Python Basics 2 Advance

Functions :

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. As you already know, Python gives you many built-in functions like print(), etc.

@ What is differece between the Function Using Print and Return ?

Print

In [1]:
```
def addition(num1,num2):
    result = num1+num2
    print(result)
```

In [2]:
```
addition(10,20)
```

```
30
```

In [3]:
```
x = addition(10,20)
```

```
30
```

In [4]:
```
print(x)
```

```
None
```

Above we can Observe when we want's see value "x" it produce "None"

Return

In [5]:
```
def addition(num1,num2):
    result = num1+num2
    return result
```

In [6]:
```
addition(10,20)
```

Out[6]:
```
30
```

In [7]:
```
x = addition(10,20)
```

In [8]:
```
print(x)
```

```
30
```

Here We can observe X store the value

@ Write a Program using function and inside function ?

In [9]:
```
def square(x):
    return x*x
```

In [10]:
```
def SumofSquares(Array, n):
    Sum = 0
    for i in range(n):
         SquaredValue = square(Array[i])
    Sum = Sum+SquaredValue
    return Sum
```

In [11]:
```
Array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
n = len(Array)
```

In [12]:
```
Total = SumofSquares(Array, n)
```

In [13]:
```
print("Sum of the Square of List of Numbers:", Total)
```

```
Sum of the Square of List of Numbers: 100
```

In [14]:
```
def Name(name):
    print("My name is {}".format(name))
```

In [15]:
```
def info(x):
    y = Name(x)
    return y
```

In [16]:
```
info("Teja")
```

```
My name is Teja
```

In [17]:
```
def converter(degree):
    print("The degree's are : {}".format(int(round((degree - 32) * 5 / 9))))
```

In [18]:
```
def infopasser(npass):
    p =  converter(npass)
    return p
```

In [19]:
```
infopasser(25)
```

```
The degree's are : -4
```

Postional Aruguments & Key-Word Aruguments :

A positional argument means its position matters in a function call. A keyword argument is a function argument with a name label. Passing arguments as keyword arguments means order does not matter.

@ What is Postional Arguments and Key-word Arguments ?

In [20]:
```
def details(name,age):
    print("My Name is :{}".format(name))
    print("My Age  is :{}".format(age))
```

In [21]:
```
details("Teja",26)
```

```
My Name is :Teja
My Age  is :26
```

Let's see what is postional Arguments and Keyword arguments with some error's ok

---

In [22]:
```
# details()
```

TypeError: details() missing 2 required positional arguments: 'name' and 'age'

If I want to pass more values then we follow below method

In [23]:
```
def details(*args,**kwargs):
    print("My Name is :{}".format(args))
    print("My age is :{}".format(kwargs))
```

In [24]:
```
list1 =["Teja","Tswarup","Swaroop"]
age_dic ={"age":27}
```

In [25]:
```
details(*list1,**age_dic)
```

```
My Name is :('Teja', 'Tswarup', 'Swaroop')
My age is :{'age': 27}
```

Here Age is Keyword Argument and List1 is postional Argument

Map - Function :

Map in Python is a function that works as an iterator to return a result after applying a function to every item of an iterable (tuple, lists, etc.). It is used when you want to apply a single transformation function to all the iterable elements. The iterable and function are passed as arguments to the map in Python

In [26]:
```
def even_or_odd(num):
    if num % 2 == 0:
        print("The Number is Even : {}".format(num))
    else:
        print("The Number is Odd : {}".format(num))
```

In [27]:
```
even_or_odd(9)
```

```
The Number is Odd : 9
```

`If I want to pass Multiple Vlaues

In [28]:
```
list1 = [1,2,3,4,5,6,7,8,9]
```

Syntax

Map(Function-Name,List-Name)

In [29]:
```
map(even_or_odd,list1)
```

Out[29]:
```
<map at 0x2906ef45eb0>
```

It Produce Complex Result Humans Can't Understand Change into Human reable form By Using list

