Object Oriented Programming

Everything in python is an object

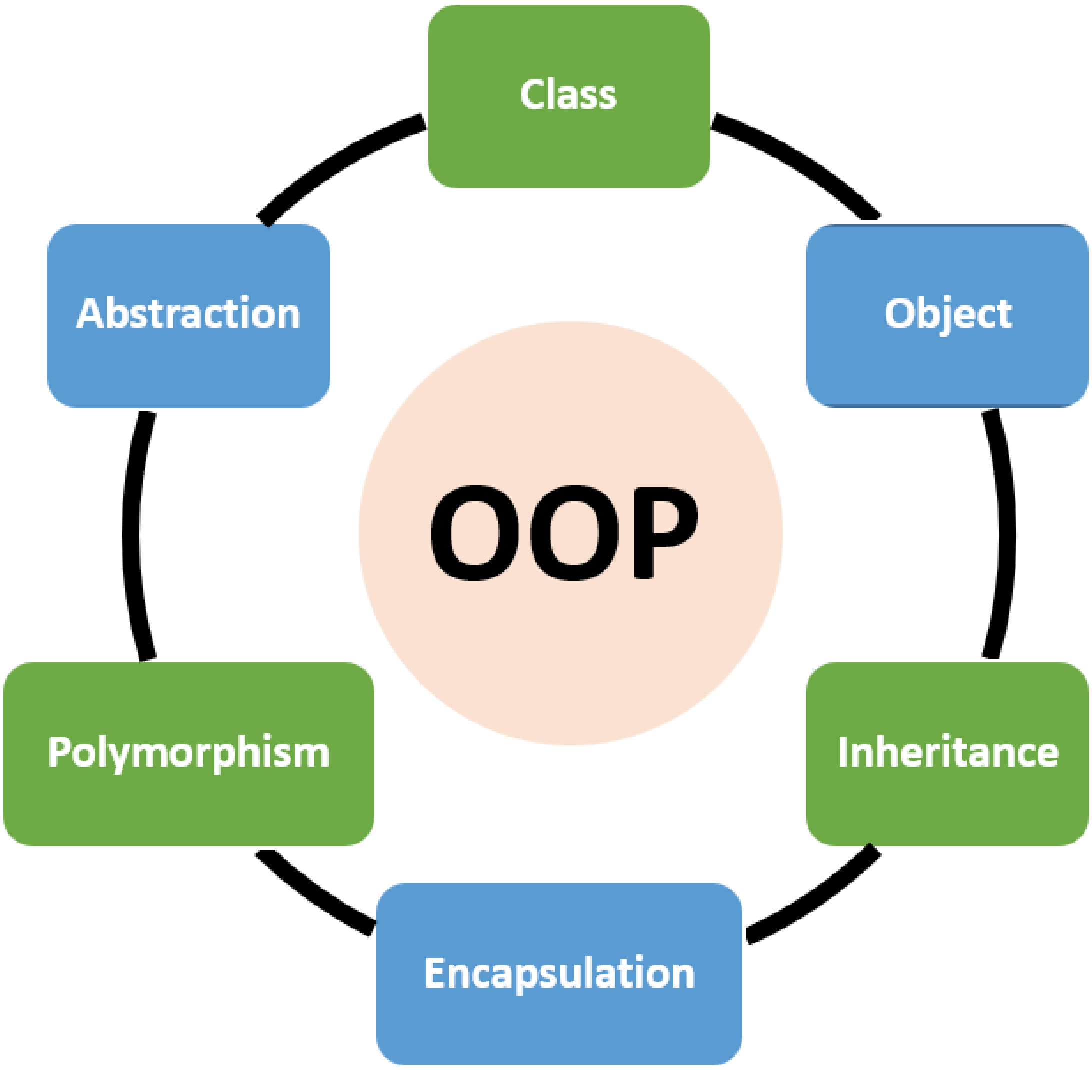
In [3]:
```
l=["a","b","c"]
l.upper()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-cfff6596d849> in <module>
      1 l=["a","b","c"]
----> 2 l.upper()

AttributeError: 'list' object has no attribute 'upper'
```

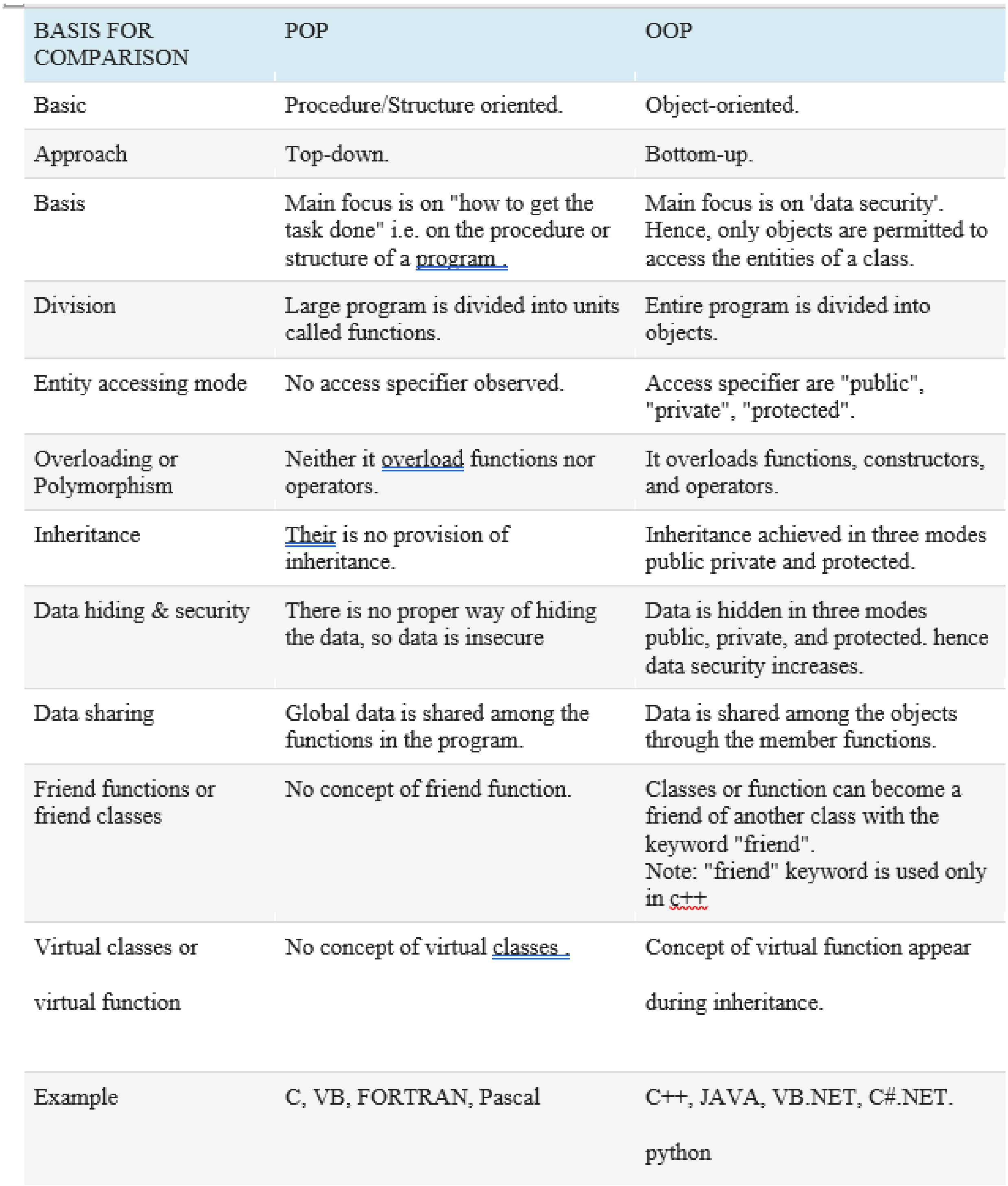
In [4]:
```
s="vishal"
s.append("x")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-4-a1b61ecaba3e> in <module>
      1 s="vishal"
----> 2 s.append("x")

AttributeError: 'str' object has no attribute 'append'
```

oop is gives power of programer to create own datatype



Differance between oop and pop(Procedure-oriented programming)



BASIS FOR COMPARISON	POP	OOP
Basic	Procedure/Structure oriented.	Object-oriented.
Approach	Top-down.	Bottom-up.
Basis	Main focus is on "how to get the task done" i.e. on the procedure or structure of a program .	Main focus is on 'data security'. Hence, only objects are permitted to access the entities of a class.
Division	Large program is divided into units called functions.	Entire program is divided into objects.
Entity accessing mode	No access specifier observed.	Access specifier are "public", "private", "protected".
Overloading or Polymorphism	Neither it overload functions nor operators.	It overloads functions, constructors, and operators.
Inheritance	Their is no provision of inheritance.	Inheritance achieved in three modes public private and protected.
Data hiding & security	There is no proper way of hiding the data, so data is insecure	Data is hidden in three modes public, private, and protected. hence data security increases.
Data sharing	Global data is shared among the functions in the program.	Data is shared among the objects through the member functions.
Friend functions or friend classes	No concept of friend function.	Classes or function can become a friend of another class with the keyword "friend". Note: "friend" keyword is used only in c++
Virtual classes or virtual function	No concept of virtual classes .	Concept of virtual function appear during inheritance.
Example	C, VB, FORTRAN, Pascal	C++, JAVA, VB.NET, C#.NET. python

class : class is blueprint

A Python class is a group of attributes and methods.

In [2]:

```python
L = [1,2,3]
print(type(L))
```

```
<class 'list'>
```

In [3]:

```
l=[1,2,3]
l.isdigit()
#here list is class,l is our object
#all data type are class.and fuctions are method of this class
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\30446165.py in <module>
      1 l=[1,2,3]
----> 2 l.isdigit()
      3 #here list is class,l is our object
      4 #all data type are class.and fuctions are method of this class

AttributeError: 'list' object has no attribute 'isdigit'
```

class

What is Attribute ?

- Attributes are represented by variable that contains data.

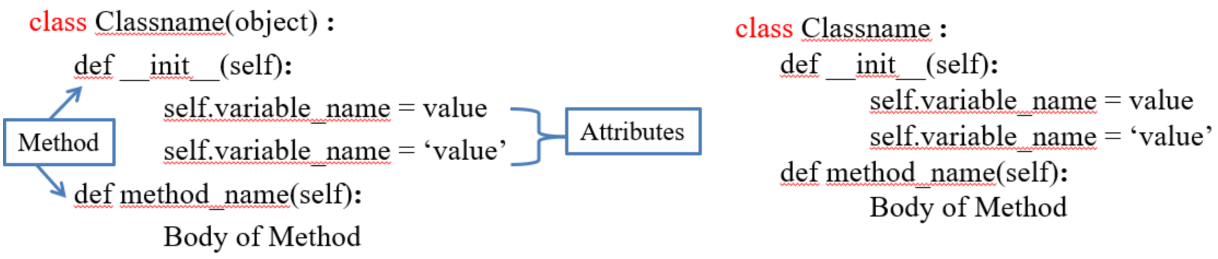
What is Method?

- Method performs an action or task. It is similar to function.

1. data or attribute or property

2. fuction or method or behavior

How to Create Class



```
class Classname(object) :
    def __init__(self):
        self.variable_name = value
        self.variable_name = 'value'
    def method_name(self):
        Body of Method
```

```
class Classname :
    def __init__(self):
        self.variable_name = value
        self.variable_name = 'value'
    def method_name(self):
        Body of Method
```

class - class keyword is used to create a class

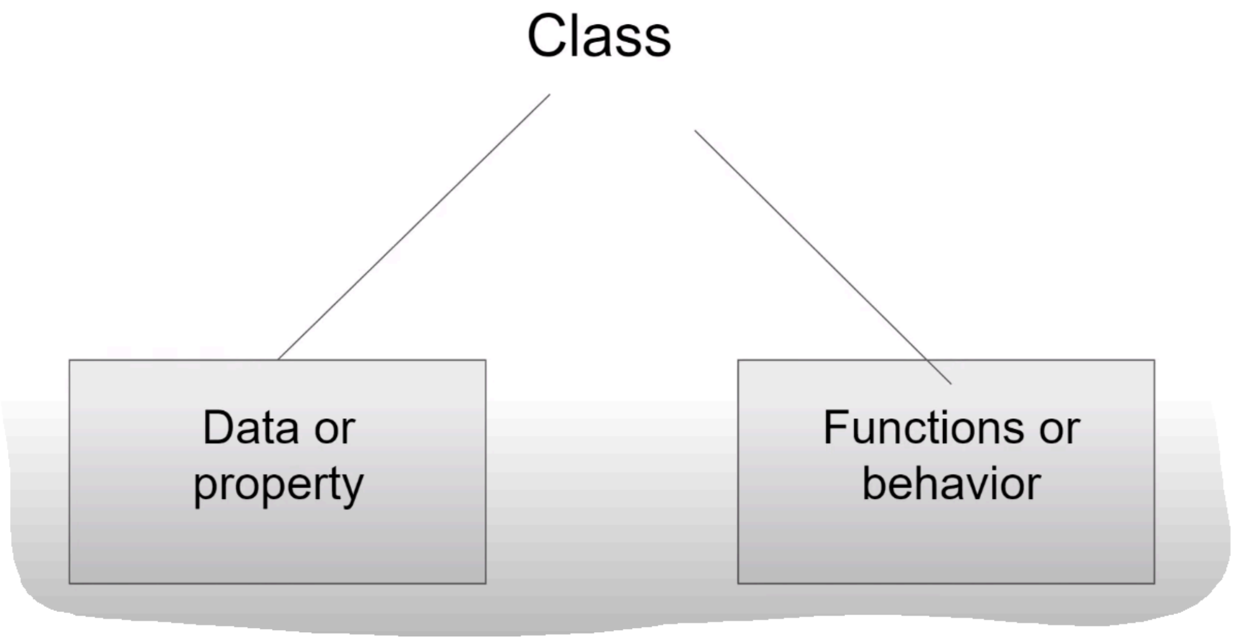
object - object represents the base class name from where all classes in Python are derived. This class is also derived from object class. This is optional.

- **init**() – This method is used to initialize the variables. This is a special method. We do not call this method explicitly.

- self – self is a variable which refers to current class instance/object.

Rule

- The class name can be any valid identifier.
- It can't be Python reserved word.
- A valid class name starts with a letter, followed by any number of letter, numbers or underscores.
- A class name generally starts with Capital Letter.



object: object is an instance of the class

The object is a class type variable or class instance. To use a class, we should create an object to the class.Instance creation represents allotting memory necessary to store the actual data of the variables. Each time you create an object of a class a copy of each variable defined in the class is created. In other words, you can say that each object of a class has its own copy of data members defined in the class.

Example:

1. Bike Honda // Honda=Bike()

2. Sport cricket// cricket=sport()

3. Animal Dog// Dog=Animal()

syntax to create an object

object_name = class_name()

object_name = class_name(arg)

In [4]:

```
class Mobile:
      def __init__(self):
            self.model = "RealMe X"
      def show_model (self):
            print("Model:", self.model)
```

In [5]:

```
realme = Mobile()
```

In [6]:

```
class Mobile:
      def __init__(self, m):
            self.model = m
      def show_model (self, p):
            price = p# Local Variable
            print('Model:', self.model, 'Price:', price)
```

In [7]:

```
realme = Mobile('RealMe X')
```

realme = Mobile()

- A block of memory is allocated on heap. The size of allocated memory is to be decided from the attributes and methods available in the class (Mobile).
- After allocating memory block, the special method **init**() is called internally. This method stores the initial data into the variables.
- The allocated memory location address of the instance is returned into object (realme).
- The memory location is passed to self.

We can access variable and method of a class using class object or instance of class.

