GeekBrains

Погружение в Python

Урок 9
Декораторы

Содержание урока

План курса

1. Основы Python
2. Простые типы данных
3. Коллекции
4. Функции
5. Итераторы и генераторы
6. Модули
7. Файлы и файловая система
8. Сериализация
9. Декораторы
10. ООП. Начало
11. ООП. Особенности Python
12. ООП. Финал
13. Исключения
14. Основы тестирования
15. Обзор стандартной библиотеки Python

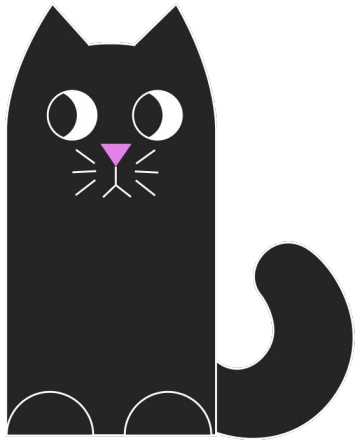
Что будет на уроке сегодня

- 📌 Разберём замыкания в программировании
- 📌 Изучим возможности Python по созданию декораторов
- 📌 Узнаем как создавать декораторы с параметрами
- 📌 Разберём работу некоторых декораторов из модуля functools

Что такое замыкания?

Области видимости и функции первого класса

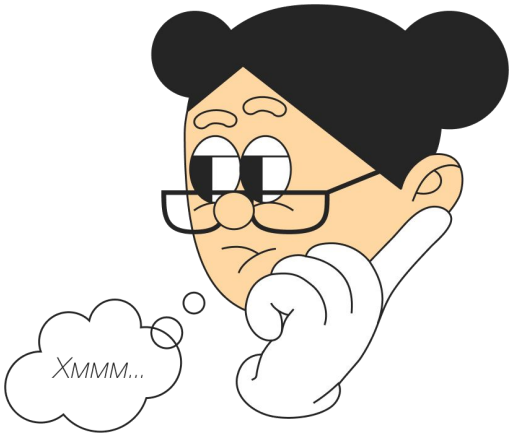
Замыкание (англ. closure) в программировании — функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами.



Замыкаем функцию с параметрами

Сохраняем результат первой функции в переменную

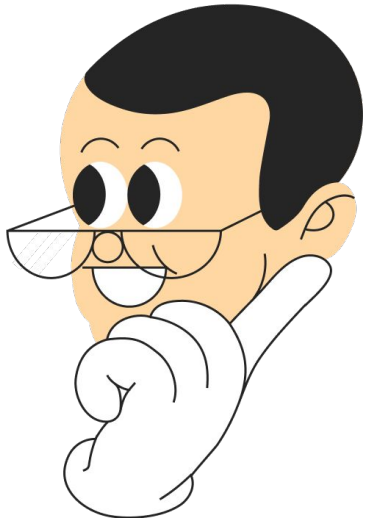
```
def one(a):
    def two(b):
        ...
        return result

    return two


closure = one(data)
```



Замыкаем изменяемые и неизменяемые объекты

Вспоминаем mutable и immutable

- **nonlocal immutable**
  явно указываем на необходимость изменения неизменяемого типа данных за пределами функции

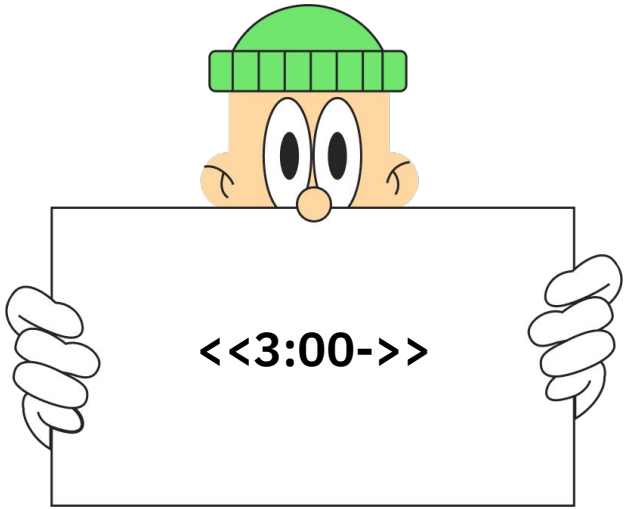


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Замыкания

```
from typing import Callable

def main(x: int) -> Callable[[int], dict[int,
int]]:
    d = {}

    def loc(y: int) -> dict[int, int]:
        for i in range(y):
            d[i] = x ** i
        return d

    return loc

small = main(42)
big = main(73)
print(small(7), big(7), small(3))
```

