

ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON

Функциональное программирование в Python

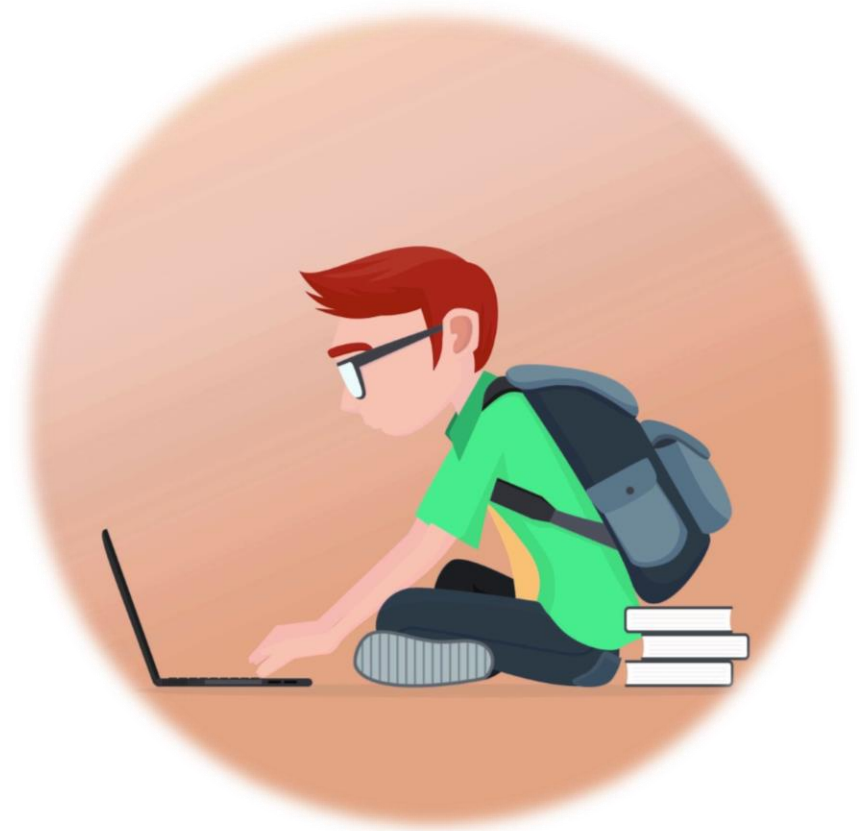


Крашенинников Роман Сергеевич

Главный специалист отдела системного администрирования
РХТУ им. Д.И. Менделеева, ведущий программист кафедры
информационных компьютерных технологий

ТЕМЫ

- Функции
- Аргументы функции
- Lambda функции
- Генераторы
- Декораторы



ПОЛЕЗНЫЕ РЕСУРСЫ

- [Функции официальная документация](#)
- [Функции](#)
- [Генераторы vs функции](#)
- [Декораторы](#)



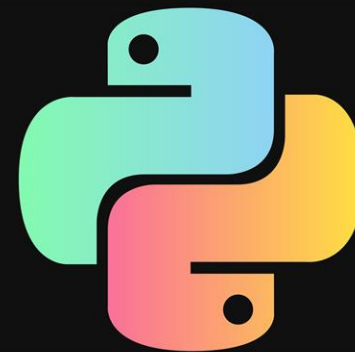
ФУНКЦИИ

Функция в python - объект, принимающий аргументы и возвращающий значение. Обычно функция определяется с помощью инструкции def.

Функции используются для организации определенного функционала в одну конструкцию, что облегчает его многократное использование и уменьшает количество работы необходимое для внесения модификаций.

Общий вид функций

```
1 def <имя функции>(<аргументы>):  
2     <тело функции>
```



ФУНКЦИИ (КЛАССИФИКАЦИЯ)

Функции



```
def func1():  
    print(5)
```

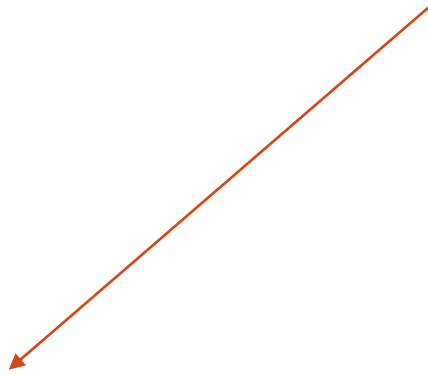
Процедуры
Ничего не возвращают

```
def func2():  
    return 5
```