In [8]:
```
#object_name.variable_name
realme.model

#object_name.method_name ( )
realme.show_model (500 );

#object_name.method_name (parameter_list)
realme.show_model(1000);
```

```
Model: RealMe X Price: 500
Model: RealMe X Price: 1000
```

In [9]:
```
# object literal so we don't follow above syntax
L = [1,2,3]
```

In [10]:
```
L = list()
L
```

Out[10]:
```
[]
```

In [11]:
```
s = str()
s
```

Out[11]:
```
''
```

class name always in Pascal Case

HelloWorld

In [17]:
```
class Atm:

  # constructor(special function)->superpower ->
  def __init__(self):
    print(id(self))
    self.pin = ''
    self.balance = 0
    print("always come")
```

In [18]:

```
obj=Atm()
```

```
2442232255296
always come
```

In [21]:

```
vis=Atm()
```

```
2442232256256
always come
```

In [19]:

```
print(type(obj))
```

```
<class '__main__.Atm'>
```

In [20]:

```
print(obj.pin)
```

In [16]:

```
print(obj.balance)
```

```
0
```

In [33]:

```
class Atm:

  # constructor
  def __init__(self):
    #1print(id(self))
    self.pin = ''
    self.balance = 0
    self.menu()

  def menu(self):
    user_input = input("""
    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    """)

    if user_input == '1':
      self.create_pin()
    elif user_input == '2':
      self.change_pin()
    elif user_input == '3':
      self.check_balance()
    elif user_input == '4':
      self.withdraw()
    else:
      exit()
  def create_pin(self):
    user_pin = input('enter your pin')
    self.pin = user_pin

    user_balance = int(input('enter balance'))
    self.balance = user_balance

    print('pin created successfully')
    self.menu()

  def change_pin(self):
    old_pin = input('enter old pin')

    if old_pin == self.pin:
      # let him change the pin
      new_pin = input('enter new pin')
      self.pin = new_pin
      print('pin change successful')
      self.menu()
    else:
      print('enter correct pin')
      self.menu()

  def check_balance(self):
    user_pin = input('enter your pin')
    if user_pin == self.pin:
      print('your balance is ',self.balance)
    else:
      print('enter correct pin')
    self.menu()

  def withdraw(self):
    user_pin = input('enter the pin')
    if user_pin == self.pin:
      # allow to withdraw
      amount = int(input('enter the amount'))
      if amount <= self.balance:
        self.balance = self.balance - amount
        print('withdrawl successful.balance is',self.balance)
      else:
        print('increase your balance')
    else:
      print('enter correct pin')
    self.menu()
```

In [34]:

```
sbi=Atm()
```

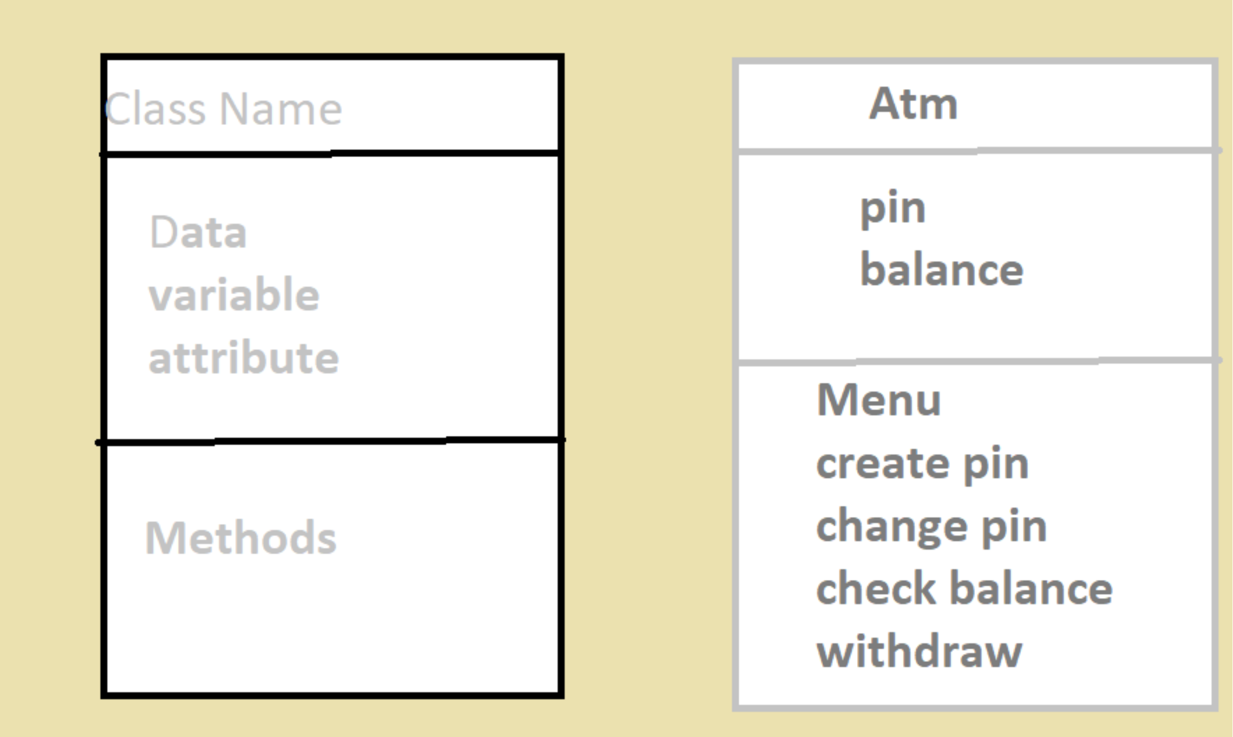
```
    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    1
enter your pin4555
enter balance10000000000
pin created successfully

    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    2
enter old pin4555
enter new pin1212
pin change successful

    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    3
enter your pin1212
your balance is  10000000000

    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    4
enter the pin1212
enter the amount5000
withdrawl successful.balance is 9999995000

    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
```

```
5. Anything else to exit
5
```

Class Diagram

"+" sign mean public

"-" sign mean private

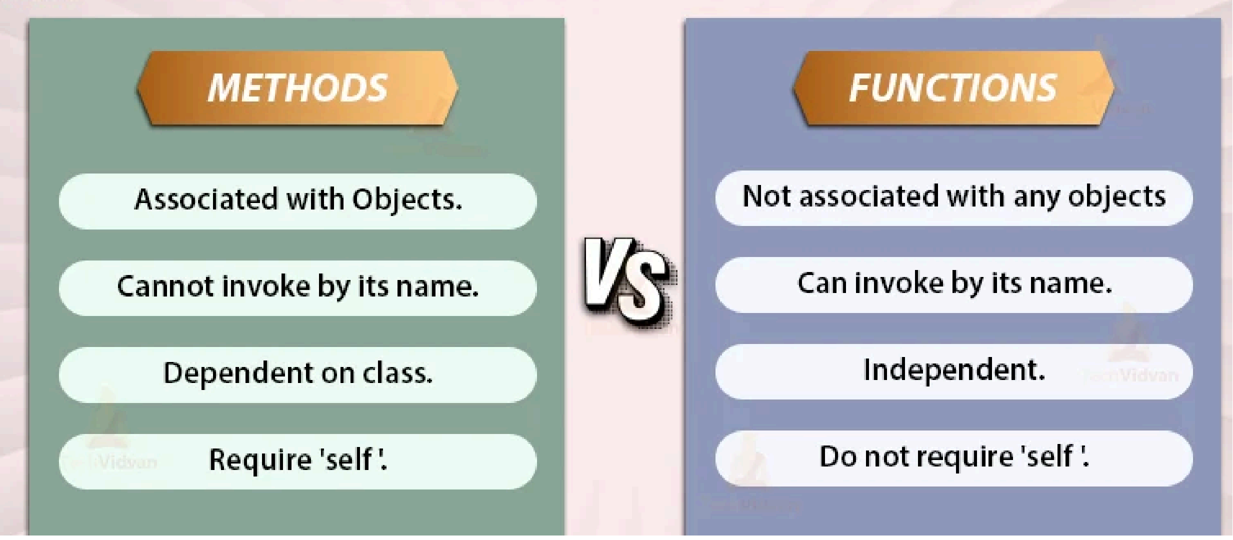


Method VS Function

Function: A function is a block of code to carry out a specific task, will contain its own scope and is called by name. All functions may contain zero(no) arguments or more than one arguments. On exit, a function can or can not return one or more values.

Method: A method in python is somewhat similar to a function, except it is associated with object/classes. Methods in python are very similar to functions except for two major differences.

- The method is implicitly used for an object for which it is called.
- The method is accessible to data that is contained within the class.