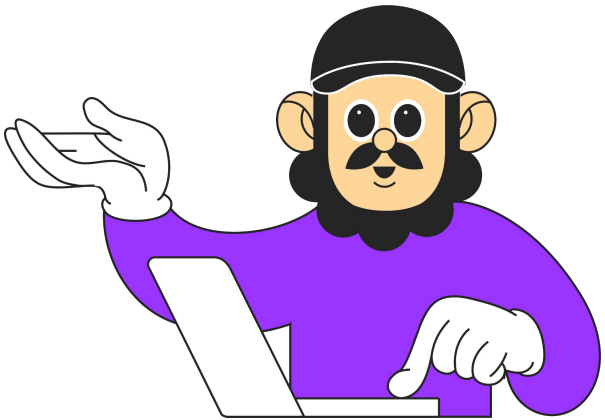


Простой декоратор без параметров

Передача функции в качестве аргумента

Функция может принимать другую функцию в качестве параметра

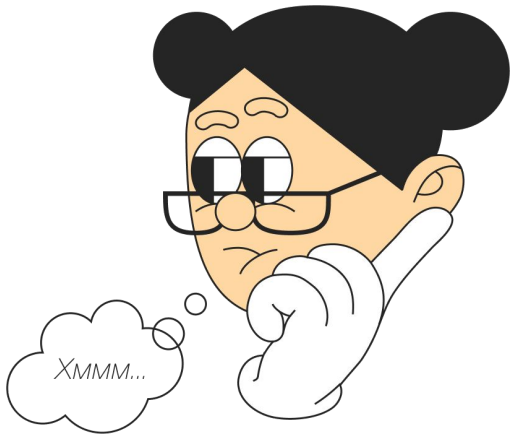
```
def main(func):
    def wrapper(*args, **kwargs):
        ...
        result = func(*args, **kwargs)
        ...
        return result
    return wrapper

def my_func(data):
    ...
    return wrapper

deco = main(my_func)
```



Синтаксический сахар Python, @

Символ @ является более простым способом создать замыкание

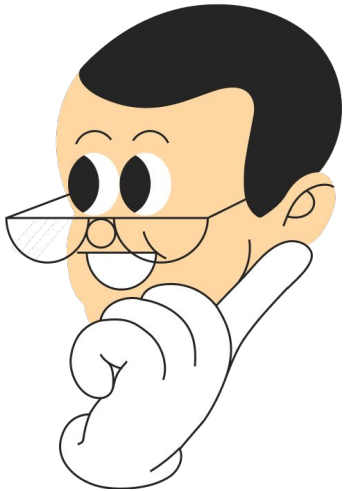
```
def main(func):
    def wrapper(*args, **kwargs):
        ...
        result = func(*args, **kwargs)
        ...
        return result
    return wrapper

@main
def my_func(data):
    ...
    return wrapper
```



Множественное декорирование

Функция может быть декорирована одновременно несколькими декораторами

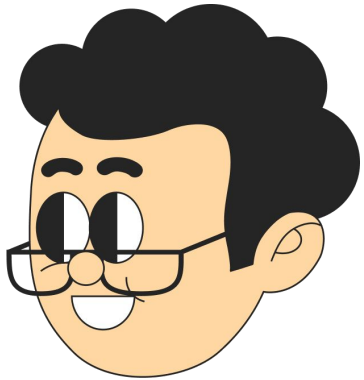
```
@deco_c
@deco_b
@deco_a
def my_func(data):
    ...
    return wrapper
```



Дополнительные переменные в декораторе

Аналогично замыканию переменных в функции,
декоратор может замыкать переменные в себе

```
def main(func):
    closure = []

    def wrapper(*args, **kwargs):
        ...
        result = func(*args, **kwargs)
        ...
        return result

    return wrapper
```

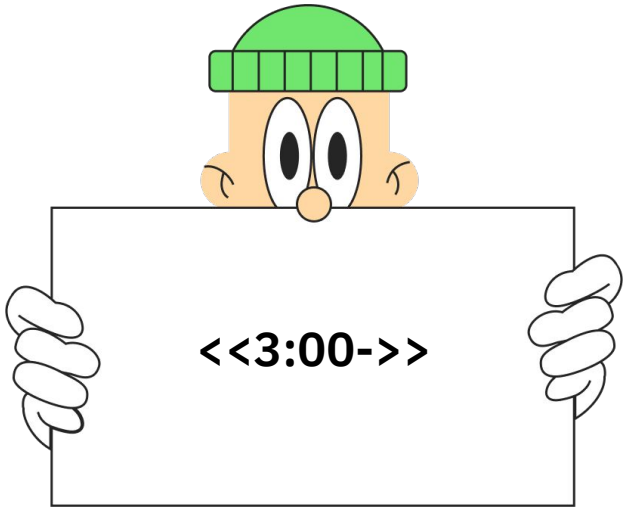


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Декоратор без параметров

```
import random
from typing import Callable

def cache(func: Callable):
    _cache_dict = {}
    def wrapper(*args):
        if args not in _cache_dict:
            _cache_dict[args] = func(*args)
        return _cache_dict[args]
    return wrapper

@cache
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)

print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
```



