Python

Functions & OOP

Agenda

In this module, we will look at the following topics:

- ✓ Python Functions
- ✓ Built-in Functions
- ✓ Lambda Functions
  - map, filter & reduce
- ✓ Function Arguments

- ✓ Object Oriented Programming in Python
- ✓ Classes, Constructors, Destructors
- ✓ Access Specifiers – Public, Private & Protected
- ✓ Instance & Class Methods/Variables
- ✓ Inheritance

Functions

function is a block of organized and reusable code that performs a specific action

Function Definition

```
def fun(a, b, c):
    print (a, b, c)
    return()
```

The `return` statement terminates the execution and returns control to the calling function

Function Call

```
fun(10, 20, 30)
```

```
10 20 30
```

__name__

Every module in python has a special attribute called **__name__** . The value of __name__ attribute is set to **__main__** when module run as main program. Otherwise the value of __name__ is set to contain the name of the module.

module1.py

```
print("==== module1.py ====")

def fun():
    print("running fun() method in module1.py")

if __name__ == "__main__":
    print("Running as main program")
    print("__name__  = ", __name__)
    fun()
```

```
==== module1.py ====
Running as main program
__name__  =  __main__
running fun() method in module1.py
```

module2.py

```
import module1

print("==== module2.py ====")

module1.fun()

print("module1.__name__ : ", module1.__name__)
print("module2.__name__  = ", __name__)
```

```
==== module1.py ====
==== module2.py ====
running fun() method in module1.py
module1.__name__ :  module1
module2.__name__  =  __main__
```

Python Built-in Functions

`sorted()`	`abs()`	`all()`	`any()`
`globals()`	`bin()`	`bool()`	`enumerate()`
`eval()`	`chr()`	`int()`	`sum()`
`open()`	`len()`	`reversed()`	`ascii()`
`bytearray()`	`exec()`	`round()`	isinstance()

Python Built-in Functions

Method	Description
abs()	returns absolute value of a number
all()	returns true when all elements in iterable is true
any()	Checks if any Element of an Iterable is True
ascii()	Returns String Containing Printable Representation
bin()	converts integer to binary string
bool()	Converts a Value to Boolean
bytearray()	returns array of given byte size
chr()	Returns a Character (a string) from an Integer
sorted()	returns sorted list from a given iterable

Python Built-in Functions

Method	Description
`globals()`	returns dictionary of current global symbol table
`enumerate()`	Returns an Enumerate Object
`eval()`	Runs Python Code Within Program
`exec()`	Executes Dynamically Created Program
`int()`	returns integer from a number or string
`sum()`	Add items of an Iterable
`open()`	Returns a File object
`reversed()`	returns reversed iterator of a sequence
`isinstance()`	Checks if a Object is an Instance of Class

Lambda Functions

Lambda functions make your functions more concise and easy to read and write.

Use the keyword `lambda` to create anonymous lambda functions.

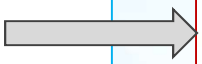
Lambda functions can not contain commands, and they can not contain more than one expression.

Lambda functions can take any number of arguments and returns the value of a single expression

Lambda Examples

```
sq2 = lambda x : x**2 if x > 10 else x**3
print(sq2(5))
print(sq2(15))

marks = [34, 78, 56, 90, 55, 80]
```



```
125
225
```

```
add = lambda x,y : x + y
print (add(10, 20))
```

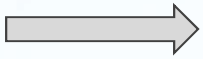
```
30
```

lambda with multiple arguments

```
def multiplier(n):
  return lambda a : a * n

mydoubler = multiplier(2)
mytripler = multiplier(3)

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



```
22
33
```

lambda as return value of a function

map

`map` applies the given function to all the items in an input list.

```
'''
# snippet 3
# map : map takes two argumnets
# agr 1 : a function definition
# arg 2 : an iterable collection
# returns : an iterable collection
'''
marks = [34, 78, 56, 90, 55, 80]
iter1 = map(lambda x: 'A' if x >= 70 else 'B', marks)
print ("TYPE:", type(iter1))
list1 = list(iter1)
print(str(list1))
```

```
TYPE: <class 'map'>
['B', 'A', 'B', 'A', 'B', 'A']
```

filter

`filter` creates a list of elements for which the function returns true

```
marks = [34, 78, 56, 90, 55, 80]

for i in filter(lambda x: x >= 60, marks):
    print(i)
```

```
78
90
80
```

reduce

reduce function reduces a list to a single value by iteratively applying a function on all the items in the list.

