Теоретический материал к занятию Декораторы 2 занятие.

Декоратор, как мы помним - функция-обертка. Но можно создать и класс декоратор. Давайте рассмотрим это на примере.

Добавив метод __call__ в класс, его можно превратить в вызываемый объект. А поскольку декоратор — это всего лишь функция, то есть, вызываемый объект, класс можно превратить в декоратор с помощью функции __call__.

```
class Decorator:
    def __init__(self, func):
        print('> Класс Decorator метод __init__')
        self.func = func

    def __call__(self):
        print('> перед вызовом класса...', self.func.__name__)
        self.func()
        print('> после вызова класса')

@Decorator
def wrapped():
    print('функция wrapped')

print('>> старт')
wrapped()
print('>> конец')
```

Результат

```
> Класс Decorator метод __init__
>> старт
> перед вызовом класса... wrapped
функция wrapped
> после вызова класса
>> конец
```

Отличие в том, что класс инициализируется при объявлении. Он должен получить функцию в качестве аргумента для метода __init__. Это и будет декорируемая функция.

При вызове декорируемой функции на самом деле вызывается экземпляр класса. А поскольку объект вызываемый, то вызывается функция __call__.

А что если функция, которую требуется декорировать, должна получать аргументы? Для этого нужно вернуть функцию с той же сигнатурой, что и у декорируемой.

```
def decorator_with_args(func):
    print('> декоратор с аргументами...')
    def decorated(a, b):
        print('до вызова функции', func.__name__)
        ret = func(a, b)
        print('после вызова функции', func.__name__)
        return ret
    return decorated

@decorator_with_args
def add(a, b):
    print('функция 1')
    return a + b

@decorator_with_args
def sub(a, b):
    print('функция 2')
    return a - b

print('>> старт')
r = add(10, 5)
print('r:', r)
g = sub(10, 5)
print('g:', g)
print('>> конец')
```

Результат

```
> декоратор с аргументами...
> декоратор с аргументами...
>> старт
до вызова функции add
функция 1
после вызова функции add
r: 15
до вызова функции sub
функция 2
после вызова функции sub
g: 5
>> конец
```

С классом тот же принцип. Нужно лишь добавить желаемую сигнатуру в функцию __call__.

```
class Decorator:
    def __init__(self, func):
        print('> Класс Decorator метод __init__')
        self.func = func

    def __call__(self, a, b):
        print('> до вызова из класса...', self.func.__name__)
        self.func(a, b)
        print('> после вызова из класса')

@Decorator
```

```
def wrapped(a, b):
    print('функция wrapped:', a, b)

print('>> старт')
wrapped(10, 20)
print('>> конец')
```

Результат

```
> Класс Decorator метод __init__
>> старт
> до вызова из класса... wrapped
функция wrapped: 10 20
> после вызова из класса
>> конец
```

Можно использовать *args и **kwargs и для функции wrapper, если сигнатура заранее неизвестна, или будут приниматься разные типы функций.

args и kwargs используются, когда мы не знаем заранее сколько аргументов передаст пользователь. args - обычные аргументы, переданные через запятую, а kwargs - именованные аргументы. Давайте рассмотрим на примере обычной функции.

```
def func(*args, **kwargs):
    print(f'{args=}')
    print(f'{kwargs=}')

func(1, 2, 3, a=5, b=1)
```

Результат

```
args=(1, 2, 3)
kwargs={'a': 5, 'b': 1}
```