Python Decorator (Link)

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behavior of a function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

In the example we will create a simple example which will print some statement before and after the execution of a function.

Here are two ways to define a decorator:

```
# How to define a decorator – the first way
# Note that inputs & returns are func/class
def smart_divide(func):
    # parameters of the nested inner() function inside the decorator
    # is the same as the parameters of functions it decorates divide.
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return
        # 'return' is no needed in this case
        # but in general it will be useful if func returns anything
        return func(a, b)
    return inner

# How to use a decorator – the first way
@smart_divide
def divide(a, b):
    print(a/b)

# How to use a decorator – the second way
def divide(a, b):
    print(a/b)
divide = smart_divide(divide)

>> divide(2, 5)
>> divide(1, 0)
```

```
I am going to divide 2 and 5
0.4
I am going to divide 1 and 0
Whoops! cannot divide

# divide(2,5) => smart_divide(divide(2,5)) => inner(2,5) => print and
check => return func(2,5) i.e. return divide(2,5) => print(2/5)

# How to define a decorator - the second way uses *args and **kwargs
# See more details below
# An equivalent version to the 1st way
def smart_divide_v2(func):
    def inner(*args, **kwargs):
        print("I am going to divide", args[0], "and", args[1])
        if args[1] == 0:
            print("Whoops! cannot divide")
            return

        return func(*args, **kwargs)
    return inner

@smart_divide_v2
def divide(a, b):
    print(a/b)

divide(2, 5)
divide(1, 0)

Output:
In [59]: runfile('/Users/zli/.spyder-py3/temp.py', wdir='/Users/zli/.spyder-py3')
I am going to divide 2 and 5
0.4
I am going to divide 1 and 0
Whoops! cannot divide

I am going to divide 2 and 5
0.4
I am going to divide 1 and 0
Whoops! cannot divide
```

The inner function takes the argument as *args and **kwargs which means that *a tuple of positional arguments* or *a dictionary of keyword arguments* can be passed of any length. This makes it a general decorator that can decorate a function having any number of arguments.

```
# Example 1
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))

myFun(first='Geeks', mid='for', last='Geeks')

>> first == Geeks
>> mid == for
>> last == Geeks

# Example 2
def myFun(*args, **kwargs):
    print(args)
    print(kwargs)

myFun('Ready', 'Set', 'Go', first='Geeks', mid='for', last='Geeks')

>> ('Ready', 'Set', 'Go')
>> {'first': 'Geeks', 'mid': 'for', 'last': 'Geeks'}
```