Алексей Васильев

Язык программирования Python

Киев 2019

Лекция 6.

Функции

- Объявление и вызов функции
- Именованные аргументы функции
- Механизм передачи аргументов
- Значения аргументов по умолчанию
- Произвольное кол-во аргументов
- Локальные/глобальные переменные
- Вложенные функции
- Лямбда-функции
- Функция как аргумент и результат
- Рекурсия
- Декораторы функций
- Функции-генераторы
- Аннотации и документирование

Объявление/вызов

Функция: блок программного кода, у которого есть имя и который можно выполнить, указав (в правильном контексте) имя функции (вызов функции)

Описание функции:

```
def имя(аргументы):
    # Описание функции
```

Функция может возвращать значение, а может не возвращать значения. Если функция возвращает значение, используем инструкцию return, после которой указывается возвращаемое значение

Программа (Funcs.py)

```
# Функция для отображения букв из переданного
# аргументом текста:
def show(txt):
    # Преобразование текста в список и его сортировка:
    symbs=sorted(list(txt))
    # Отображение содержимого списка:
    print(symbs)
# Вызов функции:
show("Python")
# Функция для вычисления суммы квадратов натуральных чисел:
def sqsum(n):
    # Создание списка из квадратов натуральных чисел:
    nums=[k*k for k in range(1,n+1)]
    # Результат функции:
    return sum(nums)
# Переменная с числовым значением:
m=10
# Вызов функции для вычисления суммы квадратов чисел:
print("Сумма квадратов чисел от 1 до",str(m)+":",sqsum(m))
```

```
['P', 'h', 'n', 'o', 't', 'y']
Сумма квадратов чисел от 1 до 10: 385
```

Функции

```
# Функция без аргументов не возвращает результат:
def next_day():
    txt=input("Какой сегодня день недели? ")
    txt=txt.lower().strip()
    if txt=="понедельник":
       new_txt="вторник"
    elif txt=="вторник":
       new_txt="среда"
    elif txt=="среда":
       new_txt="четверг"
    elif txt=="четверг":
       new_txt="пятница"
    elif txt=="пятница":
       new_txt="суббота"
    elif txt=="суббота":
       new_txt="воскресенье"
    elif txt=="воскресенье":
       new_txt="понедельник"
    else:
       print("Нет такого дня недели!")
       return
    print(f"Завтра - {new_txt}")
# Функция без аргумента возвращает результат:
def get_name():
    name=input("Добрый день! Как Вас зовут? ")
    if name.strip(" .,:;!?_")=="":
       name="Мистер Икс"
    return name
# Функция без аргументов не возвращает результат:
def hello():
    name=get_name()
    print(f"Приятно познакомиться, {name}!")
    next_day()
# Вызов функции:
hello()
```

Программа (Funcs_2.py)

Переменные, которые получают свои значения в теле функции - локальные (доступны только в теле функции). Это же правило относится и к аргументам функций. Помимо локальных, в функциях могут использоваться и глобальные переменные

```
Добрый день! Как Вас зовут? Кот Матроскин
Приятно познакомиться, Кот Матроскин!
Какой сегодня день недели? Воскресенье
Завтра - понедельник
```

```
Добрый день! Как Вас зовут? _,! ...
Приятно познакомиться, Мистер Икс!
Какой сегодня день недели? Вторница
Нет такого дня недели!
```

Функции - 2

```
# Импорт функций:
from random import *
# Функция для отображения списков,
# множеств, текста и словарей:
def show(L,symb):
    for s in L:
        print(symb,s,sep="",end="")
    print(symb)
# Исходные данные:
A=[1,2,3,4,5]           # Список
B={'A','B','C','D'}     # Множество
C="Python"              # Текст
D={"A":1,"B":2,"C":3} # Словарь
# Вызов функции:
show(A,"|")
show(B,"/")
show(C,"*")
show(D,"#")
# Функция для создания списка чисел:
def get_nums(n,state):
    if type(n)!=int:
        return []
    if state:
        L=list(2*(k+1) for k in range(n))
    else:
        L=list(2*k+1 for k in range(n))
    return L
# Вызов функции:
print(get_nums(10,True))
```

