Декораторы. ООП и магические методы в Python

@pvavilin

5 февраля 2022 г.

Outline

Что такое декораторы

```
def decorator(func):
    def inner(*args, **kwargs):
        print(
            "Сейчас будет выполнена "
            f"функция: {func.__name__}"
        )
        result = func(*args, **kwargs)
        print(
            f"функция {func.__name__} "
            "успешно выполнена"
        )
        return result
    return inner
```

Что такое декораторы

```
def mysum(x, y):
    print(f"x + y = {x+y}")
    return x + y

sum_decorated = decorator(mysum)
sum_decorated(6, 7)
```

```
Сейчас будет выполнена функция: mysum
x + y = 13
функция mysum успешно выполнена
```

Области видимости

```
def decorator(func):
    counter = 0
    def inner(*args, **kwargs):
        counter += 1
        result = func(*args, **kwargs)
        print(
            "Функция выполнена "
            f"{counter} раз"
        )
        return result
    return inner
```

Области видимости

```
counter = 0
def decorator(func):
    def inner(*args, **kwargs):
        counter += 1
        result = func(*args, **kwargs)
        print(
            "Функция выполнена "
            f"{counter} раз"
        )
        return result
    return inner
```

nonlocal

```
def decorator(func):
    counter = 0
    def inner(*args, **kwargs):
        nonlocal counter
        counter += 1
        result = func(*args, **kwargs)
        print(
            "Функция выполнена "
            f"{counter} раз"
        )
        return result
    return inner
```

global

```
counter = 0
def decorator(func):
    def inner(*args, **kwargs):
        global counter
        counter += 1
        result = func(*args, **kwargs)
        print(
            "Функция выполнена "
            f"{counter} раз"
        )
        return result
    return inner
```

Всё понятно, но вот конкретно...

```
counter = {}
def decorator(func):
    counter[func.__name__] = 0
    def inner(*args, **kwargs):
        counter[func.__name__] += 1
        result = func(*args, **kwargs)
        print(counter[func.__name__])
        return result
    return inner
mysum_decorated = decorator(mysum)
mysum_decorated(2, 2)
mysum_decorated(2, 2)
mysum_decorated(3, 1)
```

```
1
2
3
```

Не, всё понятно, но вот конкретно...

```
def decorator(func):
    counter = [0]
    def inner(*args, **kwargs):
        counter[0] += 1
        result = func(*args, **kwargs)
        print(counter[0])
        return result
    return inner
mysum_decorated = decorator(mysum)
mysum_decorated(2, 2)
mysum_decorated(2, 2)
mysum_decorated(3, 1)
```

```
1
2
3
```

Вроде всё понятно...

```
x = 0
y = 0
def f():
    x = 1
    y = 1
    class C:
        # что будет напечатано?
        print(x, y)
        x = 2
f()
```

И как это понимать?

```
x = 0
y = 0
def f():
    x = 1
    y = 1
    class C:
        # что будет напечатано?
        print(x, y)
        x = 2
f()
```

```
0 1
```

Замыкания

```
def cached(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            print("Положить в кеш")
            cache[args] = func(*args)
        else:
            print("Результат из кеша")
        return cache[args]
    return wrapper

@cached
def mysum(x, y):
    return x + y
```

Замыкания

```
mysum(1, 2)
mysum(1, 2)
mysum(2, 3)
mysum(2, 3)
mysum(1, 2)
```

```
Положить в кеш
Результат из кеша
Положить в кеш
Результат из кеша
Результат из кеша
```

Замыкания

```
def cached(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            print("Положить в кеш")
            cache[args] = func(*args)
        else:
            print("Результат из кеша")
        return cache[args]
    def invalidate(*args):
        print("Сбросить кеш")
        del cache[args]
        return wrapper(*args)
    wrapper.invalidate = invalidate
    return wrapper
```

Замыкания

```
mysum(1, 2)
mysum(1, 2)
mysum(2, 3)
mysum(2, 3)
mysum(1, 2)
mysum.invalidate(2, 3)
```

```
Положить в кеш
Результат из кеша
Положить в кеш
Результат из кеша
Результат из кеша
Сбросить кеш
Положить в кеш
```

