

Декораторы в Python

Что это такое и чем полезно

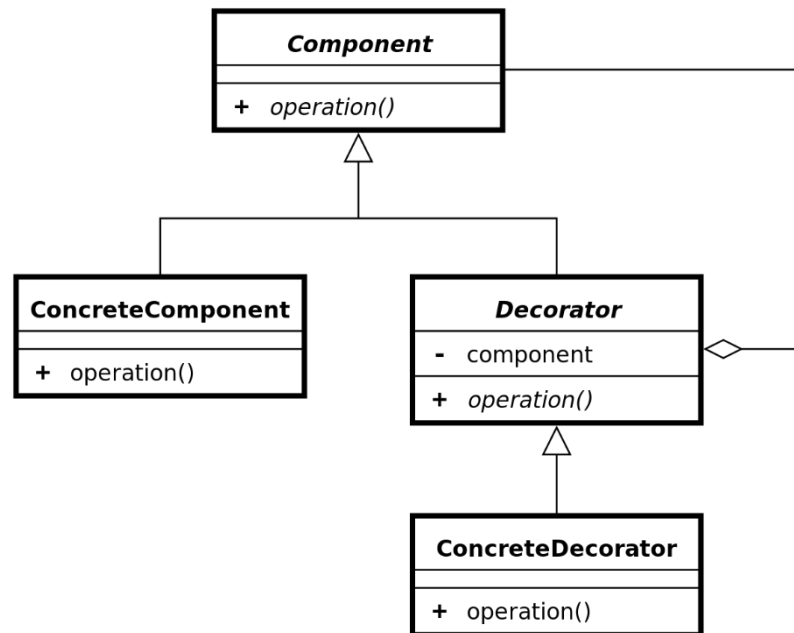


Структура презентации

1. Что такое декоратор
2. Чем полезны декораторы
3. Декорирование в Python и эквивалентная запись
4. Множественное декорирование
5. Пример простого декоратора
6. Параметризованный декоратор
7. `functools.wraps`
8. Декоратор, основанный на классе
9. Декорирование класса
10. Декорирование метода
11. Побочные действия декораторов
12. Популярные встроенные декораторы

Декоратор

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.



Чем полезен декоратор

- Добавляет функциональность к объекту без необходимости непосредственного изменения его кода
- Сокращает дублирование кода
- Упрощает чтение программы
- Увеличивает переиспользование кода
- Упростить поддержание

Декорирование в Python

```
@decorator
def foo(number, person):
    pass
```

=>

```
def foo(number, person):
    pass
foo = decorator(foo)
```

```
@decorator_with_params(param=5)
def bar(shift, date):
    pass
```

=>

```
def bar(shift, date):
    pass
bar = decorator_with_params(param=5)(bar)
```

Множественное декорирование

```
@decorator
@decorator_with_params(param=6)
@another_decorator
def spam(expected_date):
    pass
```

⇓

```
def spam(expected_date):
    pass
spam = decorator(decorator_with_params(param=6)(another_decorator(spam)))
```

Пример простого декоратора

Декорирует функцию, выводит сообщение до и после вызова функции

```
def inform_about_call(function):
    prefix = f'Function {function.__name__}: '
    def wrapper(*args, **kwargs):
        print(prefix + 'before call')
        result = function(*args, **kwargs)
        print(prefix + 'after call')

        return result

    return wrapper

@inform_about_call
def foo(a):
    print(f'foo called with {a}')

    return a
```

```
assert foo(1) == 1
# Function foo: before call
# foo called with 1
# Function foo: after call

assert foo(45) == 45
# Function foo: before call
# foo called with 45
# Function foo: after call
```

Параметризованный декоратор

Модифицированная версия декоратора предыдущего слайда

```
def print_before_and_after_call(before='Hello!', after='Bye'):  
    def inner_decorator(function):  
        def wrapper(*args, **kwargs):  
            print(before)  
            result = function(*args, **kwargs)  
            print(after)  
  
            return result  
  
        return wrapper  
  
    return inner_decorator
```

```
@print_before_and_after_call(before='Before foo call!', after='After call!')  
def foo(a):  
    print(f'foo called with {a}')  
  
    return a
```

```
assert foo(1) == 1  
# Before foo call!  
# foo called with 1  
# After call!
```

```
assert foo(45) == 45  
# Before foo call!  
# foo called with 45  
# After call!
```


