

Функции в Python

Гейне М.А.

25.09.2024

Что такое функции?

- Функция - это связь или отображение между одним или несколькими входами и набором выходов
- Пример: $z = f(x)$
- В программировании **функция** - это самодостаточный блок кода, в котором заключена конкретная задача или связанная с ней группа задач

```
>>> s = 'foobar'
```

```
>>> id(s)
```

```
56313440
```

```
>>> a = ['foo', 'bar', 'baz', 'qux']
```

```
>>> len(a)
```

```
4
```

Что такое функции?

- Нам необходимо знать интерфейс функции:
 1. Какие **аргументы** (если таковые имеются) нужны.
 2. Какие **значения** (если таковые имеются) она возвращает
- В момент вызова функции исполнение переходит в область определения функции
- После выполнения функции исполнение продолжается с места, где функция была вызвана

Для чего нужны функции?

1. Абстракция и переиспользуемость
2. Модульность, разбиение на шаги
3. Разграничение пространств имён

Объявление и вызов функций

```
def <function_name>([<parameters>]):  
    <statement(s)>
```

```
<function_name>([<arguments>])
```

```
def f():  
    pass
```

Аргументы

Позиционные аргументы

```
def f(qty, item, price):  
    print(f'{qty} {item} cost ${price:.2f}')
```

Keyword аргументы

```
f(item='bananas', price=1.74, qty=6)
```

Параметры по умолчанию

```
def f(qty=6, item='bananas', price=1.74):  
    print(f'{qty} {item} cost ${price:.2f}')
```


Аргументы

В заключение:

- **Позиционные аргументы** должны совпадать по порядку и количеству с параметрами, объявленными в определении функции.
- **Keyword аргументы** должны совпадать по количеству с объявленными параметрами, но могут быть указаны в произвольном порядке.
- **Параметры по умолчанию** позволяют опустить некоторые аргументы при вызове функции

Что, если?...

```
def f(my_list=[]):  
    my_list.append('###')  
    return my_list  
  
>>> f(['foo', 'bar', 'baz'])  
['foo', 'bar', 'baz', '###']  
  
>>> f([1, 2, 3, 4, 5])  
[1, 2, 3, 4, 5, '###']  
  
>>> f()  
['###']
```

Что будет, если мы вызовем `f()` ещё раз?

Что, если?...

```
>>> f()  
['###', '###']  
>>> f()  
['###', '###', '###']
```

“ В Python значения параметров по умолчанию **определяются только один раз** при определении функции (то есть при выполнении оператора `def`). Значение по умолчанию не переопределяется при каждом вызове функции. ”

А как исправить?

```
def f(my_list=None):  
    if my_list is None:  
        my_list = []  
    my_list.append('###')  
    return my_list
```

Side effects

- Считается, что функция Python имеет побочный эффект, если она каким-либо образом изменяет окружение своего вызова. Например, изменяет состояние переданного аргумента
- Побочные эффекты как правило скрыты или не ожидаются, что ведёт к сложно отслеживаемым ошибкам. В общем случае их стоит избегать

Возвращение ~~на родину~~

- Функции в общем случае *возвращают* значения, а не меняют окружение
- Выражение `return` позволяет:
 - Завершить исполнение функции и передать управление в точку вызова
 - Передать данные в точку вызова
- Выход из функции произойдёт и после того, как будет выполнено последнее выражение функции, однако в этом случае функция не вернёт данных

Guards!

```
def f():  
    if error_cond1:  
        return  
    if error_cond2:  
        return  
    if error_cond3:  
        return  
  
    <normal processing>
```

Возврат значений

```
>>> def f():
...     return dict(foo=1, bar=2, baz=3)
...

>>> f()
{'foo': 1, 'bar': 2, 'baz': 3}
>>> f()['baz']
3

>>> def f():
...     return 'foo', 'bar', 'baz', 'qux'
...

>>> type(f())
<class 'tuple'>
>>> t = f()
>>> t
('foo', 'bar', 'baz', 'qux')

>>> a, b, c, d = f()
>>> print(f'a = {a}, b = {b}, c = {c}, d = {d}')
a = foo, b = bar, c = baz, d = qux
```


