



Лекция 4. Декораторы

Сайфулин Дмитрий, Слободкин Евгений
ИТМО, 27 ноября 2023



MASTER OF
SOFTWARE
ENGINEERING



Функция — объект первого класса



Функция — объект первого класса

Функцию можно передать в качестве аргумента.

```
def timer(f, *args, **kwargs):  
    t_start = time.time()  
    result = f(*args, **kwargs)  
    t_finish = time.time()  
    return result, (t_finish - t_start)
```



Функция — объект первого класса

Функцию можно передать в качестве аргумента.

```
def timer(f, *args, **kwargs):  
    t_start = time.time()  
    result = f(*args, **kwargs)  
    t_finish = time.time()  
    return result, (t_finish - t_start)
```

```
def python(n):  
    time.sleep(n)  
    return "Python"
```

```
>>> timer(python, 5)  
('Python', 5.0049662590026855)
```



Дублирование кода

```
# Предположим, что мы пишем набор функций для работы с файлами. Тогда перед  
# каждым созданием файла мы должны проверить, что у нас хватит на него места.
```

```
# Получаем дублирование кода. Плюс, код проверки находится в той же функции,  
# что и код по созданию или копированию файла.
```

```
def touch_file(file_name):  
    if has_free_space():  
        ...
```

```
def cp_file(file_name):  
    if has_free_space():  
        ...
```

```
def scp_file(addr, file_name):  
    if has_free_space():  
        ...
```



Декораторы



Что такое декоратор?

Декоратор — функция, которая принимает функцию и возвращает тоже функцию.
В этом случае часто говорят, что декоратор «оборачивает» функцию.
Более аккуратно, декоратор умеет ещё и оборачивать классы и их методы,
но с классами мы будем разбираться уже на следующих лекциях.

```
def deco(f):  
    ...  
    assert callable(f_new)    # Результат функции — функция.  
    return f_new  
  
f = deco(f)    # deco — это декоратор.
```



Что такое декоратор?

Декоратор — функция, которая принимает функцию и возвращает тоже функцию.
В этом случае часто используют термин декоратор «оборачивает» функцию.
Более аккуратно, декоратор умеет ещё и оборачивать классы и их методы,
но с классами мы будем разбираться уже на следующих лекциях.

```
def deco(f):  
    ...  
    assert callable(f_new)    # Результат функции — функция.  
    return f_new
```

```
f = deco(f)    # deco — это декоратор.
```

В Python имеется специальный синтаксис для выражения той же идеи:

```
@deco  
def f():  
    pass
```

Почти то же самое, что и `f = deco(f)`. Разницу можно увидеть, вызвав `dis`.



Пишем первый декоратор



Декоратор debug

Хотим, чтобы функция, помеченная декоратором debug, печатала свои аргументы.

```
def debug(f):  
    ... # Пока реализация нас не волнует.
```



Декоратор debug

Хотим, чтобы функция, помеченная декоратором debug, печатала свои аргументы.

```
def debug(f):  
    ... # Пока реализация нас не волнует.
```

```
@debug
```

```
def run_server(host, port, log_level):  
    ...
```

```
>>> run_server("localhost", 8080, log_level="INFO")  
func: run_server  
args: (localhost, 8080)  
kwargs: {'log_level': 'INFO'}
```



Декоратор debug

Давайте внутри debug объявим функцию, которая будет вызывать переданную.
Эта функция как раз и будет результатом применения функции debug.

```
def debug(f):  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
        return f(*args, **kwargs)  
  
    return inner
```

На самом деле, данная реализация пропускает некоторые аргументы, но мы будем разбираться с этим на следующих слайдах.



Декоратор debug

Пробуем использовать наш декоратор вместе с функцией `run_server`.

```
@debug
```

```
def run_server(host, port, log_level):  
    ...
```

```
>>> run_server("localhost", 8080, log_level="INFO")  
func: run_server  
args: (localhost, 8080)  
kwargs: {'log_level': 'INFO'}
```



Проблемы в текущей реализации



У нас меняется `__name__` функции

Если `run_server = debug(run_server)`, то что такое `run_server.__name__`?

@debug

```
def run_server(host, port, log_level):  
    """Runs a server."""  
    ...
```



У нас меняется `__name__` функции

Если `run_server = debug(run_server)`, то что такое `run_server.__name__`?

```
@debug
def run_server(host, port, log_level):
    """Runs a server."""
    ...
```

```
>>> run_server.__name__
'inner'
```




И не только `__name__` ...