Функции
Возвращают значение

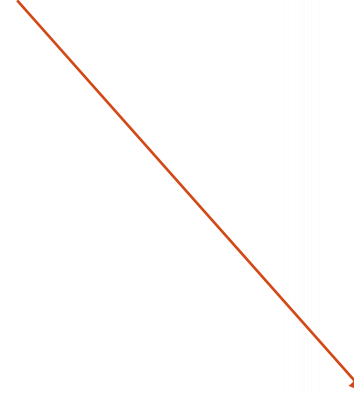
ФУНКЦИИ (КЛАССИФИКАЦИЯ)

Функции



```
def func1():  
    for i in range(10):  
        print(i)
```

Без аргументов

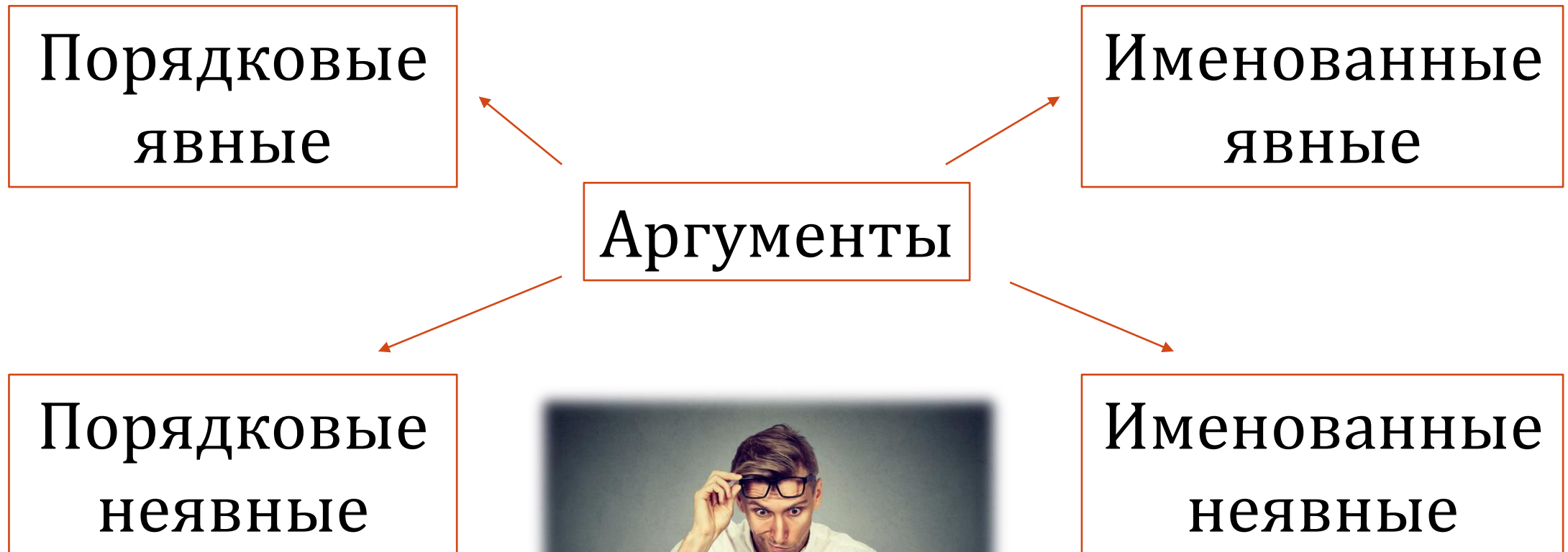


```
def func2(a, b):  
    return a + b
```

С аргументами



АРГУМЕНТЫ ФУНКЦИИ



АРГУМЕНТЫ ФУНКЦИИ

Порядковые явные

```
1 def func(a, b):  
2     return (a * b) + b  
3  
4 res1 = func(5, 2)  
5 # Можно явно указывать какому  
6 # аргументу присваивается значение  
7 res2 = func(5, b=2)  
8 res3 = func(b=2, a=5)  
9 print(res1)  
10 print(res1 == res2 == res3)
```

Наиболее простой тип аргументов, перечисляются через запятую (все указываются при вызове)

Именованные явные

```
1 def func(a, b, c=3):  
2     return (a * b) + c  
3  
4 res1 = func(2, 4)  
5 print(res1)  
6 res2 = func(2, 4, 5)  
7 print(res2)  
8 res3 = func(2, c=5, b=4)  
9 print(res3)
```

При определении функции им присваивается значение которое будет использоваться, если при вызове функции оно не будет перезаписано явно

АРГУМЕНТЫ ФУНКЦИИ

Порядковые неявные

```
1 def func(a, b, *c):  
2     print(a)  
3     print(b)  
4     print(c)  
5  
6 func(1, 2, 3, 4, 5)
```

Порядковые неявные аргументы принимают все значения которые переданы в функцию без имени и не описаны явно при определении функции

Именованные неявные

```
1 def func(a, b, c=3, **d):  
2     print(a)  
3     print(b)  
4     print(c)  
5     print(d)  
6  
7 func(1, b=2, c=3, d=4, e=5)
```

