Занятие 9: Декораторы

Практикум на ЭВМ 2020/2021

Попов Артём Сергеевич

МГУ имени М. В. Ломоносова, факультет ВМК, кафедра ММП

Определение декоратора без аргументов

Декоратор — синтаксический сахар для конструкции «функция, которая принимает другую функцию и что-то возвращает»

```
>>> @my_decorator
>>> def my_function():
...     print('I am a simple function')
```

```
>>> def my_function():
...     print('I am a simple function')
...
>>> my_function = my_decorator(my_function)
```

«Полезный» декоратор возвращает что-то похожее на функцию

Пример декоратора: печать аргументов функции

Декоратор, печатающий имя функции и её аргументы:

```
>>> def my_decorator(f):
...     def wrapper_function(*args, **kwargs):
...         """I am wrapper function"""
...         print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     return wrapper_function
...
>>> @my_decorator
... def identity_function(x):
...     """I am identity function"""
...     print('FUNCTION')
...     return x
...
>>> identity_function(4)
DECORATOR: identity_function (4,) {}
FUNCTION
Out[]: 4
```

Параметры декорируемой функции

Рассмотрим параметры функции `identity_function`:

```
>>> identity_function.__name__
'wrapper_function'
>>> identity_function.__doc__
'I am wrapper function'
```

Хотим, чтобы параметры функции соответствовали параметрам исходной функции

Как можно это сделать?

Изменение параметров декорируемой функции: способ 1

```
>>> def my_decorator(f):
...     def wrapper_function(*args, **kwargs):
...         """I am wrapper function"""
...         print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     wrapper_function.__module__ = f.__module__
...     wrapper_function.__name__ = f.__name__
...     wrapper_function.__doc__ = f.__doc__
...     return wrapper_function
...
>>> @my_decorator
... def identity_function(x):
...     """I am identity function"""
...     print('FUNCTION')
...     return x
...
>>> identity_function.__name__
'identity_function'
```

Изменение параметров декорируемой функции: способ 2

Обернём наши действия функцией update_wrapper:

```
>>> def update_wrapper(function1, function2):
...     function1.__module__ = function2.__module__
...     function1.__name__ = function2.__name__
...     function1.__doc__ = function2.__doc__
...     return function1
...
>>> def my_decorator(f):
...     def wrapper_function(*args, **kwargs):
...         """I am wrapper function"""
...         print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     return update_wrapper(wrapper_function, f)
```

Изменение параметров декорируемой функции: способ 3

Используем декоратор `@spec_update_wrapper`:

```
>>> def my_decorator(f):
...     spec_update_wrapper = lambda x: update_wrapper(x, f)
...
>>>     @spec_update_wrapper
...     def wrapper_function(*args, **kwargs):
...         """I am wrapper function"""
...         print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     return wrapper_function
```

Изменение параметров декорируемой функции: способ 4

```
>>> import functools
...
>>> def my_decorator(f):
...     @functools.wraps(f)  # декоратор с аргументами
...     def wrapper_function(*args, **kwargs):
...         """I am wrapper function"""
...         print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     return wrapper_function
...
>>> @my_decorator
... def identity_function(x):
...     """I am identity function"""
...     print('FUNCTION')
...     return x
...
>>> identity_function.__name__
'identity_function'
```

Управление состоянием декоратора

Декоратор можно выключать, используя глобальные переменные или атрибуты функции:

```
>>> def my_decorator(f):
...     @functools.wraps(f)
...     def wrapper_function(*args, **kwargs):
...         if my_decorator.flag:
...             print('DECORATOR:', f.__name__, args, kwargs)
...         return f(*args, **kwargs)
...     return wrapper_function
...
>>> my_decorator.flag = False
...
# здесь был бы прототип декорируемой функции
...
>>> identity_function(8)
FUNCTION
8
```