Если `run_server = debug(run_server)`, то что такое `run_server.__name__`?

```
@debug
def run_server(host, port, log_level):
    """Runs a server."""
    ...
```

```
>>> run_server.__name__
'inner'
```

```
>>> run_server.__doc__
None
```



Вспомним определение debug

Если `run_server = debug(run_server)`, то что такое `run_server.__name__`?

```
@debug
```

```
def run_server(host, port, log_level):  
    """Runs a server."""  
    ...
```

```
>>> run_server.__name__  
'inner'
```

```
>>> run_server.__doc__  
None
```

```
def debug(f):  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
        return f(*args, **kwargs)  
  
    return inner
```



Наивный подход к переносу аргументов

```
def debug(f):  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
        return f(*args, **kwargs)  
  
    inner.__name__ = f.__name__  
    inner.__doc__ = f.__doc__  
    ...  
  
    return inner
```

```
>>> f.__name__  
'f'
```

А если у функции есть какие-либо ещё поля?



Использование wraps для переноса аргументов

Самый адекватный способ, который обычно и используется.

```
def debug(f):  
    @functools.wraps(f)  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
        return f(*args, **kwargs)  
  
    return inner
```

```
>>> f.__name__  
'f'
```



Как реализован wraps?

Примерная реализация.

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                        '__annotations__')
WRAPPER_UPDATES = ('__dict__',)

def wraps(wrapped, assigned=WRAPPER_ASSIGNMENTS, updated=WRAPPER_UPDATES):
    return partial(
        update_wrapper,
        wrapped=wrapped,
        assigned=assigned,
        updated=updated,
    )
```



Как реализован wraps?

Примерная реализация.

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',  
                        '__annotations__')  
WRAPPER_UPDATES = ('__dict__',)  
  
def update_wrapper(wrapper, wrapped, assigned=..., updated=...):  
    for attr in assigned:  
        value = getattr(wrapped, attr)  
        setattr(wrapper, attr, value)  
  
    for attr in updated:  
        getattr(wrapper, attr).update(getattr(wrapped, attr, {}))  
  
    return wrapper
```



Проблемы в текущей реализации 2



Аргументы по умолчанию

```
def debug(f):  
    @functools.wraps(f)  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
  
        return inner  
  
@debug  
def run_server(host, port, log_level='INFO'):  
    ...
```




Аргументы по умолчанию

```
def debug(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        print('func:', f.__name__)
        print('args:', args)
        print('kwargs:', kwargs)

        return inner

    return inner

@debug
def run_server(host, port, log_level='INFO'):
    ...

>>> run_server('localhost', 8080)
func: run_server
args: ('localhost', 8080)
kwargs: # - Почему kwargs пустой?
```



Аргументы по умолчанию

```
def debug(f):  
    @functools.wraps(f)  
    def inner(*args, **kwargs):  
        print('func:', f.__name__)  
        print('args:', args)  
        print('kwargs:', kwargs)  
  
        return inner
```

```
@debug  
def run_server(host, port, log_level='INFO'):  
    ...
```

```
>>> run_server('localhost', 8080)
```

```
func: run_server
```

```
args: (localhost, 8080)
```

```
kwargs: # - Почему kwargs пустой?
```

- В kwargs находятся именованные аргументы
конкретного вызова функции.



Модуль inspect

```
import inspect

def debug(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        print('func:', f.__name__)
        print('call args', inspect.getcallargs(f, *args, **kwargs))

    return inner
```



Модуль inspect

```
import inspect

def debug(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        print('func:', f.__name__)
        print('call args', inspect.getcallargs(f, *args, **kwargs))

    return inner

@debug
def run_server(host, port, log_level='INFO'):
    ...

>>> run_server('localhost', 8080)
func: run_server
call args: {'host': 'localhost', 'port':8080, 'log_level': 'INFO'}
```



Примеры декораторов



Добавление «статической» переменной

```
def static_var(name, value):  
    def deco(f): # А где functools.wraps?  
        setattr(f, name, value)  
        return f  
  
    return deco
```

```
@static_var('secret', 'python')  
def f():  
    pass
```

```
>>> f.secret  
'python'
```



Подсчёт количества вызовов функции

```
def counter(f):  
    @functools.wraps(f)  
    def inner(*args, **kwargs):  
        inner.count += 1  
        return f(*args, **kwargs)  
  
    inner.count = 0  
    return inner
```

```
@counter  
def f():  
    pass
```

```
>>> [f() for _ in range(10)]  
>>> f.count  
10
```



Подсчёт количества вызовов функции (2)

```
def counter2(f):  
    count = 0  
    @functools.wraps(f)  
    def inner(*args, **kwargs):  
        nonlocal count  
        count += 1  
        return f(*args, **kwargs)  
  
    return inner
```

```
@counter2  
def f():  
    pass
```

```
>>> [f() for _ in range(10)]  
>>> f.count  
10
```




Порядок применения декораторов важен

```
@counter
@static_var('count', 50)
def f():
    pass
```

```
>>> [f() for _ in range(10)]
>>> f.count
10
```

```
@static_var('count', 50)
@counter
def f():
    pass
```

```
>>> [f() for _ in range(10)]
>>> f.count
60
```



Декораторы с аргументами



Декоратор timeout

- # Необходимо реализовать декоратор, который будет выбрасывать исключение,
- # если функция работала больше выделенного времени.