Именованные неявные аргументы принимают все значения которые переданы в функцию по имени и не описаны явно при определении функции

АРГУМЕНТЫ ФУНКЦИИ



```
1 def func(a, b, *c, d=4, e=5, **f):  
2     print(a)  
3     print(b)  
4     print(c)  
5     print(d)  
6     print(e)  
7     print(f)  
8  
9 func(1, 2, 3, 4, e=6, f=7, g=8)
```

Наиболее часто порядковые неявные аргументы называют `*args`, а именованные неявные - `**kwargs`

LAMBDA ФУНКЦИИ

Анонимные функции (lambda функции) используются когда нецелесообразно реализовывать целую функцию. Использование анонимных функций сильно затрудняет отладку кода, поэтому их стоит использовать крайне редко в определенных случаях. Например, lambda функции используются при передаче в качестве аргумента в другую функцию.

Пример lambda функций

```
1 f = lambda a, b: a * b
2 res1 = f(5, 2)
3 print(res)
4 res2 = (lambda a, b: a + b)(5, 2)
5 print(res2)
```



ГЕНЕРАТОРЫ

Генератор в Python — это функция с уникальными возможностями. Она позволяет приостановить или продолжить работу. Генератор возвращает итератор, по которому можно проходить пошагово, получая доступ к одному значению с каждой итерацией. Генератор создается по принципу обычной функции. Отличие заключается в том, что вместо `return` используется инструкция `yield`. Она уведомляет интерпретатор Python о том, что это генератор, и возвращает итератор.

Общий вид генератора

```
def gen_func(args):  
    ...  
    while [cond]:  
        ...  
        yield [value]
```



ГЕНЕРАТОРЫ

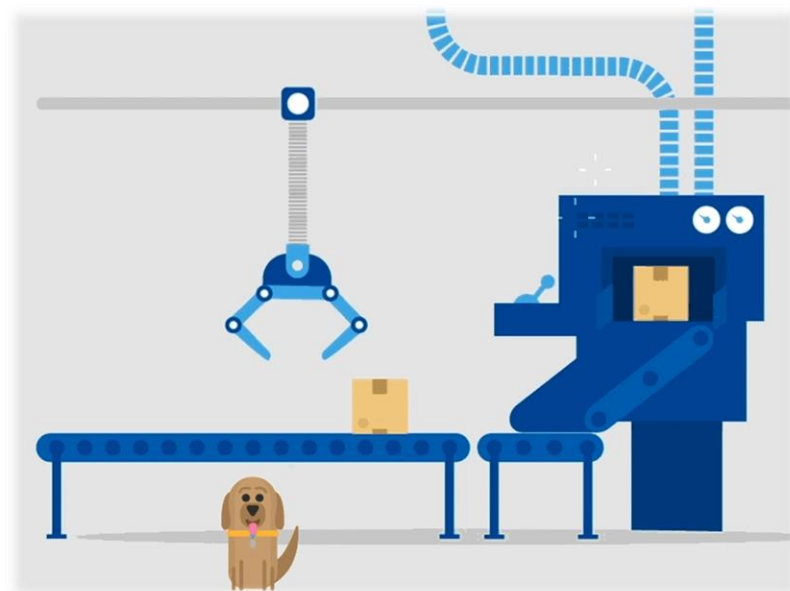
Return всегда является последней инструкцией при вызове функции, в то время как `yield` временно приостанавливает исполнение, сохраняет состояние и затем может продолжить работу позже. “Перебирать” генератор можно при помощи `next` или `for`.

Применение `next`

```
def generator1(end=5):  
    n = 0  
    while n <= end:  
        yield n ** 2  
        n += 1  
  
g = generator1(5)  
  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))  
print(next(g))
```

Применение `for`

```
def generator1(end=5):  
    n = 0  
    while n <= end:  
        yield n ** 2  
        n += 1  
  
for g in generator1(5):  
    print(g)
```