Определение декоратора с аргументами

Декоратор с аргументами — по переданным аргументам конструирует декоратор без аргументов, который применяется к функции.

```
>>> @my_deco_with_args(arg1, arg2)
... def my_function():
...     print('I am a simple function')
```

```
>>> def my_function():
...     print('I am a simple function')
...
>>> my_deco = my_deco_with_args(arg1, arg2)
>>> my_function = my_deco(my_function)
```

Пример декоратора с аргументами

```
>>> def my_decorator_with_args(useless_string):
...     def my_decorator(f):
...         @functools.wraps(f)
...         def wrapper_function(*args, **kwargs):
...             print('DECORATOR:', useless_string)
...             return f(*args, **kwargs)
...         return wrapper_function
...     return my_decorator
...
>>> @my_decorator_with_args('i want to be printed')
... def identity_function(x):
...     print('FUNCTION')
...     return x
...
>>> identity_function(15)
DECORATOR: i want to be printed
FUNCTION
15
```

Пример декоратора с аргументами: что будет выведено?

```
>>> def my_decorator_with_args(useless_string):
...     print("Я создаю декоратор.")
...     def my_decorator(f):
...         print("Я - декоратор.")
...         def wrapper_function(*args, **kwargs):
...             print(useless_string)
...             return f(*args, **kwargs)
...         print("Я возвращаю обёрнутую функцию.")
...         return wrapper_function
...     print("Я возвращаю декоратор.")
...     return my_decorator
```

Что будет выведено при данном участке кода?

```
@my_decorator_with_args("Печатаюсь перед функцией.")
def identity_function(x):
    print("Я функция, мой аргумент - {}".format(x))
identity_function(16)
identity_function(23)
```

Пример декоратора с аргументами

Вывод программного кода:

```
# создание функции
Я создаю декоратор.
Я возвращаю декоратор.
Я - декоратор.
Я возвращаю обёрнутую функцию.
# первый вызов
Печатаюсь перед функцией.
Я функция, мой аргумент - 16
# второй вызов
Печатаюсь перед функцией.
Я функция, мой аргумент - 23
```

Аргументы с значением по умолчанию

```
>>> def my_decorator_with_args(useless_string='Nothing'):
...     def my_decorator(f):
...         @functools.wraps(f)
...         def wrapper_function(*args, **kwargs):
...             print('DECORATOR:', useless_string)
...             return f(*args, **kwargs)
...         return wrapper_function
...     return my_decorator
...
>>> @my_decorator_with_args
... def identity_function(x):
...     print('FUNCTION')
...     return x
...
???
???
???
```

Аргументы с значением по умолчанию

```
>>> def my_decorator_with_args(useless_string='Nothing'):
...     def my_decorator(f):
...         @functools.wraps(f)
...         def wrapper_function(*args, **kwargs):
...             print('DECORATOR:', useless_string)
...             return f(*args, **kwargs)
...         return wrapper_function
...     return my_decorator
...
>>> @my_decorator_with_args
... def identity_function(x):
...     print('FUNCTION')
...     return x
...
>>> f = identity_function(15)
>>> f()
DECORATOR: <function identity_function at 0x7fbe70226a60>
TypeError: 'int' object is not callable
```

Значения по умолчанию: решение проблемы

```
>>> def my_decorator_with_args(useless_string='Nothing'):
...     def my_decorator(f):
...         @functools.wraps(f)
...         def wrapper_function(*args, **kwargs):
...             print('DECORATOR:', useless_string)
...             return f(*args, **kwargs)
...         return wrapper_function
...     return my_decorator
...
>>> @my_decorator_with_args()
... def identity_function(x):
...     print('FUNCTION')
...     return x
```

Как избавиться от тройной вложенности: способ 1

Декоратор с аргументами очень громоздко выглядит в коде. Хотим избавиться от тройной вложенности и задавать декоратор через двойную вложенность:

```
>>> def my_decorator_with_args(f, argument):
...     @functools.wraps(f)
...     def wrapper_function(*args, **kwargs):
...         ....
...     return wrapper_function
```