Проблема интроспекции функции

Декоратор подменяет декорируемую функцию оберткой, поэтому значения специальных атрибутов меняются. Но это можно поправить.

```
def simple_decorator(func):
    def wrapper(*args, **kwargs):
        print('Hello')

        return func(*args, **kwargs)

    return wrapper
```

```
@simple_decorator
def foo():
    pass
```

```
print(foo.__name__) # -> wrapper
```

```
from functools import wraps
```

```
def decorator_with_wraps(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Hello')

        return func(*args, **kwargs)

    return wrapper
```

```
@decorator_with_wraps
def bar():
    pass
```

```
print(bar.__name__) # -> bar
```

Декоратор, основанный на классе

```
class Decorator:
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        return self.func(*args, **kwargs)
```

```
@Decorator
def foo(a):
    print(f'foo called with {a}')

    return a
```

```
foo(10) # -> foo called with 10
foo(12) # -> foo called with 12
print(foo.count) # -> 2
print(type(foo)) # -> <class '__main__.Decorator'>
```

Декорирование класса

Декорировать можно не только функции, но и классы и их методы.

```
def count_instances(specific_class):
    count = 0
    def wrapper(*args, **kwargs):
        nonlocal count
        count += 1
        print(f'Instances: {count}')

        return specific_class(*args, **kwargs)
    return wrapper

@count_instances
class Spam:
    def __init__(self):
        print('Spam init.')

a = Spam()
# Instances: 1
# Spam init.
b = Spam()
# Instances: 2
# Spam init.
print(type(a))
# <class '__main__.Spam'>
print(type(Spam))
# <class 'function'>
```

Декорирование метода

```
def method_decorator(method):
    def wrapper(*args, **kwargs):
        print(f'Method {method.__name__} called with {args}, {kwargs}')

        return method(*args, **kwargs)

    return wrapper


class Spam:
    def __init__(self, a):
        self.a = a

    @method_decorator
    def increment(self):
        self.a += 1
        print(f'a incremented: {self.a}')


a = Spam(12)
a.increment()
# Method increment called with (<__main__.Spam object at 0x000000373AC51A20>,), {}
# a incremented: 13
```

Неправильный декоратор метода

```
class method_decorator:
    def __init__(self, method):
        self.method = method

    def __call__(self, *args, **kwargs):
        print(f'Method {self.method.__name__} called with {args}, {kwargs}')

        return self.method(*args, **kwargs)

class Spam:
    def __init__(self, a):
        self.a = a

    @method_decorator
    def increment(self):
        self.a += 1
        print(f'a incremented: {self.a}')

a = Spam(12)
a.increment()
# Method increment called with (), {}
# TypeError: increment() missing 1 required positional argument: 'self'
```

Декораторы с побочными эффектами

```
from functools import wraps

register = {}

def register_function(function):
    name = function.__name__
    if name in register:
        raise Exception(f'The name {name} has been registered already.')
    register[name] = 0

    @wraps(function)
    def wrapper(*args, **kwargs):
        register[name] += 1

        return function(*args, **kwargs)

    return wrapper

@register_function
def foo(a, b):
    return a, b

@register_function
def bar(c):
    print(c)

foo(1,2)
bar(1) # -> 1
bar(2) # -> 2
print(register) # -> {'foo': 1, 'bar': 2}
```

Популярные декораторы

```
class Spam:
    b = 1
    def __init__(self, b):
        self.b = b

    @classmethod
    def print_b(cls):
        print(cls.b)

    @staticmethod
    def print_message(message):
        print(message)

    @property
    def three_times_b(self):
        return self.b * 3

a = Spam(12)


assert a.b == 12
a.print_b() # -> 1
Spam.print_b() # -> 1
a.print_message('Hello!') # -> Hello!
Spam.print_message('Bye!') # -> Bye!
assert a.three_times_b == 36
```

```
from contextlib import contextmanager

@contextmanager
def generator_function(a):
    try:
        print('Enter context')
        yield a
    finally:
        print('Exit context')

with generator_function(5) as context: # -> Enter context
    assert context == 5
    print('Within context') # -> Within context
# -> Exit context
```

**Спасибо за
внимание**

 **artezio_software**

 **info@artezio.com**

www.artezio.com