ГЕНЕРАТОРЫ

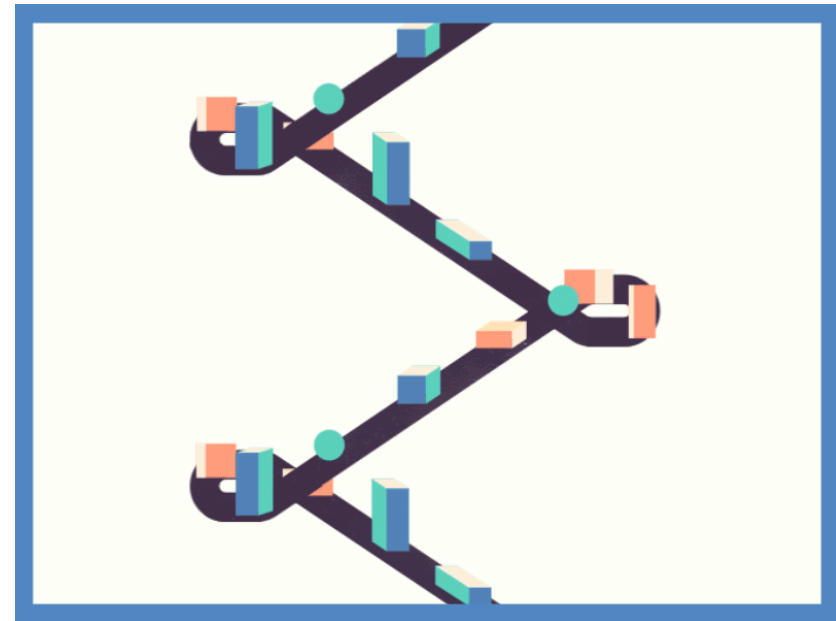
Python позволяет писать выражения генератора для создания анонимных функций генератора. Процесс напоминает создание лямбда-функций для создания анонимных функций. Синтаксис похож на используемый для создания списков с помощью цикла `for`. Однако там применяются квадратные скобки, а здесь — круглые.

Анонимный генератор

```
alist = [4, 16, 64, 256]

out = (a**(1/2) for a in alist)

print(next(out))
print(next(out))
print(next(out))
print(next(out))
print(next(out))
```



ДЕКОРАТОРЫ

Декораторы предназначены для модификации функций с помощью других функций.

Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

```
def my_decorator(func):  
    def wrap():  
        print("+++++")  
        func()  
        print("+++++")  
    return wrap  
  
@my_decorator  
def my_func():  
    print("Hello!")
```

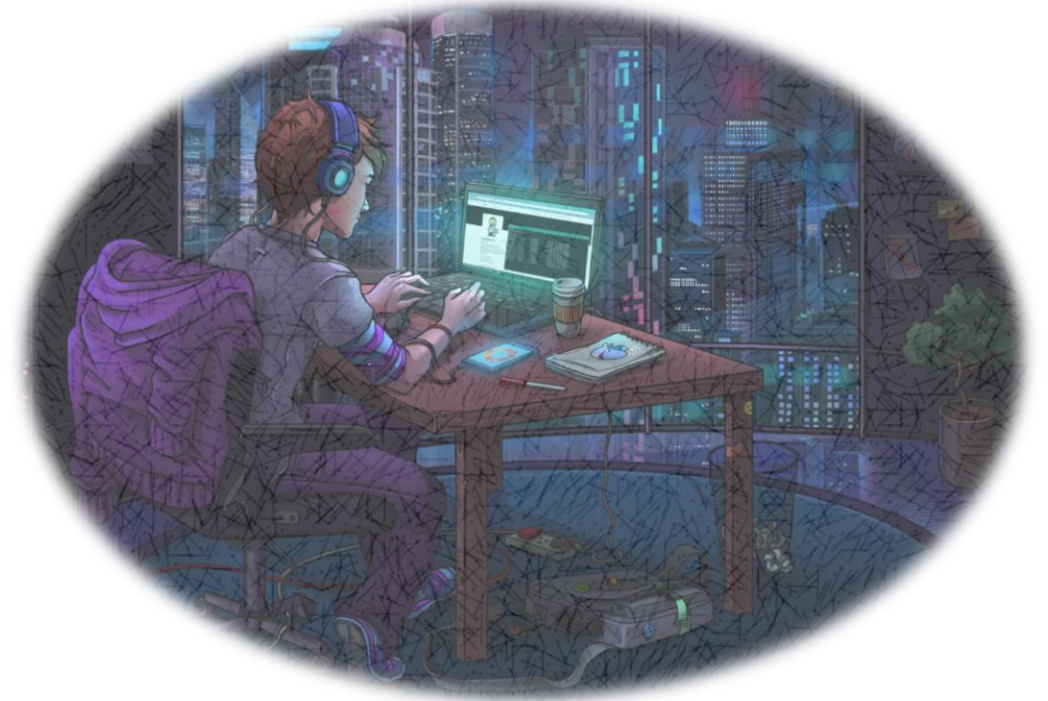
Декоратор

Декорируемая функция



ЗАДАНИЕ ДЛЯ САМОКОНТРОЛЯ

- Напишите функцию, которая принимает строку из одного или нескольких слов и возвращает ту же строку, но с перевернутыми всеми словами из пяти или более букв. **Hey fellow warriors -> Hey wollef sroirraw**
- Создать генератор N первых чисел Фибоначчи.
- Создать декоратор, для вывода аргументов и результата выполнения функции с двумя переменными.



СПАСИБО ЗА ВНИМАНИЕ!