Программа (Funcs_3.py)

```
print(get_nums(8,False))
print(get_nums(12.5,True))
# Функция для создания множества
# случайных букв:
def get_symbs(n):
    if n>10 or n<1:
        num=10
    else:
        num=n
    S=set()
    Nmin=ord("A")
    Nmax=ord("Z")
    while len(S)<num:
        S.add(chr(randint(Nmin,Nmax)))
    return S
# Инициализация генератора случайных чисел:
seed(2019)
# Вызов функции:
print(get_symbs(7))
print(get_symbs(-5))
print(get_symbs(15))
```

```
|1|2|3|4|5|
/D/A/C/B/
*P*y*t*h*o*n*
#A#B#C#
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[1, 3, 5, 7, 9, 11, 13, 15]
[]
{'U', 'Z', 'P', 'H', 'K', 'E', 'F'}
{'X', 'T', 'O', 'U', 'Y', 'C', 'Z', 'J', 'H', 'N'}
{'B', 'L', 'U', 'I', 'S', 'V', 'W', 'P', 'K', 'G'}
```

Имя функции

Программа (FuncName.py)

```
# Функции:
def alpha():
    print("Это Alpha!")
def bravo():
    print("Это Bravo!")
def hello():
    print("А это Hello!")
# Переменная с целочисленным значением:
num=123
# Вызов функций и проверка значения переменной:
print("Сначала было так:")
alpha()
bravo()
hello()
print("num =",num)
# Изменение значений:
alpha,bravo=bravo,alpha
num=hello
hello=321
# Вызов "функций" и проверка значения "переменной":
print("А стало так:")
alpha()
bravo()
num()
print("hello =",hello)
```

```
Сначала было так:
Это Alpha!
Это Bravo!
А это Hello!
num = 123
А стало так:
Это Bravo!
Это Alpha!
А это Hello!
hello = 321
```

Именованные аргументы

Программа (FuncArgs.py)

```
# Функция с тремя аргументами:
def show(first,second,third):
    print(f"[1] Первый аргумент - {first}")
    print(f"[2] Второй аргумент - {second}")
    print(f"[3] Третий аргумент - {third}")
# Вызов функции:
show(1,2,3)
show(second="B",third="C",first="A")
show(1,third=3,second=2)
```

Позиционная передача аргументов

Передача аргументов по ключу

Сначала указываются позиционные аргументы, а затем - аргументы по ключу

```
[1] Первый аргумент - 1
[2] Второй аргумент - 2
[3] Третий аргумент - 3
[1] Первый аргумент - A
[2] Второй аргумент - B
[3] Третий аргумент - C
[1] Первый аргумент - 1
[2] Второй аргумент - 2
[3] Третий аргумент - 3
```

Передача аргументов

```
# Функции:
def shift(val):
    print("Функция shift()")
    print("Исходное значение:",val)
    val=["A","B","C"]
    print("Конечное значение:",val)
def change(val):
    print("Функция change()")
    print("Исходное значение:",val)
    if type(val)==list:
        for k in range(len(val)):
            val[k]+=1
    else:
        val+=1
    print("Конечное значение:",val)
# Переменная:
num=100
# Список:
L=[10,20,30]
# Вызов функций:
print(f"Переменная num={num}")
change(num)
print(f"Переменная num={num}")
print(f"Список L={L}")
shift(L)
print(f"Список L={L}")
change(L)
print(f"Список L={L}")
```

Программа (Args.py)

