PYTHON PROGRAMMING

The Best Guide to Kick Start

Authors	Gopal Gupta, Shelley Gupta, Aatif Jamshed, Anurag Mishra, Harsh Khatter
Authorized by	
Creation/Revision Date	May 2021, December 2021
Version	1.0

Contents

Table of Contents

UNIT 6

Object-Oriented Programming

6.1. Introduction to OOPs:

Object-oriented programming is the way of perceiving the daily life properties and behavior into coding. We can clearly say this oops can model any real-life situation, making their code reusable, portable, and module-based. Oops is not a new programming concept, but it's a culture to rewrite our existing code into classes.

Food for Thought:

Let's think about what we can say about the figure 6.1 given below. Let's not discuss classes and objects; Let's take an example of car as real-life scenario.

One root vehicle(car) is black that represents class while the rest of the vehicles have colors; let's forget for a moment that black itself is a color; here, we see black as the absence of color.

So, the Black car is the car that is not holding any value for any of the car's attribute, while object 1, object 2, object three are having color values as 'Red,' " White,' ' Grey' and these are the object of the class car.

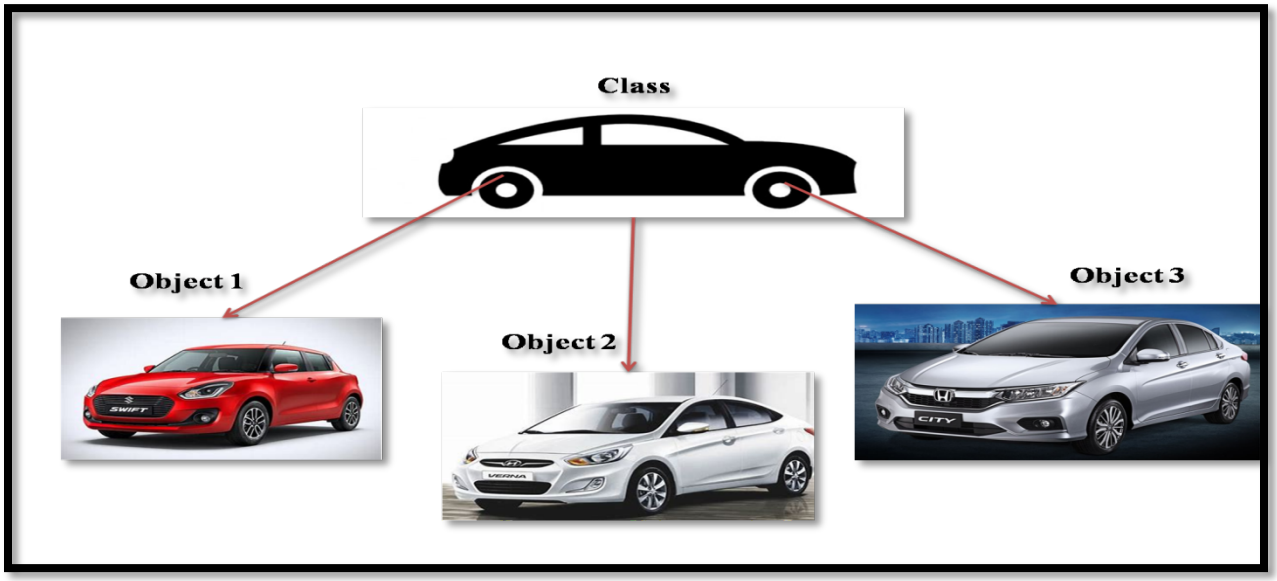


Figure 6.1 Real World Entity

6.1.1. Definition:

A widely used and popularly known approach that involves designing programs based on class and its objects. Objects consists of data and code. Data consists of attributes and code consists of methods.

6.1.2. History of OOPS:

The figure 6.1.2 elaborates the history of OOPs. The beginning of programming languages are marked around 1960. "Object-Oriented Programming" (OOP) was coined by **Alan Kay** circulate 1966. The first programming language that used objects was Simula in 1967. A breakthrough for object-oriented programming came in 1970 with the programming language Smalltalk. CPP adapted Oops in 1985, and then Java came with complete pure Oops support in 1995.

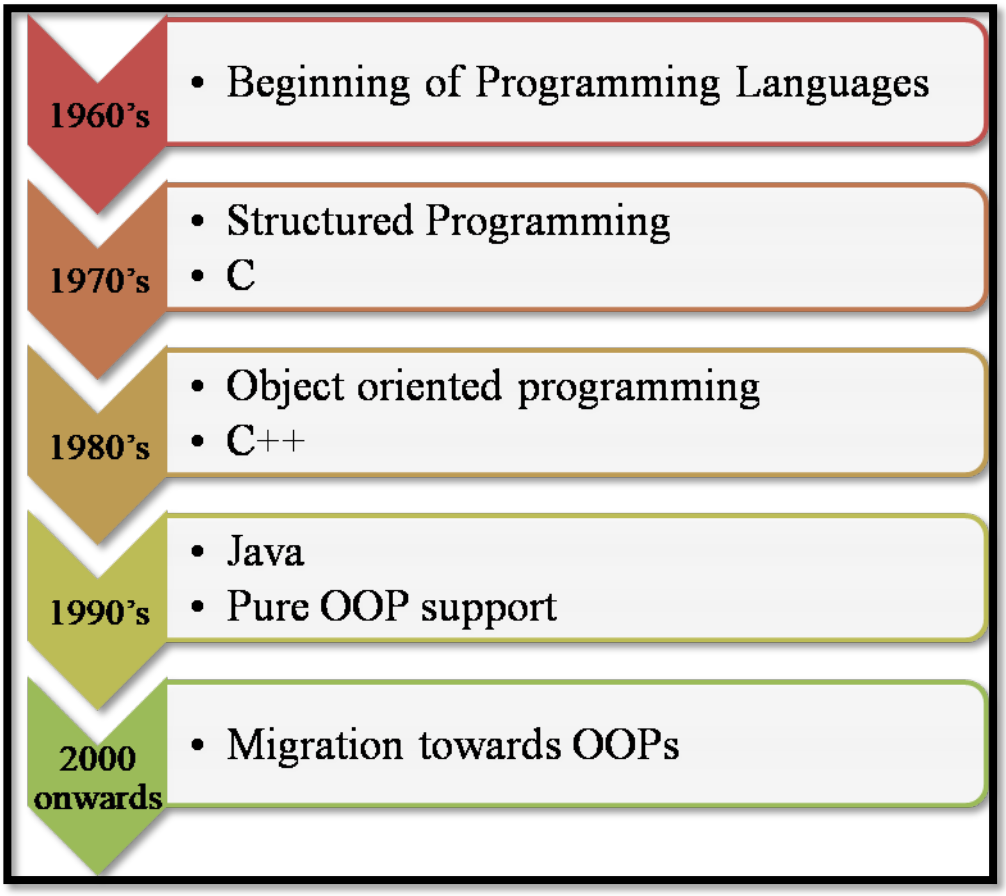


Figure 6.1.2: OOP history

The tricky part of object-oriented programming (which happens to be the challenging programming learning element) consisted of understanding the "why" factor. Why is OOP available; what is it intended to do?

6.2. Requirements of OOPs:

Oops, serve various benefits that empower the developer to make the code more readable, reusable, ease bug detection, and make programs into modules. The figure 6.2 depicts the need of migration from structured programming to Object-Oriented programing paradigm. We have divided the oops' requirement into two parts: ' Need' and 'Why.'

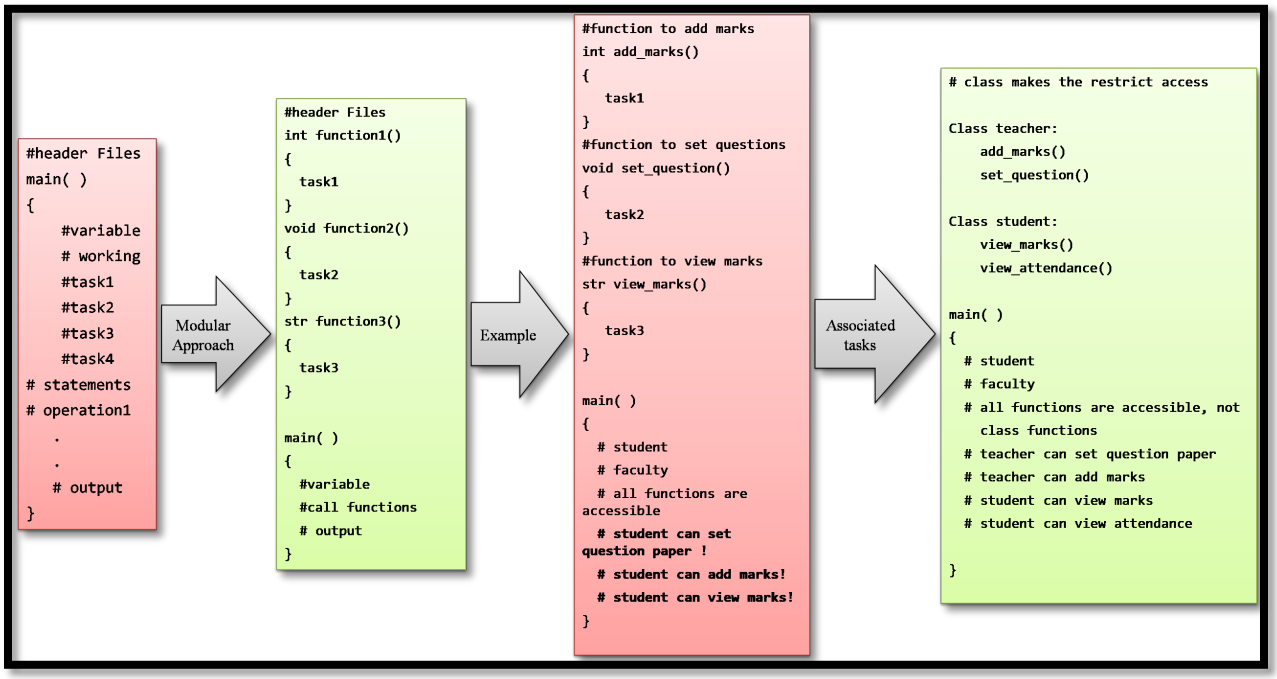


Figure 6.2: From Structured Programming to Object-Oriented Programming

As depicted in figure 6.2 structured programming programs are divided into programs or functions. It is difficult to modify and reuse structured programs. Thus, object oriented programming deals with objects consisting of data and functions both, in which it is easy to reuse the code.

6.2.1.Need of OOPS:

Because only the owner method should access its associated data and no other form should have access to such data, it is not associated with it. The figure 6.2.1. enumerates the need of OOPs. *Oops, does this for us!!*.

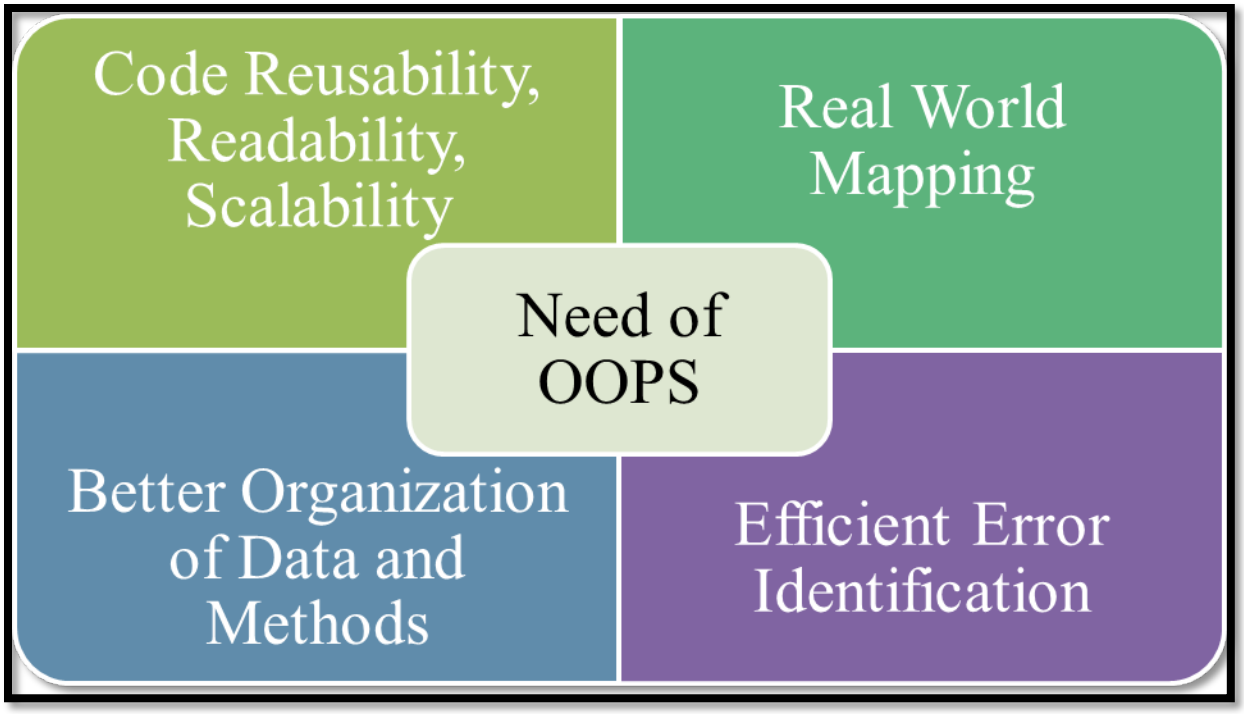


Figure 6.2.1: Need of OOP

6.2.2.Why Oops?

1. Code Reusability, Readability, Scalability
2. Real World Mapping
3. A Better organization of Data and Methods
4. Efficient Error identification
5. Better access control

The requirement of Oops can be further more understood by reading section 6.3, in which we have elaborated the topics abstraction, inheritance, data hiding, encapsulation, etc.

6.3. Outline of OOPs:

In contrast to traditional procedural programming, which seeks to solve problems using rules, Object-oriented programming (OOP) is a programming style that solves problems by thinking around real-world structures such as a car, a book, or an animal. It also assists in the organization of code, making it easier to grasp and solve complicated programs. Objects store data about their current state and behavior. The features of an entity, or the terms you'd use to describe it, are called states, and they typically take the form of is or has descriptors. You have a name, a machine is on or off, and a chair has four legs. Behaviors are the things an entity can do or the actions an intention can take, and they're normally verbs that end in **"ing"**, like You're sitting at a screen reading this paragraph.

The figure 6.3 has discussed about real world mapping by an Oops programmer with two classes:

1. Human Class: The human class consists of:
    - Data having two attributes age and gender.
    - Method are moving (), happy (), standing() and reading().
    - Objects: h1, h2, h3, h4, h6.
2. Car Class: The car class consists of:
    - Data having attribute color.
    - Method are moving (), accelate(), engine_on().
    - Objects: Car1 and Car2

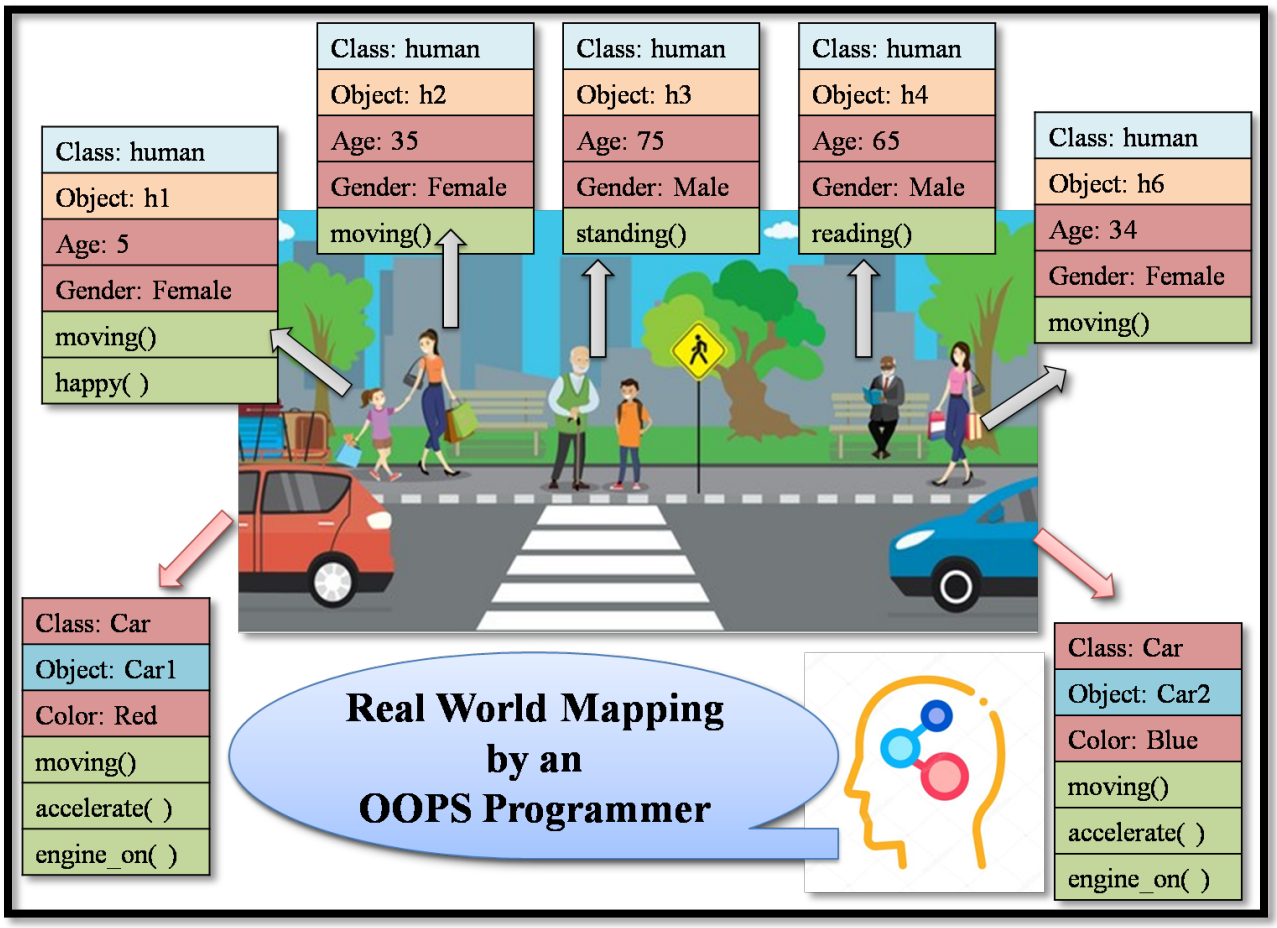


Figure 6.3: Real-world mapping by an OOPS programmer

When we read and understand the four major concepts of OOP, such as abstraction, encapsulation, inheritance, and polymorphism, we can better understand OOPs.