Избавление от тройной вложенности: способ 1

Напишем декоратор, превращающий декоратор с аргументами в «новом стиле» в декоратор с аргументами «в старом стиле».

```
>>> def with_arguments(deco_with_args):
...     @functools.wraps(deco_with_args)
...     def deco_with_args_old_style(*args, **kwargs):
...         def deco_without_args(f):
...             return deco_with_args(f, *args, **kwargs)
...         return deco_without_args
...     return deco_with_args_old_style
```

Избавление от тройной вложенности: способ 1

Теперь наш декоратор с аргументами выглядит так:

```
>>> @with_arguments
... def my_decorator_with_args(f, useless_string):
...     @functools.wraps(f)
...     def wrapper(*args, **kwargs):
...         print('DECORATOR:', useless_string)
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator_with_args('i want to be printed')
... def identity_function(x):
...     print('FUNCTION')
...     return x
...
>>> identity_function(42)
DECORATOR: i want to be printed
FUNCTION
42
```

Ещё более короткая запись

Усовершенствуем наш декоратор `@with_arguments`:

```
>>> def with_arguments(deco_with_args):
...     @functools.wraps(deco_with_args)
...     def deco_with_args_old_style(*args, **kwargs):
...         def deco_without_args(f):
...             f_wrapper = deco_with_args(f, *args, **kwargs)
...             functools.update_wrapper(f_wrapper, f)
...             return f_wrapper
...         return deco_without_args
...     return deco_with_args_old_style
```

Что это нам дало?

Ещё более короткая запись

Теперь мы можем не писать в декораторе `wraps`:

```
>>> @with_arguments
... def my_decorator_with_args(f, useless_string):
...     def wrapper(*args, **kwargs):
...         print('DECORATOR:', useless_string)
...         return f(*args, **kwargs)
...     return wrapper
...
>>> @my_decorator_with_args('i want to be printed')
... def identity_function(x):
...     """I am identity function"""
...     print('FUNCTION', x)
...
>>> identity_function.__name__
'identity_function'
```

Избавление от тройной вложенности: способ 2

Хотим придумать способ избавления от тройной вложенности, который позволит писать обе конструкции:

```
>>> @deco_with_args
... def function(x):
...     ....
>>> @deco_with_args(argument='argument')
... def function(x):
...     ....
```

Создадим конструкцию, которая будет второй способ сводить к первому способу через рекурсию.

Избавление от тройной вложенности: способ 2

```
def deco_with_args(f=None, *, message='Nothing'):
    # со скобками
    if f is None:
        def deco_without_args(g):
            return deco_with_args(f=g, message=message)
        return deco_without_args

    # без скобок
    @functools.wraps(f)
    def wrapper_function(*args, **kwargs):
        print('DECORATOR:', message)
        return f(*args, **kwargs)
    return wrapper_function
```

Зачем нужны обязательные ключевые аргументы?

Какие минусы у этого способа?

Декораторы классов

Синтаксис декораторов работает и для классов:

```
>>> @my_decorator
>>> class MyClass:
...     pass
```

```
>>> class MyClass:
...     pass
...
>>> MyClass = my_decorator(MyClass)
```

«Полезный» декоратор в этом случае возвращает что-то похожее на класс.