Аргументы передаются по значению - для аргумента автоматически создается копия

```
Переменная num=100
Функция change()
Исходное значение: 100
Конечное значение: 101
Переменная num=100
Список L=[10, 20, 30]
Функция shift()
Исходное значение: [10, 20, 30]
Конечное значение: ['A', 'B', 'C']
Список L=[10, 20, 30]
Функция change()
Исходное значение: [10, 20, 30]
Конечное значение: [11, 21, 31]
Список L=[11, 21, 31]
```

Значения по умолчанию

```
# Функция со значениями аргументов по умолчанию:
def show(first,second="Bravo",third="Charlie"):
    print(f"[1] - {first}")
    print(f"[2] - {second}")
    print(f"[3] - {third}")
    print("-"*13)
# Вызов функции:
show("Alpha")
show("A","B","C")
show(10,20)
show(100,third=300)
show(third="третий",first="первый")
```

Программа (ArgsVal.py)

```
[1] - Alpha
[2] - Bravo
[3] - Charlie
-------------
[1] - A
[2] - B
[3] - C
-------------
[1] - 10
[2] - 20
[3] - Charlie
-------------
[1] - 100
[2] - Bravo
[3] - 300
-------------
[1] - первый
[2] - Bravo
[3] - третий
-------------
```

Сначала указываются обычные аргументы, а затем - аргументы со значением по умолчанию

Аргументы можно передавать по позиции или/и по имени, причем некоторые аргументы могут отсутствовать. Команда вызова функции должна быть такой, что по ней однозначно можно определить, какое значение получает каждый аргумент

Произвольное количество аргументов

```
# Функции с произвольным количеством аргументов:
def sqr_sum(*n):
    s=0
    for a in n:
        s+=a*a
    return s
def get_sum(*n):
    s=0
    for a in n:
        if type(a)==int:
            s+=a
    return s
def get_text(*t):
    return " ".join(t)
# Вызов функций:
print("Сумма квадратов:",sqr_sum(1,3,5))
print("Сумма квадратов:",sqr_sum(2,4,6,8,10))
print("Сумма чисел:",get_sum(2,"A",4,"B",6))
print("Сумма чисел:",get_sum(1,[2,3],4))
print("Сумма чисел:",get_sum())
print("Текст:",get_text("Всем","привет"))
print("Текст:",get_text("A","B","C","D"))
```

Программа (AnyArgs.py)

```
Сумма квадратов: 35
Сумма квадратов: 220
Сумма чисел: 12
Сумма чисел: 5
Сумма чисел: 0
Текст: Всем привет
Текст: A B C D
```

Произвольное количество аргументов 2

Когда мы вызываем функцию, которая может получать произвольное количество аргументов, технически аргументы в функцию передаются в виде кортежа. В этом несложно убедиться с помощью функции

```
def show(*val):
    print("Тип:",type(val))
    print("Аргумент:",val)
```

Кроме аргумента со "звездочкой", у функции могут быть и другие аргументы. Также некоторые аргументы могут иметь значение по умолчанию

```
# Функция с разными аргументами:
def my_sum(n,*a,txt="Сумма чисел"):
    s=0
    for m in range(len(a)):
        s+=a[m]**n
    print(txt+":",s)
# Вызов функции:
my_sum(1,100,200,300)
my_sum(2,10,20,30,txt="Сумма квадратов")
```

Программа (AnyArgs_2.py)

```
Сумма чисел: 600
Сумма квадратов: 1400
```

Локальные/глобальные переменные

```
# В функции используются глобальные
# и локальные переменные:
def myfunction():
    # Глобальные переменные:
    global A,B
    # Присваивание значений переменным:
    A="Альфа"
    B="Браво"
    D="Дельта"
    # Проверка значений:
    print("В функции: A =",A)
    print("В функции: B =",B)
    print("В функции: C =",C)
    print("В функции: D =",D)
# Глобальные переменные:
A="Alpha"
C="Charlie"
D="Delta"
# Проверка значений переменных:
print("До вызова функции: A =",A)
print("До вызова функции: C =",C)
print("До вызова функции: D =",D)
# Вызов функции:
myfunction()
# Проверка значений переменных:
print("После вызова функции: A =",A)
print("После вызова функции: B =",B)
print("После вызова функции: C =",C)
print("После вызова функции: D =",D)
```