Декораторы и имя функции

```
def mysum(x,y):
    """MYSUM"""
    return x+y

mysum_decorated = decorator(mysum)
print(mysum.__name__)
print(mysum.__doc__)
print(mysum_decorated.__name__)
print(mysum_decorated.__doc__)
```

```
mysum
MYSUM
inner
None
```

wraps

```
from functools import wraps

def decorator(func):
    @wraps(func)
    def inner(*args, **kwargs):
        """INNER"""
        return func(*args, **kwargs)
    return inner

mysum_decorated = decorator(mysum)
print(mysum_decorated.__name__)
print(mysum_decorated.__doc__)
```

```
mysum
This is mysum function
```

Декораторы с аргументами

```
def benchmark(iters=3):
    def decorator(func):
        def wrapper(*a, **k):
            total = 0
            for i in range(iters):
                start = time.time()
                result = func(*a, **k)
                end = time.time()
                total += (end - start)
            print("AVG: "
                f"{total/iters:.4f}")
            return result
        return wrapper
    return decorator
```

Декораторы с аргументами

```
@benchmark()
def countdown(n):
    while n > 0:
        n -= 1

countdown(int(5e7))
```

```
AVG: 2.4185
```

Декораторы с аргументами

```
def countdown(n):
    while n > 0:
        n -= 1

countdown_decorated = \
    benchmark(5)(countdown)
countdown_decorated(int(5e7))
```

```
AVG: 2.3882
```

classmethod

Принимает в качестве первого аргумента сам класс cls а не объект self.

```
class MyDict:
    def __init__(self, d):
        self.data = d
    @classmethod
    def from_pairs(cls, pairs):
        return cls(dict(pairs))
pairs = (("a", 1), ("b", 2))
print(MyDict.from_pairs(pairs).data)
```

```
{'a': 1, 'b': 2}
```

staticmethod

Не привязан ни к текущему объекту self ни к классу cls.

```
import os

class Executor:
    def __init__(self, command):
        self.command = command
    @staticmethod
    def chdir(path):
        os.chdir(path)
    def __call__(self):
        return (
            os.popen(self.command)
            .read().strip()
        )
```

staticmethod

```
orig_path = os.getcwd()
executor = Executor("ls|wc -l")
print(os.getcwd())
print(executor())
Executor.chdir("/tmp/")
print(os.getcwd())
print(executor())
executor.chdir(orig_path)
print(os.getcwd())
```

```
/home/pimiento/yap/decorators_and_oop
33
/tmp
72
/home/pimiento/yap/decorators_and_oop
```

Классы-декораторы

```
class Decorator:
    def __cal__(self, fn):
        def wrapper(*a, **kw):
            print("BEFORE")
            result = fn(*a, **kw)
            print("AFTER")
            return result
        return wrapper

@Decorator
def mysum(x, y):
    return x + y
```

Класссы-декораторы и замыкания

пример на GitHub

Больше про декораторы

TheDecoratorsTheyWontTellYouAbout

магические методы классов в Python

magicmethods

property

```python
@dataclass
class A:
    __x: int
    @property
    def x(self):
        return self.__x
a = A(10)
print(a.x)
try:
    a.x = 100
except Exception as e:
    print(e)
```

```
10
can't set attribute
```

getter/setter/deleter

```
@dataclass
class A:
    __x: list
    @property
    def x(self):
        return self.__x[::]
    @x.setter
    def x(self, value):
        self.__x.append(value)
    @x.deleter
    def x(self):
        self.__x = []
```

getter/setter/deleter

```
a = A([])
print(a.x)
a.x = 10
print(a.x)
a.x = 100
print(a.x)
del a.x
print(a.x)
```

```
[]
[10]
[10, 100]
[]
```

Singleton

```
class Logger:
    def __init__(self):
        pass
l1 = Logger()
l2 = Logger()
print(l1 is l2)
```

```
False
```

Singleton

```
class Logger:
    _instance = None
    def __init__(self):
        raise RuntimeError("Call new() i
    @classmethod
    def new(cls):
        if cls._instance is None:
            cls._instance = cls.__new__
        return cls._instance
l1 = Logger.new()
l2 = Logger.new()
print(l1.__class__.__name__)
print(l1 is l2)
```

```
Logger
True
```

Singleton

```
class Logger:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__ne
        return cls._instance
l1 = Logger()
l2 = Logger()
print(l1.__class__.__name__)
print(l1 is l2)
```

```
Logger
True
```

Дополнительные материалы

Паттерны проектирования на Python

Head First

Вопросы