Декораторы методов класса

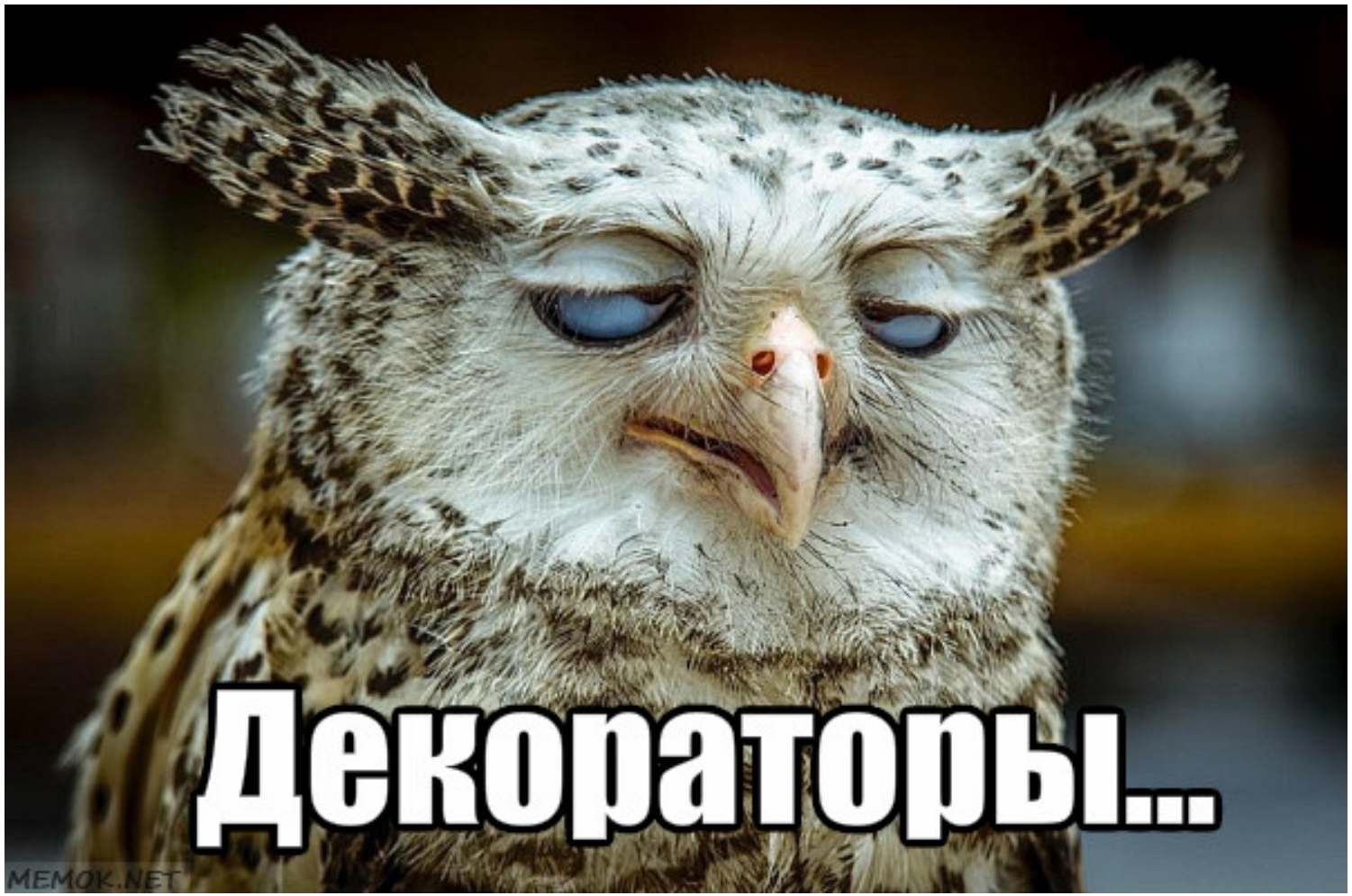
Синтаксис декораторов работает и для методов класса

```
>>> class MyClass:
...     def __init__(self, x):
...         self.x = x
...
...     @my_decorator_with_args(argument='argument')
...     def square(self):
...         return self.x ** 2
```

Реализация декоратора с помощью класса

Для реализации декоратора вместо функции используем класс:

```
>>> class my_decorator_with_args:
...     def __init__(self, useless_string='Nothing'):
...         self.useless_string = useless_string
...
...     def __call__(self, f):
...         @functools.wraps(f)
...         def wrapper(*args, **kwargs):
...             print('DECORATOR:', self.useless_string)
...             return f(*args, **kwargs)
...         return wrapper
...
>>> @my_decorator_with_args() # проблема скобок
... def identity_function(x):
...     """I am identity function"""
...     print('FUNCTION')
...     return x
```