In [30]:
```
list(map(even_or_odd,list1))
```

```
The Number is Odd : 1
The Number is Even : 2
The Number is Odd : 3
The Number is Even : 4
The Number is Odd : 5
The Number is Even : 6
The Number is Odd : 7
The Number is Even : 8
The Number is Odd : 9
```

Out[30]:
```
[None, None, None, None, None, None, None, None, None]
```

Lambda - Function :

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression

In [31]:
```
t = lambda a,b :a*b # Lambda Function
```

In [32]:
```
print(t(10,2))
```

```
20
```

In [33]:
```
even_or_odd = lambda x : x%2 == 0
```

In [34]:
```
even_or_odd(2)
```

Out[34]:
```
True
```

In [35]:
```
even_or_odd(1)
```

Out[35]:
```
False
```

In [36]:
```
def myfunc(n):
    return lambda a : a * n
```

In [37]:
```
myfunc(2)
```

Out[37]:
```
<function __main__.myfunc.<locals>.<lambda>(a)>
```

In [38]:
```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

```
22
```

In [39]:
```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

In [40]:
```
x=lambda x: x*x if x>0 else "Negative Numbers Not Taken"
```

In [41]:
```
x(5)
```

Out[41]:
```
25
```

In [42]:
```
x(-1)
```

Out[42]:
```
'Negative Numbers Not Taken'
```

Lambda - Function With Map :

In [43]:
```
list1=[1,2,3,4,5,6,7,8,9]
```

In [44]:
```
map(lambda x: x%2 == 0,list1)
```

Out[44]:
```
<map at 0x2906efcb310>
```

In [45]:
```
list(map(lambda x: x%2 == 0,list1))
```

Out[45]:
```
[False, True, False, True, False, True, False, True, False]
```

we get values in Boolean Form if we want numbers use filters

In [46]:
```
list(filter(lambda x: x%2 == 0,list1))
```

Out[46]:
```
[2, 4, 6, 8]
```

Enumerators :

The enumerate function in Python converts a data collection object into an enumerate object. Enumerate returns an object that contains a counter as a key for each value within an object, making items within the collection easier to access.

In [47]:
```
list1 = [i for i in range(1,20)]
```

In [48]:
```
list1
```

Out[48]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [49]:
```
enumerate(list1)
```

Out[49]:
```
<enumerate at 0x2906efc7dc0>
```

In [50]:
```
list(enumerate(list1))
```

Out[50]:
```
[(0, 1),
 (1, 2),
 (2, 3),
 (3, 4),
 (4, 5),
 (5, 6),
 (6, 7),
 (7, 8),
 (8, 9),
 (9, 10),
 (10, 11),
 (11, 12),
 (12, 13),
 (13, 14),
 (14, 15),
 (15, 16),
 (16, 17),
 (17, 18),
 (18, 19)]
```

Zip :

The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

In [51]:
```
l1 =[i for i in range(1,20) if i %2==0]
```

In [52]:
```
l1
```

Out[52]:
```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In [53]:
```
l2 =[i for i in range(1,20) if i %2!=0]
```

In [54]:
```
l2
```

Out[54]:
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [55]:
```
print("The L1 Values are :{}".format(l1))
print("The L2 Values are :{}".format(l2))
```

```
The L1 Values are :[2, 4, 6, 8, 10, 12, 14, 16, 18]
The L2 Values are :[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In [56]:
```
zip(l1,l2)
```

Out[56]:
```
<zip at 0x2906efd5580>
```

In [57]:
```
list(zip(l1,l2))
```

Out[57]:
```
[(2, 1),
 (4, 3),
 (6, 5),
 (8, 7),
 (10, 9),
 (12, 11),
 (14, 13),
 (16, 15),
 (18, 17)]
```

Object Orient Programming

Object-oriented programming (OOP) is a style of programming characterized by the identification of classes of objects closely linked with the methods (functions) with which they are associated.

