

Функции в Python

Гейне М.А.

26.09.2025

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
- Основная идея — инкапсуляция логики для многократного использования.

```
>>> 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

Функции как объекты первого класса

- Функции ведут себя так же, как и любые другие объекты (числа, строки, списки)
 - Функцию можно присвоить переменной.
 - Функцию можно передать в качестве аргумента другой функции.
 - Функцию можно вернуть из другой функции.
 - Функцию можно хранить в структурах данных (списках, словарях).

```
def say_hello(name):
    return f"Hello, {name}"

# 1. Присваиваем функцию переменной
greet = say_hello
print(greet("World")) # Вывод: Hello, World

# 2. Передаём функцию как аргумент
def process_greeting(func, name):
    print(f"Executing greeting: {func(name)}")

process_greeting(say_hello, "Alice") # Вывод: Executing greeting: Hello, Alice
```

Для чего нужны функции?

1. Абстракция и переиспользуемость (DRY - Don't Repeat Yourself)
 - Определите логику один раз и используйте её многократно
2. Модульность и декомпозиция
 - Разбивайте сложную задачу на более мелкие, управляемые подзадачи. Программа становится проще для понимания и отладки.
3. Разграничение пространств имён
 - Переменные, созданные внутри функции (локальные), не конфликтуют с переменными вне её.

Объявление и вызов функций

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

```
<function_name>(<arguments>)
```

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

- **Параметры** — это переменные, перечисленные в определении функции.
- **Аргументы** — это конкретные значения, которые передаются в функцию при её вызове.

Объявление и вызов функций

```
def add(x, y): # x и y - параметры
    result = x + y
    return result

sum_result = add(10, 20) # 10 и 20 - аргументы
```

Аргументы

Позиционные аргументы

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

Именованные (Keyword) аргументы

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")  
  
describe_pet(pet_name="Willie", animal_type="dog")
# I have a dog named Willie.
```

Параметры по умолчанию

```
def describe_pet(pet_name, animal_type="dog"):  
    print(f"I have a {animal_type} named {pet_name}.")  
  
describe_pet("Willie") # animal_type='dog' по умолчанию  
# I have a dog named Willie.  
  
describe_pet("Whiskers", "cat") # Переопределяем значение  
# I have a cat named Whiskers.
```

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
- **Важно:** Параметры со значениями по умолчанию должны идти после параметров без них.

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

```
def get_user_info(user_id):
    if user_id < 0:
        return None # Явный возврат при ошибке
    # ... логика поиска пользователя ...
    return {'id': user_id, 'name': 'John Doe'}
```

```
user = get_user_info(-1)
print(user) # Вывод: None
```

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
```

Возврат значений

```
>>> 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
```

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
```

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
...     else:
...         return None
...
>>> oper(3, 4, op='+')
7

>>> 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
```

Пространства имён и правило LEGB

Когда вы обращаетесь к переменной, Python ищет её в 4 областях (пространствах имён) в строгом порядке:

1. **L (Local)** — Локальная область: внутри текущей функции.
2. **E (Enclosing)** — Замыкающая область: в локальных областях всех внешних функций (для вложенных функций).
3. **G (Global)** — Глобальная область: на уровне модуля (файла .py).
4. **B (Built-in)** — Встроенная область: предопределённые в Python имена (`len`, `print`, `str` ...).

Пример LEGB

```
x = "global" # G

def outer():
    x = "enclosing" # E

    def inner():
        x = "local" # L
        print(x) # Находит 'local' на шаге L

    inner()
    print(x) # Находит 'enclosing' на шаге L (для outer)

outer()
print(x) # Находит 'global' на шаге G
```

Вложенные функции и Замыкания (Closures)

- **Вложенная функция** — это функция, определённая внутри другой функции.
- **Замыкание (closure)** — это особый вид вложенной функции. Она "помнит" и имеет доступ к переменным из той области, в которой она была создана, даже после того, как внешняя функция завершила свою работу.

Вложенные функции и Замыкания (Closures)

```
def multiplier_factory(n): # Внешняя функция
    # n находится в Enclosing области для `multiplier`
    def multiplier(x): # Внутренняя функция
        return x * n # Использует `n` из своего окружения
    return multiplier # Возвращаем саму функцию, а не результат её вызова

# Создаем функции-фабрики
double = multiplier_factory(2) # n=2 "запомнилось" в замыкании
triple = multiplier_factory(3) # n=3 "запомнилось" в замыкании

# Используем созданные функции
print(double(5)) # Вывод: 10
print(triple(5)) # Вывод: 15
```

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

@my_decorator
def say_whee():
    print("Whee!")

# Этот код полностью эквивалентен:
# say_whee = my_decorator(say_whee)

>>> say_whee()
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
```

Проблема: Декораторы "крадут" метаданные

Декорированная функция на самом деле является wrapper - ом.

```
@my_decorator
def say_whee():
    """Простая функция, которая кричит Whee!
    print("Whee!")

print(say_whee.__name__) # 'wrapper', а не 'say_whee'
print(say_whee.__doc__) # None, а не docstring
```

Решение: `functools.wraps`

`@functools.wraps` — это декоратор, который помогает сохранить метаданные исходной функции.

```
import functools

def my_decorator(func):
    @functools.wraps(func) # <-- Вот магия!
    def wrapper(*args, **kwargs):
        print("До вызова...")
        result = func(*args, **kwargs)
        print("После вызова...")
        return result
    return wrapper

@my_decorator
def say_whee():
    """Простая функция, которая кричит Whee!"""
    print("Whee!")

    print(say_whee.__name__) # 'say_whee'
    print(say_whee.__doc__) # 'Простая функция, которая кричит Whee!'
```

Практический пример: декоратор-таймер

```
import time
import functools

def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Функция {func.__name__!r} выполнилась за {run_time:.4f} с")
        return result
    return wrapper

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])  
  
waste_some_time(1)
waste_some_time(10)
```

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
```

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

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.

Рекурсия

```
# Классический пример: вычисление факториала
def factorial(n):
    # Базовый случай
    if n == 0 or n == 1:
        return 1
    # Шаг рекурсии
    else:
        return n * factorial(n - 1)

print(factorial(5)) # 120 (5 * 4 * 3 * 2 * 1)
```

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.
 - Python имеет ограничение на максимальное количество вложенных вызовов (обычно около 1000) для предотвращения переполнения стека.
 - В отличие от некоторых других языков, Python **не выполняет** оптимизацию хвостовой рекурсии.

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
```

Аннотации типов и Docstrings

Docstring — это строка в начале функции, которая описывает её назначение, аргументы и возвращаемое значение.

Доступна через `help(func)` или `func.__doc__`.

Аннотации типов (Type Hints) — способ указать ожидаемые типы аргументов и возвращаемого значения. Они не влияют на исполнение кода, но используются статическими анализаторами (`труу`), IDE и для самодокументирования.

Аннотации типов и Docstrings

```
from typing import List, Optional

def find_user(user_id: int, users: List[dict]) -> Optional[dict]:
    """
    Находит пользователя в списке словарей по его ID.

    Args:
        user_id: Идентификатор искомого пользователя.
        users: Список словарей, где каждый словарь представляет пользователя.

    Returns:
        Словарь с данными пользователя или None, если пользователь не найден.
    """
    for user in users:
        if user.get("id") == user_id:
            return user
    return None

help(find_user)
```

Спасибо за внимание! :)