Декораторы...
MEMOK.NET

Подсчёт времени выполнения функции

```
>>> import time
...
>>> def time_count(func=None, *, n_iter=100):
...     if func is None:
...         return lambda func: time_count(func, n_iter=n_iter)
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         print(func.__name__, end=" ... ")
...         total_time = 0
...         for i in range(n_iter):
...             tick = time.clock()
...             result = func(*args, **kwargs)
...             total_time += time.clock() - tick
...         print(f"mean time is {total_time / n_iter:.3f}")
...         return result
...     return wrapper
```

Подсчёт времени выполнения функции

```
>>> # код с предыдущего слайда
...
>>> import numpy as np
...
>>> @time_count
... def count_nonzero_elements(x):
...     return np.sum(x != 0)
...
>>> arr = np.random.randint(0, 2, size=1000)
>>> count_nonzero_elements(arr)
count_nonzero_elements ... mean time is 0.002
500262
```

Подсчёт количества запусков

```
>>> def profiled(func):
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         wrapper.ncalls += 1
...         return func(*args, **kwargs)
...     wrapper.ncalls = 0
...     return wrapper
...
>>> @profiled
... def identity_function(x):
...     return x
...
>>> for i in range(10):
...      identity_function(1)
>>> identity_function.ncalls
10
```

Мемоизация

```
>>> def memoized(func):
...     cache = {}
...     @functools.wraps(func)
...     def wrapper(*args, **kwargs):
...         key = args + tuple(sorted(kwargs.items()))
...         if key not in cache:
...             cache[key] = func(*args, **kwargs)
...         return cache[key]
...     return wrapper
...
>>> @memoized
... def ackermann(m, n):
...     if not m:
...         return n + 1
...     elif not n:
...         return ackermann(m - 1, 1)
...     else:
...         return ackermann(m - 1, ackermann(m, n - 1))
```

Мемоизация

Подобная функция уже есть в модуле `functools`

```
>>> import functools
...
>>> @functools.lru_cache(maxsize=64)
... def ackermann(m, n):
...     # код функции
>>> ackermann(3, 4)
125
>>> ackermann.cache_info()
CacheInfo(hits=65, misses=315, maxsize=64, currsize=64)
```

Мемоизация может быть полезна при работе с библиотеками предобработки. Например, мемоизация часто применяется для ускорения лемматизации.

Предупреждение о неподдерживаемых функциях

```
>>> import warnings
...
... def deprecated(func):
...     warnings.warn(func.__name__ + " is deprecated.")
...     return func
...
>>> @deprecated
... def identity_function(x):
...     return x
/usr/local/lib/python3.5/dist-packages/ipykernel_launcher.py:5:
UserWarning: identity_function is deprecated.
```

@classmethod

Хотим задать метод класса, а не метод объекта класса.

```
>>> class MyClass:
...     def __init__(self, x):
...         self.x = x ** 2
...
>>>     @classmethod
...     def from_disk(cls, path_to_x):
...         with open(path_to_x, 'r') as f:
...             x = int(f.read())
...         new_class = cls(x)
...         return new_class
```

@staticmethod

Хотим задать функцию внутри класса. Например, если функция связана с классом, но не является его методом.

```
>>> class MyClass:
...     def __init__(self, x):
...         self.x = x ** 2
...
...     @classmethod
...     def from_disk(cls, path_to_x):
...         x = cls.read_int_numbers(path_to_x)
...         new_class = cls(x)
...         return new_class
...
...     @staticmethod
...     def read_int_numbers(path_to_file):
...         with open(path_to_file, 'r') as f:
...             x = int(f.read())
...         return x
```

Вспоминаем property

```
>>> class Mine(object):
...     def __init__(self):
...         self._x = None
...
...     def get_x(self):
...         return self._x
...
...     def set_x(self, value):
...         self._x = value
...
...     def del_x(self):
...         self._x = 'No more'
...
...     x = property(get_x, set_x, del_x, 'Это свойство x.')
```

