Теоретический материал к занятию Декораторы.

Для того, чтобы понять, как работают декораторы, в первую очередь следует вспомнить, что функции в python являются объектами, соответственно, их можно возвращать из другой функции или передавать в качестве аргумента. Также следует помнить, что функция в python может быть определена и внутри другой функции.
Вспомнив это, можно смело переходить к декораторам. Декораторы — это, по сути, "обёртки", которые дают нам возможность изменить поведение функции, не изменяя её код.

По сути декоратор — это функция, которая принимает функцию, делает что-то и возвращает другую функцию.

```
def decorator(func):
    def wrapper():
        print('функция-оболочка')
        func()
    return wrapper

def basic():
    print('основная функция')

wrapped = decorator(basic)
print('старт программы')
basic()
wrapped()
print('конец программы')
```

Результат

```
старт программы
основная функция
функция-оболочка
основная функция
конец программы
```

Разберемся с тем, что здесь произошло. Функция decorator — это, как можно понять по названию, декоратор. Она принимает в качестве параметра функцию func. Крайне оригинальные имена. Внутри функции объявляется другая под названием wrapper. Объявлять ее внутри не обязательно, но так проще работать.

В этом примере функция wrapper просто вызывает оригинальную функцию, которая была передана в декоратор в качестве аргумента, но это может быть любая другая функциональность.
В конце возвращается функция wrapper. Напомним, что нам все еще нужен вызываемый объект. Теперь результат можно вызывать с оригинальным набором возможностей, а также новым включенным кодом.

Но в Python есть синтаксис для упрощения такого объявления. Чтобы декорировать функции, используется символ @ рядом с именем декоратора. Он размещается над функцией, которую требуется декорировать

```
def decorator(func):
    '''Основная функция'''
    print('декоратор')
    def wrapper():
        print('-- до функции', func.__name__)
        func()
        print('-- после функции', func.__name__)
    return wrapper

@decorator
def wrapped():
    print('--- обернутая функция')

print('- старт программы...')
wrapped()
print('- конец программы')
```

Результат

```
декоратор
- старт программы...
-- до функции wrapped
--- обернутая функция
-- после функции wrapped
- конец программы
```

Можно использовать тот же декоратор с любым количеством функций, а также — декорировать функцию любым декоратором.

```
def decorator_1(func):
    print('декоратор 1')
    def wrapper():
        print('перед функцией')
        func()
    return wrapper

def decorator_2(func):
    print('декоратор 2')
    def wrapper():
        print('перед функцией')
        func()
    return wrapper

@decorator_1
@decorator_2
def basic_1():
    print('basic_1')

@decorator_1
def basic_2():
    print('basic_2')
```

```
print('>> старт')
basic_1()
basic_2()
print('>> конец')
```

Результат

```
декоратор 2
декоратор 1
декоратор 1
>> старт
перед функцией
перед функцией
basic_1
перед функцией
basic_2
>> конец
```

Когда у функции несколько декораторов, они вызываются в обратном порядке относительно того, как были вызваны. То есть, такой вызов:

```
@decorator_1
@decorator_2
def wrapped():
    pass
```

аналогичен

```
a = decorator_1(decorator_2(wrapped))
```

Если декорированная функция возвращает значение, и его нужно сохранить, то нужно сделать так, чтобы его возвращала и функция-обертка.