Декораторы на python

Суханов А.Я.

Зачем нужны декораторы?

- Можно заменить какую-либо функцию, чаще всего стороннюю, добавив дополнительные особенности при ее выполнении.

Декоратор. Функция более высокого порядка.

```
from random import random

def decorator(func):
    def wrapper_for_func():
        print("Запускаем что-либо до вызова функции")
        x = func()
        print(x)
        print("Запускаем что-либо после вызова функции")
        return x
    return wrapper_for_func

print(random())
random = decorator(random)
print(random())
```

Результат работы декоратора

- 0.5948731855992081
- Запускаем что-либо до вызова функции
- 0.5985262870159087
- Запускаем что-либо после вызова функции
- 0.5985262870159087

Декоратор с использованием синтаксиса языка

```
from random import random
def decorator_name(func):
    def wrapper_for_func():
        print("Запускаем что-либо до вызова функции")
        x = func()
        print(x)
        print("Запускаем что-либо после вызова функции")
        return x
    return wrapper_for_func
@decorator_name
def add():
    return 1
print(add())
```

Результат

- Запускаем что-либо до вызова функции
- 1
- Запускаем что-либо после вызова функции
- 1

Декоратор для определения времени работы функции

```
import time
def decorator_time(func):
    def wrapper_for_func():
        print("")
        t = time.time()
        x = func()
        dt = time.time()-t
        print(f"Время исполнения функции {func.__name__} равно {dt}")
        return x
    return wrapper_for_func
@decorator_time
def add():
    i = 0.0
    for i in range(10000):
        i=i+i
    return i
print(add())
```

Результат

- Время исполнения функции add равно 0.0006504058837890625
- 19998
- >>>

```
@decorator_time
def add():
    i = 0.0
    for i in range(10000):
        i=i+i
    return i
print(add())
@decorator_time
def rand():
    return random.random()
print(rand())
```

- Время исполнения функции add равно 0.0008225440979003906
- 19998

- Время исполнения функции rand равно 2.86102294921875e-06
- 0.8971549149038762
- >>>

Декоратор для определения количества запусков функции.

```
def decorator_count(func):
    def wrap_func():
        wrap_func.__ecount__ += 1
        return func()
    wrap_func.__ecount__ = 0
    return wrap_func

@decorator_count
def my_func():
    boomboom = 0
    return boomboom

for i in range(100):
    my_func()

print(my_func.__ecount__)
```

```
import time
def decorator_time(func):
    def wrapper_for_func():
            t = time.time()
            print(f"time start")
            x = func()
            dt = time.time()-t
            print(f"Время исполнения функции
{func.__name__} равно {dt}")
            print(f"time end")
            return x
    return wrapper_for_func
def decorator_count(func):
    def wrap_func():
            print("count start")
            wrap_func.__ecount__ += 1
            x = func()
            print("count end")
            return x
    wrap_func.__ecount__ = 0
    return wrap_func
```

Декоратор над декоратором над функцией.
Если поменять местами count и time,
__ecount__ уже не будет доступен.

```
@decorator_count
@decorator_time
def my_func():
    boomboom = 0
    return boomboom
for i in range(2):
    print(my_func())
    print(my_func.__ecount__)
```

Результат

- count start
- time start
- Время исполнения функции my_func равно 5.245208740234375e-06
- time end
- count end
- 0
- 1
- count start
- time start
- Время исполнения функции my_func равно 6.6757021484375e-06
- time end
- count end
- 0
- 2

Декораторы с аргументами

```
def decorator_with_args(func):
    def wrap_func(x=0,y=0):
        print(f"Get for {func.__name__} x = {x} y = {y}")
        return func(x,y)

    return wrap_func

@decorator_with_args
def add(x=0,y=0):
    return x+y

print(add(x=2,y=2))
```

Get for add x = 2 y = 2
4

Неименованные аргументы

```
def decorator_with_args(func):
    def wrap_func(arg1,arg2):
        print(f"Get for {func.__name__} x = {arg1} y = {arg2}")
        return func(arg1,arg2)

    return wrap_func

@decorator_with_args
def add(x,y):
    return x+y

print(add(2,2))
```

Неименованные аргументы

```
def decorator_with_args(func):
    def wrap_func(*arg):
        print(f"Get for {func.__name__} {arg}")
        return func(*arg)

    return wrap_func

@decorator_with_args
def add(x,y):
    return x+y

print(add(2,2))
```

Разыменовывание *,** в f{} использовать нельзя

```
def decorator_with_args(func):
    def wrap_func(*arg, **kwargs):
        print(f"Get for {func.__name__} {arg} {kwargs}")
        return func(*arg, **kwargs)

    return wrap_func

@decorator_with_args
def add(z, x =0 ,y= 0):
    return x+y+z

print(add(1, x = 2, y = 2))
```

```
Get for add (1,)  {'x': 2, 'y': 2}
5
>>>
```