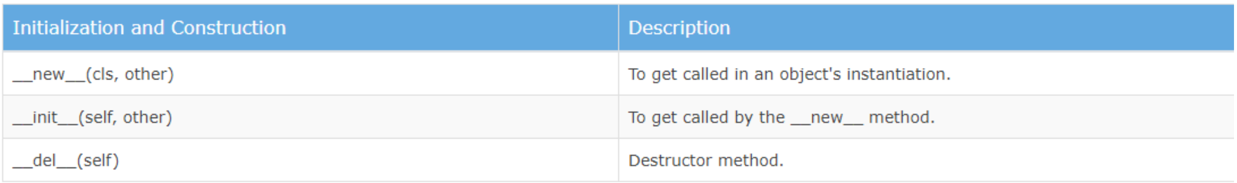


In [1]:

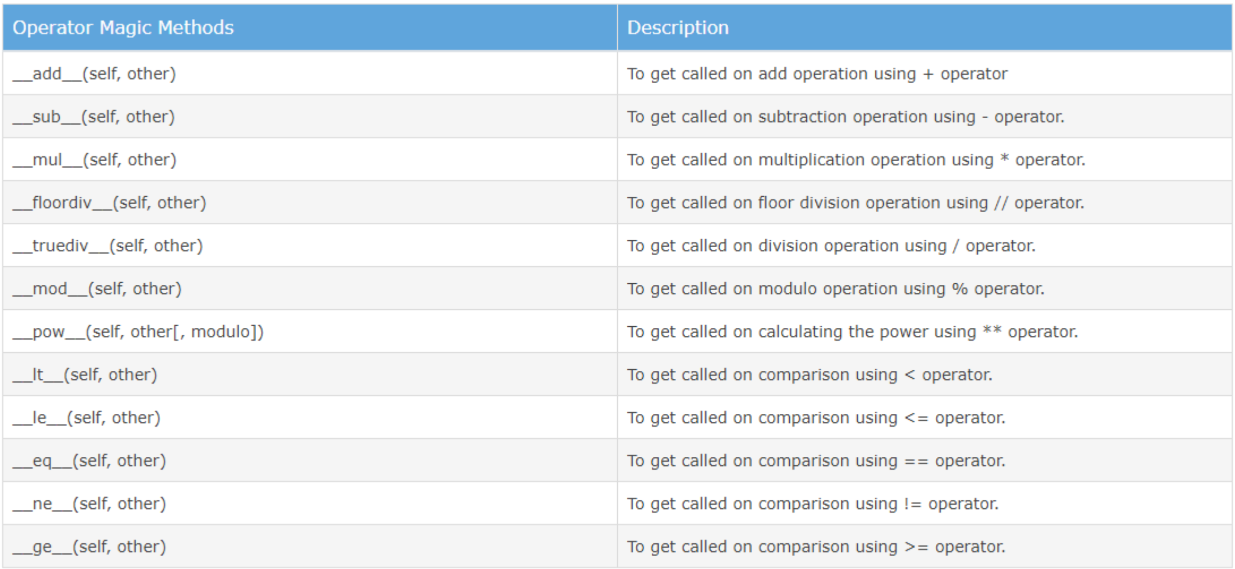
```
#example
s="hello"
print(len(s))#function bcz it is outsode string class
print(s.upper())#method bcz it is inside string class
```

```
5
HELLO
```

Magic Methods (Dunder Methods)



Initialization and Construction	Description
__new__(cls, other)	To get called in an object's instantiation.
__init__(self, other)	To get called by the __new__ method.
__del__(self)	Destructor method.



Operator Magic Methods	Description
__add__(self, other)	To get called on add operation using + operator
__sub__(self, other)	To get called on subtraction operation using - operator.
__mul__(self, other)	To get called on multiplication operation using * operator.
__floordiv__(self, other)	To get called on floor division operation using // operator.
__truediv__(self, other)	To get called on division operation using / operator.
__mod__(self, other)	To get called on modulo operation using % operator.
__pow__(self, other[, modulo])	To get called on calculating the power using ** operator.
__lt__(self, other)	To get called on comparison using < operator.
__le__(self, other)	To get called on comparison using <= operator.
__eq__(self, other)	To get called on comparison using == operator.
__ne__(self, other)	To get called on comparison using != operator.
__ge__(self, other)	To get called on comparison using >= operator.

What is a Constructor?

A constructor is a unique function that gets called automatically when an object is created of a class. The main purpose of a constructor is to initialize or assign values to the data members of that class. It cannot return any value other than none.

- main application: if your app connect to net connection,database connect or any other function which not depended on user

- Syntax of Python Constructor

"def **init**(self):

```
# initializations"
```

- init is one of the reserved functions in Python. In Object Oriented Programming, it is known as a constructor.

Rules of Python Constructor

- It starts with the def keyword, like all other functions in Python.

- It is followed by the word init, which is prefixed and suffixed with double underscores with a pair of brackets, i.e., **init**().

- It takes an argument called self, assigning values to the variables.

- Self is a reference to the current instance of the class. It is created and passed automatically/implicitly to the **init**() when the constructor is called.

```
                                          Types of
Constructors in Python
```

- Parameterized Constructor

- Non-Parameterized Constructor

- Default Constructor

1. Parameterized Constructor in Python

When the constructor accepts arguments along with self, it is known as parameterized constructor.

These arguments can be used inside the class to assign the values to the data members. Let's see an example:

In [2]:

```
class Family:
    # Constructor - parameterized
    members=5
    def __init__(self, count):
        print("This is parametrized constructor")
        self.members = count
    def show(self):
        print("No. of members is", self.members)

object = Family(10)
object.show()
```

```
This is parametrized constructor
No. of members is 10
```

2. Non-Parameterized Constructor in Python

When the constructor doesn't accept any arguments from the object and has only one argument, self, in the constructor, it is known as a non-parameterized constructor.

This can be used to re-assign a value inside the constructor. Let's see an example:

In [3]:

```
class Fruits:
    favourite = "Apple"

    # non-parameterized constructor
    def __init__(self):
        self.favourite = "Orange"

    # a method
    def show(self):
        print(self.favourite)


# creating an object of the class
obj = Fruits()

# calling the instance method using the object obj
obj.show()
```

```
Orange
```

3. Default Constructor in Python

When you do not write the constructor in the class created, Python itself creates a constructor during the compilation of the program.

It generates an empty constructor that has no code in it. Let's see an example:

In [4]:

```
class Assignments:
    check= "not done"
    # a method
    def is_done(self):
        print(self.check)

# creating an object of the class
obj = Assignments()

# calling the instance method using the object obj
obj.is_done()
```

```
not done
```

- The constructor is a method that is called when an object is created of a class.
- The creation of the constructor depends on the programmer, or else Python will automatically generate the default constructor.
- It can be used in three types - Parameterized Constructor, Non-Parameterized Constructor, Default Constructor.

What is Destructor in Python?

When an object is erased or destroyed in object-oriented programming, a destructor is invoked. Before deleting an object, the destructor in python executes clean-up operations such as memory management. Destructor and constructor in python are quite diametric in nature. Constructor is automatically called when an object is created, whereas destructor is called when an object is destroyed.

- Syntax of destructor in Python

"""def **del**(self):""" #body of destructor

Here,

- def is a keyword used to define a method in python.
- """__del __() Method: In Python, the __del __() is referred to as a destructor method. When all references to an object have been erased, i.e., once an object's garbage is collected, this method is invoked."""
- self: The self-argument reflects one of the given class's instances (objects).

Example 1: Using Destructor

Destructor was automatically invoked because we used the del keyword to delete all references to the object.

In [8]:

```python
# Create a Class Computer
class Computer:

    # initialize the class
    def __init__(self):
        print('Class Computer is created.')

    def __del__(self):
        print('Computer is deleted.')

# this is where the object is created and the constructor is called
object = Computer()

# here the destructor function gets called
del object
```

```
Class Computer is created.
Computer is deleted.
```

Example 2: Invoking destructor at the end of the program

The destructor is invoked when the program is finished or when all references to the object are erased, not when the object is removed from scope. This is demonstrated in the following example

In [14]:

```python
# Create Class Computer
class Computer:

    #  Initialize the class
    def __init__(self):
        print('Class Computer is created.')
    def show(self):
        print("hi")

    # Call the destructor
    def __del__(self):
        print('The destructor is called.')

def Create_obj(object):
    print('The object is created.')
    object = Computer()
    object.show()
    print('Function ends here.')
    return object

print('Call the Create_obj() function.')
h = Create_obj("o")
print('The Program ends here.')
o.show()
```

```
Call the Create_obj() function.
The object is created.
Class Computer is created.
hi
Function ends here.
The destructor is called.
The Program ends here.
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\3606683550.py in <module>
     22 h = Create_obj("o")
     23 print('The Program ends here.')
---> 24 o.show()

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

In [11]:

```
# destructor
class Example:

  def __init__(self):
    print('constructor called')

  # destructor
  def __del__(self):
    print('destructor called')

  def show(self):
    print("vishal")

obj = Example()
a = obj
del obj
del a
a.show()
```

```
constructor called
destructor called
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-8d75fa3e4e5d> in <module>
     16 del obj
     17 del a
---> 18 a.show()

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

Self

only object access of all class methods and data.