6.3.1. Real-life scenario for better understanding of OOPs:

A house can be thought of as an application and as depicted in figure 6.3.1. Drawing room, Bedroom, Storeroom, Kitchen, bathroom, etc., can be considered classes. According to the

functionality, we put different items (data) in the other part (classes) of the house, such as Bed in the bedroom, sofa, TV in the drawing-room, utensils, and stove in the kitchen, soap, shampoo, buckets in the bathroom, etc. Here, we are trying to bind or place the things as their usages. We call this concept encapsulation.

Most of the items are accessible to all the house members, but some valuables such as money and jewelry are kept hidden in the closet, i.e., neither all the house members nor outsiders have access to them. This is **data hiding.**

We use various appliances such as TV, Fridge, Fan, AC, etc. Neither do we create them, nor we have any interest in their internal workings. We operate them, i.e., their inner working is abstracted from us. This is **an abstraction.**

Figure 6.3.1: Data Organization

let's go through each of these concepts and try to understand them better:

6.3.2. Abstraction

Abstraction is the method of covering the particulars and exposing/showing only the essential aspects of an idea or entity, omitting any context information or description. Consider an ATM as a clear example. You are just concerned with getting your money from the ATM; you are unconcerned with how it is dispatched, stocked, or calculates currency.

Since the most important thing for an ATM is to dispense cash, you can abstract away the specifics and only show the dispense () method when making an ATM Machine class. There are several abstraction degrees; as you get further, the abstraction level increases, and some bits of knowledge decrease. As you lower the level of abstraction, you begin to notice more information.

For example, an airplane, a bus, a train, and a ship are all modes of transportation that carry people at a high altitude, but when you get down to earth, you'll note that an airplane flies, a bus travels on the ground, a train travels on a railway track, and a ship sail on the sea.

Let's take a better example:
How we make coffee?

1. Pour milk into the container
2. Mix coffee and sugar
3. Drink the coffee

Wouldn't it be easier if someone else can make it for us? An espresso machine shown in figure 6.3.2 can do this for us by pushing a small button. This depicts abstraction as we are getting our coffee by pressing the button, without caring about the inner operating details of the machine. The same happens in programming. If there are steps, we need to repeat multiple times, then we create a function for that.

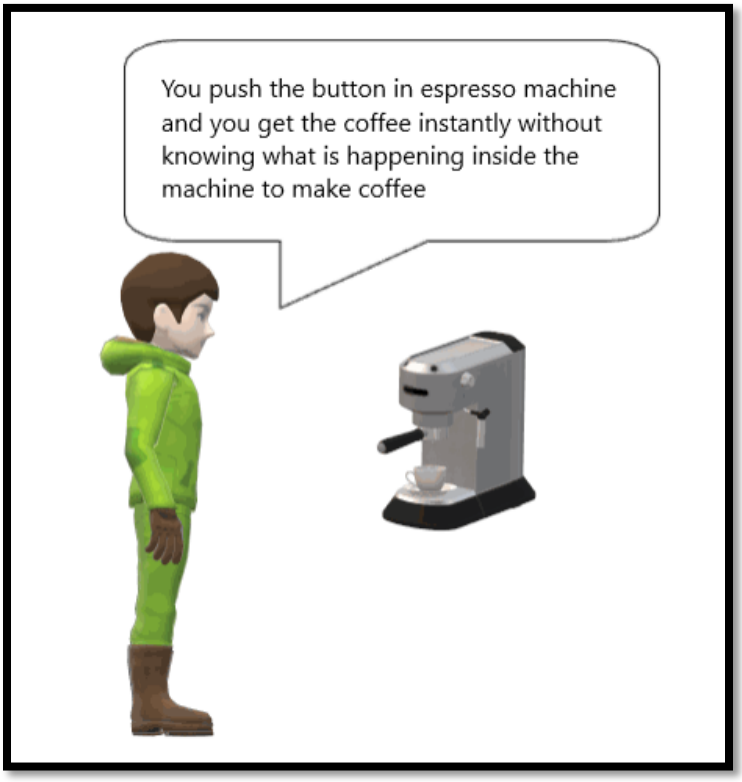


Figure 6.3.2.: Use of the vending machine as an abstraction

6.3.3. Encapsulation

Encapsulation is similar to abstraction in that it masks implementation data as well, but the goal is distinct. It keeps implementation specifics hidden from the rest of the world so that you can modify them later. As depicted in figure 6.3.3. encapsulation is the process of shielding a class's or object's internal workings from outside powers, making them available only by techniques such as "getters" and "setters." Encapsulation helps avoid unintended modifications to data and

functions within a given class/object from external sources, resulting in serious harm in certain software parts.

By limiting data access to methods, classes and objects can properly handle change and determine which external sources can use or modify the data stored within, depending on the available methods. This eliminates the risk of breaking other code in our software that operates our specific class or entity, even though we make improvements elsewhere.

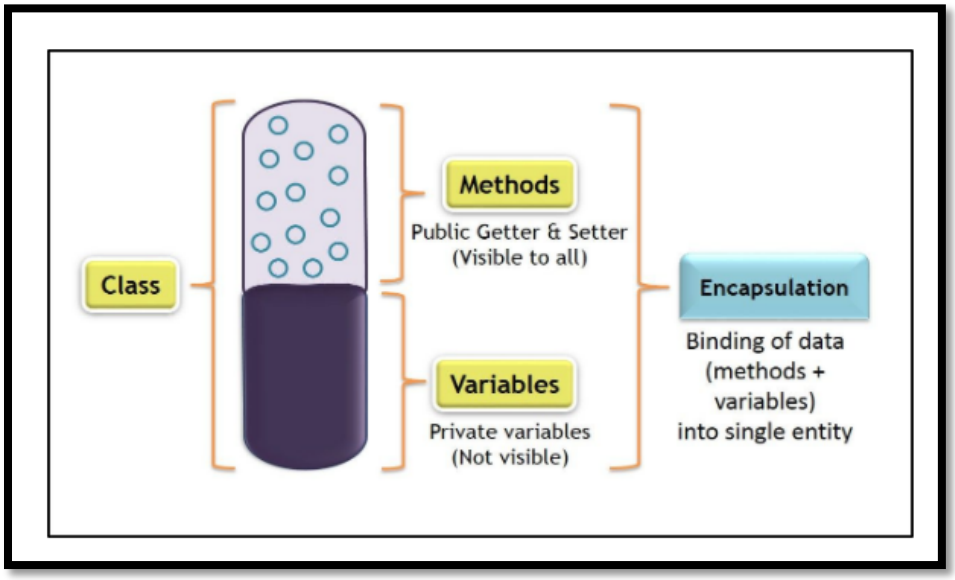


Figure 6.3.3.: The binding of methods and variables in a class

6.3.4. Polymorphism

Polymorphism is a Greek word that refers to "one name, multiple ways," which is just what the meaning entails. Inheritance and polymorphism are related in the same way as abstraction and encapsulation are (and encapsulation).

Polymorphism in OOP allows for various acts to be done based on which entity is executing them. It defines a trend in which classes use the same interface but have different features.

Build a new subclass named Dog using the Animal superclass and the Bird subclass as an example. Both dogs and birds make noises, but dogs bark and birds' chirp. Each subclass will have a Sound () form, which will yield different results despite being named the same thing. As discussed in figure 6.3.4. the Sound () method for a dog will bark, the Sound() method for a cat will meow, and the Sound() method for a lion will roar.

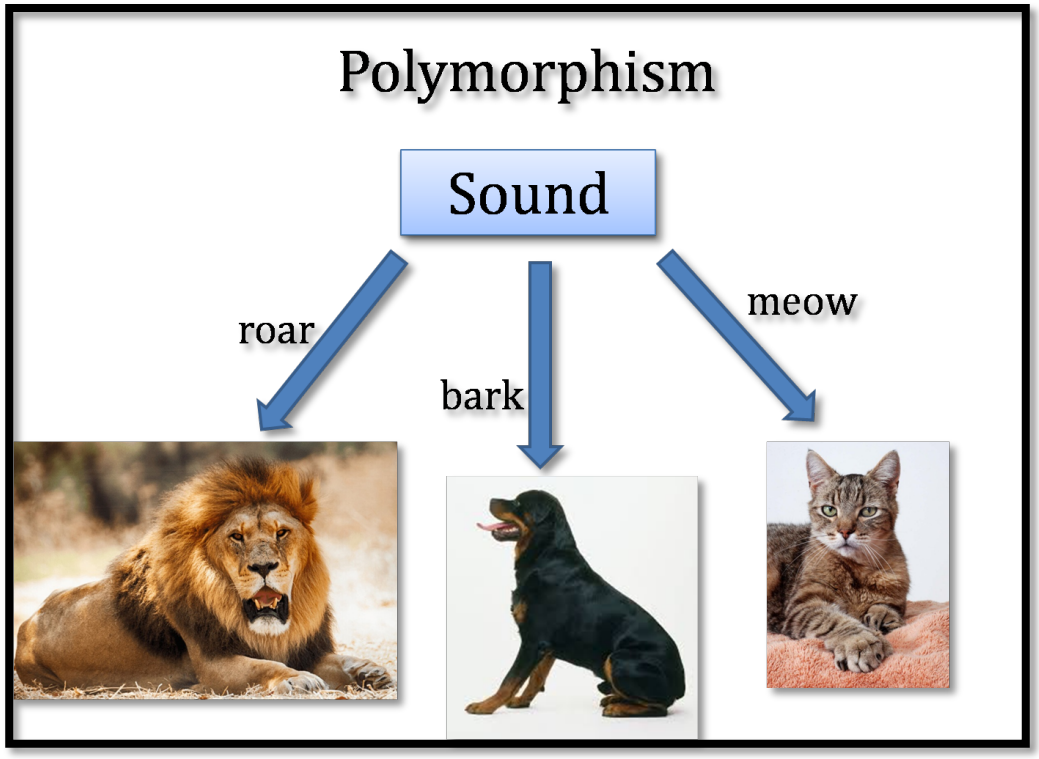


Figure 6.3.4: Various types of Sound

6.3.5. Inheritance

Inheritance is an exciting and essential concept in the Object-Oriented Programming Language. Python inheritance means creating a new class that inherits (takes) all the parent class's functionalities and allows them to add more. No object-oriented programming language would be worthy of looking at or using if it didn't support inheritance.

Introduction and Definition: Inheritance is a way of arranging objects in a hierarchy from the most general to the most specific.

Figure 6.3.5. shows a new Python child class (derived class) is created by inheriting the functionalities of parent class (base class or super class). This child class can be further inherited by other classes.

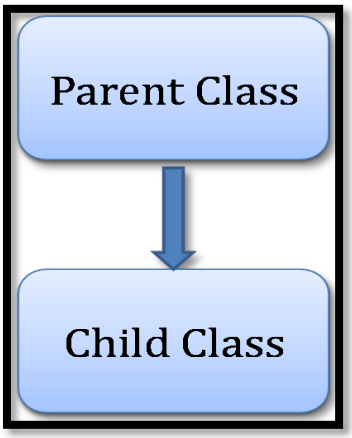


Figure 6.3.5: Parent class and its child class

Parent Class: It contains the attributes and methods common and reusable for the number of times.
Child Class: It inherits all the attributes and methods that are available in its Python Parent Class. It also contains its own attributes and methods.
Think about Employee Class: What would be the attributes and methods?
Attributes: emp Id, age, name, hire date, gender, address, salary
Methods: add () ,delete(), update(), salaryhike()

So, you can declare them in the parent class and use them in the child classes. Similarly, some calculations like the Salary Hike percentage might be the same. We can define a function/method in the parent class for one time, and you can call it from multiple child classes.

Syntax:
class ParentClass: # parent class
class ChildClass(ParentClass): # some attributes of the child class

6.4. Deep-Dive into Programming (let's move towards implementation)

6.4.1.Basic Terminologies: Class, Object and Instance

The figure 6.4.1. (a) depicts the mapping of real-world entity to software entity making use of Oops.

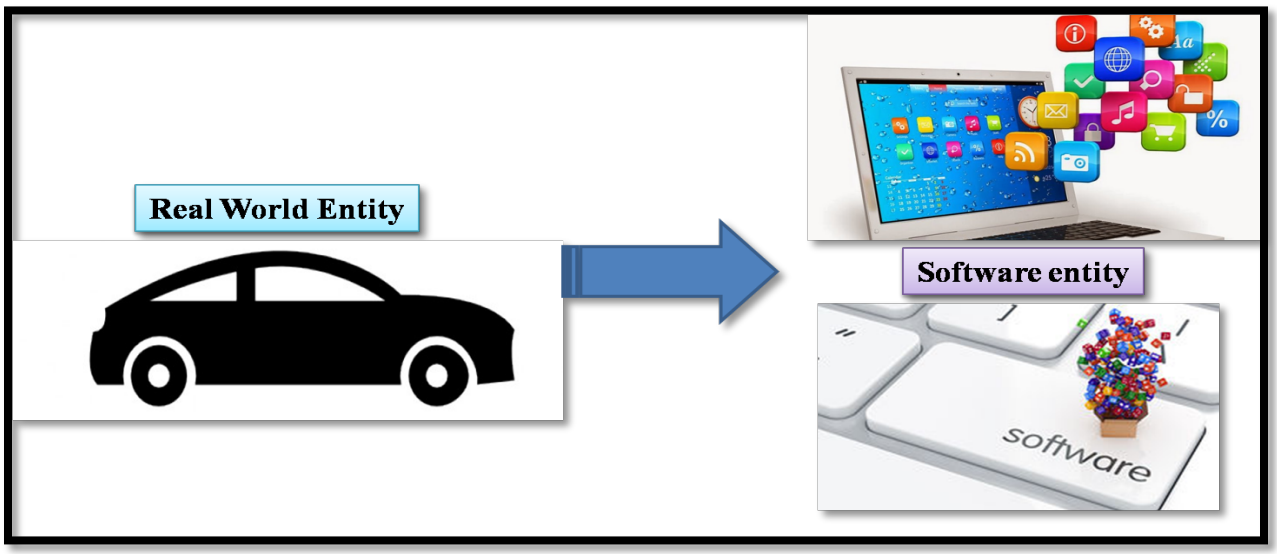


Figure 6.4.1. (a) Real World Entity to Software Entity

Class

Class is the way to get organized real-world things into coding. That's why it is called a blueprint or template. It's all story revolves around the object.

Object

An Object can be defined as a data container with unique values for attributes and behavior defined in its class.