- # При этом хочется сделать опциональные параметры под тип и текст ошибки.

- # Для простоты мы будем дожидаться окончания работы функции, но, отметим, что
- # так можно выбрасывать исключение сразу как только вышли за лимит времени.



Декоратор timeout

```
@timeout(5)
def a(): time.sleep(6)

>>> a()
TimeoutError # 6 секунд спустя.
```



Декоратор timeout

```
@timeout(5)
def a(): time.sleep(6)

>>> a()
TimeoutError # 6 секунд спустя.
```



```
@timeout(5, err_msg='Time Limit Exceeded')
def b(): time.sleep(6)

>>> b()
TimeoutError: Time Limit Exceeded
```



Декоратор timeout

```
@timeout(5)
```

```
def a(): time.sleep(6)
```

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

```
TimeoutError # 6 секунд спустя.
```

```
@timeout(5, err_msg='Time Limit Exceeded')
```

```
def b(): time.sleep(6)
```

```
>>> b()
```

```
TimeoutError: Time Limit Exceeded
```

```
@timeout(5, err_cls=RuntimeError, err_msg='Time Limit Exceeded')
```

```
def c(): time.sleep(5)
```

```
>>> c()
```

```
RuntimeError: Time Limit Exceeded
```



Реализация декоратора timeout

Вызов timeout всегда должен возвращать декоратор.

```
def timeout(seconds, *, err_cls=TimeoutError, err_msg=None):  
    def deco(f):  
        @functools.wraps(f)  
        def inner(*args, **kwargs):  
            ...  
  
        return inner  
  
    return deco
```



Реализация декоратора timeout

Вызов timeout всегда должен возвращать декоратор.

```
def timeout(seconds, *, err_cls=TimeoutError, err_msg= None):
    def deco(f):
        @functools.wraps(f)
        def inner(*args, **kwargs):
            start = time.time()
            result = f(*args, **kwargs)
            finish = time.time()
            if finish - start > seconds:
                raise err_cls(err_msg)

            return result

        return inner

    return deco
```




Декораторы с аргументами и без



Декоратор debug

Вновь делаем декоратор debug, но в этот раз добавим возможность менять формат.

```
@debug
```

```
def foo(x, y):  
    pass
```

```
@debug(fmt='[{f}] >> ARGS: {args}, KWARGS: {kwargs}')
```

```
def bar(x, y):  
    pass
```

```
>>> foo(1, 2)  
func: foo; args: (1, 2); kwargs: {}
```

```
>>> bar(1, y=2)  
[bar] >> ARGS: (1,), KWARGS: {'y': 2}
```



Декоратор debug

Перепишем код с прошлого слайда, используя только применение функции.

```
@debug
def foo(x, y):
    pass
```

```
foo = debug(foo)
```

```
@debug(fmt='[{f}] >> ARGS: {args}, KWARGS: {kwargs}' )
def bar(x, y):
    pass
```

```
bar = debug(fmt='[{f}] >> ARGS: {args}, KWARGS: {kwargs}')(bar)
```



Декоратор debug

Перепишем код с прошлого слайда, используя только применение функции.

```
@debug
def foo(x, y):
    pass
```

```
foo = debug(foo)
```

```
@debug(fmt='[{f}] >> ARGS: {args}, KWARGS: {kwargs}')
def bar(x, y):
    pass
```

```
bar = debug(fmt='[{f}] >> ARGS: {args}, KWARGS: {kwargs}')(bar)
```

То есть debug возвращает либо декоратор, либо новую функцию, вызов которой в свою очередь возвращает декоратор.



