Декораторы

Декоратор - если говорить простым языком - это возможность Python обернуть какую-либо функцию пользователя в еще одну функцию. Таким образом, когда мы запускаем нашу функцию - до нее и после нее будет выполнен определенный в декораторе код.

Декораторы являются более продвинутой темой и не все с ними умеют полноценно работать. Однако декораторы приносят большую пользу для упрощения работы со скриптами.

Рассмотрим пример простого декоратора:

```
def myDecorator( function = None ):
    def wrapper():
        print "Before"
        function()
        print "After"
    return wrapper

@myDecorator
def myFoo():
    print "body"

myFoo()

"""
Output:
    Any start code
    body
    Any end code

"""
```

В данном примере wrapper это непосредственно функция-обертка, которая добавляет функционал к нашей вызываемой функции. Знак @ означает что данная функция обернута в ту самую обертку. Если попытаться все это написать без декоратора, пример будет выглядеть вот так:

```
print "Before"
myFoo()
print "After"
```

Рассмотрим еще один пример, в котором мы проверяем аргумент функции, является ли он отрицательным или нет. Если отрицательный - вывести сообщение об ошибке.

```
def check_non_negative(f):
    def wrap(*args):
        for i in args:
            if i < 0:
                raise ValueError("Argument {} must be a positive number".format(i))
        return f(*args)
    return wrap

@check_non_negative
def test(*args):
    print args

test(3,4,-5,6,6)
```

Декораторы также могут быть с аргументами. Для этого мы должны саму функцию декоратора обернуть в еще одну функцию, в которую мы будем передавать наш аргумент. В примере ниже декоратор фактически начинается с функции divideBy, в которую мы передаем аргумент (в примере - 10). Следующая функция - та, в которую передается ссылка на декорируемую функцию. Далее следует уже сама функция-обертка, которая принимает args аргумент, который фактически содержит в себе все аргументы функции add.

[Пример на сл. странице]

```
def divideBy(arg1):
    """
    Divide acts like a modificator to our 'add' method
    It lets us not touch the actual 'add' method and
        process the result to give us new result
    """
    def wrap(f):

        def wrapper(*args):

            print ("wrapper begins")

            # passing *args means we pass original add() args - 17 and 3
            # if we wanna pass something different - we pass arg1 - 10
            result = f(*args)
            print ("wrapper ends")

            return result/float(arg1)

        return wrapper

    return wrap


@divideBy(10)
def add(x, y):
    return (x + y)

print (add(1,3)) # output 0.4
```

Декораторов может быть применено сразу несколько к функции. Каждый из декораторов выполняет свою роль, и служит оберткой для других декораторов (и только лишь один для самой функции).

```
@dec1
@dec2
@dec3
def foo()
```

Порядок вызова декораторов и самой функции следующий:

dec1 -> dec2 -> dec3 -> foo() -> dec3.close() -> dec2.close() -> dec1.close()

Рассмотрим пример:

```
def dec1(f):
    def wrapper(*args, **kwargs):
        print "Dec1 Start"
        f(*args, **kwargs)
        print "Dec1 End"
    return wrapper

def dec2(f):
    def wrapper(*args, **kwargs):
        print "Dec2 Start"
        f(*args, **kwargs)
        print "Dec2 End"
    return wrapper

def dec3(f):
    def wrapper(*args, **kwargs):
        print "Dec3 Start"
        f(*args, **kwargs)
        print "Dec3 End"
    return wrapper

@dec1
@dec2
@dec3
def Foo(value = 0):
    print value

Foo(13)

"""
Output:
  Dec1 Start
  Dec2 Start
  Dec3 Start
  0
  Dec3 End
  Dec2 End
  Dec1 End
"""
```