In [14]:
```
class Vishal():
    def __init__(self):
        self.value=50
        show()
    def show(self):
        print(self.value)
```

In [16]:
```
hi=Vishal()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-686c8ca82a6b> in <module>
----> 1 hi=Vishal()

<ipython-input-14-3362008cce12> in __init__(self)
      2     def __init__(self):
      3         self.value=50
----> 4         show()
      5     def show(self):
      6         print(self.value)

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

In [17]:
```
class Vishal():
    def __init__(self):
        self.value=50
        self.show()
    def show(self):
        print(self.value)
```

In [18]:
```
hi=Vishal()
```

```
50
```

In [19]:
```
class Vishal():
    def __init__(self):
        print(id(self))
        self.value=50
        self.show()
    def show(self):
        print(self.value)
```

In [20]:
```
obj1=Vishal()
```

```
2104985141696
50
```

In [22]:

```
print(id(obj1))
```

```
2104985141696
```

Introduction to Python self

- Suppose you have a class named as a student with three instance variables student_name, age, and marks. Now you want to access these members in a class function. What should you do? You cannot call these members directly. This will cause an error because python will search these variables outside the class instead of the class. Now how can we solve this problem? The answer is using a self word (a reference to class). We pass the self as an argument in the method and access these members, or also we can manipulate the state of these members. Let's dive deep and understand some characteristics or properties of the self.

What is self in python?

- The self is an instance of the class; by using the self, we can access or manipulate the state of the instance members of the class. Also, we can create new instance members with the help of self. The self is mostly used for initializing the instance members of the class.

- Syntax

The self is passed as an argument in the method of the class, and by default, it is compulsory to pass the reference of the class in all the methods of the class.

In [23]:

```
class A():
    a = 3
    # Passing the self in the method
    def hello(self):
        print(self.a)

a = A()
a.hello()
```

```
3
```

In [24]:

```
class A():
    a = 3
    def hello(self):
        # This is the valid way to access the instance variable
        print(self.a)
    def printa():
        # This line will cause an error
        # we cannot access the variable directly
        print(a)
a = A()
a.hello()
a.printa()
```

```
3
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-24-e7955a2bf07f> in <module>
     10 a = A()
     11 a.hello()
---> 12 a.printa()

TypeError: printa() takes 0 positional arguments but 1 was given
```

Why self is defined explicitly in Python

- According to the zen of the python "Explicit is better than implicit". Because explicitly writing the code(clearly defining the state of something even if it is obvious) helps to increase the readability of the code. In a python programming language, we have to pass the reference of the class (using the self word) as an argument in every method of the class

In [25]:

```
class Scaler():
    # Defining the method in the class
    # object.hello(number) is converted to class.hello(object,number) inte
    # That's why the self is passed in this method
    def hello(self, number):
        print("This is the number", number)


a = Scaler()
a.hello(1)
```

```
This is the number 1
```

Is the self a keyword in python?

The self word is not a keyword in python. For the sake easiest naming convention, we often use the self word in the place of the other word, and it is advisable to use the self word instead of the other word because one of the major reasons for this is most of the internal python libraries used word self to represent the reference of the object. So to reduce the conflict between the programmers and the inbuilt libraries, we often use the self word.

In [26]:

```
class human():
    def __init__(self, age, sex="?"):
        self.age = age
        self.sex = sex

    # Passing hello word in the place of the self word
    def speak(hello):
        print(hello.age)


man = human(12, 'M')
man.speak()
```

```
12
```

How can we skip self in python?

- Now, Suppose you want to skip the self word as a parameter in the class methods.

What should you do?

- We can skip self as an argument in the method by adding the @staticmethod decorator on the method, which makes the method static. The static methods don't need the reference of the class. A static method cannot access or modify the class members. We generally use static methods when we write some operations that are not supposed to be changed in the future, like some fixed arithmetic calculations. Code:

In [27]:

```
class Scaler():
    @staticmethod
    # Method is not containing self parameter
    def hello():
        print("This is the method")
```

```
a = Scaler()
a.hello()
```

```
This is the method
```

Self is used for accessing the instance members of the class.

Self is not a keyword in python.

We have to pass self as a parameter in every class method by default.

We can skip the self as a parameter in a method by using the @staticmethod decorator on the method of the class.

Self is always defined explicitly.

How objects access attributes

In [16]:

```
class Person:

  def __init__(self,name_input,country_input):
    self.name = name_input
    self.country = country_input

  def greet(self):
    if self.country == 'india':
      print('Namaste',self.name)
    else:
      print('Hello',self.name)
```

In [17]:

```
# how to access attributes
p = Person('vishal','india')
```

In [18]:

```
p.country
```

Out[18]:

```
'india'
```

In [19]:

```
p.greet()
```

```
Namaste vishal
```

In [25]:

```
print(p.gender)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_20248\269207858.py in <module>
----> 1 print(p.gender)

AttributeError: 'Person' object has no attribute 'gender'
```

Attribute creation from outside of the class

In [27]:

```
p.gender = 'male'
```

In [28]:

```
print(p.gender)
```

```
male
```

In [29]:

```
# Python code for accessing attributes of class
class emp:
    name='Harsh'
    salary='25000'
    def show(self):
        print (self.name)
        print (self.salary)
e1 = emp()
# Use getattr instead of e1.name
print (getattr(e1,'name'))

# returns true if object has attribute
print (hasattr(e1,'name'))

# sets an attribute
setattr(e1,'height',152)

# returns the value of attribute name height
print (getattr(e1,'height'))

# delete the attribute
delattr(emp,'salary')
```

```
Harsh
True
152
```

Reference Variables

- Reference variables hold the objects
- We can create objects without reference variable as well
- An object can have multiple reference variables
- Assigning a new reference variable to an existing object does not create a new object

In [38]:

```
# object without a reference
class Person:

  def __init__(self):
    self.name = 'vishal'
    self.gender = 'male'

Person()
```

Out[38]:

```
<__main__.Person at 0x2a23e996c40>
```

In [39]:

```
# object without a reference
class Person:

  def __init__(self):
    self.name = 'vishal'
    self.gender = 'male'

p = Person()
q = p
```

In [40]:

```
# Multiple ref
print(id(p))
print(id(q))
```

```
2895857778304
2895857778304
```

change attribute value with the help of 2nd object

In [41]:

```
print(p.name)
print(q.name)
q.name = 'kavit'
```

```
print(q.name)
print(p.name)
```

```
vishal
vishal
kavit
kavit
```

Pass by reference

In [42]:

```
class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

# outside the class -> function
def greet(person):
  print('Hi my name is',person.name,'and I am a',person.gender)
  p1 = Person('kavit','male')
  return p1

p = Person('vishal','male')
x = greet(p)
print(x.name)
print(x.gender)
```

```
Hi my name is vishal and I am a male
kavit
male
```

In [43]:

```
class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

# outside the class -> function
def greet(person):
  print(id(person))
  person.name = 'kavit'
  print(person.name)

p = Person('vishal','male')
print(id(p))
greet(p)
print(p.name)
```

```
2895858209456
2895858209456
kavit
kavit
```

object is mutable

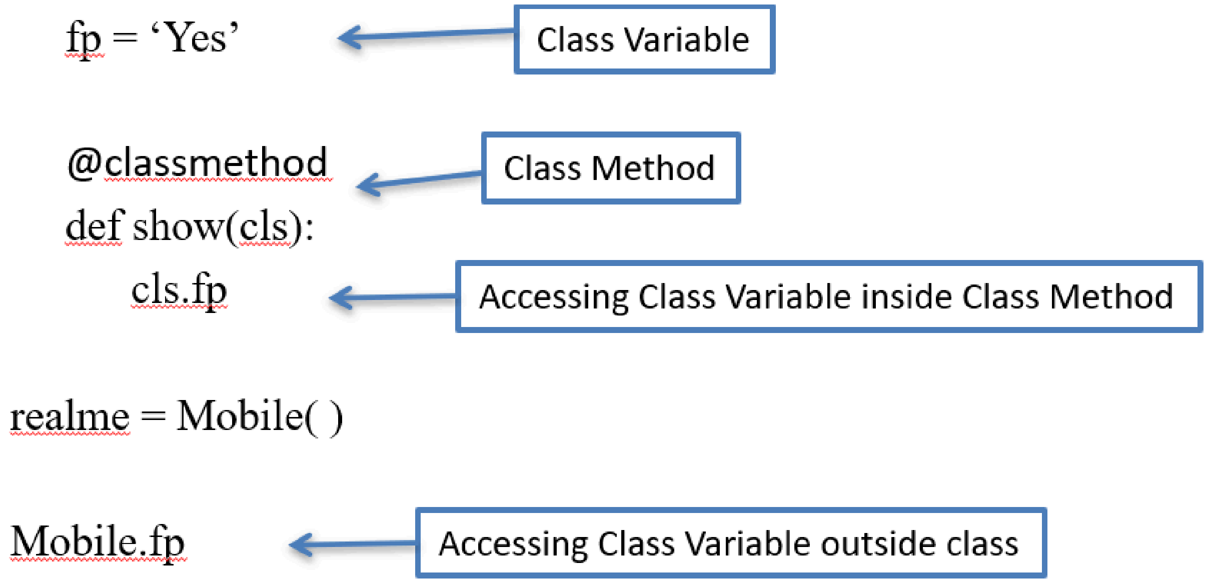
In [ ]:
```
class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