```
from functools import reduce

marks = [34, 78, 56, 90, 55, 80]
reduceMarks = reduce(lambda x, y: x + y, marks)
reduceFactorial = reduce(lambda x, y: x*y, range(1,6))
print(reduceMarks, reduceFactorial)
```

```
393 120
```

Variable Scope

Global Variables

Variables that are declared outside a function are global in scope. They can be used anywhere in the program.

Local Variables

Variables that are declared inside a function can be used within the function only.

```
i = 10

def fun():
    j = 20
    print("j = ", j)
    print("i = ", i)

fun()
print("i = ", i)
print("j = ", j)
```

```
j =  20
i =  10
i =  10
Traceback (most recent call last):

  File "<ipython-input-267-8422d559797c>",
line 11, in <module>
    print("j = ", j)

NameError: name 'j' is not defined
```

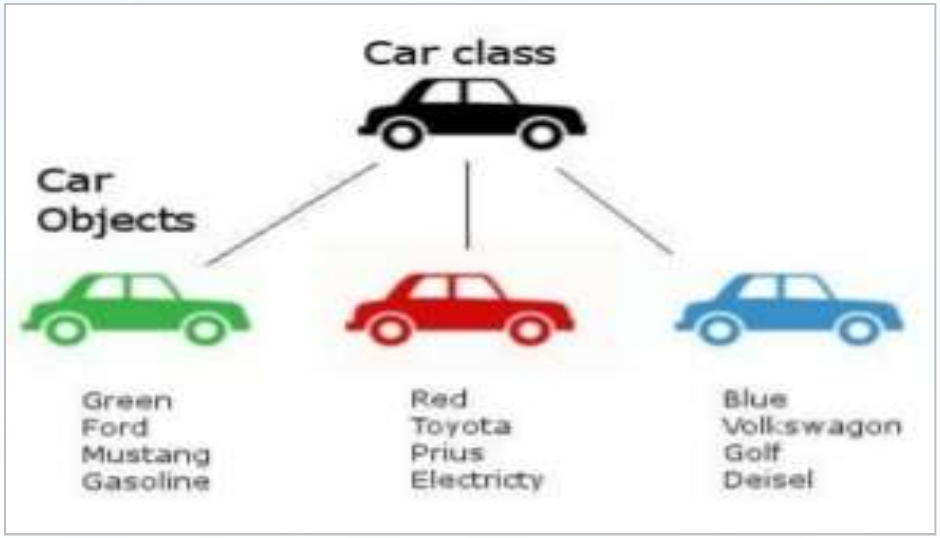
Because ***j*** is ***local*** to the function it can not be accessed outside the function.

Object Oriented Programming

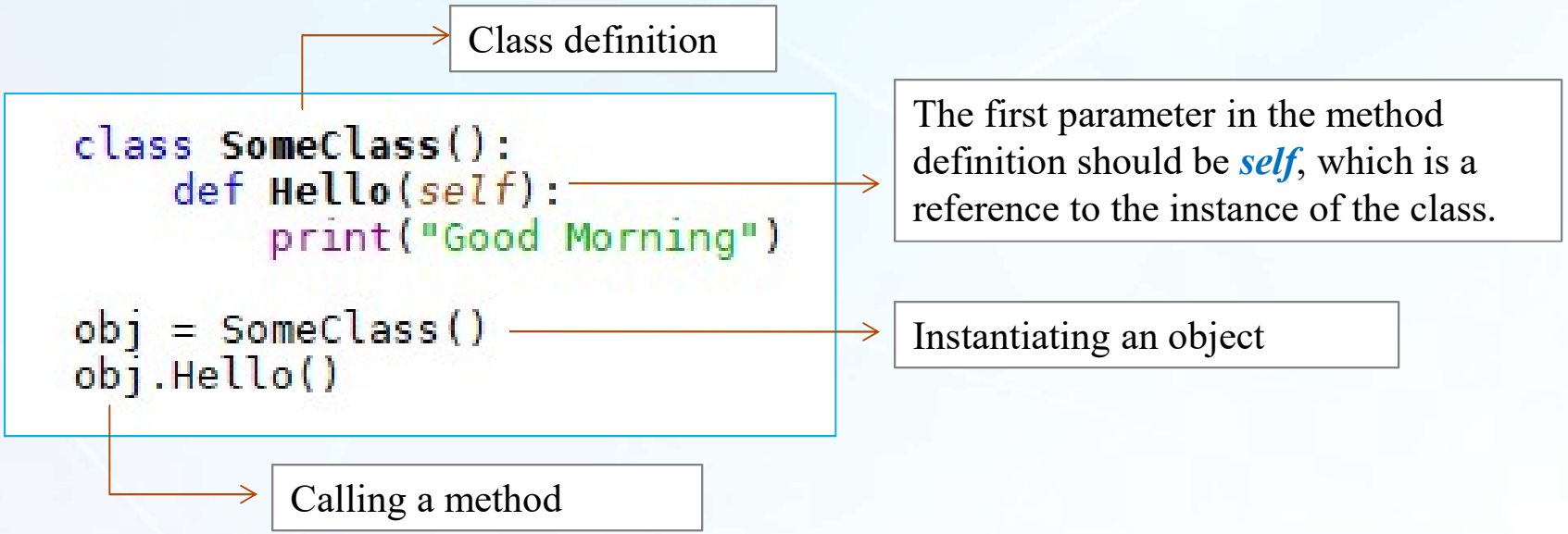
Class & Objects

Class is a blueprint used to create objects having some property or attribute as its class.

An object is an instance of a class which contains properties and methods.



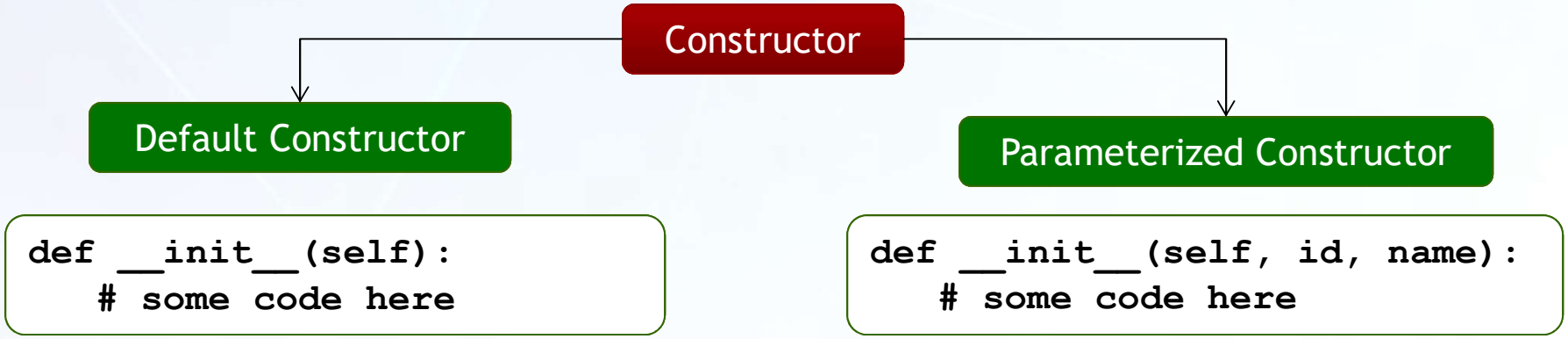
Creating a Class



Every non-static method in Python needs the first parameter to be 'self', which is a reference to the class object.

Constructors

- Constructors are generally used for instantiating an object. The task of constructors is to initialize the data members of the class.
- In Python the `__init__()` method is called the constructor and is always called when an object is created.



Constructors

