Assigning Functions to Variables

Create a function that will add one to a number whenever it is called. Then assign the function to a variable and Use this variable to call the function

In [ ]:
```
def plus_one(number):
    return number + 1



add_one=plus_one

add_one(5)
```

Out[ ]:
```
6
```

Defining Functions Inside other Functions

Define a function inside another function in Python.

In [ ]:
```
def plus_one(number):

    #inside function
    def add_one(number):
        return number + 1

    result = add_one(number)
    return result

plus_one(4)
```

Out[ ]:
```
5
```

Passing Functions as Arguments to other Functions

Functions can also be passed as parameters to other functions.

```
In [ ]: def plus_one(number):
            return number + 1

        def function_call(function):
            return function(5) #plus_one(5)

        #function_call(plus_one)

        plus_one(function_call(plus_one)) #function_call(plus_one)-->6 plus_one(6)--->7
```

```
Out[ ]: 7
```

Functions Returning other Functions

A function can also generate another function.

```
In [ ]: def hello_function():

            #Inside function``
            def say_hi():
                return "Hi"

            return say_hi

        hello = hello_function()
        hello()
```

```
Out[ ]: 'Hi'
```

Nested Functions have access to the Enclosing Function's Variable Scope

Python allows a nested function to access the outer scope of the enclosing function. This is a critical concept in decorators -- this pattern is known as a `Closure`

In [ ]:
```
#Enclosing Function

# def message_sender(message):
#     print(message)

def print_message(message):

    def message_sender():
        #Nested Function
        print(message)

    message_sender(message=message)

print_message(message="Some random message")
```

```
Some random message
```

Decorators

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are typically applied to functions, and they play a crucial role in enhancing or modifying the behavior of functions. Traditionally, decorators are placed before the definition of a function you want to decorate

In [ ]:
```
#outside
def outside_function(function): #functions as a parameter

    def inside_function():
        func=function() #say_hi()
        make_uppercase = func.upper() #HELLO THERE
        return make_uppercase # A string

    return inside_function #function-->#string
```

In [ ]:
```
# A Decorator would be used on a function,
# only when we know that that particular function
# is going to be passed as a parameter to \
# our decorator function.

@outside_function #Decorator syntax
def say_hi():
    return 'hello there'

say_hi()
```

Out[ ]:
```
'HELLO THERE'
```

Using a `Decorator` to have `Encapsulation` Feature

In [ ]:
```
class Employee:
    def __init__(self):
        #Default Constructor
        self._name = '' #private
        self._age= 0

    #get the employee name
    def get_name(self):
        print('Name getter Method')
        return self._name

    #set the employee name
    def set_name(self, value):
        print('Name Setter Method')
        self._name = value.upper()

    #get the employee age
    def get_age(self):
        print('Age getter Method')
        return self._age

    #set the employee age
    def set_age(self, value):
        print('Age Setter Method')
        self._age=value
    name=property(get_name, set_name) #Property() is an inbuilt python object. #Property() is a an outside function that takes
    age=property(get_age,set_age)
```

In [ ]:
```
empObj=Employee()
```

In [ ]:
```
empObj.name='Subham'

empObj._name='XYZ'
```

In [ ]:
```
empObj.name='Subham'
```

```
Name Setter Method
```

In [ ]:
```
print(empObj.name)
```

```
Name getter Method
XYZ
```

In [ ]:
```
empObj.age=25
print(empObj.age)
```

```
Age Setter Method
Age getter Method
25
```

In [ ]:
```
class Employee:
    def __init__(self):
        self._name = ''
        self._age= 0

    @property #Decorator
    def name(self):
        print('Name getter Method')
        return self._name

    @name.setter
    def name(self, value):
        print('Name Setter Method')
        self._name = value.upper()

    @property
    def get_age(self):
        print('Age getter Method')
        return self._age

    @get_age.setter
    def set_age(self, value):
        print('Age Setter Method')
        self._age=value
```

In [ ]:
```
empObj=Employee()
```

In [ ]:
```
empObj.name='Subham'
```

```
Name Setter Method
```

In [ ]:
```
empObj.name
```

```
Name getter Method
```

Out[ ]: `'SUBHAM'`

In [ ]: