Декораторы

Дмитрий Яковлев

Санкт-Петербургский государственный университет

24 марта 2022 г.

План лекции

Декораторы

Задача

Как реализовать функцию, которая при повторном вызове не будет исполняться?

```
def init_db():
    print("Init!")
init_db()
```

Решение

Можно создать вспомогательную функцию с глобальной переменной

```
called = False
def init_db2():
    global called
    if not called:
        called = True #плохо?
        return init_db()
    raise Exception()
```

Можно ли избавиться от глобальной переменной?

Решение

Можно воспользоваться аргументом по умолчанию:

```
def init_db2(called=[]):
    if not called:
        called.append(True)
        return init_db()
    raise Exception()
```

Декораторы: Напоминание

- Функции в Python'е являются объектами
- Функции можно передавать в качестве аргумента
- Функции можно возвращать из функции

```
>>> def makebold(fn):
...     def wrapped():
...         return "<b>" + fn() + "</b>"
...     return wrapped
...
>>> makebold(lambda: "Hello!")()
'<b>Hello!</b>'
```

Декораторы: Общие сведения

Декоратор - функция, которая принимает другую функцию и возвращает обертку, которая делает что-то своё «вокруг» вызова основной функции.

```
>>> import time
>>> def logging(func):
...     def inner(*args, **kwargs):
...         print(time.strftime("%Y-%m-%d %H:%M"),
...               func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
>>> @logging
... def indetity(x):
...     return x
>>> indetity(15)
2016-12-10 18:23 indetity (15,) {}
15
```

Декораторы: Пример once 1

```
>>> def once(func):
...     called = False
...     def inner(*args, **kwargs):
...         nonlocal called
...         if not called:
...             result = func(*args, **kwargs)
...             called = True
...             return result
...         raise Exception() # return?
...     return inner
>>> @once
... def init_db():
...     print("Init!")
>>> init_db()
Init!
>>> init_db()
Exception
```

Декораторы: Пример timer

```
>>> def timer(func):
...     def inner(*args, **kwargs):
...         start_time = time.time()
...         res = func(*args, **kwargs)
...         end_time = time.time()
...         run_time = end_time - start_time #Плохо?
...         print("Running time %.3f sec" % run_time)
...         return res
...     return inner
>>> @timer
... def f():
...     N = 10000000
...     res = [sqrt(x) for x in range(N)]
>>> f()
Running time 1.655 sec
```

Декораторы: Пример debug

Можем управлять работой декоратора:

```
DEBUG = False
def debug(func):
    if not DEBUG:
      return func
    @functools.wraps(func)
    def wrapper_debug(*args, **kws):
        args_repr = [repr(a) for a in args]
        kws_repr = [f"{k}={v!r}" for k, v in kws.items()]
        signature = ", ".join(args_repr + kws_repr)
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kws)
        print(f"{func.__name__!r} returned {value!r}")
        return value
    return wrapper_debug
```

Декораторы: Пример вызова debug

Можем управлять работой декоратора:

```
>>> import math
>>> math.factorial = debug(math.factorial)
>>>
>>> def approximate_e(N):
...     return sum(1 / math.factorial(n) for n in range(N))
...
>>> approximate_e(4)
Calling factorial(0)
'factorial' returned 1
Calling factorial(1)
'factorial' returned 1
Calling factorial(2)
'factorial' returned 2
Calling factorial(3)
'factorial' returned 6
2.666666666666665
```

Декораторы: Проблема с декорируемой функцией (1)

Возникает проблема с внутренними атрибутами декорируемой функции:

```
>>> @timer
... def f():
...     N = 10000000
...     res = [sqrt(x) for x in range(N)]
>>> f.__name__
'inner'
```

Аналогично:

```
>>> def f():
...     N = 10000000
...     res = [sqrt(x) for x in range(N)]
>>> f = timer(f)
>>> f.__name__
'inner'
```

Декораторы: Проблема с декорируемой функцией (2)

Решение:

```
>>> import functools
>>> def logging(func):
...     @functools.wraps(func)
...     def inner(*args, **kwargs):
...         print(time.strftime("%Y-%m-%d %H:%M"),
...               func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     return inner
>>> @logging
... def indetity(x):
...     return x
>>> indetity.__name__
'indetity'
```

Как можно было бы иначе решить?

Декораторы: Проблема с декорируемой функцией (3)

Альтернатива:

```
>>> def logging(func):
...     def inner(*args, **kwargs):
...         print(time.strftime("%Y-%m-%d %H:%M"),
...               func.__name__, args, kwargs)
...         return func(*args, **kwargs)
...     inner.__module__ = func.__module__
...     inner.__name__ = func.__name__
...     inner.__doc__ = func.__doc__
...     return inner
>>> @logging
... def indetity(x):
...     return x
>>> indetity.__name__
'indetity'
```

Декораторы: Передача аргументов (1)

```
>>> def decorator(*arguments):
...     def real_decorator(func):
...         def wrapper(*args, **kwargs):
...             print(arguments)
...             func(*args, **kwargs)
...         return wrapper
...     return real_decorator
...
>>> def print_args(*args):
...     print(args)
...
>>> dec = decorator("arguments")
>>> dec(print_args)(1, 2, 3)
('arguments',)
(1, 2, 3)
```

Декораторы: Передача аргументов (2)

```
>>> def decorator(*arguments):
...     def real_decorator(func):
...         def wrapper(*args, **kwargs):
...             print(arguments)
...             func(*args, **kwargs)
...         return wrapper
...     return real_decorator
...
>>> @decorator("arguments")
... def print_args(*args):
...     print(args)
...
>>> print_args(1, 2, 3)
('arguments',)
(1, 2, 3)
```

Декораторы: Опциональные аргументы (1)

```
def handle(func=None, *, err_msg="Error!"):
    # со скобками
    if func is None:
        def decorator(func):
            return handle(func, err_msg=err_msg)
        return decorator
    # без скобок
    @functools.wraps(func)
    def inner(*args, **kwargs):
        res = func(*args, **kwargs)
        if res:
            print(func.__name__, args, kwargs, err_msg)
    return inner
```

Декораторы: Опциональные аргументы (2)

Со скобками:

```
>>> @handle(err_msg="Error in  file upload")
... def upload_file(filename):
...     return 1
...
>>> upload_file("test.txt")
upload_file ('test.txt',) {} Error in  file upload
```

Без скобок:

```
>>> @handle
... def add_item(x):
...     return 1
...
>>> add_item("test.txt")
add_item ('test.txt',) {} Error!
```

Полезные декораторы

Декораторы: count

Пример добавления поля к функции:

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

Как можем реализовать декоратор once, чтобы вместо исключения получать результат?

Декораторы: once

Если мы хотим получать посчитанный результат

```
def once(func):
    @functools.wraps(func)
    def inner(*args, **kwargs):
        try:
            return inner._result
        except AttributeError:
            inner._result = func(*args, **kwargs)  # Плохо?
            return inner._result
    return inner
```

Декораторы: singledispatch 1

Переопределние методов

```
from functools import singledispatch
from datetime import date, datetime, time
@singledispatch
def format(arg):
    return arg
@format.register
def _(arg: date):
    return f"{arg.day}-{arg.month}-{arg.year}"
@format.register
def _(arg: datetime):
    return f"{arg.year} {arg.hour}:{arg.minute}"
@format.register(time)
def _(arg):
    return f"{arg.hour}:{arg.minute}:{arg.second}"
```

Декораторы: singledispatch 2

```
print(format("today"))
# today
print(format(date(2021, 5, 26)))
# 26-5-2021
print(format(datetime(2021, 5, 26, 17, 25, 10)))
# 2021 17:25
print(format(time(19, 22, 15)))
# 19:22:15
```

Декораторы: числа Фибоначчи

```
def fib(n):
    if n > 2:
        return fib(n - 1) + fib(n - 2)
    return 1

%timeit -r 1 fib(35)
2.2 s ± 0 ns per loop
```

Как можно ускорить?

Декораторы: memoize

```
def memoize(func):
    cache = {}
    def inner(*args, **kwargs):
        key = args
        if key not in cache:
            cache[key] = func(*key)
        return cache[key]
    return inner

@memoize
def fib(n):
    if n > 2:
        return fib(n - 1) + fib(n - 2)
    return 1
```

Будет ли работать?

Декораторы: memoize

```
def memoize(func):
    cache = {}
    def inner(*args, **kwargs):
        key = args + tuple(sorted(kwargs.items()))
        if key not in cache:
            cache[key] = func(*key)
        return cache[key]
    return inner
@memoize
def fib(n):
    if n > 2:
        return fib(n - 1) + fib(n - 2)
    return 1
%timeit -r 1 fib(35)
484 ns ± 0 ns per loop
```

Декораторы: lru_cache

```
from functools import lru_cache
@lru_cache
def fib(n):
    if n > 2:
        return fib(n - 1) + fib(n - 2)
    return 1
%timeit -r 1 fib(35)
76.3 ns ± 0 ns per loop
```

Декораторы: Flask приложение (1)

Декоратор для проверки токенов:

```
api_tokens = ["TRUE_TOKEN"]
def validate_api_token(validation_func):
    def decorator(func):
        @wraps(func)
        def decorated_function(*args, **kws):
            api_token = request.headers.get('Token')
            is_valid_api_token = validation_func(api_token)
            if is_valid_api_token:
                return func(*args, **kws)
            return 'Invalid API Token', 401
        return decorated_function
    return decorator
def simple_api_token_validation(api_token):
    return api_token in api_tokens
```

Декораторы: Flask приложение (2)

Сервер:

```
from flask import Flask, request
from functools import wraps

app = Flask(__name__)

@app.route('/', methods=['POST'])
@validate_api_token(simple_api_token_validation)
def post_hello():
    return 'POST Hello, World!'

if __name__ == '__main__':
    app.run()
```

Декораторы: Flask приложение (3)

Запуск сервера:

```
python3 main.py
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Serving Flask app 'main' (lazy loading)
 * Environment: production
   WARNING: This is a development server.
   Use a production WSGI server instead.
 * Debug mode: off
```

Отправка запроса:

```
curl -X POST http://127.0.0.1:5000/ -H 'token: TRUE_TOKEN'
POST Hello, World!
```

Задачи

Задачи: doc

- **doc** - функция, которая выводит информацию о функции (модуль, имя, документация)

```
>>> @doc
... def f(x):
...     """DOC"""
...     return x
...
>>> f(5)
__main__
f
DOC
5
```

Задачи: counting

- **counting** - считает количество вызовов функции и записывает в атрибут **call**

```
>>> @counting
... def f(x):
...     return x
...
>>> f(0)
0
>>> f(1)
1
>>> f.ncalls
2
```

Задачи: once

- **once** - позволяет только один раз вызвать функцию

```
>>> @once
... def init():
...     print("Init.")
...
>>> init()
Init.
>>> init()
```

Задачи: partial, timer

- **partial** - фиксация позиционных и ключевых аргументов

```
>>> basetwo = partial(int, base=2)
>>> basetwo("10010")
18
```

- **timer** - вычисляет среднее время работы функции

```
>>> result = timer(n_iter=5)(sum)(range(10**7))
0 0.3050308240053709 6
1 0.3600884760089684 3
2 0.3906337890075519 7
3 0.3495317990018520 5
4 0.3489224300137721
Mean time is -  0.3508414636075031
```

Задачи: memorize, const

- **memorize** - хранит результаты выполнения функции

```
>>> @memorize
... def fib(n):
...     if n > 2:
...         return fib(n - 1) + fib(n - 2)
...     return 1
...
>>> timer(n_iter=1)(fib)(105)
0 1.0908988770097494e-05
Mean time is -  1.0908988770097494e-05
392841376460687116573O
```

- **const** - проверяет, что переданные аргументы не были изменены, иначе выводит на экран ошибку

Задачи: precondition, postcondition

- **precondition** - проверка предусловия

```
>>> @precondition(lambda x: x != 0, "null argument")
... def f(x):
...     return 5 / x
>>> f(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in inner
AssertionError: null argument
```

- **postcondition** - проверка постусловия

Спасибо за внимание!