# outside the class -> function
def greet(person):
  person.name = 'ankit'
  return person

p = Person('nitish','male')
print(id(p))
p1 = greet(p)
print(id(p1))
```

Class Variables

Declared inside the class definition (but outside any of the instance methods). They are not tied to any particular object of the class, hence shared across all the objects of the class. Modifying a class variable affects all objects instance at the same time. or

Class variables are the variables whose single copy is available to all the instance of the class. If we modify the copy of class variable in an instance, it will effect all the copies in the other instance.

To access class variable, we need class methods with cls as first parameter then we can access class variable using cls.variable_name



```
fp = 'Yes'

@classmethod
def show(cls):
    cls.fp

realme = Mobile( )

Mobile.fp
```

Class Variable
Class Method
Accessing Class Variable inside Class Method
Accessing Class Variable outside class

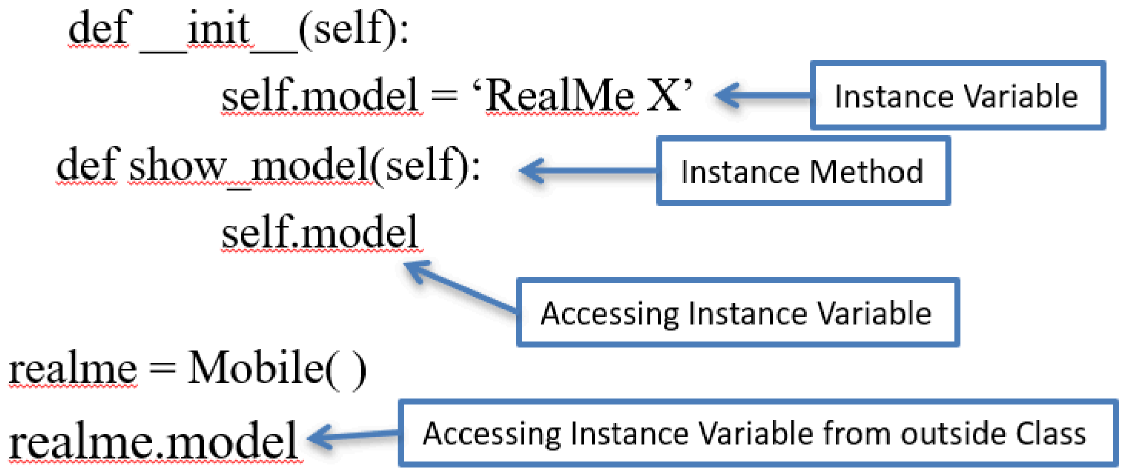
In [33]:

```python
class Fruit:
    name = 'Fruitas'

    @classmethod
    def printName(cls):
        print('The name is:', cls.name)

Fruit.printName()
apple = Fruit()
berry = Fruit()

Fruit.printName()
Fruit.name="banana"
apple.printName()
apple.name="mango"
berry.printName()
```

```
The name is: Fruitas
The name is: Fruitas
The name is: banana
The name is: banana
```

Instance Variable

Declared inside the constructor method of class (the **init** method). They are tied to the particular object instance of the class, hence the contents of an instance variable are completely independent from one object instance to the other.



```
def __init__(self):
        self.model = 'RealMe X'
def show_model(self):
        self.model

realme = Mobile( )
realme.model
```

Instance Variable

Instance Method

Accessing Instance Variable

Accessing Instance Variable from outside Class

In [44]:

```
class Car:
    wheels = 4    # <- Class variable
    def __init__(self, name):
        self.name = name    # <- Instance variable
```

In [47]:

```
jag = Car('jaguar')
fer = Car('ferrari')
print(jag.name, fer.name)
print(jag.wheels, fer.wheels)
print(Car.wheels)
car.name
```

```
jaguar ferrari
4 4
4
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_20248\670756080.py in <module>
      4 print(jag.wheels, fer.wheels)
      5 print(Car.wheels)
----> 6 car.name

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

In [48]:

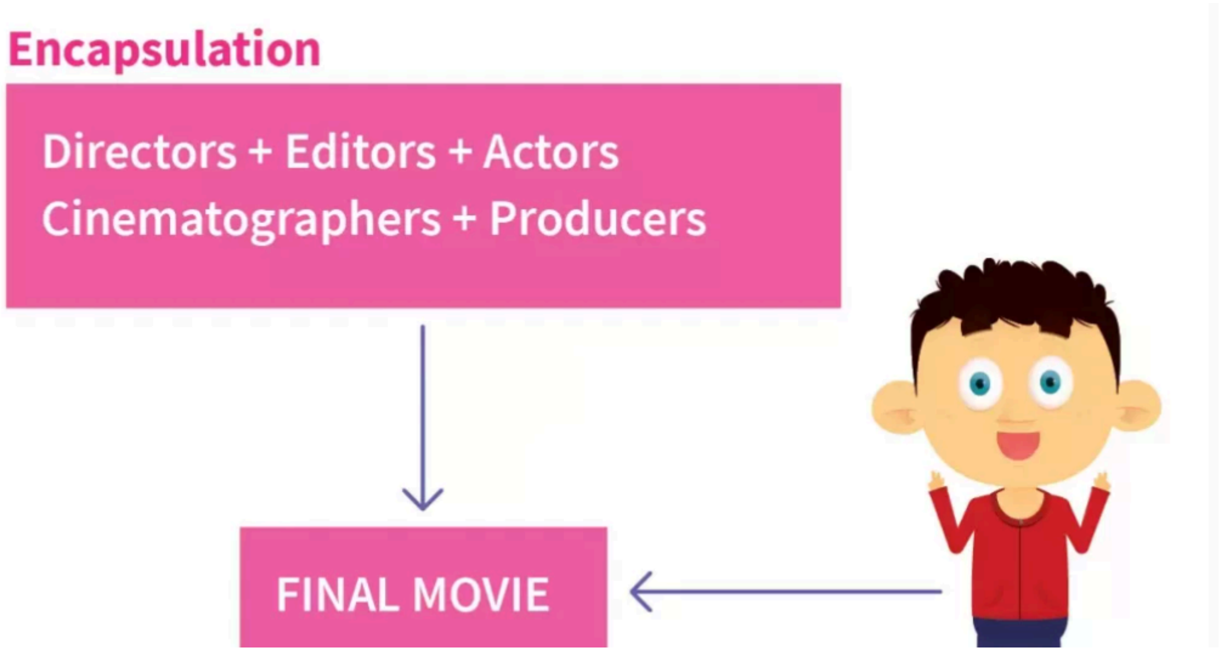
```
# instance var
class Person:

  def __init__(self,name_input,country_input):
    self.name = name_input
    self.country = country_input

p1 = Person('vishal','india')
p2 = Person('kavit','australia')
```

In [49]:
```
print(id(p1.name))
print(p1.name)
print(id(p1))
print(id(p2.name))
print(p2.name)
print(id(p2))
```

```
2895858912560
vishal
2895858212672
2895858075376
kavit
2895858211952
```

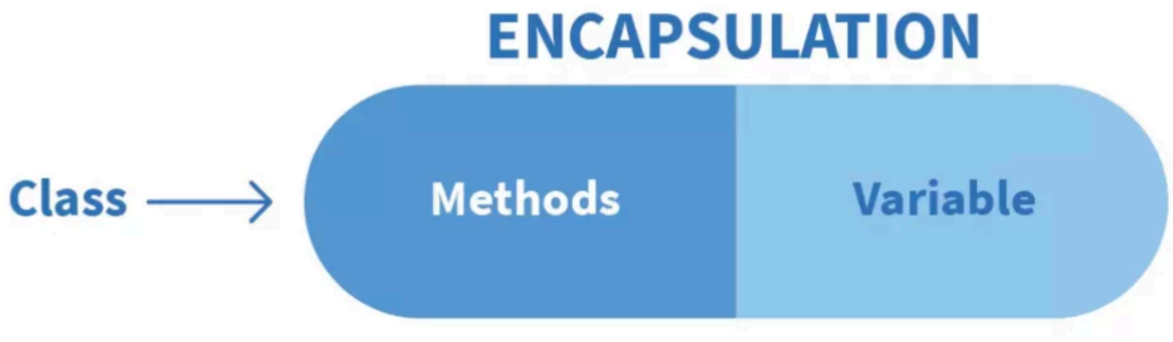
Encapsulation

Encapsulation is one of the critical features of object-oriented programming, which involves the bundling of data members and functions inside a single class. Bundling similar data members and functions inside a class also helps in data hiding. Encapsulation also ensures that objects are self-sufficient functioning pieces and can work independently.



```
                                          What is
   Encapsulation in Python?
```

Encapsulation is one of the cornerstone concepts of OOP. The basic idea of Encapsulation is to wrap up both data and methods into one single unit. The way that data and methods are organized does not matter to the end-user. The user is only concerned about the right way to provide input and expects a correct output on the basis of the inputs provided.