Модуль functools

- Содержит полезные декораторы
- lru_cache (кэширование последних max вызовов функции)
- partials (создание функции с частично заданными аргументами)
- singledispatch (перегрузка функции в соответствии с типом первого аргумента)
- wraps (декоратор для внутренней wrap функции, чтобы, например, по имени возвращалось описание самой функции, а не врапера)

lru_cache

- Кэширует результат работы функции.
- Можно использовать, например, при обращении к данным на сайте, после чего они будут кэшированы.
- При расчетах какой-либо функции, если ее какие-либо входные данные могут быть теми же самыми.

```
@lru_cache(30)
def example(a,b,func,*args):
    e = 1e-9
    def romberg(a1,b1,n1,n2):
        iv = 0.0
        dx1 = (b1-a1)/n1
        f1 = func(a1,*args)
        for i in range(n1):
            x = a1+(i+1)*dx1
            f2 = func(x,*args)
            sq = (f1+f2)*dx1*0.5
            ff1 = f1
            f1 = f2
            x1 = a1+i*dx1
            x2 = x
            dx2 = (x2-x1)/n2
            sqv = 0.0
            for j in range(n2):
                x = x1+(j+1)*dx2
                ff2 = func(x,*args)
                sqv = sqv+(ff1+ff2)*dx2*0.5
                ff1 = ff2
            if(abs(sqv-sq)>e):
                iv = iv+romberg(x1,x2,n1,n2)
            else:
                iv = iv + sqv
        return iv
    return romberg(a,b,2000,50)
```

Результаты

```
t = time.time()
print(example(-100,11,gauss,0.0,0.00005))
t = time.time() - t
print(f"time of first function {t}")
t = time.time()
print(example(-100,11,gauss,0.0,0.00006))
t = time.time() - t
print(f"time of second function {t}")
t = time.time()
print(example(-100,11,gauss,0,0.00005))
t = time.time() - t
print(f"time of third function {t}")
```

```
1.0000000000000795
time of first function 2.861814022064209
1.0000000000001261
time of second function 3.1440503597259 52
1.0000000000000795
time of third function 2.9802322387695312e-05
```

partial

```
from functools import partial

def add(x,y):
    return x+y
# преобразуем в функцию без аргументов
# суммирующую две константы
p_add0 = partial(add,2,5)
print(p_add0())

# преобразуем в функцию без аргументов
# суммирующую переменную с константу 2
p_add1 = partial(add,2)
print(p_add1(5))

# оставляем функцию с тем же
# количеством аргументов
p_add2 = partial(add)
print(p_add2(2,5))
```

Атрибуты возвращаемого partial-объекта

partial(func, args, keywords)

func Функция, к которой будет перенаправлен вызов с применением аргументов.

args Позиционные аргументы, которые будут переданы в вызываемую функцию при вызове объекта.

keywords Именованные аргументы, которые будут переданы в вызываемую функцию при вызове объекта.

Partial. Сценарии.

```
from functools import partial

def add(x, y):
        return x + y

def multiply(x, y):
        return x * y

def run(func):
        print(func())

a1 = partial(add, 1, 2)
m1 = partial(multiply, 5, 8)
run(a1)
run(m1)
```

singledispatch – сработает функция в соответствии с первым типом в register

```
from functools import singledispatch
@singledispatch
def add(a, b):
    raise NotImplementedError('Unsupported type error')
@add.register(int)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)
@add.register(str)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)
@add.register(list)
def _(a, b):
    print("First argument is of type ", type(a))
    print(a + b)
add(1, 2)
add('Python', 'Programming')
add([1, 2, 3], [5, 6, 7])
add(1.0, 1) # вернет ошибку Unsupported type
```

Результат

- First argument is of type <class 'int'>
- 3
- First argument is of type <class 'str'>
- PythonProgramming
- First argument is of type <class 'list'>
- [1, 2, 3, 5, 6, 7]
- Traceback (most recent call last):
- raise NotImplementedError('Unsupported type error')
- NotImplementedError: Unsupported type error

functools.wraps

```
from functools import wraps
def another_function(func):
    """
    Функция которая принимает другую функцию
    """
    @wraps(func)
    def wrapper():
        """
        Оберточная функция
        """
        val = "The result of %s is %s" % (func(),
        eval(func()))
        return val
    return wrapper

@another_function
def a_function():
    """Обычная функция"""
    return "1+1"

print(a_function.__name__) # a_function (без wraps wrapper)
print(a_function.__doc__) # Обычная функция(без wraps Оберточная функция)
```