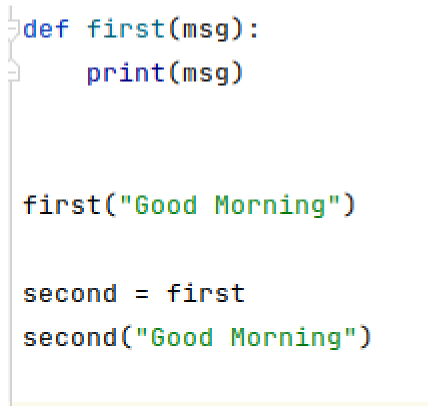
Python Decorators

A decorator takes in a function, adds some functionality and returns it.

Python has an interesting feature called **decorators** to add functionality to an existing code.

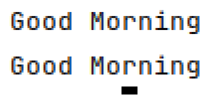
This is also called **metaprogramming** because a part of the program tries to modify another part of the program at compile time.

Prerequisites for learning decorators:



```python
def first(msg):
    print(msg)


first("Good Morning")

second = first
second("Good Morning")
```

Output:

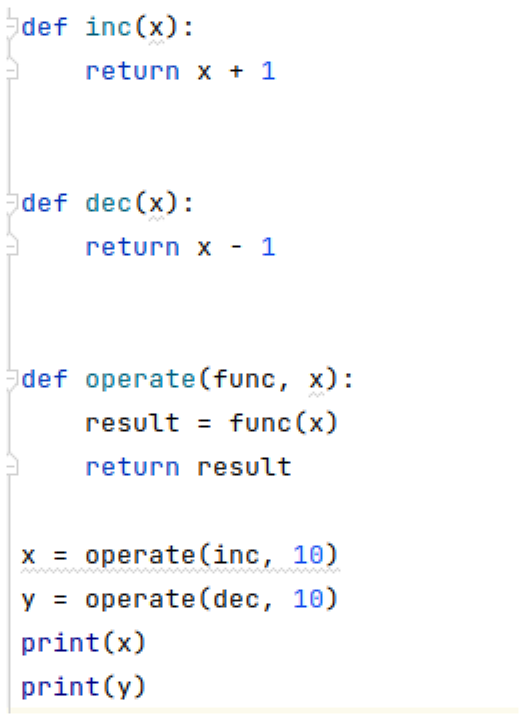


```
Good Morning
Good Morning
```

When you run the code, both functions first and second give the same output. Here, the names first and second refer to the same function object.

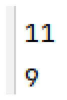
Functions can be passed as arguments to another function.

Such functions that take other functions as arguments are also called **higher order functions**. Here is an example of such a function.



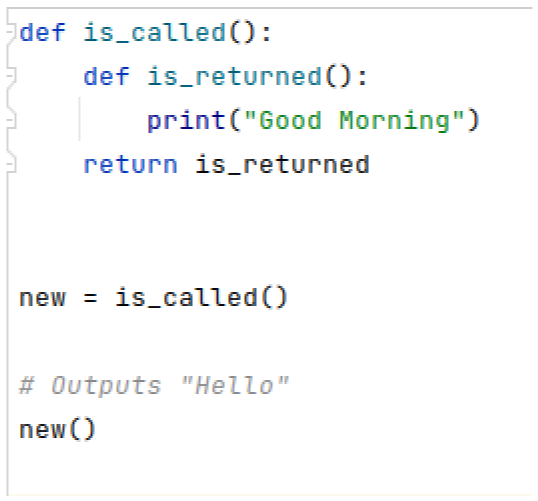
```python
def inc(x):
    return x + 1


def dec(x):
    return x - 1


def operate(func, x):
    result = func(x)
    return result

x = operate(inc, 10)
y = operate(dec, 10)
print(x)
print(y)
```

Output:



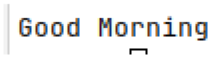
```
11
9
```

Furthermore, a function can return another function.



```python
def is_called():
    def is_returned():
        print("Good Morning")
    return is_returned


new = is_called()

# Outputs "Hello"
new()
```

Output:

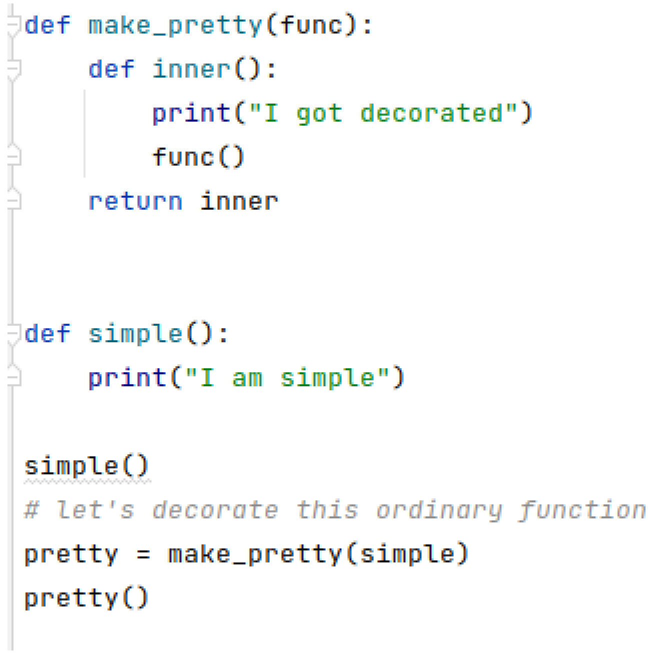


```
Good Morning
```

Here, is_returned() is a nested function which is defined and returned each time we call is_called().

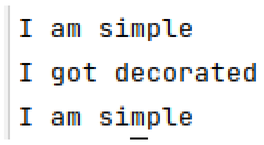
Getting back to Decorators:

Basically, a decorator takes in a function, adds some functionality and returns it.



```
def make_pretty(func):
    def inner():
        print("I got decorated")
        func()
    return inner


def simple():
    print("I am simple")

simple()
# let's decorate this ordinary function
pretty = make_pretty(simple)
pretty()
```

Output:



```
I am simple
I got decorated
I am simple
```

In the example shown above, make_pretty() is a decorator. In the assignment step:

pretty = make_pretty(simple)

The function simple() got decorated and the returned function was given the name pretty.

We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).

Generally, we decorate a function and reassign it as:

```
simple = make_pretty(simple)
```

We can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example:

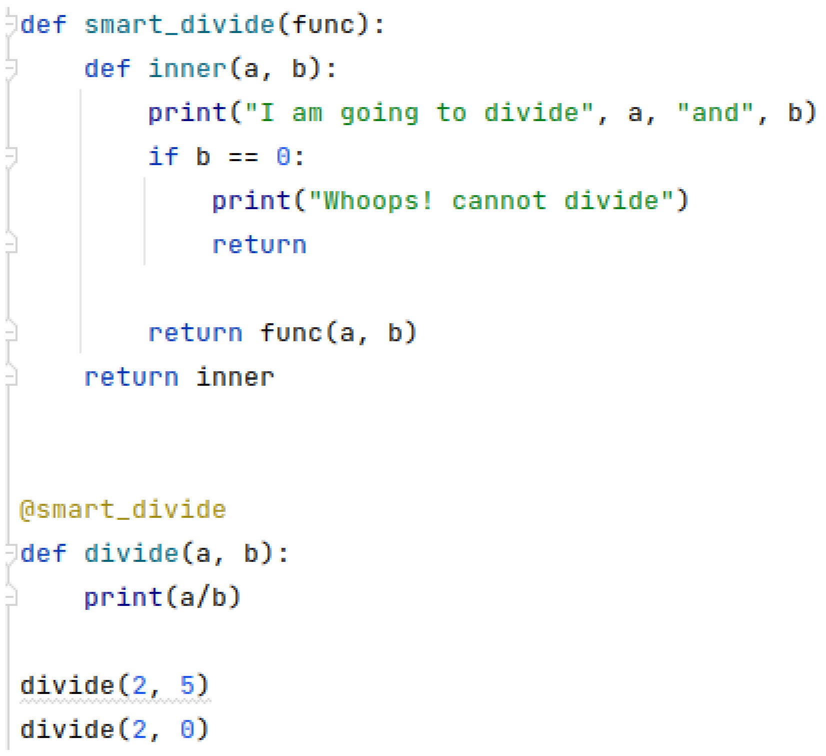
```
@make_pretty
def simple():
    print("I am simple")
```

is equivalent to:

```
def simple():
    print("I am simple")
simple = make_pretty(simple)
```

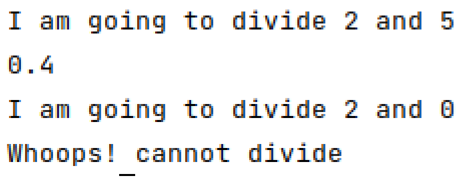
Decorating functions with parameters:

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like:



```python
def smart_divide(func):
    def inner(a, b):
        print("I am going to divide", a, "and", b)
        if b == 0:
            print("Whoops! cannot divide")
            return

        return func(a, b)
    return inner


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

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

Output:

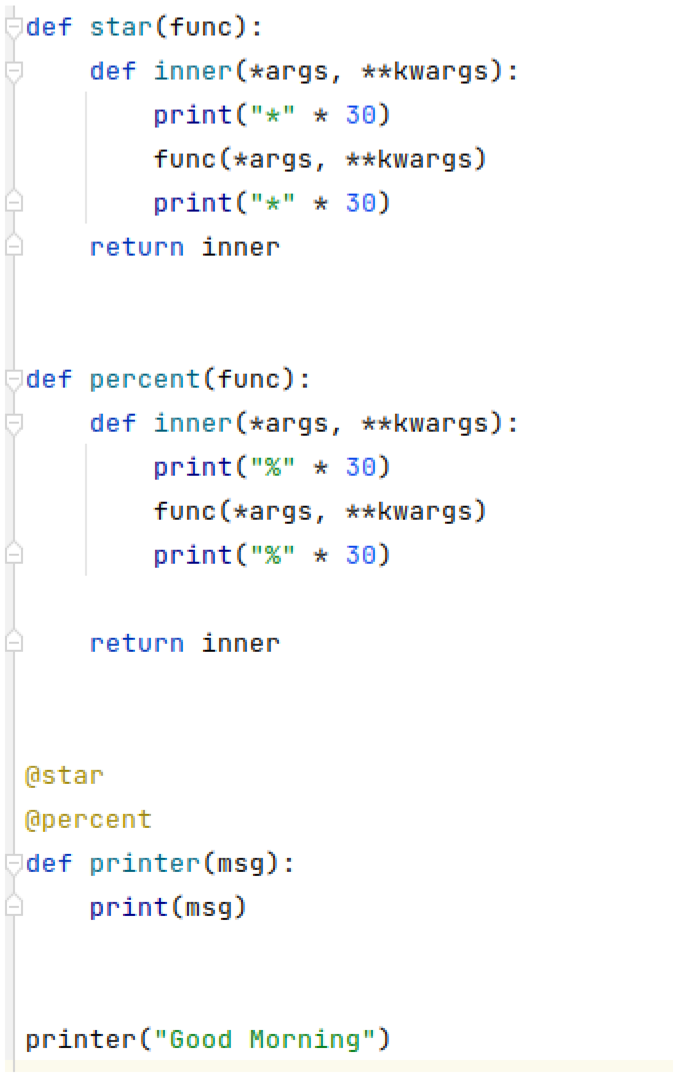


```
I am going to divide 2 and 5
0.4
I am going to divide 2 and 0
Whoops! cannot divide
```

Chaining Decorators in Python:

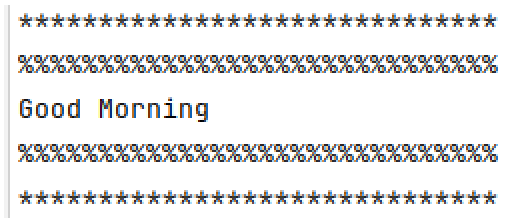
Multiple decorators can be chained in Python.

This is to say, a function can be decorated multiple times with different (or same) decorators. We simply place the decorators above the desired function.



```python
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner


def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)

    return inner


@star
@percent
def printer(msg):
    print(msg)


printer("Good Morning")
```

Output:



```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Good Morning
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of:

```
@star
@percent
def printer(msg):
   print(msg)
```

is equivalent to:

```
def printer(msg):

   print(msg)

printer = star(percent(printer))
```

The order in which we chain decorators matters. If we had reversed the order as:

```
@percent

@star

def printer(msg):

   print(msg)
```

Output:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```