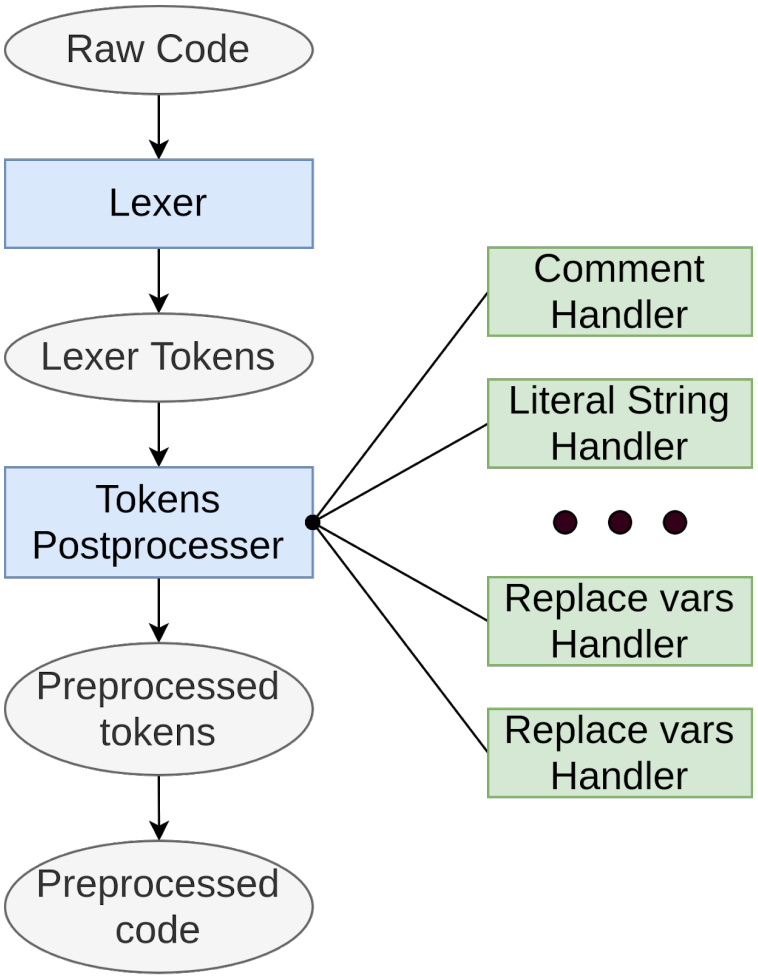
@property

```
>>> class Mine(object):
...     def __init__(self):
...         self._x = None
...
...     @property
...     def x(self):
...         return self._x
...
...     @x.setter
...     def x(self, value):
...         self._x = value
...
...     @x.deletter
...     def x(self):
...         self._x = 'No more'
```

@numpy.vectorize

```
>>> import numpy as np
...
>>> @np.vectorize
... def my_func(x):
...     if x > 5:
...         return 25
...     else:
...         return x**2
...
>>> data = np.array([1, 2, 3, 4, 5, 6])
>>> my_func(data)
array([ 1,  4,  9, 16, 25, 25])
```

Сложные пайплайны обработки данных



x	<-	"42"	\n
Name	Oper.	Lit. Str.	Text

Входные токены

True	False	False	False

Обработан ли токен?

var1	None	None	None

Выходные значения

Сложные пайплайны обработки данных

Пример: каждая сущность пайплайна должна быть классом, наследованным от класса BaseHandler, с некоторыми заданными атрибутами и методами.

Не хочется заводить отдельный класс для простых элементов пайплайна (например, для приведения к нижнему регистру).

Можно создавать по функции классы и их экземпляры внутри декоратора динамически:

```
>>> def handler(simple_preprocessing_function):
...         class NewHandler(BaseHandler):
...             def __init__(self, handler_function):
...                 self.handler_function = handler_function
...             def handle(self, input_data):
...                 return self.handle(input_data)
```

Заключение

- Декораторы — синтаксический сахар для конструкций вида «функция, которая принимает другую функцию и что-то возвращает»
- Главное — один раз понять принцип устройства декоратов, тогда потом будет несложно пользоваться этим инструментом
- Написание декораторов — частая задача на собеседовании