Concepts :

- classes/objects
- encapsulation/data hiding
- inheritance
- polymorphism
- interfaces/methods

class :

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods.A class is Blueprint of the object

In [58]:
```
class Information:
    def physcialMeasurements(self,height,weight,color):
        print("Height is :",height)
        print("Weight is :",weight)
        print("Color is :",color)
```

In [59]:
```
obj1 = Information()  # Object Creations
```

In [60]:
```
obj1.physcialMeasurements(5.8,50,"Black")
```

```
Height is : 5.8
Weight is : 50
Color is : Black
```

Here obj1 is OBJECT , By using Class Name [Template Or BluePrint] we can create Multiple OBJECTS

In [61]:
```
class Information:
    color ="Black"
    height = 5.7
    weight = 50

    def phyinfo(self,name):
        print("This is Phyinfo Method :{}".format(name))
```

In [62]:
```
obj = Information()
```

In [63]:
```
obj.phyinfo("Teja")
```

```
This is Phyinfo Method :Teja
```

In [64]:
```
obj.color
```

Out[64]: 'Black'

In [65]:
```
Information.color
```

Out[65]: 'Black'

We can access the static variable by using the obj and and class name

@ Create the class Using the constructor ?

In [75]:
```
class Information:
    def __init__(self,name,age,gender):
        self.name = name
        self.age = age
        self.gender = gender
    def details(self):
        print("Name of Empolyee : {}".format(self.name))
        print("Age of Empolyee : {}".format(self.age))
        print("Gender of Empolyee : {}".format(self.gender))

    def onlyage(self):
        print("Age",self.age)
```

In [76]:
```
obj3 = Information("Teja",29,"Male")
```

In [71]:
```
obj3.details()
```

```
Name of Empolyee : Teja
Age of Empolyee : 29
Gender of Empolyee : Male
```

In [77]:
```
obj3.onlyage()
```

```
Age 29
```

Inheritance :

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

Single Inheritances :

In python single inheritance, a derived class is derived only from a single parent class and allows the class to derive behaviour and properties from a single base class.

In [92]:
```
class father:
    gold = "20kg"
    money =1000000
    building = 3

    def properties(self):
        print("I had {} gold".format(self.gold))
        print("I had {} money".format(self.money))
        print("I had {} buildings".format(self.building))

class son_or_daughter(father):
    money = 250000

    def nothing(self):
        print("I am derived class, I mean i am child class")
```

In [93]:
```
obj = son_or_daughter()
```

In [94]:
```
obj.properties()
```

```
I had 20kg gold
I had 250000 money
I had 3 buildings
```

In [95]:
```
obj.nothing()
```

```
I am derived class, I mean i am child class
```

In [102]:
```
class father:
    gold = "20kg"
    money =1000000
    building = 3

    def properties(self):
        print("I had {} gold".format(self.gold))
        print("I had {} money".format(self.money))
        print("I had {} buildings".format(self.building))

class son_or_daughter(father):
    moneyc = 250000

    def nothing(self):
        print("I am derived class, I mean i am child class")
        print("I had money",self.money+self.moneyc)
```

In [103]:
```
obj1 = son_or_daughter()
```

In [104]:
```
obj1.nothing()
```

```
I am derived class, I mean i am child class
I had money 1250000
```

Multi-Level Inheritances :

n python, Multilevel inheritance is one type of inheritance being used to inherit both base class and derived class features to the newly derived class

In [110]:
```
class GrandFather:
    moneyG = 10000000

    def grandpa_saying(self):
        print("I am GrandFather class Method")


class father(GrandFather):
    gold = "20kg"
    money =1000000
    building = 3

    def properties(self):
        print("I am father Method")
        print("I had {} gold".format(self.gold))
        print("I had {} money".format(self.money))
        print("I had {} buildings".format(self.building))

class son_or_daughter(father):
    moneyc = 250000

    def nothing(self):
        print("I am derived class, I mean i am child class")
        print("I had money",self.money+self.moneyc+self.moneyG)
```

