DECORATORS

Copying of function:

Even after deleting the RHS function, the copy of the returned value will still persist in the LHS variable. This is called copying of the function.

In [1]:
```
def welcome():
    print("welcome you")
```

In [2]:
```
welcome()
```

```
welcome you
```

In [3]:
```
wel=welcome()
```

```
welcome you
```

In [4]:
```
wel
```

In [5]:
```
del welcome
```

lets nake a new function

In [6]:
```
def welcome():
    return "welcome you"
```

In [7]:
```
wel=welcome()    # we are doing a function copying
```

In [8]:
```
wel
```

Out[8]:
```
'welcome you'
```

In [9]:
```python
del welcome
```

In [10]:
```python
wel
```

Out[10]:
```
'welcome you'
```

closures:

Defining functions inside a functions. Child fucntion is called closure function. The nested child function, in its definition, can be able to call or use the function and/or variables of the parent function. Closure remember its values and variables in its enclosing scope, even its parent function gets destroyed.

The child function is just its defination. To use the child function, one needs to call it somewhere, using the parent function, or, be it in technical concept, the child function eeds to be called from its parent scope.

In [11]:
```python
def print_msg(msg):
    greet="Hello"

    def printer():
        print(greet, msg)

    printer()    # calling the child function in the scope of the parent function.
```

In [12]:
```python
print_msg("Python theek thaak hi hai!")
```

```
Hello Python theek thaak hi hai!
```

In [13]:
```
def main_welcome(stringz):
    msg="kaise ho aap log"
    def sub_welcome():
        print("Welcome to my world")
        print(msg, stringz)
        print("kahtam")
    return sub_welcome()    # Its a kind of calling a child function from the parent function's scope.
```

In [14]:
```
main_welcome("sharan here")
```

```
Welcome to my world
kaise ho aap log sharan here
kahtam
```

In [15]:
```
def main_welcome(stringz):
    msg="kaise ho aap log"
    def sub_welcome():
        print("Welcome to my world")
        print(msg, stringz)
        print("kahtam")
    return sub_welcome    # Its a kind of calling a child function from the parent function's scope, but the whole f
```

In [16]:
```
main_welcome("sharan here")
```

Out[16]:
```
<function __main__.main_welcome.<locals>.sub_welcome()>
```

In [17]:
```
func_itself = main_welcome("sharan here")    # perfect example of function copying. Here, 'func_itself' becomes func
```

In [18]:
```
func_itself
```

Out[18]:
```
<function __main__.main_welcome.<locals>.sub_welcome()>
```

In [19]:
```
func_itself()
```

```
Welcome to my world
kaise ho aap log sharan here
kahtam
```

closures and initial concept decorators

passing [inbuilt] function as argument to a function receiving function as a parameter.

In [20]:
```
def main_welcome(func):
    msg="kaise ho aap log"
    def sub_welcome():
        print("Welcome to my world")
        print(msg)
        func("ki haal?")
        print("kahtam")
    return sub_welcome()
```

In [21]:
```
main_welcome(print) # passing an inbuilt function
```

```
Welcome to my world
kaise ho aap log
ki haal?
kahtam
```

passing an user defined function

In [22]:
```
def main_welcome(func):
    msg="kaise ho aap log"
    def sub_welcome():
        print("Welcome to my world")
        print(msg)
        print("executing now the", func.__name__, "function.")
        func()
        print("kahtam")
    return sub_welcome
```

In [23]:
```
def pet_name():
    print("nahi bataunga")
```

In [24]:
```
whl_func = main_welcome(pet_name)
```

In [25]:
```
whl_func()
```

```
Welcome to my world
kaise ho aap log
executing now the pet_name function.
nahi bataunga
kahtam
```

PURE DECORATOR:

by definition, python decorators are the functions that takes another functions, add some functionality to it and then returns it.

Decorators makes extensive use of decorators.

Calling a function from the another function. Just put the name of the calling function just before the to be called function name, prepending it with the '@', and run it. This will automatically pass that to be called function to the '@' prepended function, and will run that calling function.

This is helpful when we need to call one function to several function. So we mention calling function names before the to be called function name.

In [26]:
```
@main_welcome
def pet_name():
    print("nahi bataunga")
```

In [27]:
```
# now calling the pet_name will ensure that pet_name will get pass to the decorator, and eventually decorator will g
pet_name()
```

```
Welcome to my world
kaise ho aap log
executing now the pet_name function.
nahi bataunga
```

```
kahtam
```

decorating functions with parameters

Passing the parameters of decorators' parameter function's parameters to closure.

In [28]:
```
def smart_divide(func):
    def denom_check(a, b):    # since the inner function replaces our original function, the parametrs should be pas
        print("Dividing", a, "by", b)
        if b == 0:
            print("Cannot divide by 0")
            return    # returning 'None' from the decorator
        #else part
        return func(a, b)
    return denom_check

@smart_divide
def divide(a, b):
    return a/b
```

In [29]:
```
val1 = divide(15, 3)
print(val1)

val2 = divide(5, 0)
print(val2)
```

```
Dividing 15 by 3
5.0
Dividing 5 by 0
Cannot divide by 0
None
```

Decorating function with same/different decorators

In [30]:
```
def star(func):
    def inner(arg):
        print("*"*30)
        func(arg)
        print("*"*30)
    return inner
```

In [31]:
```
def percent(func):
    def inner(arg):
        print("%" * 30)
        func(arg)
        print("%" * 30)
    return inner
```

the below two cells are similar as writing:

```
def printer(msg):
    print(msg)

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

In [32]:
```
@star
@percent
def printer(msg):
    print(msg)
```

In [33]:
```
printer("Decorators are wonderful")
```

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Decorators are wonderful
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```