модуля.

Из программы список sys.path можно изменить, используя его методы:

```
import sys
sys.path.append(r"C:\Мои модули") #в конец списка
sys.path.insert(0, r"C:\Мои модули") #в начало списка
```

Повторная загрузка модулей

Модуль загружается один раз при первой операции импорта.

Если в модуль вносились изменения, то его нужно перезагрузить. Иначе вы будете работать со старой версией модуля.

Для повторной загрузки модуля выполните команды:

```
>>> import imp
>>> reload(mm)
```

Или, если псевдоним не использовался,

```
>>> from imp import reload
>>> reload(mod_test)
```