`Why do we need Encapsulation in Python?`

The advantages of Encapsulation in Python can be summed up as follows –

1. Encapsulation provides well-defined, readable code

- The primary advantage of using Encapsulation in Python is that as a user, we do not need to know the architecture of the methods and the data and can just focus on making use of these functional, encapsulated units for our applications. This results in a more organized and clean code. The user experience also improves greatly and makes it easier to understand applications as a whole.

2. Prevents Accidental Modification or Deletion

- Another advantage of encapsulation is that it prevents the accidental modification of the data and methods. Let's consider the example of NumPy again, if I had access to edit the library, then I might make a mistake in the implementation of the mean function and then because of that mistake, thousands of projects using NumPy would become inaccurate.

3. Encapsulation provides security

- Encapsulation in Python is achieved through the access modifiers. These access modifiers ensure that access conditions are not breached and thus provide a great user experience in terms of security.

Access Modifiers in Python encapsulation

- Sometimes there might be a need to restrict or limit access to certain variables or functions while programming. That is where access modifiers come into the picture.

- Now when we are talking about access, 3 kinds of access specifiers can be used while performing Encapsulation in Python. They are as follows :

Access Modifier: Public

- The members declared as Public are accessible from outside the Class through an object of the class.

Access Modifier: Protected

- The members declared as Protected are accessible from outside the class but only in a class derived from it that is in the child or subclass.

Access Modifier: Private

- These members are only accessible from within the class. No outside Access is allowed.

Public Members

Public members (generally methods declared in a class) are accessible from outside the class. The object of the same class is required to invoke a public method. This arrangement of private instance variables and public methods ensures the principle of data encapsulation.

All members in a Python class are public by default. Any member can be accessed from outside the class environment.

In [20]:

```python
class Student:
    schoolName = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
```

```
std = Student("Steve", 25)
print(std.schoolName)
print(std.name)
print(std.age)
```

```
XYZ School
Steve
25
```

In [21]:

```
std.age=20
print(std.age)
```

```
20
```

Protected Members

Protected members of a class are accessible from within the class and are also available to its sub-classes. No other environment is permitted access to it. This enables specific resources of the parent class to be inherited by the child class.

Python's convention to make an instance variable protected is to add a prefix _ (single underscore) to it. This effectively prevents it from being accessed unless it is from within a sub-class.

In [22]:

```
# illustrating protected members & protected access modifier
class details:
    _name="Jason"
    _age=35
    _job="Developer"
class pro_mod(details):
    def __init__(self):
        print(self._name)
        print(self._age)
        print(self._job)

# creating object of the class
obj = pro_mod()
# direct access of protected member
print("Name:",obj.name)
print("Age:",obj.age)
```

```
Jason
35
Developer
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\482597799.py in <module>
     13 obj = pro_mod()
     14 # direct access of protected member
---> 15 print("Name:",obj.name)
     16 print("Age:",obj.age)

AttributeError: 'pro_mod' object has no attribute 'name'
```

In [23]:

```
class Student:
    _schoolName = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age # protected instance attribute
std = Student("Steve", 25)
print(std._schoolName)
print(std._name)
print(std._age)
```

```
XYZ School
Steve
25
```

In [24]:

```
std._age=20
print(std._age)
```

```
20
```

Private Members

Python doesn't have any mechanism that effectively restricts access to any instance variable or method. Python prescribes a convention of prefixing the name of the variable/method with a single or double underscore to emulate the behavior of protected and private access specifiers.

The double underscore __ prefixed to a variable makes it private. It gives a strong suggestion not to touch it from outside the class. Any attempt to do so will result in an AttributeError:

In [26]:

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def __display(self):  # private method
        print('This is private method.')
```

In [27]:

```
std = Student("Bill", 25)
```

In [28]:

```
std.__schoolName
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\2535855424.py in <module>
----> 1 std.__schoolName

AttributeError: 'Student' object has no attribute '__schoolName'
```

In [29]:

```
std.__name
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\1120992331.py in <module>
----> 1 std.__name

AttributeError: 'Student' object has no attribute '__name'
```

In [30]:

```
std.__display()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\4181530199.py in <module>
----> 1 std.__display()

AttributeError: 'Student' object has no attribute '__display'
```

In [31]:

```
class Student:
    __schoolName = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__salary=age # private instance attribute
    def display(self):  # private method
        print('This is private method.')
        print(self.__salary)
```

In [32]:

```
std=Student("vishal",25)
```

In [33]:

```
std.display()
```

```
This is private method.
25
```

In [34]:

```
std.__salary=70
```

In [35]:

```
std.display()
```

```
This is private method.
25
```

In [36]:

```
# illustrating private members & private access modifier
class Rectangle:
  __length = 0 #private variable
  __breadth = 0#private variable
  def __init__(self):
    #constructor
    self.__length = 5
    self.__breadth = 3
    #printing values of the private variable within the class
    print(self.__length)
    print(self.__breadth)

rect = Rectangle() #object created
#printing values of the private variable outside the class
print(rect.length)
print(rect.breadth)
```

```
5
3
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\3023363869.py in <module>
     13 rect = Rectangle() #object created
     14 #printing values of the private variable outside the class
---> 15 print(rect.length)
     16 print(rect.breadth)

AttributeError: 'Rectangle' object has no attribute 'length'
```

Python performs name mangling of private variables. Every member with a double underscore will be changed to _object._class__variable. So, it can still

be accessed from outside the class, but the practice should be refrained.

In [37]:

```
std = Student("Bill", 25)
print(std._Student__name)
std._Student__name = 'Steve'
print(std._Student__name)
std._Student__display()
```

```
Bill
Steve
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18208\3630278853.py in <module>
      3 std._Student__name = 'Steve'
      4 print(std._Student__name)
----> 5 std._Student__display()

AttributeError: 'Student' object has no attribute '_Student__display'
```

In [38]:

```
class Atm:

  # constructor(special function)->superpower ->
  def __init__(self):
    print(id(self))
    self.pin = ''
    self.__balance = 0
    #self.menu()

  def get_balance(self):
    return self.__balance

  def set_balance(self,new_value):
    if type(new_value) == int:
      self.__balance = new_value
    else:
      print('enter correct value')

  def __menu(self):
    user_input = input("""
    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    """)
    if user_input == '1':
      self.create_pin()
    elif user_input == '2':
      self.change_pin()
    elif user_input == '3':
      self.check_balance()
    elif user_input == '4':
      self.withdraw()
    else:
      exit()

  def create_pin(self):
    user_pin = input('enter your pin')
    self.pin = user_pin

    user_balance = int(input('enter balance'))
    self.__balance = user_balance

    print('pin created successfully')

  def change_pin(self):
    old_pin = input('enter old pin')

    if old_pin == self.pin:
      # let him change the pin
      new_pin = input('enter new pin')
      self.pin = new_pin
      print('pin change successful')
    else:
      print('enter correct process')

  def check_balance(self):
    user_pin = input('enter your pin')
    if user_pin == self.pin:
      print('your balance is ',self.__balance)
    else:
      print('correct pin')

  def withdraw(self):
    user_pin = input('enter the pin')
    if user_pin == self.pin:
      # allow to withdraw
      amount = int(input('enter the amount'))
      if amount <= self.__balance:
        self.__balance = self.__balance - amount
        print('withdrawl successful.balance is',self.__balance)
      else:
        print('increase balance')
    else:
      print('correct pin')
```

concept of encapsulation: get and set method

In [39]:

```
obj = Atm()
```

```
1714908088784
```

In [40]:

```
obj.get_balance()
```

Out[40]:

```
0
```

In [41]:

```
obj.set_balance(1000)
```

In [42]:

```
obj.withdraw()
```

```
enter the pin400
correct pin
```

In [43]:

```
class Library:
 def __init__(self, id, name):
     self.bookId = id
     self.bookName = name

 def setBookName(self, newBookName): #setters method to setthe book name
     self.bookName = newBookName

 def getBookName(self): #getters method to get the bookname
     print(f"The name of book is {self.bookName}")

book = Library(101,"The Witchers")
book.getBookName()
book.setBookName("The Witchers Returns")
book.getBookName()
```

```
The name of book is The Witchers
The name of book is The Witchers Returns
```

Collection of objects

In [44]:

```
# list of objects
class Person:
  def __init__(self,name,gender):
    self.name = name
    self.gender = gender

p1 = Person('vishal','male')
p2 = Person('kavit','male')
p3 = Person('deepika','female')

L = [p1,p2,p3]

for i in L:
  print(i.name,i.gender)
```

```
vishal male
kavit male
deepika female
```

In [45]:

```
# dict of objects
# list of objects
class Person:

  def __init__(self,name,gender):
    self.name = name
    self.gender = gender
p1 = Person('vishal','male')
p2 = Person('kavit','male')
p3 = Person('deepika','female')

d = {'p1':p1,'p2':p2,'p3':p3}

for i in d:
  print(d[i].gender)
```

```
male
male
female
```

In [90]:

```
class Atm:

  # constructor(special function)->superpower ->
  def __init__(self):
    print(id(self))
    self.pin = ''
    self.__balance = 0
    self.id=0
    self.id+=1
    #self.menu()

  def get_balance(self):
    return self.__balance
  def set_balance(self,new_value):
    if type(new_value) == int:
      self.__balance = new_value
    else:
      print('enter correct value')

  def __menu(self):
    user_input = input("""
    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    """)

    if user_input == '1':
      self.create_pin()
    elif user_input == '2':
      self.change_pin()
    elif user_input == '3':
      self.check_balance()
    elif user_input == '4':
      self.withdraw()
    else:
      exit()

  def create_pin(self):
    user_pin = input('enter your pin')
    self.pin = user_pin

    user_balance = int(input('enter balance'))
    self.__balance = user_balance

    print('pin created successfully')

  def change_pin(self):
    old_pin = input('enter old pin')

    if old_pin == self.pin:
      # let him change the pin
      new_pin = input('enter new pin')
      self.pin = new_pin
      print('pin change successful')
    else:
      print('enter correct process')

  def check_balance(self):
    user_pin = input('enter your pin')
    if user_pin == self.pin:
      print('your balance is ',self.__balance)
    else:
      print('correct pin')

  def withdraw(self):
    user_pin = input('enter the pin')
    if user_pin == self.pin:
      # allow to withdraw
      amount = int(input('enter the amount'))
      if amount <= self.__balance:
        self.__balance = self.__balance - amount
        print('withdrawl successful.balance is',self.__balance)
      else:
        print('increase balance')
    else:
      print('correct pin')
```

In [92]:

```
sbi=Atm()
```

```
2895858209648
```

In [93]:

```
axis=Atm()
```

```
2895858209264
```

In [94]:

```
sbi.id
```

Out[94]:

```
1
```

In [95]:

```
axis.id
```

Out[95]:

```
1
```

In [103…

```
class Atm:
  __counter = 1

  # constructor(special function)->superpower ->
  def __init__(self):
    print(id(self))
    self.pin = ''
    self.__balance = 0
    self.cid = Atm.__counter
    Atm.__counter = Atm.__counter + 1
    #self.menu()

  # utility functions
  @staticmethod
  def get_counter():
    return Atm.__counter

  def get_balance(self):
    return self.__balance

  def set_balance(self,new_value):
    if type(new_value) == int:
      self.__balance = new_value
    else:
      print('enter correct value')

  def __menu(self):
    user_input = input("""
    Hi how can I help you?
    1. Press 1 to create pin
    2. Press 2 to change pin
    3. Press 3 to check balance
    4. Press 4 to withdraw
    5. Anything else to exit
    """)

    if user_input == '1':
      self.create_pin()
    elif user_input == '2':
      self.change_pin()
    elif user_input == '3':
      self.check_balance()
    elif user_input == '4':
      self.withdraw()
    else:
      exit()

  def create_pin(self):
    user_pin = input('enter your pin')
    self.pin = user_pin

    user_balance = int(input('enter balance'))
    self.__balance = user_balance

    print('pin created successfully')

  def change_pin(self):
    old_pin = input('enter old pin')

    if old_pin == self.pin:
      # let him change the pin
      new_pin = input('enter new pin')
      self.pin = new_pin
      print('pin change successful')
    else:
      print('enter correct process')

  def check_balance(self):
    user_pin = input('enter your pin')
    if user_pin == self.pin:
      print('your balance is ',self.__balance)
    else:
      print('correct pin')

  def withdraw(self):
    user_pin = input('enter the pin')
    if user_pin == self.pin:
      # allow to withdraw
      amount = int(input('enter the amount'))
      if amount <= self.__balance:
        self.__balance = self.__balance - amount
        print('withdrawl successful.balance is',self.__balance)
      else:
        print('increase balance')
    else:
      print('correct pin')
```

In [104…

```
sbi=Atm()
```

```
2895859067392
```

In [105…

```
axis=Atm()
```

```
2895859065376
```

In [106…

```
Atm.get_counter()
```

Out[106…

```
3
```

In [108…

```
axis.get_counter()
```

Out[108…

```
3
```

In [109…

```
vis=Atm()
```

```
2895858198848
```

In [110…

```
vis.get_counter()
```

Out[110…

```
4
```

Static Variables(Vs Instance variables)

Points to remember about static

- Static attributes are created at class level.
- Static attributes are accessed using ClassName.
- Static attributes are object independent. We can access them without creating instance (object) of the class in which they are defined.
- The value stored in static attribute is shared between all instances(objects) of the class in which the static attribute is defined.

In [6]:

```
class Lion:
  __water_source="well in the circus"

  def __init__(self,name, gender):
      self.__name=name
      self.__gender=gender

  def drinks_water(self):
      print(self.__name,
      "drinks water from the",Lion.__water_source)

  @staticmethod
  def get_water_source():

      return Lion.__water_source

simba=Lion("Simba","Male")
simba.drinks_water()
print( "Water source of lions:",simba.get_water_source())
```

```
Simba drinks water from the well in the circus
Water source of lions: well in the circus
```

In [9]:

```
class Lion:
  __water_source="well in the circus"

  def __init__(self,name, gender):
      self.__name=name
      self.__gender=gender
  def drinks_water(self):
      print(self.__name,
      "drinks water from the",Lion.__water_source)

  @staticmethod
  def get_water_source(k):
      Lion.__water_source+=k
      return Lion.__water_source

simba=Lion("Simba","Male")
simba.drinks_water()
print( "Water source of lions:",simba.get_water_source(" kj"))
```

```
Simba drinks water from the well in the circus
Water source of lions: well in the circus kj
```

What are Python Generators?

Python's generator functions are used to create iterators(which can be traversed like list, tuple) and return a traversal object. It helps to transverse all the items one at a time present in the iterator. Generator functions are defined as the normal function, but to identify the difference between the normal function and generator function is that in the normal function, we use the return keyword to return the values, and in the generator function, instead of using the return, we use yield to execute our iterator.

In [112…

```
def gen_fun():
    yield 10
    yield 20
    yield 30
for i in gen_fun():
    print(i)
```

```
10
20
30
```

In [14]:

```
class Fib:
    def __init__(self):
        self.a, self.b = 0, 1

    def __iter__(self):
        return self
    def __next__(self):
        result = self.a
        self.a, self.b = self.b, self.a + self.b
        return result

f = Fib()

for i in range(3):
    print(next(f))
```

```
0
1
1
```

What is a Generator

- Python generators are a simple way of creating iterators.

In [46]:

```
def square(num):
    for i in range(1,num+1):
        yield i**2
```

In [48]:

```
gen = square(10)

print(next(gen))


for i in gen:
    print(i)
```

```
1
4
9
16
25
36
49
64
81
100
```

In [3]:

```
g=(i for i in range(1,10))
print(type(g))
```

```
<class 'generator'>
```

In [4]:

```python
for i in g:
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

In [ ]:

```python

```