In [111]:
```
obj1 = son_or_daughter()
```

In [107]:
```
# Calling GrandFather Method and variable
obj1.grandpa_saying()
```

```
I am GrandFather class Method
```

In [108]:
```
obj1.moneyG
```

Out[108]:
```
10000000
```

In [109]:
```
GrandFather.moneyG
```

Out[109]:
```
10000000
```

In [112]:
```
# Calling father
obj1.properties()
```

```
I am father Method
I had 20kg gold
I had 1000000 money
I had 3 buildings
```

In [113]:
```
obj1.gold
```

Out[113]:
```
'20kg'
```

In [114]:
```
obj1.money
```

Out[114]:
```
1000000
```

In [115]:
```
obj1.building
```

Out[115]:
```
3
```

In [116]:
```
obj1.nothing()
```

```
I am derived class, I mean i am child class
I had money 11250000
```

Hierarichical Inheritances :

When more than one derived classes are created from a single base – it is called hierarchical inheritance. In this program, we have a parent (base) class name Details and two child (derived) classes named Employee and Doctor.

In [134]:
```
class Father:
    def properties(self):
        print("I am millionaire")
class son(Father):
    def sonmethod(self):
        print("I am  son class method")
class daughter(Father):
    def daughtermethod(self):
        print("I am daughter class")
```

son

In [125]:
```
objs = son()
```

In [126]:
```
objs.properties()
```

```
I am millionaire
```

In [127]:
```
objs.sonmethod()
```

```
I am  son class method
```

daughter

In [135]:
```
objd = daughter()
```

In [136]:
```
objd.properties()
```

```
I am millionaire
```

In [137]:
```
objd.daughtermethod()
```

```
I am daughter class
```

Multiple Inheritances :

Inheritance is the mechanism to achieve the re-usability of code as one class(child class) can derive the properties of another class(parent class)

In [155]:
```
class Father:
    salaryf =  25000
    def fatherm(self):
        print("This is father method")

class Mother:
    salarym = 25000
    def motherm(self):
        print("this is Mother method")

class children(Father,Mother):
    def chidmethod(self):
        print("I am child method i had money {}".format(self.salaryf+self.salaryr
```

In [156]:
```
objchild = children()
```

In [157]:
```
objchild.fatherm()
```

```
This is father method
```

In [158]:
```
objchild.motherm()
```

```
this is Mother method
```

In [159]:
```
objchild.chidmethod()
```

```
I am child method i had money 50000
```

Decorators :

Decorators are a very powerful and useful tool in Python since it allows programmers to modify the behaviour of a function or class. Decorators allow us to wrap another function in order to extend the behaviour of the wrapped function, without permanently modifying it. But before diving deep into decorators let us understand some concepts that will come in handy in learning the decorators.

@ write a program to find the execution time of your code with out using decorators ?

In [1]:
```
import time
```

In [3]:
```
def square1(num):
    start = time.time()
    for i in range(1,num):
        result = i*i
        end = time.time()
        print("The Excution time taken is"+str((end-start)*1000))
    return result
```

In [7]:
```
test1 = square1(5)
```

```
The Excution time taken is0.0
The Excution time taken is0.0
The Excution time taken is0.0
The Excution time taken is0.0
The Excution time taken is0.0
```

Note :

- Every time we have to write time means it is waste of time and also it effect program execution time when you running complex programs. so we use decorators concept ok

@ write a program to find the execution time of your code using decorators ?

In [8]:
```
def timer(func):
    def wrapper(*args,**kwargs):
        start = time.time()
        result = func(*args,**kwargs)
        end = time.time()
        print(func.__name__,"The execution time is"+str((end-start)*1000))
        return result
    return wrapper
```

In [19]:
```
@timer
def square2(num):
    result =[]
    for i in num:
        result.append(i*i)
    return result
```

In [20]:
```
arr = range(1,100)
```

In [22]:
```
test2 = square2(arr)
```

```
square2 The execution time is0.0
```

In [ ]:
```

```