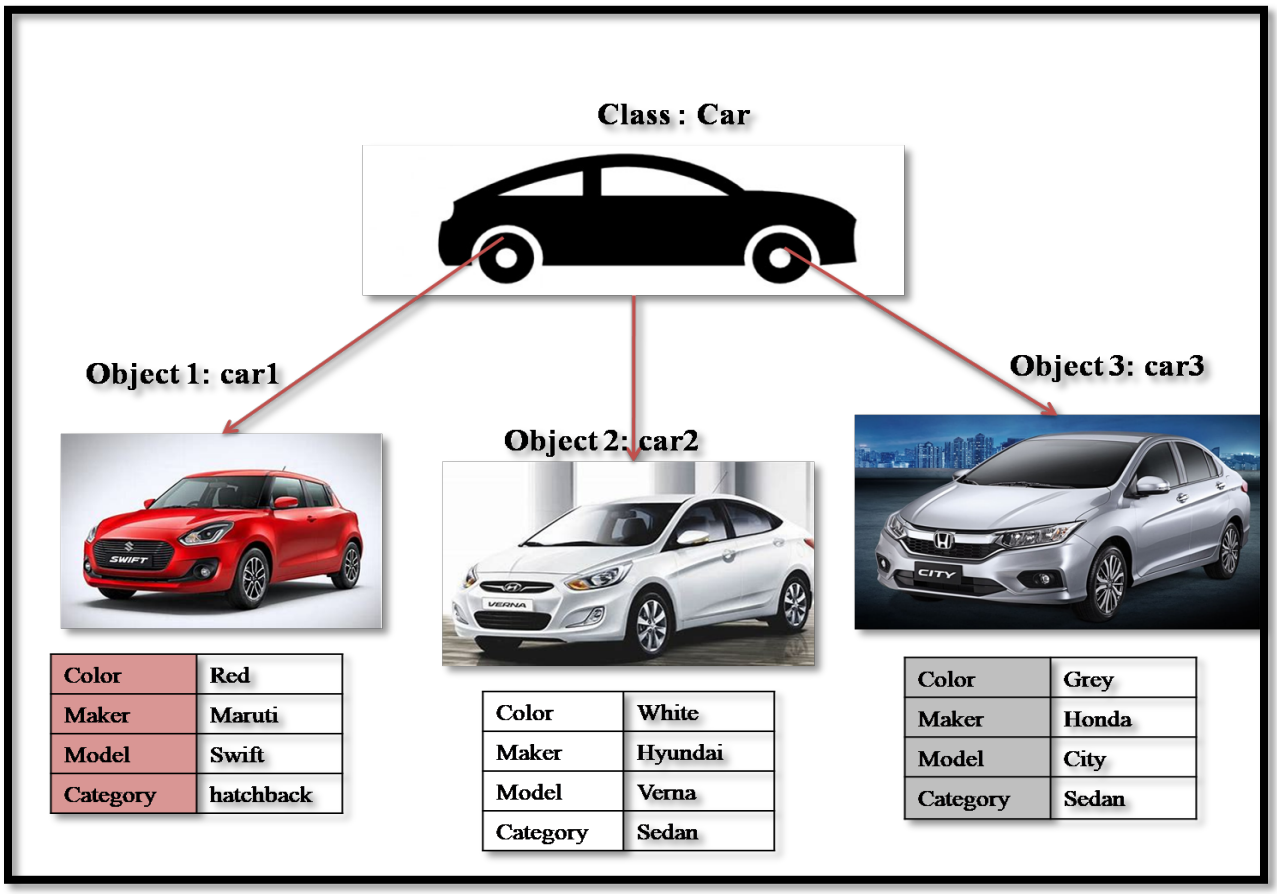


Figure 6.4.1.(b) Mapping of Real Entity as class-object

In the figure 6.4.1 (b), the class car has attributes like color, maker, model, etc. The objects of car like car1, car2 and car3 will have all the attributes of car but with its own attribute value. For example, the first object car1 has four attributes: color is red, maker of a car is Maruti, Model is Swift, and category of a car is Hatchback. Similarly, the second object, object 2, is car 2 with attributes; color is white, maker of a car is Hyundai, Model is Verna, and car category is the Sedan.

Instance

As depicted in figure 6.4.1 (b), we have taken blueprint of car as a class and created three objects for it. The more objects of car can be created and initiated. But each individual object created,

presently active and initiated is known as an instance of respective object or class. Thus, an instance is an object that is created, initiated and presently active.

6.4.2. Representation of class and instances

The figure 6.4.2 represents the class and its instances.

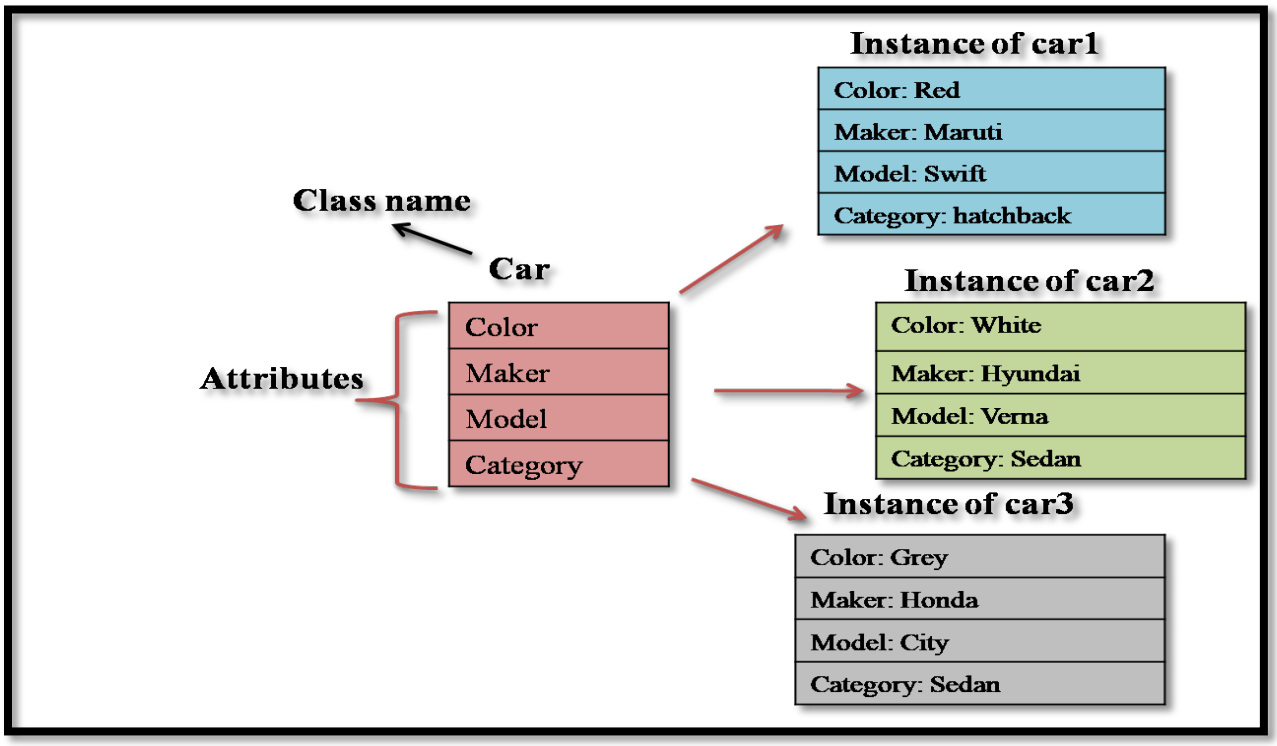


Figure 6.4.2. Class and its instances

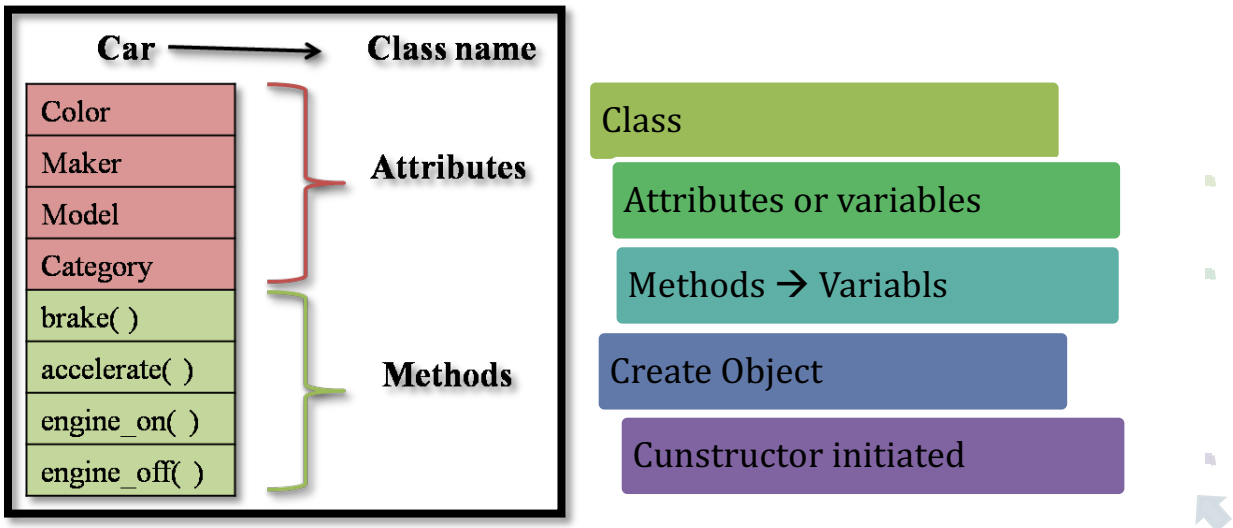


Figure 6.4.4. Classification of a Class and Workflow of class and its components

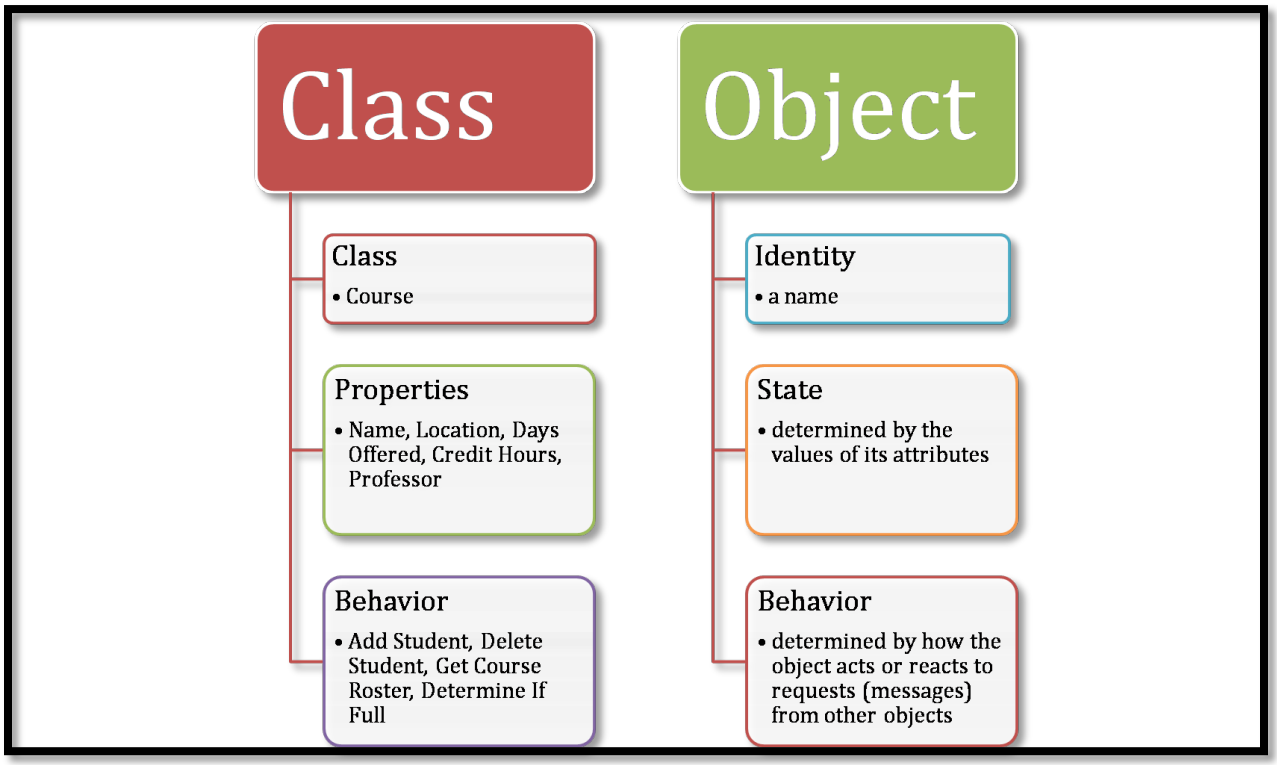


Figure 6.4.5: Components of Class and Objects

6.4.3. Creation of Class and objects in python

The figure 6.4.3 (a) shows that class creation syntax.

```
'''
class <className>
    'this is the syntax how we create class'
    '''
class car:#here car is the class name
    pass
```

Figure 6.4.3. (a) Class creation syntax

The figure 6.4.3. (b) shows that object creation of a class.

```
'''
class <className>
    'this is the syntax how we create class '
    '''
class car:# here car is the class name
    pass
car1=car() #creation of object with the name car1
car2=car() #creation of object with the name car2
car3=car() #creation of object with the name car3

#Here we have created three car objects
```

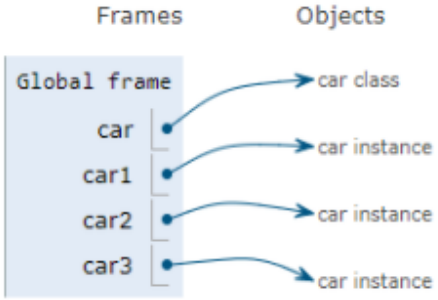


Figure 6.4.3. (b) Creation of Objects

6.4.4. Concept of Self in Python:

Definition: Self is a reference to the current instance of a class.

The program must be able to identify different objects uniquely. Self has solved this purpose. Self is a variable that binds the arguments of an instance of an object with attributes of the class. It is not the keyword in python, and it is a positional variable, and it would always take 1st place in the argument list. By convention, we should use 'self.' However, different variable name can also be used other than *'self'* as 1st argument in methods of class.

The figure 6.4.4 (a) and 6.4.4 (b) when a teacher asks students to tell their marks. The method 'printmarks' will be initiated by the respective student to tell their marks. The marks attribute values which were assigned to respective students using *'self' in* __init__ method will be printed by method 'printmarks'.

Figure 6.4.4. (c) demonstration of the usage of self-using human class with objects man and woman.

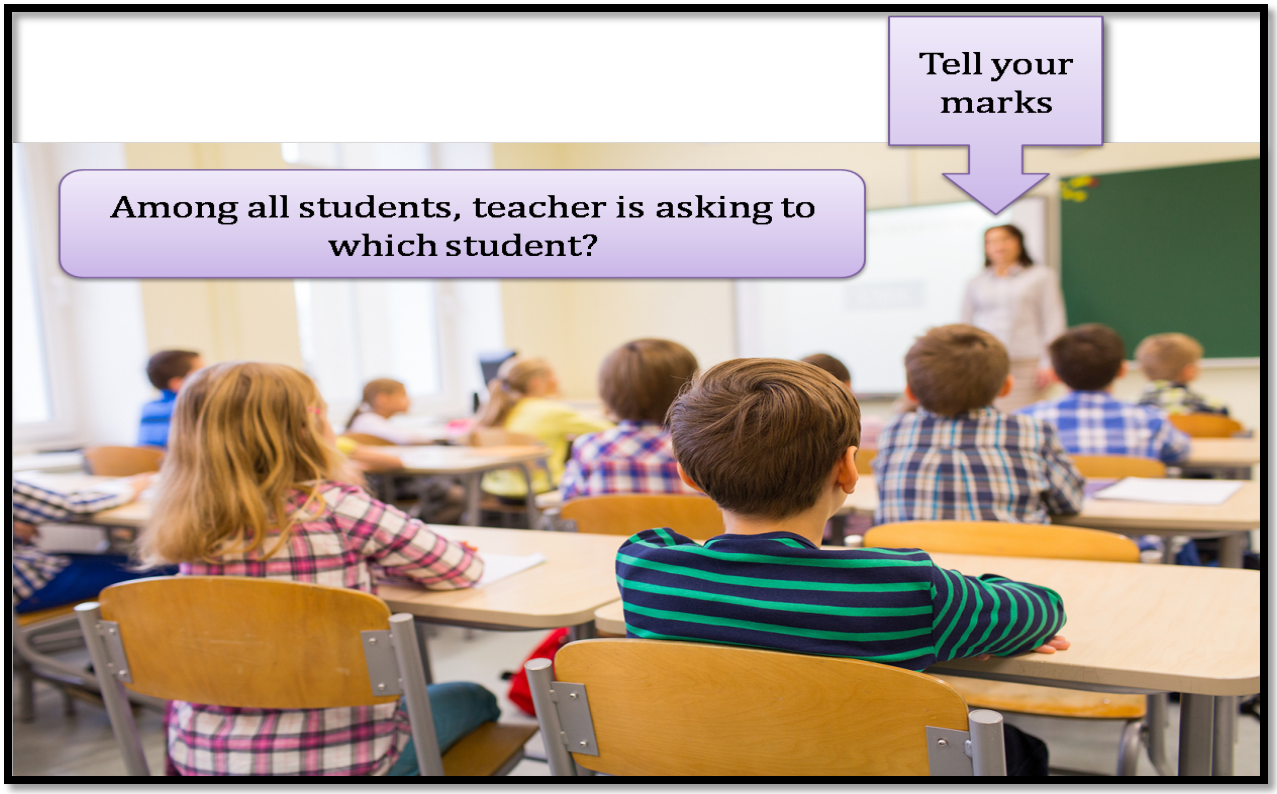


Figure 6.4.4 (a): Class Method called

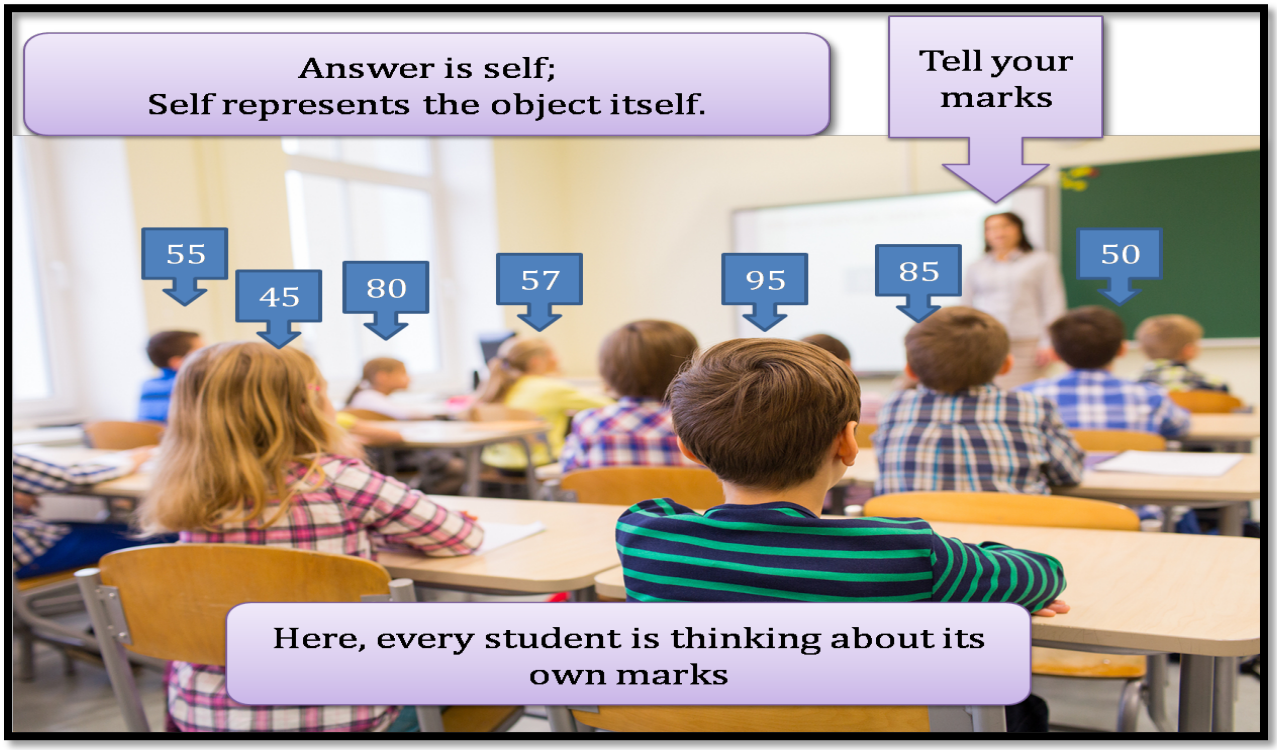


Figure 6.4.4 (b): Use of self in a class object

```
def method_name(self,var1):
    pass #position 1 taken by self
```

```
def methd_name(var1,var2):
    pass #here var1 would do the same work as self
```

