Декораторы.

Декораторы

Как мы уже выяснили, функции в python также являются объектами:

```
def debug_with_args(f, *args, **kwargs):
    print(f'Calling function {f.__name__} with args {args} and
{kwargs}')
    return f(*args, **kwargs)

def sum(a, b):
    return a + b

print(debug_with_args(sum, 1, 2))
# Calling function sum with args (1, 2) and {}
# 3
```

Декораторы

На предыдущем слайде мы сделали так, чтобы при вызове функции выводилась дополнительная информация о функции и аргументах.

Но для этого нам пришлось делегировать вызов `sum` другой функции, что не всегда удобно.

Вместо этого можно было бы изменить саму функцию `sum`.

Декораторы

```
def make_debug_with_args(f):
    def not_f(*args, **kwargs):
        print(f'Calling function {f.__name__} with args {args}
and {kwargs}')
        return f(*args, **kwargs)

    return not_f

def sum(a, b):
    return a + b
sum = make_debug_with_args(sum)

print(sum(1, 2))
# Calling function sum with args (1, 2) and {}
# 3
```

Декораторы

Функции, наподобие `make_debug_with_args` в python называются декораторами.

Т.е. декоратор — это такая функция-обертка, которая принимает функцию и возвращает функцию.

Декоратор может быть применен к функции с помощью специального синтаксиса:

```
@decorator
def foo():
    ...
```

Декораторы

На самом деле декоратор может быть применён и к классам и к методам, тогда он должен возвращать класс/метод соответственно.

```
def id(cls):
    return cls

@id
class A:
    pass
```

Декораторы

```
def make_debug_with_args(f):
    def not_f(*args, **kwargs):
        print(f'Calling function {f.__name__} with args {args}
and {kwargs}')
        return f(*args, **kwargs)

    return not_f

# тоже самое, что и sum = make_debug_with_args(sum)
@make_debug_with_args
def sum(a, b):
    """Nonempty docstring"""
    return a + b

print(sum(1, 2))
# Calling function sum with args (1, 2) and {}
# 3
```

Декораторы

Но с функцией, созданной таким образом, могут быть некоторые сложности: важные атрибуты функции будут потеряны, как и доступ к оригинальной функции.

```
print(sum.__name__)
# not_f

print(sum.__doc__)
# None
```

Декораторы

Первый (и плохой) способ это исправить — копировать атрибуты в декораторе:

```
def my_deco(f):
    def inner(*args, **kwargs):
        print("Red")
        return f(*args, **kwargs)

    inner.__wrapped__ = f
    inner.__doc__ = f.__doc__
    inner.__name__ = f.__name__
    return inner
```

Декораторы

Второй способ — использовать `functools.update`

```
import functools

def my_deco(f):
    def inner(*args, **kwargs):
        print("Red")
        return f(*args, **kwargs)

    functools.update_wrapper(inner, my_deco)
    return inner
```

Декораторы

Третий (и самый адекватный) способ — использовать декоратор `functools.wraps`.

```
import functools

def my_deco(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        print("Red")
        return f(*args, **kwargs)

    return inner
```

Как реализован wraps?

Как ещё не надо писать декораторы

```
import functools

def deco(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        # это бесполезный декоратор!
        return f(*args, **kwargs)
    return inner

def deco(f):
    # f -- это функция, она изменяемая, можно
    # её изменить в этом месте,
    # вернуть её же саму также никто не мешает
    return f
```

Декораторы

Декоратор, который добавляет "статическую" переменную функции:

```
def static_var(name, value):
    def deco(f):
        # а где functools.wraps?
        setattr(f, name, value)
        return f

    return deco


@static_var("calls", 42)
def foo():
    return foo.calls
print(foo())
# 42
```

Декораторы

Декоратор, который считает сколько раз функция была вызвана:

```
def count_calls(f):
    @functools.wraps(f)
    def inner(*args, **kwargs):
        inner.counter += 1
        return f(*args, **kwargs)
    inner.counter = 0

    return inner
```

Декораторы

```
def count_calls2(f):
    counter = 0
    @functools.wraps(f)
    def inner(*args, **kwargs):
        nonlocal counter
        counter += 1
        return f(*args, **kwargs)

    return inner
```

Декораторы

К одной функции может быть применено несколько декораторов, при этом порядок применения важен:

Декораторы

```
def deco_print_e(f):
    print('e')
    return f

def deco_print_o(f):
    print('o')
    return f

@deco_print_o
@deco_print_e
def a(): # при создании a будет выведено 'e', затем 'o'
    pass

@deco_print_e
@deco_print_o
def b(): # при создании b будет выведено 'o', затем 'e'
    pass
```

Декораторы с аргументами

Напишем декоратор, который можно было бы применять с опциональными аргументами:

```
@debug
def foo():
    pass

@debug(description="Bar func from __main__")
def bar():
    pass
```

Декораторы с аргументами

Поймем для этого, что код с предыдущего слайда аналогичен следующему:

```
def foo():
    pass
foo = debug(foo)

def bar():
    pass
bar = debug(description="Bar func from __main__")(bar)
```

Декораторы с аргументами

Т.е. `debug` в зависимости от переданных аргументов возвращает

- либо новую функцию (если передали только функцию), которую нужно использовать вместо продекорированной,
- либо декоратор (если передали "опциональный" аргумент), который должен быть применен к функции.

Декораторы с аргументами

```
from functools import wraps
def debug(f=None, *, description=""):
    def deco(f):
        @wraps(f)
        def inner(*args, **kwargs):
            print(f"Called {f.__name__} with arg {args}
{kwargs}")
            return f(*args, **kwargs)

        return inner

    # если передали функцию, то нужно ее продекорировать
    if f is not None:
        return deco(f)
    # иначе возвращаем декоратор
    return deco
```

Что ещё почитать?

Документацию к модулю functools, конечно.

Обратите внимание на декораторы:

- `lru_cache`
- `total_ordering`
- `singledispatch`

Как они могли бы быть реализованы?