```
class ConstructDestruct:
    account_type = 0

    def __init__(self, acct_id, name, password):
        # these are container attributes
        self.id = acct_id
        self.name = name
        self.__password = password

        print("Constructing the object")

    def hello(self):
        print("Hello {}".format(self.name))

    def __del__(self):
        print("Good Bye {}".format(self.name))

obj = ConstructDestruct(10, 'spyder', 'python')
print(obj.__dict__)
obj.hello()
del obj
```

Parameterized constructor

```
Constructing the object
{'id': 10, 'name': 'spyder',
'_ConstructDestruct__password': 'python'}
Hello spyder
```

Destructor

- The destructor is defined using `__del__(self)`.
- The destructor is called when the object is removed out of the memory.
- It is either called directly when you remove the object using "***`del obj`***" command or when Garbage Collector removes the object from memory.

Destructor

```
class ConstructDestruct:
    account_type = 0

    def __init__(self, acct_id, name, password):
        # these are container attributes
        self.id = acct_id
        self.name = name
        self.__password = password

        print("Constructing the object")

    def hello(self):
        print("Hello {}".format(self.name))

    def __del__(self):
        print("Good Bye {}".format(self.name))

obj = ConstructDestruct(10, 'spyder', 'python')
print(obj.__dict__)
obj.hello()
del obj
```

Destructor

Destructor is called here

Attributes

Class attributes are attributes that are shared by all class objects.

Built-in attributes

```
__dict__
__doc__
__name__
__module__
__bases__
```

User defined attributes

Defined by user within a class

Built-in Attributes