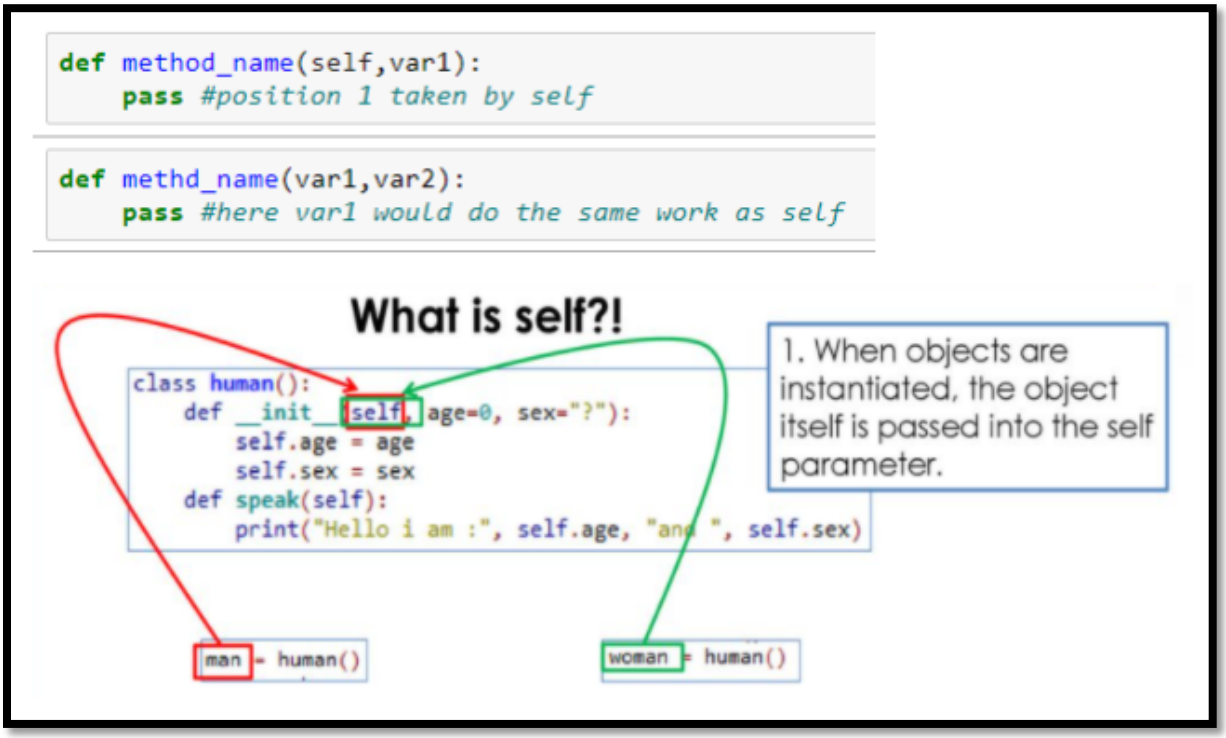


Figure 6.4.4. (c) Demonstration of the usage of self

***6.4.4.1. Can we use self as a variable name in the class? Yes*. Refer figure 6.4.4.1. (a)**

```
class car:#here car is the name

    def new1(self,v1):
        self=4
        print(self,v1)

car1=car() #creation of object with the name car1
car2=car() #creation of object with the name car2
car3=car() #creation of object with the name car3
car3.new1(3)

#Here we have created three car objects
```

```
4 3
```

Figure 6.4.4.1 (a) Self as a variable name

We have taken one more example for better understanding of self, refer figure 6.4.4.1 (b)

```
class Student:
  language = []# mistaken use of a class variable

  def __init__(self, name): # instance variable unique to each instance
    self.name = name

  def add_lang(self, lang_name):
    self.language.append(lang_name)

student1 =Student("Anurag")
student2 =Student("Gopal")
student1.add_lang("Python")
student2.add_lang("Java")
student1.language
```

```
['Python', 'Java']
```

Figure 6.4.4.1. (b)

In figure 6.4.4.1 (b)**, there** is a class named as Student and we have declared a global/class variable named as lanugae.We have created two objects student1 and students 2.Now we call add_lang method to add python and java laguages by student 1 and student2 respectively . If student1 try to access class variable language than both the languages are mapped to student1. But this shouldn't be happening. **In other words,** when you append any value in the list type variable 'language', then all the objects of the class can access the same value. Say, student1 learnt python and the value 'python' appended in the list, then student2 can also get the value 'python'. This is the main issue in class variables.

The properties which are related to object1 should not be conflict with object2. The solution of this issue is use of 'self'

```
class Student:
  # mistaken use of a class variable

  def __init__(self, name): # instance variable unique to each instance
    self.name = name
    self.language = []

  def add_lang(self, lang_name):
    self.language.append(lang_name)

student1 =Student("Anurag")
student2 =Student("Gopal")
student1.add_lang("Python")
student2.add_lang("Java")
print(student1.language)
print(student2.language)
```

```
['Python']
['Java']
```

Figure 6.4.4.1. (c)

6.5.Constructor

The data members of an object are initialized by using constructors. In class, generally, we have

- Attributes/ Properties
- Methods
- Objects

All classes have constructors by default: if you do not write a constructor code yourself, Python creates one for you. However, then you are not able to set initial values for object attributes.
Figure 6.5.(a) provides the **syntax of init()**

```
def __init__(self, input_parameters):
 #initialization code
 ....
 ....
```

Figure 6.5. (a) Constructor

In ___init__(), the first argument is always defaulted "Self." A builder may or may not have other input parameters, i.e., optional input parameters. Init is the initialization acronym, and ___(double underscore) indicates the special init process.
In Python, when you create a new instance of a class.
For example, **Car1 =CarClass()** first the special method __new__() is called to create the object, and then the special Method __init__() is called to initialize it.
Figure 6.5.(b) figure represents how the constructor of a class can create multiple instances of the car object.

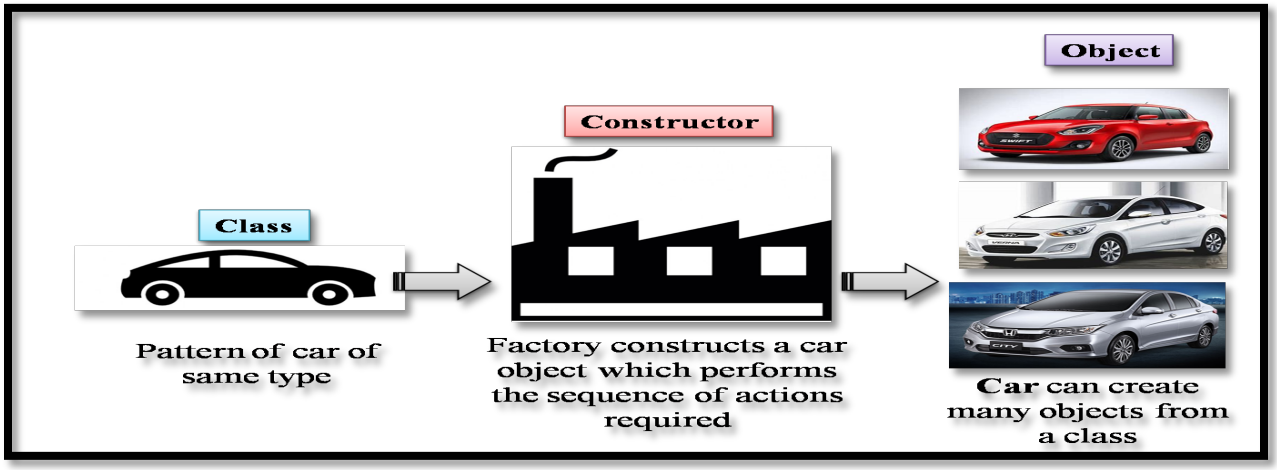


Figure 6.5.(b) The above figure represents how the constructor of a class can creates multiple instances of the car object

6.5.1 Concept of __new__

__new__() method is a magic method used to create an instance of a respective class. __new__() is a static method that takes a class as a first argument.

Two steps are involved in the creation of an object in python. In the first step, __new__( ) creates an instance of a class, and the second step is to initialize that instance by the __init__( ) method.

Whenever we create an object of a class, __new__( ) method of a super class will be called and return an instance of that class. There is no need to define (override the super class method) a __new__() method inside the class until you want to customize the object's creation. An instance of a class can be created inside __new__() method using super function. Figure 6.5.1. represents concept of __new__.

Syntax -

Instance = super(class-name,cls).__new__(cls, *args, **kwargs)

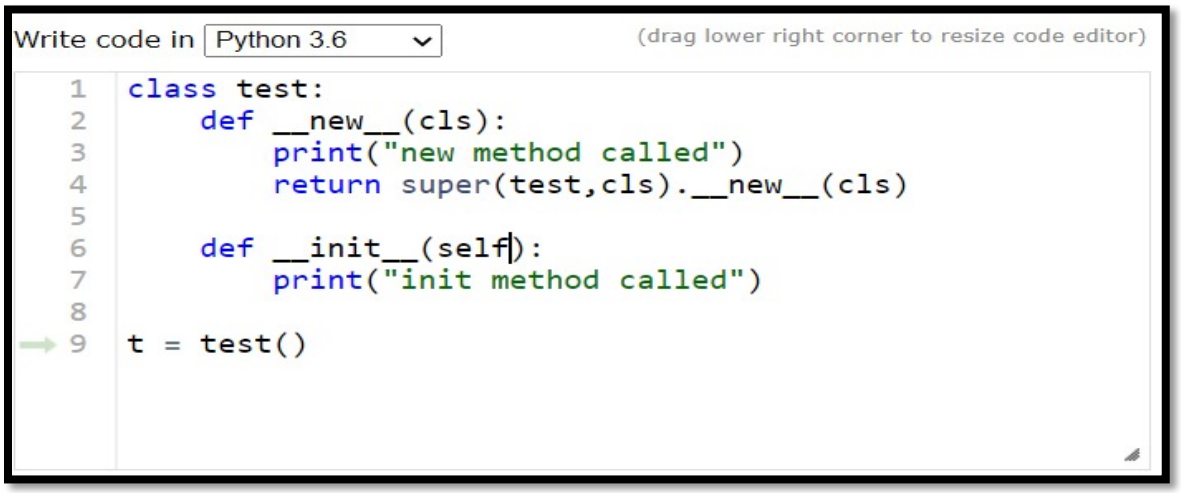


Figure 6.5.1(a) New calling super class

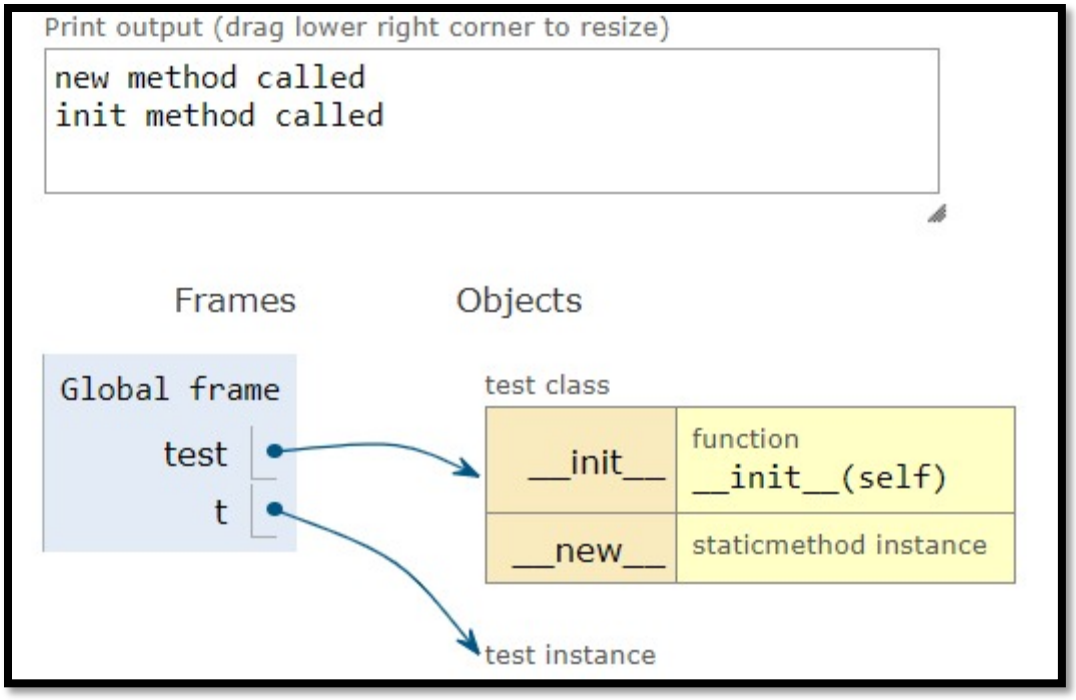


Figure 6.5.1(b) Concept of __new__

In the above example, It is clearly shown that __new__() method is called before the __init__() method for creating an object of class test.

If both, __init__() and __new__() methods exist in the class, then the __new__() method is executed first and decide whether to use __init__ method or not.

In the above example, the class constructor, i.e., __init__( ), is called by the new method using super function. If we omit super from __new__() method, then __init__() method will not execute as shown in the Figure 6.5.1. (b)

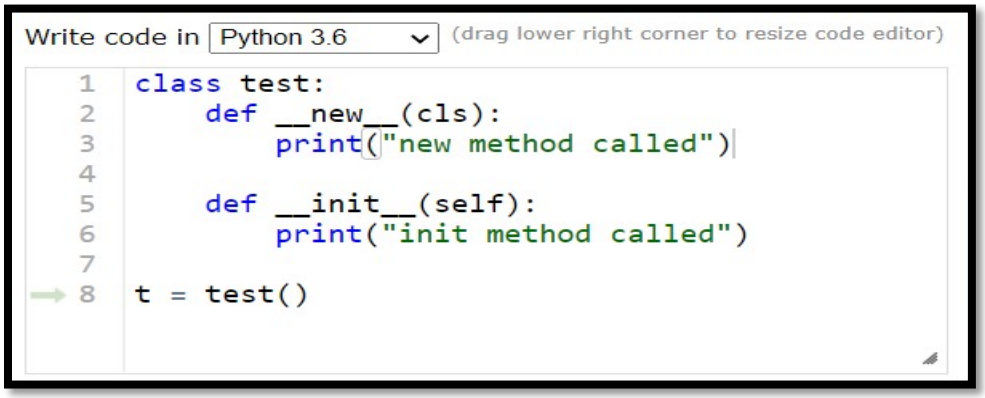


Figure 6.5.1(c) New without calling super class

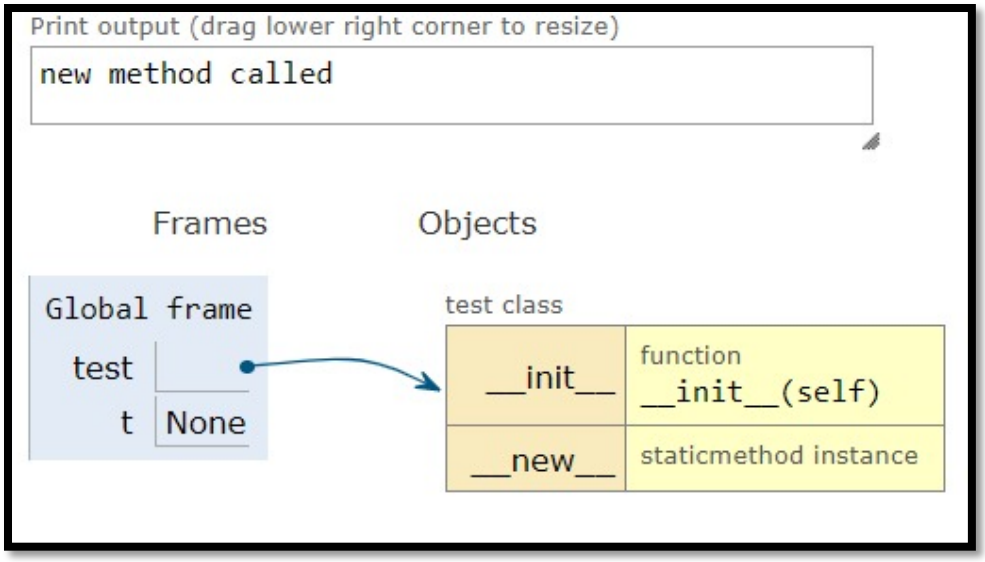


Figure 6.5.1.(d) Visulaization

6.5.2. Types of the constructor:

There are two types of constructors as depicted in figure 6.5.2 (a) and figure 6.5.2 (a)

Parameterized

Parameterized constructors are described as constructors that accept arguments during the object development process. That can be seen with constructors like these, which are used to initialize the object's instance members.

Non-Parameterized or default

Unless we have a constructor in our program, an object cannot be created. Python does it for us if we don't declare a builder in our software.

We can still build an object for the class if we don't have a constructor. This is because, during the software's compilation, a default constructor is implicitly injected by Python.

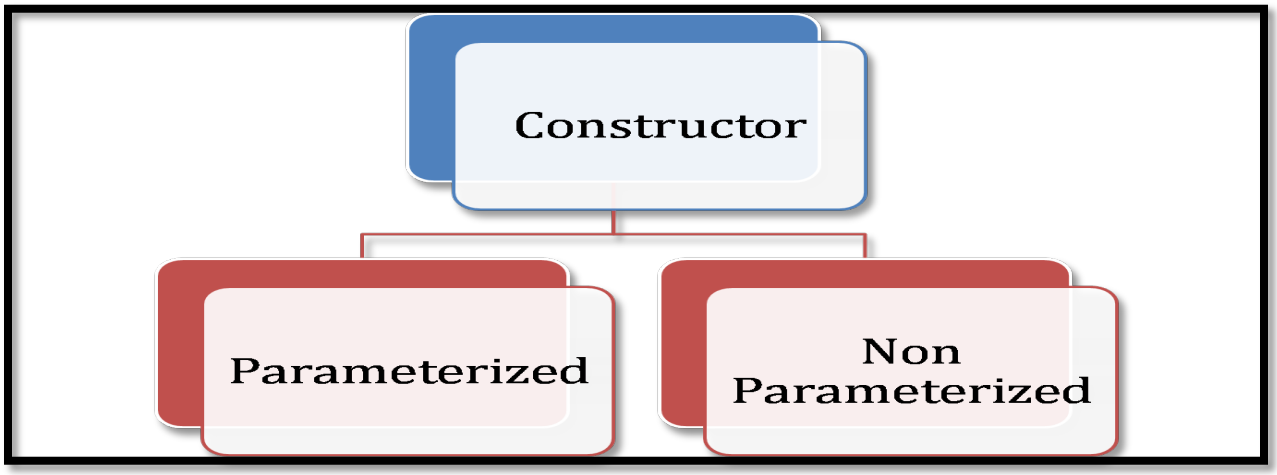


Figure 6.5.2. (a): Types of Constructors

```
#parameterized
def __init__(self,color,maker,model):
    pass

#Non parameterized
def __init__(self):
    pass
```

Figure 6.5.2. (b): Types of Constructors

6.5.3. Default values within the constructor

You may also give the constructor arguments default values as shown in figure 6.5.3. If no value is transferred when instantiating the class, the instance variables are set to default values.

```
class Person:
  def __init__(self, name='Unknown', age=0):
    self.name = name
    self.age = age

  def display_data(self):
    print(self.name)
    print(self.age)

person = Person()
person.display_data()
```

```
Unknown
```

Figure 6.5.3. Demonstration with default values

6.6. Variable

Definition

Object-oriented programming allows for variables to be used at the class level or the instance level. Variables are essentially symbols that stand in for a value you're operating in a program demonstrated in figure 6.6.

```
class car:
    def __init__(self, color, maker):
        self.color = color
        self.maker = maker
```

Figure 6.6. Demonstration of Variables

6.6.1. Types of variables in Python

There are two types of variables used in Python, demonstrated in figure 6.6.1. (a) and 6.6.1. (b).

- Class variables
- Instance variables

Class variables

Class variables are declared inside the class, but these are kept outside the scope of class methods, and these class variables are also called global variables. For example:

```
class A:
    class_var = 10 # class variable
```

Instance variable

Instance variables (also called data attributes) are unique to each instance of the class, and they are defined within a class method. For example:

```
class A:
    class_var=10 # Class Variable
    def __init__(self,int_var):
        self.int_var = int_var  #instance variable
```

The values we give to the object are passed to these variables (Instance), and that's why these variables are called local variables.

We can observe the difference among both types of variables below the examples mentioned. Here' radius' is the instance variable and 'pi' is the class variable.

```
# instance variable and class variable
# program to find out circumference of car wheel
# to access class varibale using self or class name

class Carwheel:
    pi=3.14   #class variable and its accessible throughout the class-+

    def __init__(self,radius):
        self.radius=radius # Instance Variable
    def circumferance(self):
        return 2*Carwheel.pi*self.radius #class variable is accessed through Class Name as well as self
                                         # instance varible is accessed through self.
c=Carwheel(4)
print(c.circumferance())
```

```
25.12
```

Figure 6.6.1. (a) Class and instance variables demonstration

```
# instance variable and class variable
# program to find out circumference of car wheel
# to access class varibale using self or class name
# if we are using global keyword then we can access through variable name itself
class Carwheel:
    global pi
    pi=3.14  #class variable and its accessible throughout the class-+

    def __init__(self,radius):
        self.radius=radius # Instance Variable
    def circumferance(self):
        return 2*pi*self.radius #class variable is accessed directly though variable name
                                # instance varible is accessed through self.
c=Carwheel(4)
print(c.circumferance())
```

pi is class variable

radius is instance variable

```
25.12
```

Figure 6.6.1. (b) Class and instance variables demonstration explained

Note-In the above example, 'c' is the name of an object created by Carwheel class. Even though the method __init__ method has two arguments(self, radius), we will pass only 'radius' while creating the object. The value for the self is not given as arguments (Implicit calling).

6.6.2. How to access instance variables

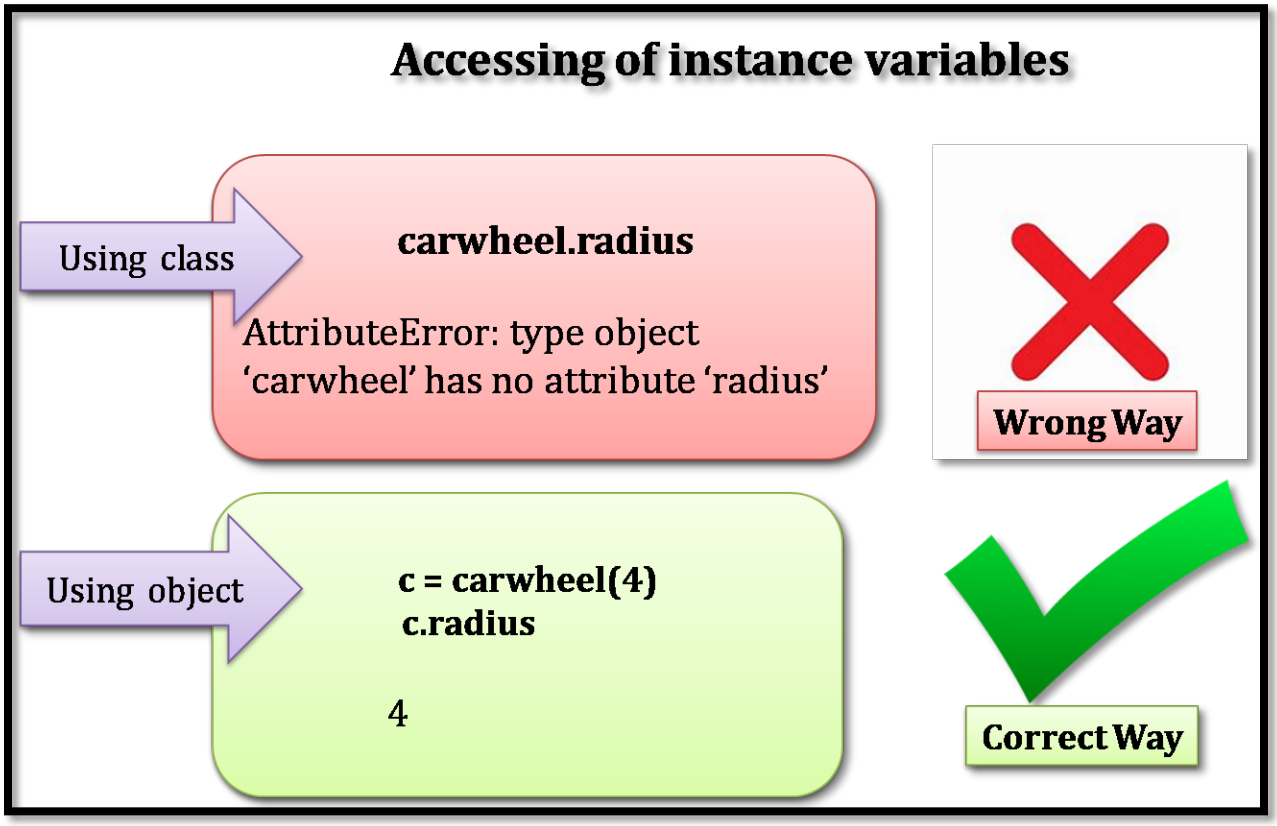


Figure 6.6.2: Use of variables in a class

Here, ***carwheel.radius*** returns an error as it cannot be accessed because the object has not been created

6.7.Methods

A method is just a function object created by a def statement. The method works in the same way as a simple function.

6.7.1. Types of methods in Python:

- Static Method
- Non-Static Method

6.7.1.1.Non-Static Methods / Instance Methods

This is the standard type of method used to set or get the details about instances. The instance method can access the properties that are unique to objects, and it takes self-variable as the first argument, and these methods are also called object or regular Method. These methods' functionality is almost similar to the functions we discussed before except for the concept of 'self.'

Note: Instance variables are used with instance methods.

Explanation:

The figure 6.7.1 represents simple bank enquiry model. The figure 6.7.1.1. represents non static method. In this example, cid, name, and balance are instance variables, and these get initialized when we create an object for the Customer class. If we want to call the deposit () function Non-Static(instance) Method, we must make a thing for the class. If we look at the program, the self-keyword says that those are instance variables and methods quickly.

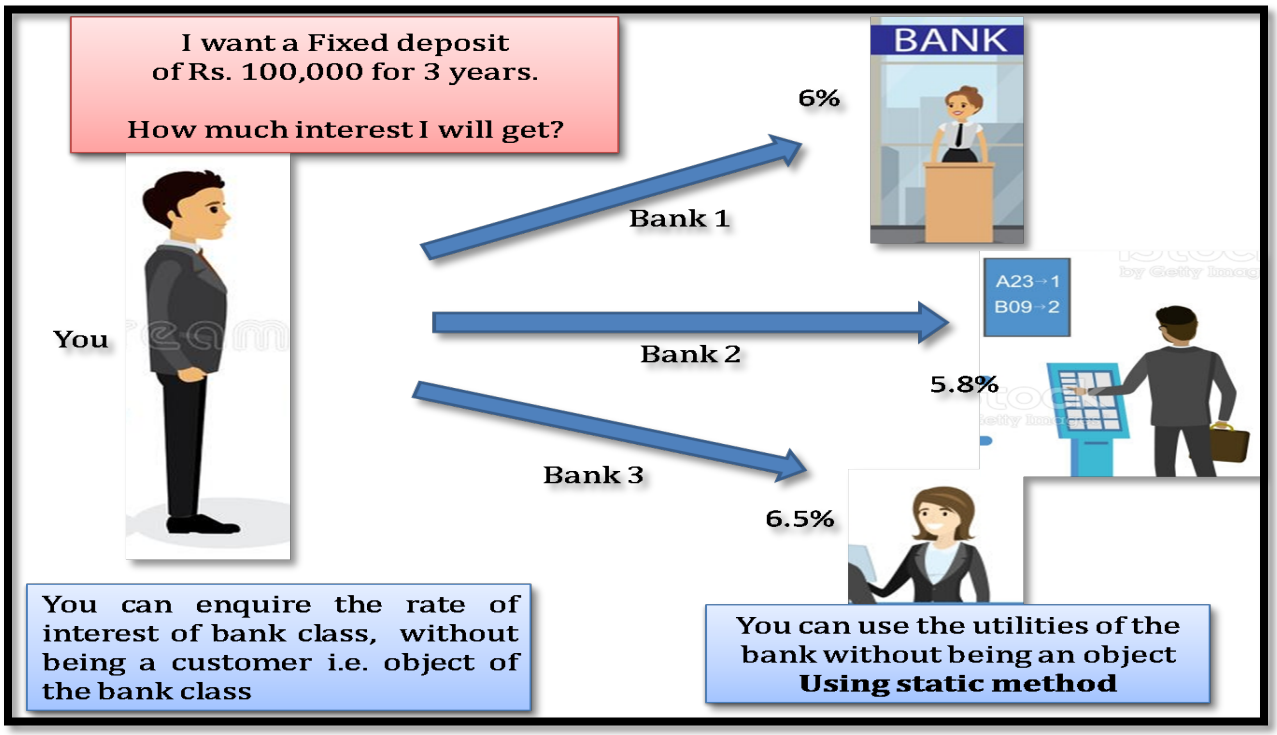
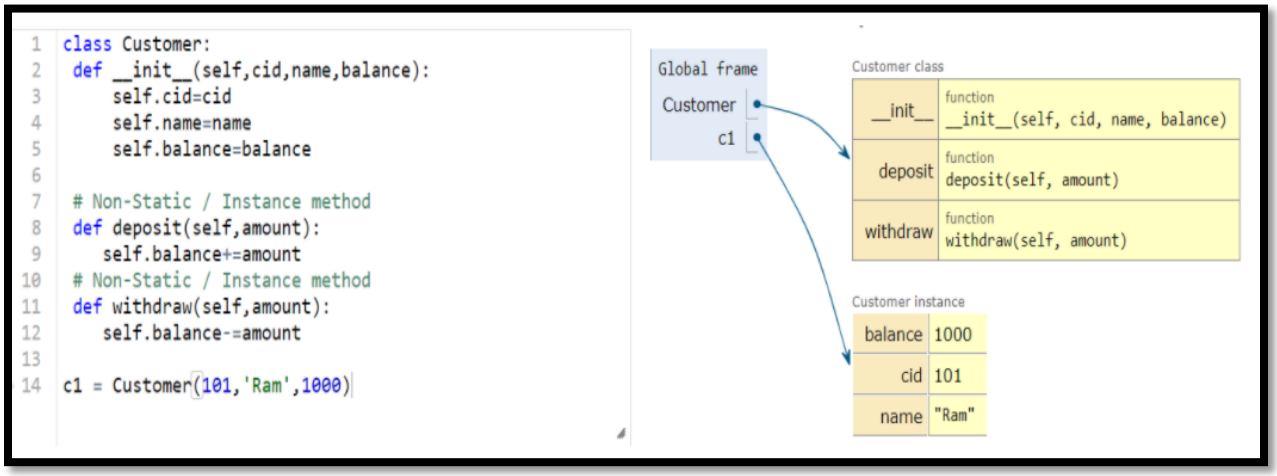


Figure 6.7.1.: Simple Bank Enquiry Model



```
1  class Customer:
2   def __init__(self,cid,name,balance):
3       self.cid=cid
4       self.name=name
5       self.balance=balance
6
7   # Non-Static / Instance method
8   def deposit(self,amount):
9      self.balance+=amount
10  # Non-Static / Instance method
11  def withdraw(self,amount):
12     self.balance-=amount
13
14 c1 = Customer(101,'Ram',1000)
```


Figure 6.7.1.1.: Non-Static Method

6.7.1.2.Static Method:

Static methods are utility functions; whatever is written inside the Method, they perform the assigned task by taking some parameters.