Программа (Vars.py)

- Переменные, созданные в функции - **локальные** и доступны только в функции
- **Глобальные** переменные создаются вне тела функции
- Если в теле функции переменной присваивается значение, то такая переменная интерпретируется как локальная, даже если есть глобальная переменная с таким же именем
- Если значение переменной только считывается, то используется глобальная переменная
- Если мы хотим использовать в теле функции глобальную переменную (в том числе и присваивать ей значение), ее необходимо объявить с ключевым словом `global`

```
До вызова функции: A = Alpha
До вызова функции: C = Charlie
До вызова функции: D = Delta
В функции: A = Альфа
В функции: B = Браво
В функции: C = Charlie
В функции: D = Дельта
После вызова функции: A = Альфа
После вызова функции: B = Браво
После вызова функции: C = Charlie
После вызова функции: D = Delta
```

Вложенные функции

Программа (InnerFuncs.py)

```
# Функция с вложенной функцией:
def mysum(*a):
    # Список:
    txt=["чисел","квадратов","кубов"]
    # Вложенная функция:
    def calc(n):
        s=0
        for m in range(len(a)):
            s+=a[m]**n
        return s
    # Вызов вложенной функции:
    for k in range(len(txt)):
        print("Сумма",txt[k]+":",calc(k+1))
# Вызов функции:
mysum(1,3,5,7)
```

```
Сумма чисел: 16
Сумма квадратов: 84
Сумма кубов: 496
```

- Функцию можно описывать в другой функции: это вложенная или внутренняя функция
- Вложенная функция доступна внутри той функции, в которой она описана
- Вложенная функция имеет доступ к переменным во внешней функции
- Внешняя функция доступа к локальным переменным вложенной функции не имеет

Лямбда-функции

Лямбда-выражение: выражение, которое определяет функцию (без имени)

Синтаксис:

```
lambda аргументы: результат
```

```
Нечетные числа:
1 3 5 7 9 11 13 15 17 19
Степени двойки:
1 2 4 8 16 32 64 128 256 512
Квадраты чисел:
1 4 9 16 25 36 49 64 81 100
Вызов F(3,5): 15
Вызов f(3,5): 8
Вызов calc(3,5): 15
```

Программа (Lambda.py)

```
num=10
# Функция на основе лямбда-выражения:
L=lambda n: 2*n+1
# Проверка результата:
print("Нечетные числа:")
for k in range(num):
    print(L(k),end=" ")
# Новое значение:
L=lambda n: 2**n
# Проверка результата:
print("\nСтепени двойки:")
for k in range(num):
    print(L(k),end=" ")
# Прямой вызов лямбда-функции:
print("\nКвадраты чисел:")
for k in range(num):
    print((lambda x: x*x)(k+1),end=" ")
# Обычная функция:
def calc(x,y):
    return x+y
# Использование функции в лямбда-выражении:
F=lambda x,y: calc(x,y)
# Переменной присваивается имя функции:
f=calc
# Имени функции присваивается лямбда-выражение:
calc=lambda x,y: x*y
# Проверка результата:
print("\nВызов F(3,5):",F(3,5))
print("Вызов f(3,5):",f(3,5))
print("Вызов calc(3,5):",calc(3,5))
```