```
class BltInAttribs():
    emp_count = 0

print(BltInAttribs.__dict__)
print(BltInAttribs.__name__)
print(BltInAttribs.__doc__)
print(BltInAttribs.__module__)
print(BltInAttribs.__bases__)
```

- Dictionary containing class' namespace
- Class name
- Class documentation string
- Module in which class is defined
- List of base classes of this class.

__dict__

__name__

__doc__

__module__

__bases__

```
{'__module__': '__main__', 'emp_count': 0, '__dict__':
'__weakref__': <attribute '__weakref__' of 'BltInAttrib
BltInAttribs
None
__main__
(<class 'object'>,)
```

Access Specifiers

public attributes can be freely used from all class objects

protected attributes can only be accessed from within the class and from the class' subclasses.

They are prefixed with _. (ex: _name)

private attributes can only be accessed from within the class definition.

They are prefixed with __ (ex: __salary)

Access Specifiers

```
class Account:
    account_type = 'Savings'      # public attribute
    _balance = 10000              # protected attribute.
    __loan_amount = 100000        # private attribute.

account = Account()
print(account.account_type)
print(account._balance)
print(account.__loan_amount)
```

```
Savings
10000
Traceback (most recent call last):

  File "<ipython-input-2-82ae8ff24c0e>", line 10, in <module>
    print(account.__loan_amount)

AttributeError: 'Account' object has no attribute '__loan_amount'
```

- **`private`** attributes can not be accessed from an object of the class

Access Specifiers

```
class Account:

    def public_method(self):
        print("public method")

    def _protected_method(self):
        print("protected method")

    def __private_method(self):
        print("private method")

account = Account()
account.public_method()
account._protected_method()
account._Account__private_method()
account.__private_method()
```

```
public method
protected method
private method
Traceback (most recent call last):

  File "<ipython-input-36-81276acc6919>", line
17, in <module>
    account.__private_method()

AttributeError: 'Account' object has no
attribute '__private_method'
```

protected & private

- In Python, **`protected`** and **`private`** access specifiers are indicators for the programmers to refrain from using them from outside a class.
- A **`protected`** attribute/method can still be accessed from a class object.
- Even a **`private`** attribute/method can be accessed from an object using _<ClassName> notation.

```
class Employee:
    def __init__(self, name, sal):
        self._name=name   # protected attribute
        self.__salary=sal # private attribute

e1 = Employee("Raju",10000)
print(e1._name)
print(e1._Employee__salary)
```

```
Raju
10000
```

Instance & Class Variables

```
class Student:
    stream = 'CSE'
    def __init__(self,name,roll):
        self.name = name
        self.roll = roll

# Objects of CSStudent class
a = Student('Rahul', 1)
b = Student('Madhu', 2)

print(a.stream)
print(b.stream)

print(a.name)
print(b.name)
print(a.roll)
print(b.roll)

# Class variables can be accessed
# using class name also
print(Student.stream)
```

Class Variable

Instance Variables

Class Variable is shared by both instances

Each object has their own instance variables

Class Variable can also be accessed like this

Output

```
CSE
CSE
Rahul
Madhu
1
2
CSE
```

Instance & Class Methods

```
class Account:
    account_type = 'Savings'

    @classmethod
    def cls_method(cls, prefix):
        print(prefix, cls.account_type)

    def set_account_type(self, account_type):
        self.account_type = account_type


account = Account()

Account.cls_method(1)

account.set_account_type("Current")

Account.cls_method(2)

account.cls_method(3)

print(account.account_type)
```

Create class method using `@classmethod` annotation and by passing `cls` as first parameter

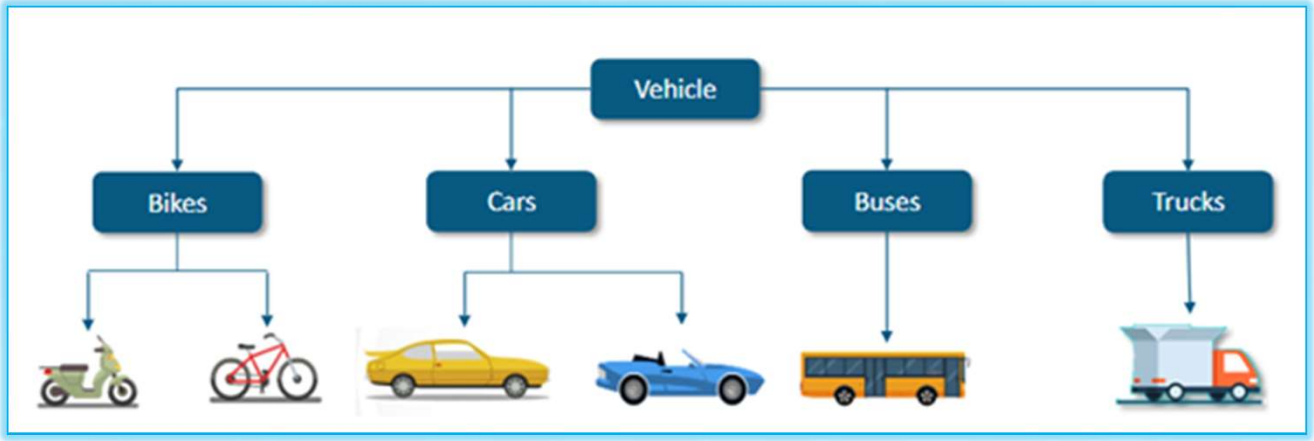
Instance method takes `self` as first parameter.