It does not take a positional argument (self-etc.). Static methods can be called without creating the class's object, just by using the class name. The figure 6.7.1.2. represents calling static method *interest* in line no. 11. To create a static Method in Python, we use @staticmethod

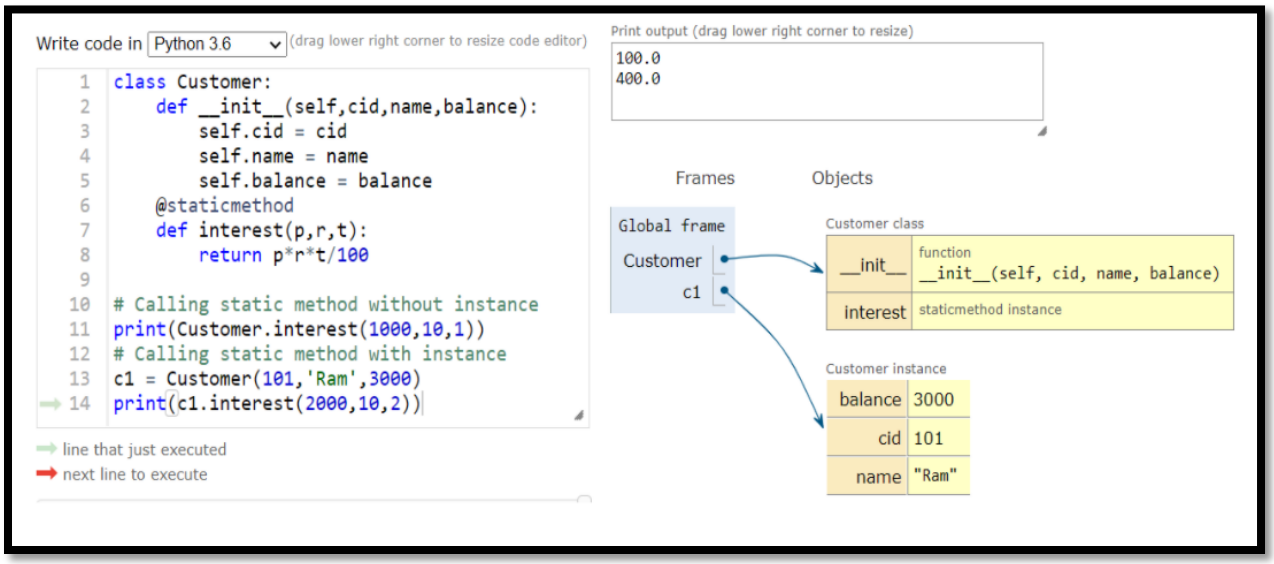


Figure 6.7.1.2. Static method

6.8. Pillars of Oops

- **Inheritance**
- **Encapsulation**
- **Abstraction**
- **Polymorphism**

6.8.1.Inheritance

As discussed during the introduction, inheritance is one of the core concepts of object-oriented programming (OOP) languages.

Inheritance is the way of using the attributes and methods of a class (Base class or Super Class or Parent Class) into another class (Derived class or Child class or inherited class or Subclass). The figure .6.8.1. represents the syntax of inheritance.

Syntax:

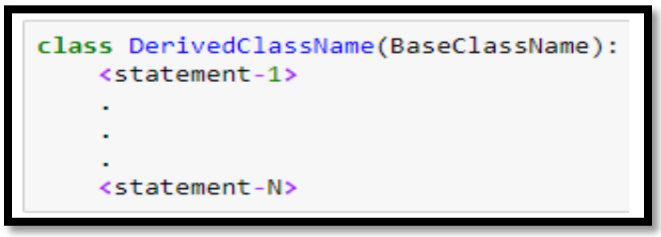


Figure 6.8.1. Syntax

Types of Inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

6.8.1.1.Single inheritance

It enables the child class to inherit from a single base class only. The figure 6.8.1.1 demonstrates the single inheritance showing child class inherited parent class. Now, child class has two functions display2 and dislay1.

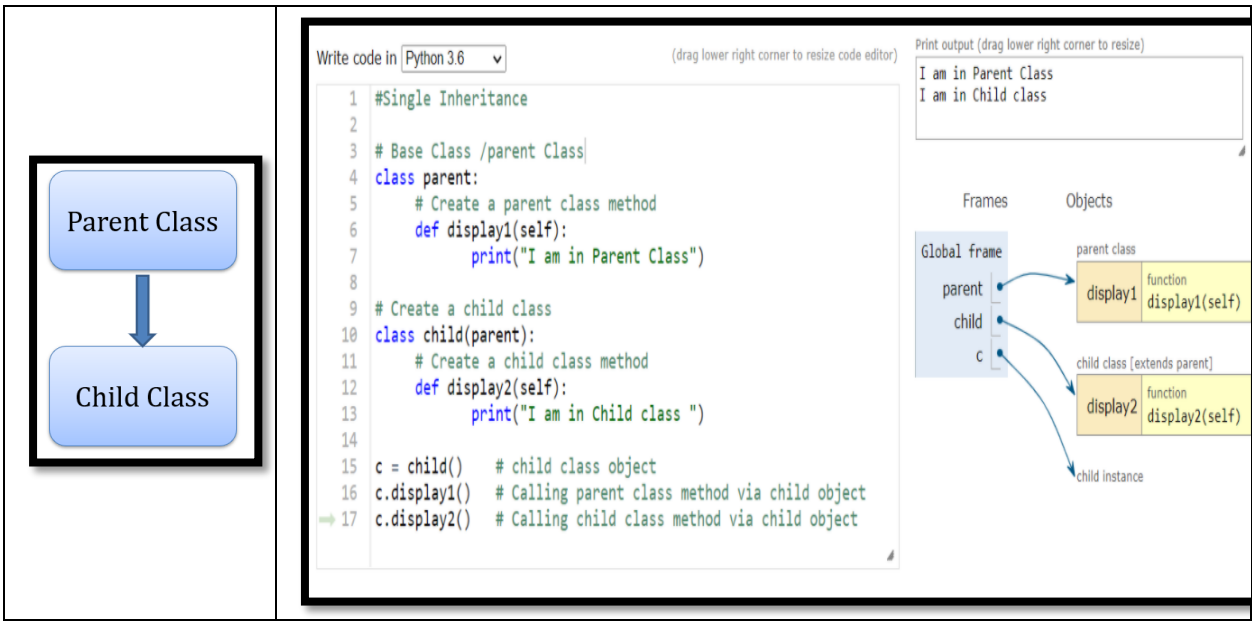


Figure 6.8.1.1. Single Inheritance

6.8.1.2.Multiple inheritance:

When a child class is enabled to inherit from more than one parent class is called Multiple Inheritance. Figure 6.8.1.2. shown an example of multiple inheritance. It has two base class *parent1* and *parent2*. Both of the parent classes are inherited by *child* class. Now *child* class three functions *display1, display2 and display3*.

6.8.1.3.Multilevel inheritance

If a derived class is further inherited into another derived class, then it is called multiple inheritance. Figure 6.8.1.3. shown an example of multilevel inheritance. In this example the base *grandparent* is inherited by the derived class *parent.* Further, *parent* class is inherited by *child* class. Thus, class *child* inhibits multilevel inheritance.

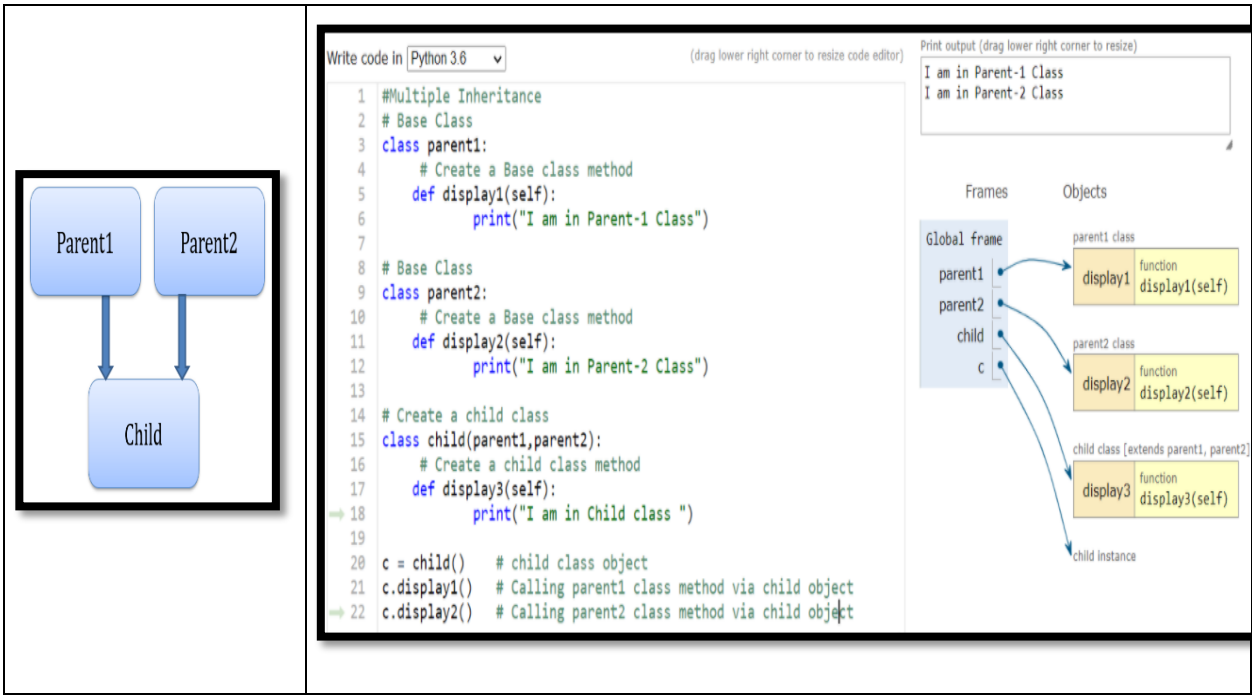


Figure 6.8.1.2. Multiple Inheritance

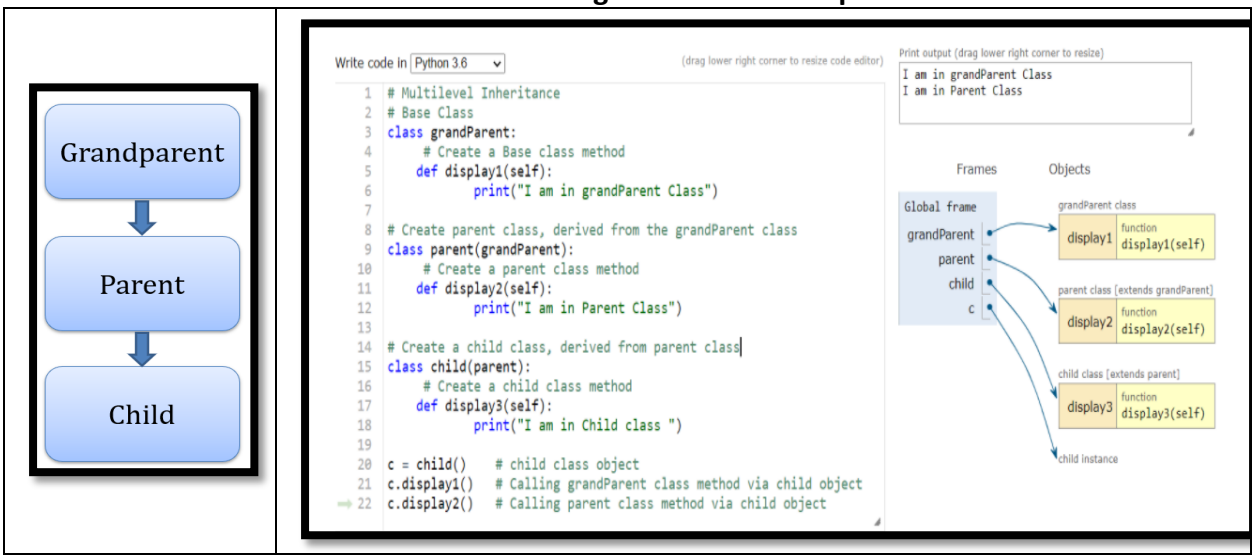
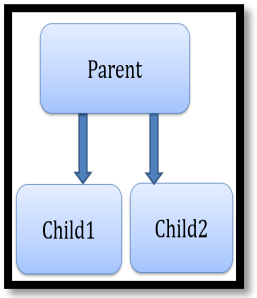


Figure 6.8.1.3. Multilevel Inheritance

6.8.1.4. Hierarchical inheritance

When a Parent class is inherited with more than one child, it is called Hierarchical Inheritance. Figure 6.8.1.4. shown an example of hierarchical inheritance. In this example the base *parent* is inherited by the two derived class *child1* and *child2.* Further, *parent* class. This represents hierarchal inheritance.



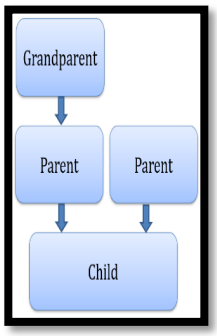

```
# Hierarchical Inheritance
# Base Class
class parent:
     # Create a parent class method
    def display(self):
            print("I am in parent Class")

# Create child class
class child1(parent):
     pass

# Create a child class
class child2(parent):
     pass

c1 = child1()    # child1 class object
c2 = child2()    # child2 class object
c1.display()   # Calling parent class method via child1 object
c2.display()   # Calling parent class method via child2 object
```

```
I am in parent Class
I am in parent Class
```


Figure 6.8.1.4. Hierarchal Inheritance

6.8.1.5. Hybrid inheritance

When in a single program more than one type of inheritance happens is called hybrid inheritance. Figure 6.8.1.5. shown an example of hybrid inheritance. In this example the multilevel inheritance and multiple inheritance taking place in same program.




```
# Hybrid Inheritance

class grandparent:
    def display1(self):
        print("I am in Grand Parent")

class parent1(grandparent):
    pass

class parent2:
    def display2(self):
        print("I am in Parent2")

class child(parent1,parent2):
    pass

c = child()
c.display1()
c.display2()
```

```
I am in Grand Parent
I am in Parent2
```


Figure 6.8.1.5. Hybrid Inheritance

6.8.2. Inherit Constructor

Let's look at the below example. The child class inherits the parent class, so the child class inherits the parent class's constructor, depicted in figure 6.8.2.

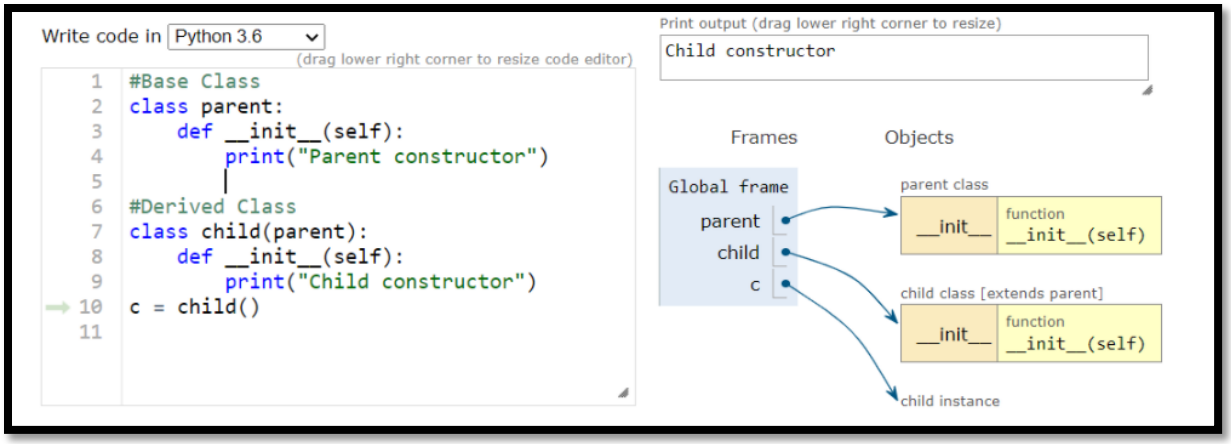


Figure 6.8.2. Inherit Constructor

While creating a child object, we need to pass value because the child class inherits the parent's constructor.

6.8.3. What will happen if our child class has its constructor?

Parent constructor will not be inherited if a child has its constructor.
The figure 6.8.3 shows that both parent and child class have their constructor. When we create an object of the child class, then only the child class constructor is called.