Функция как аргумент и результат

Программа (ArgRes.py)

```
x       1    2    3    4    5
A(x)    1    4    9   16   25
B(x)    1    8   27   64  125
C(x)    5    9   13   17   21
F(x->x*x)(5):  625
F(x->2*x+1)(5): 23
```

```
# Аргумент функции - функция (и два числа):
def display(f,a,b):
    for k in range(a,b+1):
        print("{0:4}".format(f(k)),end=" ")
    print()
# Результат функции - функция:
def mypow(n):
    return lambda x: x**n
# Аргументы функции - функции. Результат - функция:
def apply(f,h):
    def calc(x):
        return f(h(x))
    return calc
# Определение функций:
A=mypow(2)
B=mypow(3)
C=apply(lambda x: 2*x+1,lambda x: 2*x)
# Проверка результата:
print("x   ",end="")
display(lambda x: x,1,5)
print("A(x)",end="")
display(A,1,5)
print("B(x)",end="")
display(B,1,5)
print("C(x)",end="")
display(C,1,5)
# Определение функции:
F=lambda f: lambda x: f(f(x))
# Проверка результата:
print("F(x->x*x)(5): ",F(lambda x: x*x)(5))
print("F(x->2*x+1)(5):",F(lambda x: 2*x+1)(5))
```

Рекурсия

Рекурсия — способ описания функции, когда функция вызывает сама себя

```
Сумма чисел:
0 1 3 6 10 15 21 28 36 45 55 66
Числа Фибоначчи:
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Инверсия текста:
|n|o|h|t|y|P| |o|l|l|e|H|
Инверсия списка:
|5|4|3|2|1|
```

Программа (Recursion.py)

```
# Функция для вычисления суммы чисел:
def mysum(n):
    if n==0:
        return 0;
    else:
        return n+mysum(n-1)
# Функция для вычисления чисел Фибоначчи:
def fib(n):
    if n==1 or n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
# Функция для инверсного отображения
# текста/списка:
def show(txt):
    if len(txt)==0:
        print("|")
    else:
        print("|",txt[-1],end="",sep="")
        show(txt[:-1])
# Вызов функций:
print("Сумма чисел:")
for k in range(12):
    print(mysum(k),end=" ")
print("\nЧисла Фибоначчи:")
for k in range(15):
    print(fib(k+1),end=" ")
print("\nИнверсия текста:")
show("Hello Python")
print("Инверсия списка:")
show([1,2,3,4,5])
```

Декораторы

```
# Функции для использования в декораторах:
def A(h):
    return lambda x: h(x)*h(7-x)
def B(h):
    return lambda x,y: h(x,y)+h(y,x)
def C(h):
    return lambda x: h(x,10-x)
# Функции с декоратором:
@A
def f(x):
    return 2*x-1
@B
def F(x,y):
    return (8-x)*(y+1)
@C
def H(x,y):
    return x*y
# Проверка результата:
print("f(3) =",f(3))
print("F(5,7) =",F(5,7))
print("H(6) =",H(6))
```

Программа (Decorators.py)

Декоратор функции: инструкция вида `@A` перед описанием некоторой функции `h()`, причем `A` - имя функции, у которой аргумент - функция и результат - тоже функция. В результате функция `h()` переопределяется согласно правилу `h=A(h)`

```
f(3) = 35
F(5,7) = 30
H(6) = 24
```

Генераторы

Программа (Generators.py)

```
# Функции-генераторы:
def names():
    yield "Дядя Федор"
    yield "Пес Шарик"
    yield "Кот Матроскин"
def colors():
    L=["Красный","Желтый","Зеленый","Синий"]
    for clr in L:
        yield clr
def myrange(n):
    for k in range(n):
        yield 2*k+1
# Использование функций-генераторов:
print("Они из Простоквашино:")
for name in names():
    print(name)
print(list(names()))
R=colors()
print("Цветовой спектр:")
for r in R:
    print(r,end=" ")
print("\nЕще одна попытка...")
for r in R:
    print(r,end=" ")
print("Ничего нет? Это нормально.")
print("Нечетные числа:")
print(list(myrange(10)))
print(tuple(myrange(10)))
N=myrange(8)
A=list(N)
print("A =",A)
B=list(N)
print("B =",B)
for num in myrange(8):
    print(num,end=" ")
print()
```

```
Они из Простоквашино:
Дядя Федор
Пес Шарик
Кот Матроскин
['Дядя Федор', 'Пес Шарик', 'Кот Матроскин']
Цветовой спектр:
Красный Желтый Зеленый Синий
Еще одна попытка...
Ничего нет? Это нормально.
Нечетные числа:
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
A = [1, 3, 5, 7, 9, 11, 13, 15]
B = []
1 3 5 7 9 11 13 15
```