Реализация декоратора debug

```
def debug(f, *, fmt='func: {func}; args: {args}; kwargs: {kwargs}'):
    def deco(f):
        @functools.wraps(f)
        def inner(*args, **kwargs):
            log = fmt.format(func=f.__name__, args=args, kwargs=kwargs)
            print(log)
            return f(*args, **kwargs)

        return inner

    return deco
```



Реализация декоратора debug

```
def debug(f, *, fmt='func: {func}; args: {args}; kwargs: {kwargs}'):
    def deco(f):
        @functools.wraps(f)
        def inner(*args, **kwargs):
            log = fmt.format(func=f.__name__, args=args, kwargs=kwargs)
            print(log)
            return f(*args, **kwargs)

        return inner

    return deco
```

Случай с переданным параметром одолели!

```
@debug(fmt='{func} - {args} - {kwargs}')
def a(x, y): ...
```

```
>>> a(1, y=2)
a - (1,) - {'y': 2}
```



Реализация декоратора debug

```
def debug(f, *, fmt='func: {func}; args: {args}; kwargs: {kwargs}'):
    def deco(f):
        @functools.wraps(f)
        def inner(*args, **kwargs):
            log = fmt.format(func=f.__name__, args=args, kwargs=kwargs)
            print(log)
            return f(*args, **kwargs)

        return inner

    return deco
```

Но ещё не работает, если не передавать никакие аргументы.

```
@debug
def a(x, y): ...
```

```
>>> a(1, y=2)
```

```
TypeError: debug.<locals>.deco() got an unexpected keyword argument 'y'
```



Реализация декоратора debug

```
def debug(f=None, *, fmt='func: {func}; args: {args}; kwargs: {kwargs}'):
    def deco(f):
        @functools.wraps(f)
        def inner(*args, **kwargs):
            log = fmt.format(func=f.__name__, args=args, kwargs=kwargs)
            print(log)
            return f(*args, **kwargs)

        return inner

    if f is not None: # Если функция передана, то продекорлируем её.
        return deco(f)

    return deco
```

```
>>> a(1, y=2)
func: add; args: (1,); kwargs: {'y': 2}
```




Декораторы из стандартной библиотеки



lru_cache

```
@lru_cache(maxsize=None)
```

```
def fib(n):
```

```
    if n < 2:
```

```
        return n
```

```
    return fib(n-1) + fib(n-2)
```

```
>>> [fib(n) for n in range(16)]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

```
>>> fib.cache_info()
```

```
CacheInfo(hits=28, misses=16, maxsize=None, currsize=16)
```



singledispatch

```
@singledispatch
```

```
def icon(event: Event):  
    return '✿'
```

```
@icon.register
```

```
def _(update: CommentCreated):  
    return '💬'
```

```
@icon.register
```

```
def _(update: MRCreated):  
    return '🦊'
```

```
>>> icon(MRCreated(...))  
'🦊'
```

```
>>> icon(IssueCreated(...))  
'✿'
```



Декораторы из модуля typing

- final
- override
- runtime_checkable

Как реализован final?



Декораторы из модуля typing

- `final`
- `override`
- `runtime_checkable`

Как реализован final?

```
def final(f):  
    try:  
        f.__final__ = True  
    except (AttributeError, TypeError):  
        pass  
  
    return f
```



Декораторы, которые скоро нас ждут

- `property`
- `classmethod`
- `staticmethod`
- `abstractmethod`



Бонус: декоратор может быть классом



Уточнение определение

Декоратор — **функция**, которая принимает функцию и возвращает тоже функцию.
На самом деле, декоратор может быть и классом, у которого определён `__call__`.



Уточнение определение

Декоратор — функция, которая принимает функцию и возвращает тоже функцию.
На самом деле, декоратор может быть и классом, у которого определён `__call__`.

```
class Timer:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        t_start = time.time()
        result = self.func(*args, **kwargs)
        t_finish = time.time()
        return result, (t_finish - t_start)
```

```
@Timer
def foo():
    ...
```



Уточнение определение

Декоратор — функция, которая принимает функцию и возвращает тоже функцию.
На самом деле, декоратор может быть и классом, у которого определён `__call__`.

```
class Timer:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        t_start = time.time()
        result = self.func(*args, **kwargs)
        t_finish = time.time()
        return result, (t_finish - t_start)

@Timer
def foo():
    ...
```

Но такой способ используется крайне редко, поэтому останавливаться мы на нём



Бонус: типизация декораторов



ParamSpec

Для типизации декораторов в Python появился класс ParamSpec.
Он умеет хранить в себе спецификации аргументов.

Например, наш debug можно протипизировать так:

```
P = ParamSpec('P')
R = TypeVar('R')

def debug(f: Callable[P, R]) -> Callable[P, R]:
    def inner(*args: P.args, **kwargs: P.kwargs) -> R:
        print(f.__name__, args, kwargs)
        return f(*args, **kwargs)

    return inner
```