Декоратор
с параметрами

Декоратор с параметрами

Три уровня вложенности позволяют передавать аргументы в декоратор

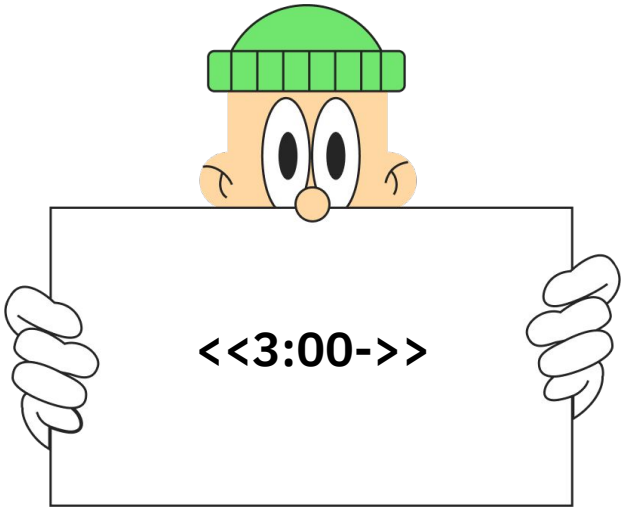
```
def count(param):
    def deco(func: Callable):
        def wrapper(*args, **kwargs):
        ...
            return result

        return wrapper

    return deco
```

Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Декоратор с параметрами

```
def count(num: int = 1):
    def deco(func: Callable):
        counter = []
        def wrapper(*args, **kwargs):
            for _ in range(num):
                result = func(*args, **kwargs)
                counter.append(result)
            return counter
        return wrapper
    return deco

@count(10)
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)

print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 100) = }')
print(f'{rnd(1, 1000) = }')
```



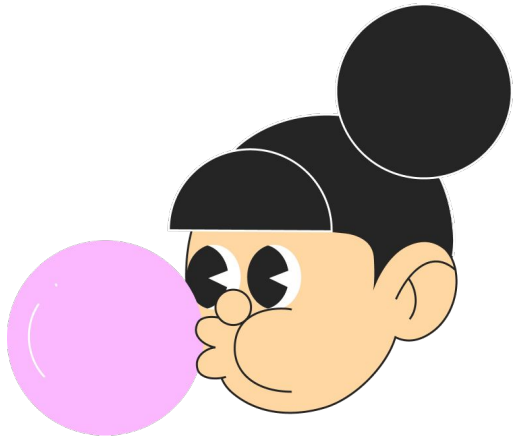
💡 import random
from typing import Callable

Просто импорт не влез слева

Декораторы functools

Декоратор wraps

```
def count(param):
    def deco(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
        ...
            return result

        return wrapper

    return deco
```

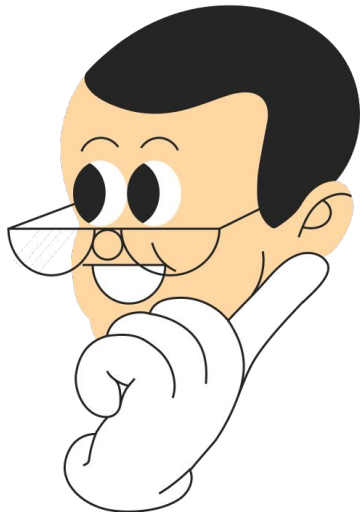


- ✔ __name__ получает имя декорируемой функции
- ✔ help() возвращает справку декорируемой функции

Декоратор cache

cache позволяет кэшировать результат работы функции

```
@cache
def my_func(data):
    ...
```



Итоги занятия

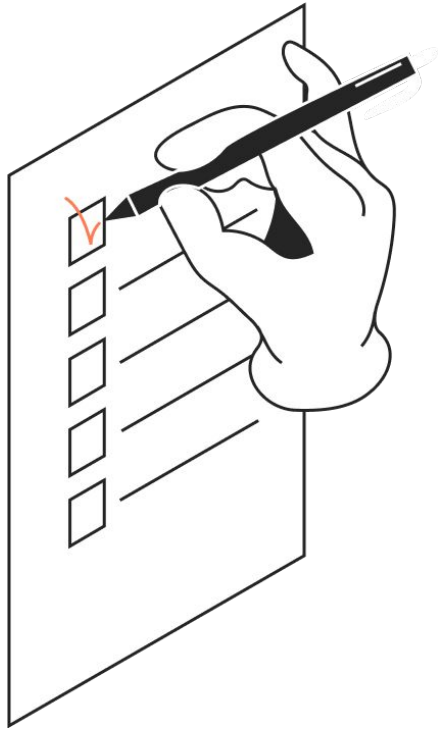
На этой лекции мы

- 📌 Разобрали замыкания в программировании
- 📌 Изучили возможности Python по созданию декораторов
- 📌 Узнали как создавать декораторы с параметрами
- 📌 Разобрали работу некоторых декораторов из модуля functools

Задание

1. Примените рассматриваемые на лекции декораторы к функциям, созданным на прошлых уроках.

2. Попробуйте создать свои декораторы. Например вы можете написать декоратор, который считает количество вызовов функции.



Спасибо за внимание