```
1 Savings
2 Savings
3 Savings
Current
```

Inheritance

Inheritance 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.

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```



Inheritance Example

```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def input_sides(self):
        self.sides = [float(input("Enter side " + str(i+1 )+ " : "))
                      for i in range(self.n)]

    def display_sides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])

class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def find_area(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)

t = Triangle()
t.input_sides()
t.display_sides()
t.find_area()
```

---> Base Class

---> Derived Class

Output

```
Enter side 1 : 3

Enter side 2 : 4

Enter side 3 : 5
Side 1 is 3.0
Side 2 is 4.0
Side 3 is 5.0
The area of the triangle is 6.00
```

Multiple Inheritance

In multiple inheritance, the features of all the base classes are inherited into the derived class.

```
class BaseClass1:
    def __init__(self):
        super(BaseClass1, self).__init__()
        print("BaseClass1")

class BaseClass2:
    def __init__(self):
        super(BaseClass2, self).__init__()
        print("BaseClass2")

class Derived(BaseClass2, BaseClass1):
    def __init__(self):
        super(Derived, self).__init__()
        print("Derived")

derived = Derived()
```

--->

```
BaseClass1
BaseClass2
Derived
```

Multilevel Inheritance

We can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.

```
class animal:
    def do(self):
        print("I can eat")

class person(animal):
    def do(self):
        super().do()
        print("I can think")

class woman(person):
     def do(self):
        super().do()
        print("I can give birth to a baby")

class mother(woman):
    def do(self):
        super().do()
        print("I gave birth to a baby")

class grandmother(mother):
    def do(self):
        super().do()
        print("My daughter gave birth to a baby")

a = grandmother()
a.do()
```

```
I can eat
I can think
I can give birth to a baby
I gave birth to a baby
My daughter gave birth to a baby
```

Method Overriding

Derived classes can override methods defined in the base class.

```
class Rectangle:
    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def get_area(self):
        return self.length * self.breadth

class Square(Rectangle):
    def __init__(self, side):
        self.side = side
        Rectangle.__init__(self, side, side)

    def get_area(self):
        return self.length * self.breadth

s = Square(4)
print("Area of Square: ", s.get_area())

r = Rectangle(3, 4)
print("Area of Rectangle: ", r.get_area())
```

--->

```
Area of Square:  16
Area of Rectangle:  12
```

Getters & Setters

```
class GetSet:
    def __init__(self, name):
        self.name = name

    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

obj = GetSet("Bill Gates")
print(obj.get_name())

obj.set_name("Jeff Bezos")
print(obj.get_name())
```

-->

```
Bill Gates
Jeff Bezos
```

Magic Methods

Magic Methods

Magic methods in Python are the special methods which are not meant to be invoked directly by you, but the invocation happens internally from the class on a certain action. For example, when you add two numbers using the + operator, internally, the __add__() method will be called.

- Built-in classes in Python define many magic methods.
- Use the **`dir()`** function to see the number of magic methods inherited by a class. For example, **`dir(int)`** lists all the attributes and methods defined in the int class.

Magic Methods

```
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
'__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real',
'to_bytes']
```

- We can implement our own magic functions for our classes by overriding the above methods such as __str__, __add___, etc.,

Method Overriding

Derived classes can override methods defined in the base class.

Static Methods

- Static methods does not require an object to be instantiated.
- We call them directly at class level
- Generally used for utility functions
- Use `@staticmethod` decorator to declare a function as static

```
class Sum:
    @staticmethod
    def get_sum(*args):
        s = 0
        for i in args:
            s += i
        return s

def main():
    print("Sum: ", Sum.get_sum(1,2,3))

main()
```


THANK
YOU