- Функции-генераторы позволяют создавать особые итерируемые объекты, которые можно затем использовать в операторах цикла или создавать на их основе списки и другие последовательности
- В функции-генераторе вместо ключевого слова `return` используют инструкцию `yield`

Аннотации

- Функция реализуется в виде специального объекта класса `function`
- У функции есть некоторые свойства, обусловленные тем, что по сути это объект

- В описании функции сразу после строки с названием функции можно размесить текст - ***текст документирования***. При выполнении кода функции он игнорируется. Его можно "прочитать", указав после имени функции (без круглых скобок) через точку поле `__doc__`

- Можно создавать специальные "пояснения" для аргументов функции и результата функции - ***аннотации***

- Аргумент с аннотацией:

```
аргумент: аннотация=значение
```

- Аннотация для результата:

```
def функция(аргументы)->аннотация:
    # Код функции
```

- Поле `__annotations__`: словарь, ключи — названия аргументов, значения элементов — аннотации. Аннотация к результату запоминается с ключом `"return"`

Аннотации - 2

Программа (Annotations.py)

```
# Функции с документированием:
def show(txt):
    "Это функция show() с одним аргументом."
    print("Единственный аргумент:",txt)
def display(a,b):
    "Это функция display() с двумя аргументами."
    print("[1] Первый аргумент:",a)
    print("[2] Второй аргумент:",b)
# Функция без документирования:
def hello():
    print("Всем привет!")
# Вызов функций и проверка документирования:
print(show.__doc__)
show("A")
print(display.__doc__)
display("B","C")
# Переменная ссылается на функцию:
f=show
# Вызов функций и проверка документирования:
print(f.__doc__)
f("D")
# Новый текст документирования для функции:
display.__doc__="Новый текст для display()"
# Проверка результата:
print(display.__doc__)
display("E","F")
# Создается документирование для функции:
hello.__doc__="Функция hello()"
# Проверка результата:
print(hello.__doc__)
hello()
```

```
Это функция show() с одним аргументом.
Единственный аргумент: A
Это функция display() с двумя
аргументами.
[1] Первый аргумент: B
[2] Второй аргумент: C
Это функция show() с одним аргументом.
Единственный аргумент: D
Новый текст для display()
[1] Первый аргумент: E
[2] Второй аргумент: F
Функция hello()
Всем привет!
```

Программа (Annotations_2.py)

```
Функция show()
{'txt': 'Текст', 'return': 'Результата нет'}
txt - Текст
return - Результата нет
```

```
# Функция с аннотациями:
def show(txt:"Текст"="Функция show()")->"Результата нет":
print(txt)
# Вызов функции:
show()
# Словарь аннотаций:
print(show.__annotations__)
# Аннотации:
for k in show.__annotations__:
    print(k,"-",show.__annotations__[k])
```

Задание - 1

Напишите программу с функцией, аргументом которой передается числовой список, а результатом является еще один список, в который включены только нечетные числа из списка-аргумента

Задание - 2

Напишите программу, в которой описывается функция с произвольным количеством числовых аргументов, а результатом возвращается список из трех элементов: среднее значение аргументов, максимальное значение среди аргументов и минимальное значение среди аргументов

Домашнее задание

[1] Напишите программу, в которой создается функция с одним текстовым аргументом и произвольным количеством целочисленных аргументов. Результатом является текст, сформированный из букв первого текстового аргумента. Целочисленные аргументы определяют индексы букв, которые нужно включить в текст-результат

[2] Напишите программу, в которой используется функция-генератор, создающая итерируемый объект со степенями двойки. Количество элементов определяется аргументом функции-генератора