Ещё про аргументы

Argument tuple packing

```
def avg(a, b, c):  
    return (a + b + c) / 3  
def avg(*args):  
    total = 0  
    for i in args:  
        total += i  
    return total / len(args)  
>>> avg(1, 2, 3)  
2.0  
>>> avg(1, 2, 3, 4, 5)  
3.0  
def avg(*args):  
    return sum(args) / len(args)
```

Argument tuple unpacking

```
>>> def f(x, y, z):  
...     print(f'x = {x}')
```

```
...     print(f'y = {y}')
```

```
...     print(f'z = {z}')
```



```
>>> f(1, 2, 3)  
x = 1  
y = 2  
z = 3
```

```
>>> t = ('foo', 'bar', 'baz')  
>>> f(*t)  
x = foo  
y = bar  
z = baz
```

“ Можно применить не только к кортежам, но также к спискам и наборам ”

Argument Dictionary Packing

```
>>> def f(**kwargs):  
...     print(kwargs)  
...     print(type(kwargs))  
...     for key, val in kwargs.items():  
...         print(key, '->', val)  
...  
  
>>> f(foo=1, bar=2, baz=3)  
{'foo': 1, 'bar': 2, 'baz': 3}  
<class 'dict'>  
foo -> 1  
bar -> 2  
baz -> 3
```

Argument Dictionary Unpacking

```
>>> def f(a, b, c):  
...     print(F'a = {a}')
```

```
...     print(F'b = {b}')
```

```
...     print(F'c = {c}')
```

```
...
```



```
>>> d = {'a': 'foo', 'b': 25, 'c': 'qux'}
```

```
>>> f(**d)
```

```
a = foo
```

```
b = 25
```

```
c = qux
```

```
>>> def f(a, b, *args, **kwargs):  
...     print(F'a = {a}')
```

...

```
...     print(F'b = {b}')
```

...

```
...     print(F'args = {args}')
```

...

```
...     print(F'kwargs = {kwargs}')
```

...


```
>>> f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)  
a = 1  
b = 2  
args = ('foo', 'bar', 'baz', 'qux')  
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

Keyword-only arguments

```
>>> def concat(prefix='-> ', *args):  
...     print(f'{prefix}{".".join(args)}')  
...  
>>> concat(prefix='//', 'a', 'b', 'c')  
File "<stdin>", line 1  
SyntaxError: positional argument follows keyword argument  
>>> concat('a', 'b', 'c', prefix='... ')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: concat() got multiple values for argument 'prefix'
```

Keyword-only arguments

```
>>> def concat(*args, prefix='-> '):  
...     print(f'{prefix}{".".join(args)}')
```

```
...
```

```
>>> concat('a', 'b', 'c', prefix='... ')  
... a.b.c
```

```
>>> def concat(*args, prefix):  
...     print(f'{prefix}{".".join(args)}')
```

```
...
```

```
>>> concat('a', 'b', 'c', prefix='... ')  
... a.b.c
```

```
>>> concat('a', 'b', 'c')
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: concat() missing 1 required keyword-only argument: 'prefix'
```

Keyword-only arguments

```
>>> def oper(x, y, *, op='+'):
...     if op == '+':
...         return x + y
...     elif op == '-':
...         return x - y
...     elif op == '/':
...         return x / y
...     else:
...         return None
...

>>> oper(3, 4, op='+')
7
>>> oper(3, 4, op='/')
0.75

>>> oper(3, 4, "I don't belong here")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: oper() takes 2 positional arguments but 3 were given

>>> oper(3, 4, '+')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: oper() takes 2 positional arguments but 3 were given
```


Positional-only arguments

```
>>> # This is Python 3.8
>>> def f(x, y, /, z):
...     print(f'x: {x}')
...     print(f'y: {y}')
...     print(f'z: {z}')
...

>>> f(1, 2, 3)
x: 1
y: 2
z: 3

>>> f(1, 2, z=3)
x: 1
y: 2
z: 3

>>> f(x=1, y=2, z=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got some positional-only arguments passed as keyword arguments:
'x, y'
```

```
>>> # This is Python 3.8
>>> def f(x, y, /, z, w, *, a, b):
...     print(x, y, z, w, a, b)
...

>>> f(1, 2, z=3, w=4, a=5, b=6)
1 2 3 4 5 6

>>> f(1, 2, 3, w=4, a=5, b=6)
1 2 3 4 5 6
```

Docstrings

```
>>> def foo(bar=0, baz=1):  
...     """Perform a foo transformation.  
...  
...     Keyword arguments:  
...     bar -- magnitude along the bar axis (default=0)  
...     baz -- magnitude along the baz axis (default=1)  
...     """  
...     <function_body>  
...
```

Аннотации функций

```
>>> def f(a: '<a>', b: '<b>') -> '<ret_value>':  
...     pass  
...  
>>> f.__annotations__  
{ 'a': '<a>', 'b': '<b>', 'return': '<ret_value>' }  
  
>>> def f(a: int, b: str) -> float:  
...     print(a, b)  
...     return(3.5)  
...  
>>> f.__annotations__  
{ 'a': <class 'int'>, 'b': <class 'str'>, 'return': <class 'float'> }  
>>> f('foo', 2.5)  
foo 2.5  
(1, 2, 3)
```

Рекурсия

- **Dictionary.com:** Действие или процесс возвращения или бегства назад.
- **Викисловарь:** Акт определения объекта (обычно функции) в терминах самого этого объекта
- **Свободный словарь:** Метод определения последовательности объектов, таких как выражение, функция или множество, где дается некоторое количество начальных объектов и каждый последующий объект определяется в терминах предыдущих объектов

Рекурсия

- Многие задачи в программировании возможно решать без рекурсий, однако отдельные задачи, особенно основанные на вложенности и самоопределении, намного эффективнее решаются рекурсией
- Пример: обход древовидных структур

Рекурсия

- Стоит учитывать:
 - Для некоторых задач рекурсивное решение, хотя и возможно, будет скорее неудобным, чем элегантным.
 - Рекурсивные реализации часто занимают больше памяти, чем нерекурсивные.
 - В некоторых случаях использование рекурсии может привести к замедлению времени выполнения.

Рекурсия

```
def function():  
    x = 10  
    function()
```

```
>>> function()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 3, in function  
  File "<stdin>", line 3, in function  
  File "<stdin>", line 3, in function  
  [Previous line repeated 996 more times]  
RecursionError: maximum recursion depth exceeded
```


Рекурсия

Рекурсивные функции обычно следуют шаблону:

- Существует один или несколько базовых случаев, которые решаются напрямую, без необходимости дальнейшей рекурсии.
- Каждый рекурсивный вызов постепенно приближает решение к базовому случаю.

Анонимные функции

```
lambda x, y: x + y
```

```
>>> (lambda x, y: x + y)(2, 3)
```

```
5
```

```
>>> high_ord_func = lambda x, func: x + func(x)
```

```
>>> high_ord_func(2, lambda x: x * x)
```

```
6
```

Декораторы

```
def decorator(func):  
    def wrapper():  
        print("Something is happening before the function is called.")  
        func()  
        print("Something is happening after the function is called.")  
    return wrapper  
  
def say_whee():  
    print("Whee!")  
  
say_whee = decorator(say_whee)  
  
>>> say_whee()  
Something is happening before the function is called.  
Whee!  
Something is happening after the function is called.
```

Декораторы

```
@decorator  
def say_whee():  
    print("Whee!")
```

“ Проще говоря, декоратор оборачивает функцию, изменяя ее поведение, и возвращает эту измененную функцию. ”

Спасибо за внимание! :)