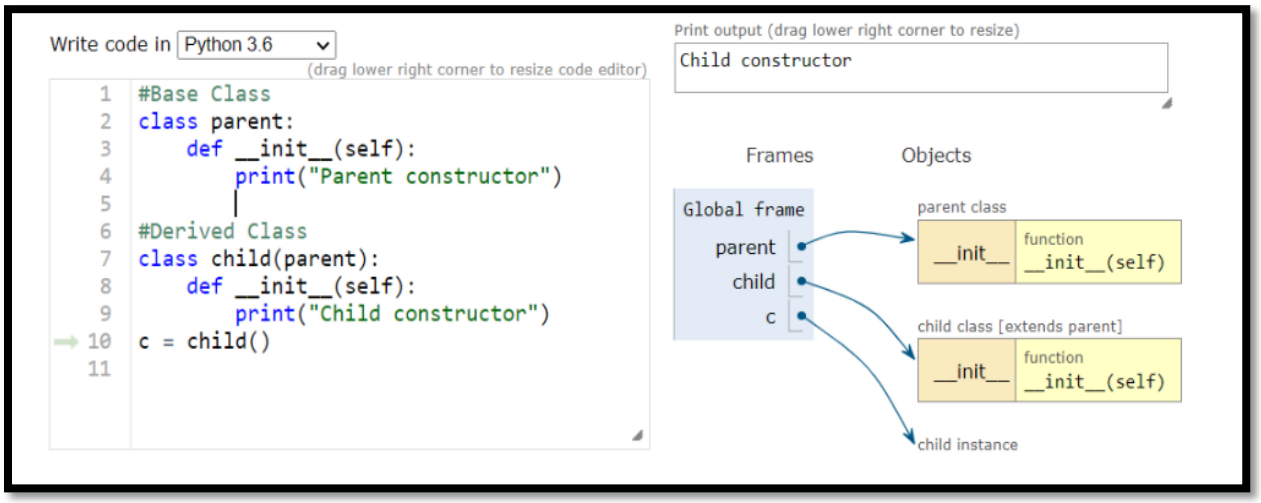


Figure 6.8.3. Demonstration

6.8.4. How to inherit the parent constructor if the child has its constructor?

To inherit the parent constructor (same as true for oops), we use the super () function in Python as shown in figure 6.8.4(a) and figure 6.8.4(b).

```python
#Base Class
class parent:
    def __init__(self):
        print("Parent constructor")

#Derived Class
class child(parent):
    def __init__(self):
        print("Child constructor")
        super().__init__() # Invoke parent constructor
c = child()
```

Figure 6.8.4. (a) Use of Super

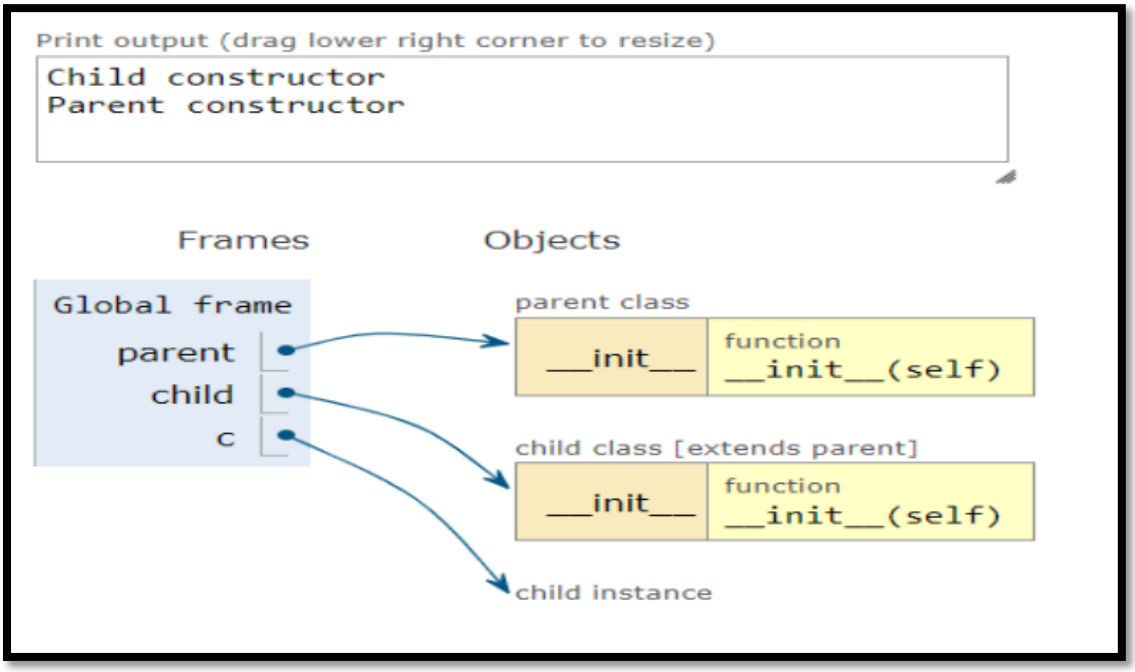


Figure 6.8.4. (b)Visualization

6.8.5. Method Resolution Order (MRO)

The order defined for searching attributes and methods in a class is called as Linearization and the set of rules defined for searching attributes and methods in a class is called MRO (Method Resolution Order). In case, a method is available in both super and child class, then the interpreter needs a way to decide which method to be initiated.

```python
# Python program showing
# how MRO works

class ClassSuper:
    def rk(self):
        print(" In class SUPER")
class ClassChild(ClassSuper):
    def rk(self):
        print(" In class CHILD")

r = ClassChild()
r.rk()
```

```
 In class CHILD
```

Figure 6.8.5 (a) Method Resolution Order

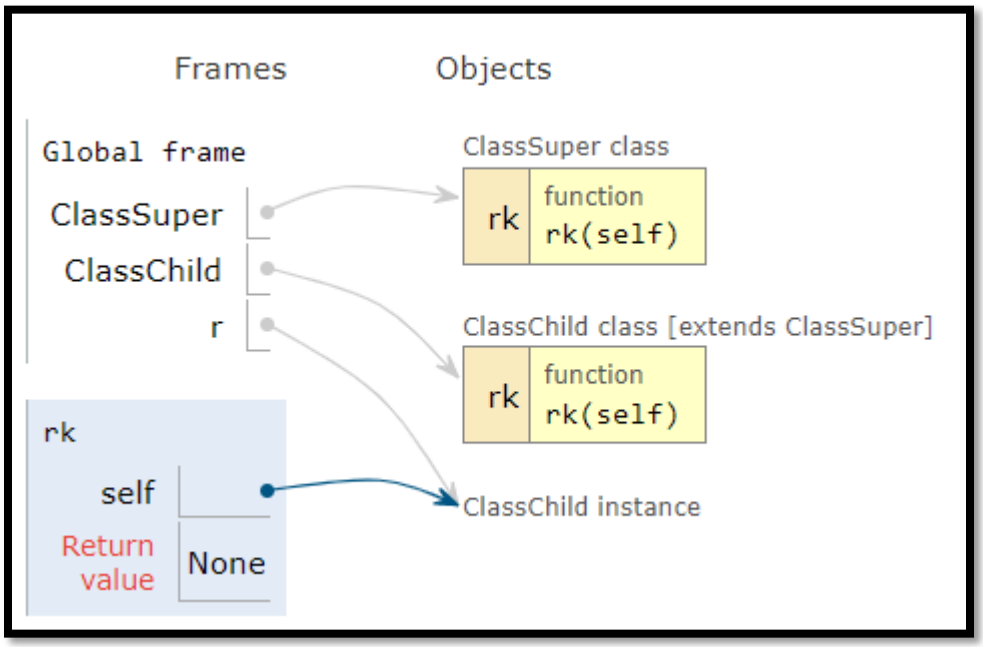


Figure 6.8.5 (a) Method Resolution Order

The figure 6.8.5 (a) and 6.8.5 (b) shows that the method 'rk' is invoked from ClassChild not ClassSuper. Therefore, the order follows is ClassChild -> ClassSuper

Now, consider an example for multiple inheritance as given below:

```
# Python program showing
# how MRO works

class A1:
    def method1(self):
        print(" In class A1")
class A2(A1):
    def method1(self):
        print(" In class A2")
class A3(A1):
    def method1(self):
        print("In class A3")

# classes ordering
class A4(A2, A3):
    pass

r = A4()
r.method1()
```

```
 In class A2
```

Figure 6.8.5 (c) Method Resolution Order: Multiple Inheritance

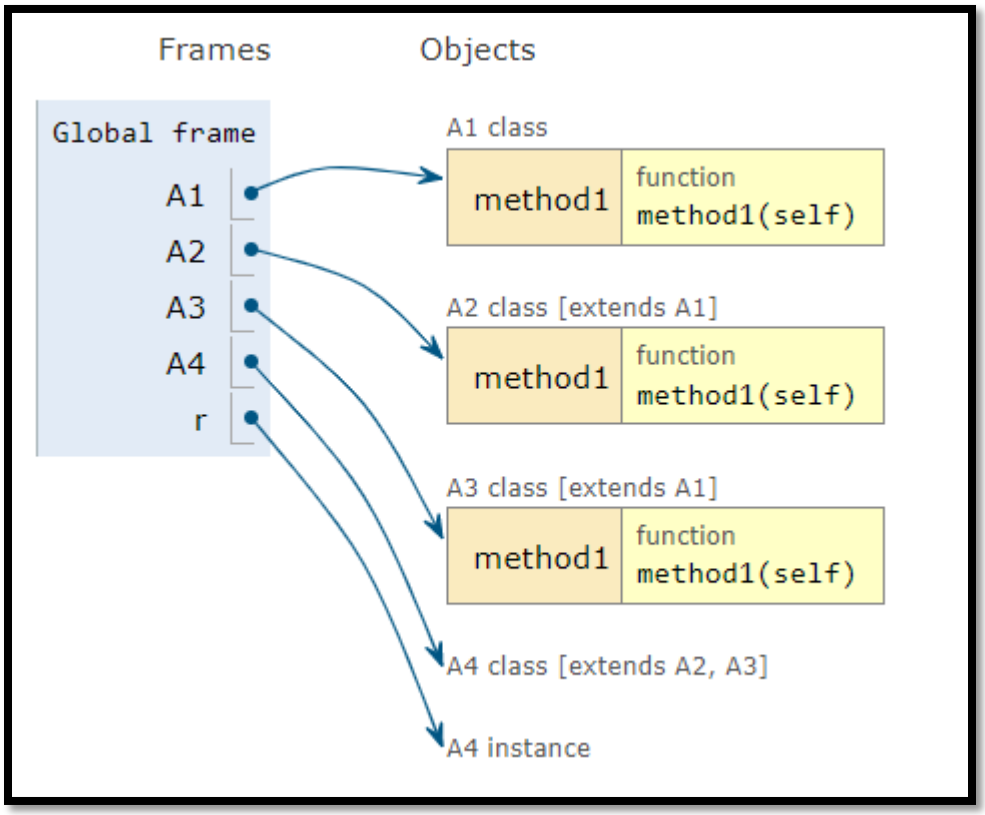


Figure 6.8.5 (d) Method Resolution Order: Multiple inheritance

The figure 6.8.5 (a) shows that Python follows depth first order in multiple inheritance. It ended at calling method from class A1. Using Method Resolution Order, the order followed is A4 -> A2 -> A3 -> A1.

6.8.2.Encapsulation

Need: Preventing unauthorized parties' direct access

When communicating with groups and confidential statistics, giving everyone global access to all variables is not a brilliant idea. Encapsulation allows one to control the necessary variables without granting complete software access to them.

Explicitly specified methods for updating, modifying, and removing data from variables can be used. The advantage of using this approach to programming is that you have more power over the data you submit, and it's safer.

In all object-oriented programming languages, the principle of encapsulation is the same. When the ideas are extended to specific languages, the distinction becomes apparent if we compared to the other programming languages like Java, where variables and methods have access modifiers (public or private).

Python offers global access to all variables and processes. We'll use a few different ways to manage the access to variables within a Python program because we don't have access modifiers in Python.

6.8.1.1.Access Control (Access Modifier)

Python offers multiple methods to limit variable and method access across the program. Let's go over the methods in detail.

Case Study:

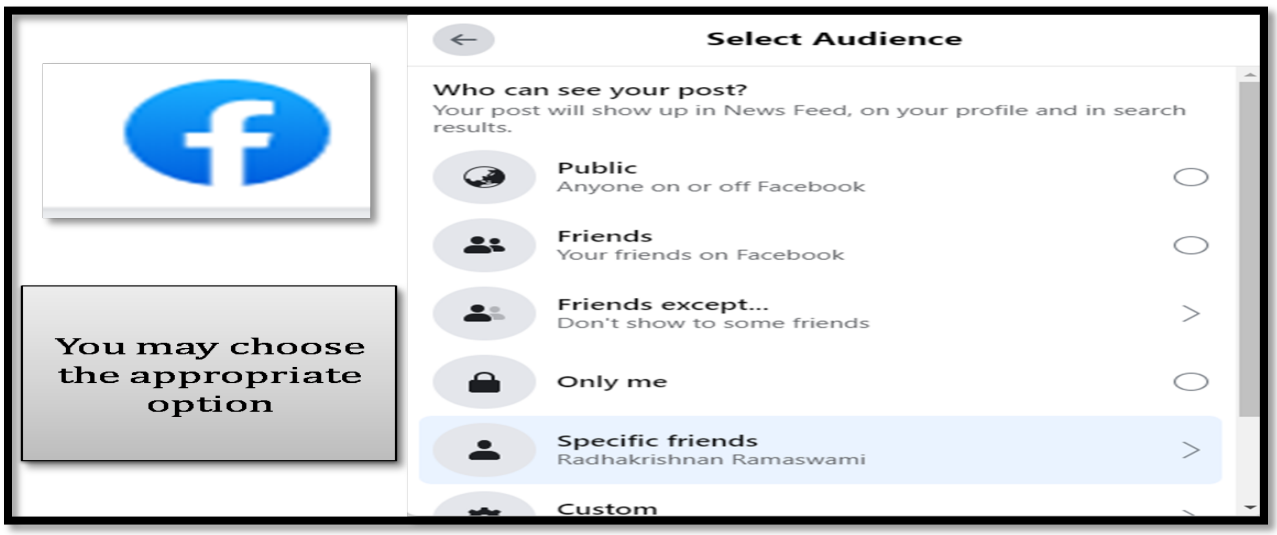


Figure 6.8.1.1: Example of Access Modifier

While posting a post on Facebook, you will get an option to select all your friends, or anyone on Facebook, or a set of specific people as you can opt for any one chance to share the post among a particular group of people. You can relate this as the access modifiers in object-oriented programming as depicted in figure 6.8.1.1.

6.8.2.2.Protected or Single Underscore[_]

Prefixing a private variable with an underscore is a standard Python programming convention for identifying it. Now, on the programmer side of things, this makes no difference. The attribute can be accessed directly as before. However, since it is a programming convention, it informs all programmers that the variables or methods should only be used within the class's limits.
Protected members of a class can be accessed both by the class and by its sub-classes. It is not accessible to any other world. Members allow the child class to inherit unique properties from the parent class. To render an instance variable safe in Python, add the prefix (single underscore) to it. It essentially forbids entry to it unless it is from inside a sub-class. The figure 6.8.2.2 depicts the use of protected or single underscore.

```
class Person:
   def __init__(self, name, age):
      self.name = name
      self._age = age

   def display(self):
      print(self.name)
      print(self._age)

person = Person('Harsh', 32)
#accessing using class method
person.display()
#accessing directly from outside
print(person.name)
print(person._age)
```

```
Harsh
32
Harsh
32
```

Figure 6.8.2.2. Demonstration of Protected

6.8.2.3.Privateor Double Underscores[__]

If you want to render class members, such as methods and variables, private, use double underscores before them. However, Python has some support for the private modifier. Name

mangling is the term for this process. It is also possible to reach the members of the class from outside of it. When you declare a data member secret, you're saying that no one in the class should be able to view it. In Python, however, the keyword 'personal' is not written directly. No function in Python essentially limits access to any instance variable or procedure. Python recommends prefixing the name of the variable/method with a single or double underscore to mimic the actions of secure and private access specifiers.
A variable becomes private when it has the double underscore prefixed to it. It implies that no one outside the class can touch it. **Any attempt to do so will result in an AttributeError** as shown in figure 6.8.2.3.

```
class Person:
   def __init__(self, name, age=0):
      self.name = name
      self.__age = age

   def display(self):
      print(self.name)
      print(self.__age)

person = Person('Gopal', 45)
#accessing using class method
person.display()
#accessing directly from outside
print('Trying to access variables from outside the class ')
print(person.name)
print(person.__age)
```

```
Gopal
45
Trying to access variables from outside the class
Gopal

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-d557657c7f74> in <module>()
     14 print('Trying to access variables from outside the class ')
     15 print(person.name)
---> 16 print(person.__age)

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

Figure 6.8.2.3. Using Private

Mangling
Any identifier with __Var in Python is rewritten as _Classname__Var by a Python interpreter, and the class name remains the same. In Python, this name-changing process is known as Name Mangling.

Using Getter and Setter methods to access private variables as depicted in figure 6.8.2.4.

```
class Person:
  def __init__(self, name, age=0):
    self.name = name
    self.___age = age

  def display(self):
    print(self.name)
    print(self.___age)

  def getAge(self):
    print(self.___age)

  def setAge(self, age):
    self.___age = age

person = Person('Aatif', 34)
#accessing using class method
person.display()
#changing age using setter
person.setAge(35)
person.getAge()
```

```
Aatif
34
35
```

Figure 6.8.2.4. Getter and setter

6.8.3.Abstraction

Need: To let the user know the main functionality of the module without going into details of implementation.

For example, consider the following scenario: you've just purchased a new mobile device. Along with the software, you get a user guide that describes how to use the program, but it provides little detail about the device's internal workings.
Another example is when you use the TV remote and don't understand how hitting a key in the remote switches the channel internally. You're already aware that clicking the + volume key raises the volume.

6.8.3.1.How to implement abstraction in Python?

The abstract class is generated in Python by deriving from the metaclass ABC, which is part of the ABC module (Abstract Base Class) as depicted in figure 6.8.3.1.

Syntax:

```
from abc import ABC
Class MyClass(ABC):
```

Figure 6.8.3.1. Synatx for creation of Abstract class

In the ABC module, ABC metaclass has to be imported, and the abstract class has to inherit the ABC class to be considered an abstract class.

6.8.3.2. Abstract Method

A process must be decorated with the @abstractmethod decorator to describe abstract methods in an abstract class. To use the annotation, you must import the @abstractmethod decorator from the ABC module as depicted in figure 6.8.3.2 (a).

Syntax:

```
from abc import ABC, abstractmethod
Class MyClass(ABC):
 @abstractmethod
 def mymethod(self):
  #empty body
  pass
```

Figure 6.8.3.2. (a) Abtract method

Points to be noted:

1. An abstract class can have both standard methods as well as abstract methods.
2. The abstract class works as a template for other classes. Using an abstract class, you can define a generalized structure without providing complete implementation of every Method. Methods that provide standard functionality for all derived classes are described as concrete methods in the abstract class, whereas the methods where implementation may vary are defined as abstract methods.
3. Abstract class can't be instantiated, so it is impossible to create objects of an abstract class.
4. Derived classes must implement the abstract Method of a superclass. If the derived class Python does not implement any abstract method throws an error.
5. Refer figure 6.8.3.3 (b) for abstract method in detail.

