Python Decorators

1. Understand Functions

In python, functions are objects.

- Functions can be passed as variables into other functions.
- Functions can be returned as variables from other functions.
- Functions can be defined inside another function, which are called **Inner Functions**.

Example:

Depends on input parameter, following function `my_func` returns different function.

- Note: Function object is returned **without** parentheses `()`

In [1]:

```
def my_func(name):
    def double(val):
        return val * 2

    def triple(val):
        return val * 3

    if name == 'double':
        return double
    elif name == 'triple':
        return triple
    else:
        return None
```

In [2]:

```
f = my_func('double')
f(2)
```

In [3]:

```
f = my_func('triple')
f(2)
```

2. Decorators

Time the Execution of a Function

Following is a simple function returns a list.

In [4]:

```
def myjob():
    result = []
    for i in range(1000000):
        result.append(i**10)
    return result
```

We would like to time the execution of this function.

In [5]:

```
import time
start = time.time()
myjob()
end = time.time()
print('Elapsed in seconds:', end - start)
```

```
Elapsed in seconds: 0.8437469005584717
```

We can add the code directly to the function `myjob()` .

- But original function is modified.

In [6]:

```
def myjob():
    import time
    start = time.time()

    result = []
    for i in range(1000000):
        result.append(i**10)

    end = time.time()
    print('Elapsed in seconds:', end - start)

    return result
```

What if we would like to measure the execution time of multiple functions?

What is the best way of implementing such feature to all functions without modifying the function directly?

Understand Decorator

Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

Example:

Following function `my_decorator()` takes in a function `func()` , and return another function `inner()` .

- The `inner()` function executes `func()` .

In [7]:

```
def my_decor(func):
    def inner():
        print('before')
        func()
        print('after')

    return inner
```

Following is a simple function `greet()` which prints `hi` .

In [8]:

```
def greet():
    '''This is to say hi'''
    print('hi')

greet()
```

```
hi
```

What happens when we appply `my_decor` as a decorator to the function `greet()` ?

In [9]:

```
@my_decor
def greet():
    '''This is to say hi'''
    print('hi')

greet()
```

```
before
hi
after
```

Get Back Identity

If you check the `__name__` and `__doc__` value of `greet()` function, you will the values of `inner()` function.

In [10]:

```
print(greet.__name__)
print(greet.__doc__)
```

```
inner
None
```

This is undesirable because it will be confusing to other users of `greet()` function.

How can we get back the correct `__name__` and `__doc__` value of `greet()` after applying the decorator?

Using `functools.wraps(func)`

To fix this, the inner function of decorator should decorated by the `@functools.wraps` decorator, which will preserve information about the original function.

In [11]:

```
import functools

def my_decor(func):
    @functools.wraps(func)
    def inner():
        print('before')
        func()
        print('after')

    return inner
```

Check the identity of the `greet()` function again.

In [12]:

```
@my_decor
def greet():
    '''This is to say hi'''
    print('hi')

greet()
```

```
before
hi
after
```

In [13]:

```
print(greet.__name__)
print(greet.__doc__)
```

```
greet
This is to say hi
```

Full Decorator

Above decorator cannot handle function with input parameters and/or return value.

How to allow decorated function accept parameters, and return values?

- Decorated function may accept any number of positional arguments, and keyword arguments.
- Decorated function can return a value too.

Let's implement a decorator to time execution of a function.

In [14]:

```
import functools

def time_execution(func):

    @functools.wraps(func)
    def inner(*args, **kwargs):
        import time
        start = time.time()

        result = func(*args, **kwargs)

        end = time.time()
        print('Elapsed in seconds:', end - start)

        return result

    return inner
```

Use above decorator `time_execution` to decorate `myjob()` function.

- Add a docstring `'''This is my job'''` for `myjob()` too.

In [15]:

```
@time_execution
def myjob():
    '''This is my job'''
    result = []
    for i in range(1000000):
        result.append(i**10)
    return result

result = myjob()
```

```
Elapsed in seconds: 0.9504611492156982
```

Check the `__name__` and `__doc__` values of `myjob()` .

In [16]:

```
print(myjob.__name__)
print(myjob.__doc__)
```

```
myjob
This is my job
```