```
from abc import ABC, abstractmethod

class Parent(ABC):
 #common functionality
 def common_to_all(self):
     print('In common method of Parent')

 @abstractmethod
 def different_from_all(self):
     pass

class Child1(Parent):
  def different_from_all(self):
     print('In vary method of Child1')

class Child2(Parent):
  def different_from_all(self):
     print('In vary method of Child2')

# object of Child1 class
obj = Child1()
obj.common_to_all()
obj.different_from_all()
# object of Child2 class
obj = Child2()
obj.common_to_all()
obj.different_from_all()
```

Output:

```
In common method of Parent
In vary method of Child1
In common method of Parent
In vary method of Child2
```

Figure 6.8.3.3. (b) Demonstartion full program

6.8.4.Polymorphism

One of the four basic OOPS terms is polymorphism. Polymorphism describes the tendency of a function of the same name to perform several functions. It provides a structure that can handle several object types like water can have many forms with different forms as shown in figure 6.8.4 (a).

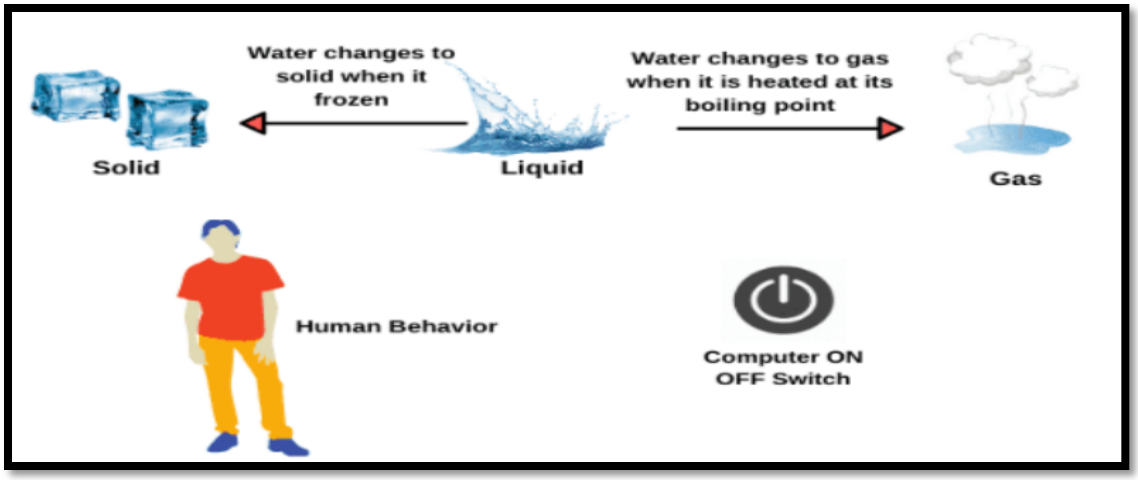


Figure 6.8.4 (a): Multiple forms of an entity

Polymorphism is supported in object-oriented languages through:

- Method/Function overloading
- Method/Function overriding
- Operator overloading

Python supports polymorphism as an object-oriented language, but only by method overriding and operator overloading. ***Python does not support method overloading in the conventional context.***

Polymorphism allows Python to use various kinds of classes in the same way. A loop that iterates through a tuple of objects can be used to do this. After that, you can call the methods without understanding what sort of class the entity belongs to!

```
class Cat():
    def age(self):
        print("This function determines the age of Cat.")

    def color(self):
        print("This function determines the color of Cat.")

class Lion():
    def age(self):
        print("This function determines the age of Lion.")

    def color(self):
        print("This function determines the color of Lion.")

obj1 = Cat()
obj2 = Lion()
for type in (obj1, obj2): #loop to iterate through the obj1, obj2
    type.age()
    type.color()
```

Output:

```
This function determines the age of Cat.
This function determines the color of Cat.
This function determines the age of Lion.
This function determines the color of Lion.
```

Figure 6.8.4 (b) Polymorphism

The figure 6.8.4 (b) shows method *age* and *color* having two different forms in classes and *Lion*.

6.8.4.1.Method overloading

In Python, method overloading is something where similar method names exist in the same class but vary in form or number of arguments transferred, which is not supported. **Note** that in Python, attempting to use the same name does not cause a compile-time error; however, only the last specified Method is remembered in such a scenario; calling some other overloaded method causes an error as depicted in figure. 6.8.4.1 (a).

```
class method_overload:
  # first sum method, two parameters
  def sum(self, a, b):
    s = a + b
    print(s)
  # overloaded sum method, three parameters
  def sum(self, a, b, c):
    s = a + b + c
    print(s)

od =  method_overload()
od.sum(7, 8,9)
```

Output will be: 24

But If we call any other overloaded method will results in an error.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-c728d217489b> in <module>()
     10
     11 od =  method_overload()
---> 12 od.sum(7, 8)

TypeError: sum() missing 1 required positional argument: 'c'
```

Figure 6.8.4.1.(a) Trying overloading

Achieving Method overloading in Python

Since overloading a function of the same name is difficult in Python, method overloading is achieved by making a single method with several parameters. You must validate the number of arguments passed to the Method and then continue with the operation.

For example, if we use the same sum () method as above but transfer two or three parameters, we can do method overloading in Python, as seen in figure 6.8.4.1 (b)

```
class method_overload:
 # sum method with default as None for parameters
 def sum(self, a=None, b=None, c=None):
  # When three params are passed
  if a!=None and b!=None and c!=None:
   s = a + b + c
   print('Sum = ', s)
  # When two params are passed
  elif a!=None and b!=None:
   s = a + b
   print('Sum = ', s)

od =  method_overload()
od.sum(7, 8)
od.sum(7, 8, 9)
```

```
Sum =  15
Sum =  24
```

Figure 6.8.4.1. (b) The correct way of overloading

6.8.4.2.Method overriding

Method overriding allows you to modify the execution of a child class method that is already specified in its superclass. When a child class has a superclass and a Function method of the same name and number of arguments, the child class method overrides the parent class method. The parent class method is implemented when the Method is called with the parent class object. When a method is named for a child class object, the child class's method is executed. Polymorphism is shown where the required overridden Method is called based on the object type. Method overriding is depicted in figure 6.8.4.2 (a).

The super () function is used to gain access to a parent or sibling class's methods and properties.

Method overriding provides the ability to change the child class method's implementation, which is already defined in its superclass. If there are a superclass and method with the same name and the same number of arguments in a child class, then the child class method is overriding the parent class method.

Note: *When the Method is called with the parent class object, the parent class method is executed. When Method is called with child class object, Method of the child class is*

performed. The appropriate overridden Method is named based on the object type, which is an example of polymorphism.

The **super ()** function gives access to a parent or sibling class's methods and properties.

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

  def displayData(self):
    print('In parent class displayData method')
    print(self.name)
    print(self.age)

class Employee(Person):
  def __init__(self, name, age, id):
    super().__init__(name, age)# calling constructor of super class
    self.empId = id

  def displayData(self):
    print('In child class displayData method')
    print(self.name)
    print(self.age)
    print(self.empId)

#Person class object
person = Person('Anurag', 40)
person.displayData()
#Employee class object
emp = Employee('Anurag', 40, '027**')
emp.displayData()
```

Output:

```
In parent class displayData method
Anurag
40
In child class displayData method
Anurag
40
027**
```

Figure 6.8.4.2 (a) Demonstration of overriding

When the displayData() method is called for a Person class entity, the Person class's displayData() method is called. When the displayData() method is called on an Employee class object, the Employee class's displayData() method is called. As a result, the proper overridden Method is called depending on the object type, a Polymorphism example.

Overriding the parent class the kid class's method

You may use the following methods to call the superclass's overridden method:

1. Making use of ClassName.method (self), refer figure 6.8.4.2. (b)
2. Make use of super (), refer figure 6.8.4.2. (c)

Calling superclass method using class name

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

  def displayData(self):
    print('In parent class displayData method')
    print(self.name)
    print(self.age)

class Employee(Person):
  def __init__(self, name, age, id):
    # calling constructor of super class
    super().__init__(name, age)
    self.empId = id

  def displayData(self):
    print('In child class displayData method')
    Person.displayData(self) #Calling super class method using class name
    print(self.empId)

#Employee class object
emp = Employee('Anurag', 40, '027**')
emp.displayData()
```

Figure 6.8.4.2 (b) Calling superclass method using class name

Calling superclass method using super ()

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

    def displayData(self):
        print('In parent class displayData method')
        print(self.name)
        print(self.age)

class Employee(Person):
    def __init__(self, name, age, id):
        # calling constructor of super class
        super().__init__(name, age)
        self.empId = id

    def displayData(self):
        print('In child class displayData method')
        super().displayData() #Calling super class method using Super()
        print(self.empId)

#Employee class object
emp = Employee('Anurag', 40, '027**')
emp.displayData()
```

Output:

```
In parent class displayData method
Anurag
40
In child class displayData method
Anurag
40
027**
```

Figure 6.8.4.2. (c) Calling superclass method using super ()

6.8.4.3 Operator overloading

As the name implies, operator overloading refers to the potential to overload an operator to provide additional functions in addition to its primary purpose. The '+' operator, for example, is used with numbers to execute addition operations. However, when used for two strings in Python, the '+' operator concatenates the strings and merges two lists. Likely, the '+' operator in the str and list classes is overwhelmed to provide extra features, depicted in figure 6.8.4.3 (a).

```
#using + operator with integers to add them
print(9 + 9)
#using + operator with Strings to concatenate them
print('Aatif ' + 'Jamshed')
a = [1, 2, 3]
b = [4, 5, 6]
# using + operator with List to concatenate them
print(a + b)
```

```
18
Aatif Jamshed
[1, 2, 3, 4, 5, 6]
```

Figure 6.8.4.3. (a) Operator overloading

Need of Operator Overloading

Refer the figure 6.8.4.3 (b), the '+' operator operated for Strings and Lists because it was already overwhelmed to provide String and List features. What if you try to use a custom object with an operator? If you're going to use the '+' operator on your custom class objects, for example.
There is a class Point with two variables, x, and y, in the example. Two Point type objects are instantiated, and you attempt to merge their data (p1.x + p2.x) and (p1.y + p2.y) into a single object.

```
class addobject:
  def __init__(self, x, y):
    self.x = x
    self.y = y

p1 = addobject(1, 2)
p2 = addobject(3, 4)

print(p1+p2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-18-e0f719b7b634> in <module>()
      7 p2 = addobject(3, 4)
      8
----> 9 print(p1+p2)

TypeError: unsupported operand type(s) for +: 'addobject' and 'addobject'
```

Figure 6.8.4.3. (b) Demonstration.

Note: Python specifies methods for all operators implicitly to provide functionality for such operators. The special method __add__, for example, provides functionality for the '+' operator (). Internally, if the '+' operator is used, the __add__() Method is called to perform the procedure.

Magic or extraordinary methods in Python are internal methods that have utility for the operators. When matching operators are used, these unique methods are automatically invoked; if you want an operator to deal with custom objects, you must override the corresponding particular Method that offers that operator features as shown in figure 6.8.4.3. (b).

```
class addobject:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  #overriding magic method
  def __add__(self, other):
    return self.x + other.x, self.y + other.y
p1 = addobject(1, 2)
p2 = addobject(3, 4)

print(p1+p2)
```

```
(4, 6)
```

Figure 6.8.4.3. (c) Demonstration.

6.8.4.4.Special/Dunder/Magic methods

The methods prefixed with two underscores and suffixed with two underscores are known as magic methods in Python. In Python, these mystical approaches are known as Dunders (Double UNDERscores). Magical methods for some of the most critical operators are described in the figure 6.8.4.4 (a)

Operator	Special Method	Remarks
[+]	__add__	It is used as an Additive operator
[-]	__sub__	It is used as a Subtraction operator
[*]	__mul__	It is used as a Multiplication operator
[/]	__truediv__	It is used as Division with the fractional result
[%]	__mod__	It is used as the Remainder operator
[//]	__floordiv__	Division with integer result, discarding any fractional part
[**]	__pow__	Return a to the power b for a and b numbers.
[<]	__lt__	It is used as less than

[<=]	__le__	It is used as less than or equal to
[==]	__eq__	It is used as equal to
[!=]	__ne__	It is used as not equal to
[>]	__gt__	It is used as greater than
[>=]	__ge___	greater than or equal to

Figure 6.8.4.4 (a) Magic method for operators

__dict__

It is a special attribute. It actually represents any module's system table.

Syntax: object. __dict__

```
1  class MyClass(object):
2      class_var = 1
3
4      def __init__(self, i_var):
5          self.i_var = i_var
6
7  foo = MyClass(2)
8  bar = MyClass(3)
9
10 print(MyClass.__dict__)
11 print(foo.__dict__)
12 print(bar.__dict__)
```

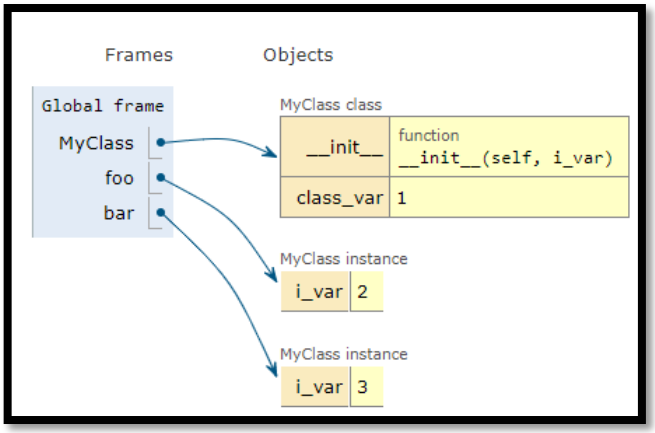


Figure 6.8.4.4 (b) __dict__

Output:

{'__module__': '__main__', 'class_var': 1, '__init__': <function MyClass.__init__ at 0x7f81774d26a8>, '__return__': None, '__dict__': <attribute '__dict__' of 'MyClass' objects>, '__weakref__': <attribute '__weakref__' of 'MyClass' objects>, '__doc__': None}

{'i_var': 2}

{'i_var': 3}
The figure 6.8.4.4 (b) shows __dict__ contains all the attributes of object.

__repr__

This is another special method. Which is used to represent object as a string in python. It is called by repr() function which is built in.

Syntax: object.__repr__(self)

A class, Person, is created with the __repr__ method. We can then call this method using the built-in Python function repr(). It would return the object representation as string, refer figure 6.8.4.4 (c). The **output** will be Person (John, 20).

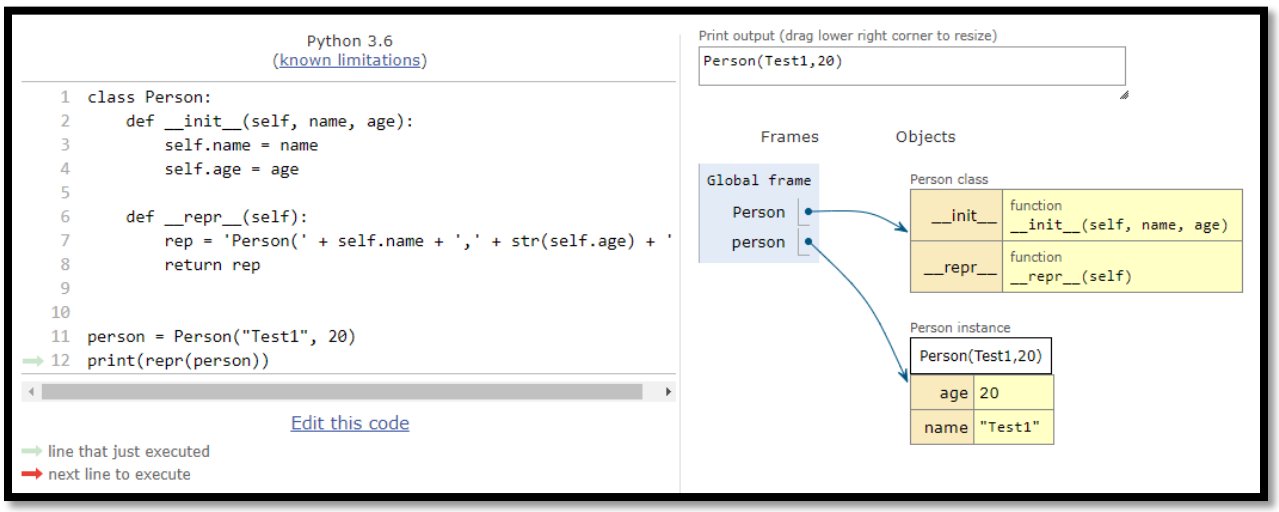


Figure 6.8.4.4 (c) __repr__

__str__

__str__ method in python represents Class objects as string. __str__ method is called when below functions are called on the object and return a string.

- print()
- str()

If we have not defined the __str__ , then it will call the __repr__ method. Refer figure 6.8.4.4 (d).

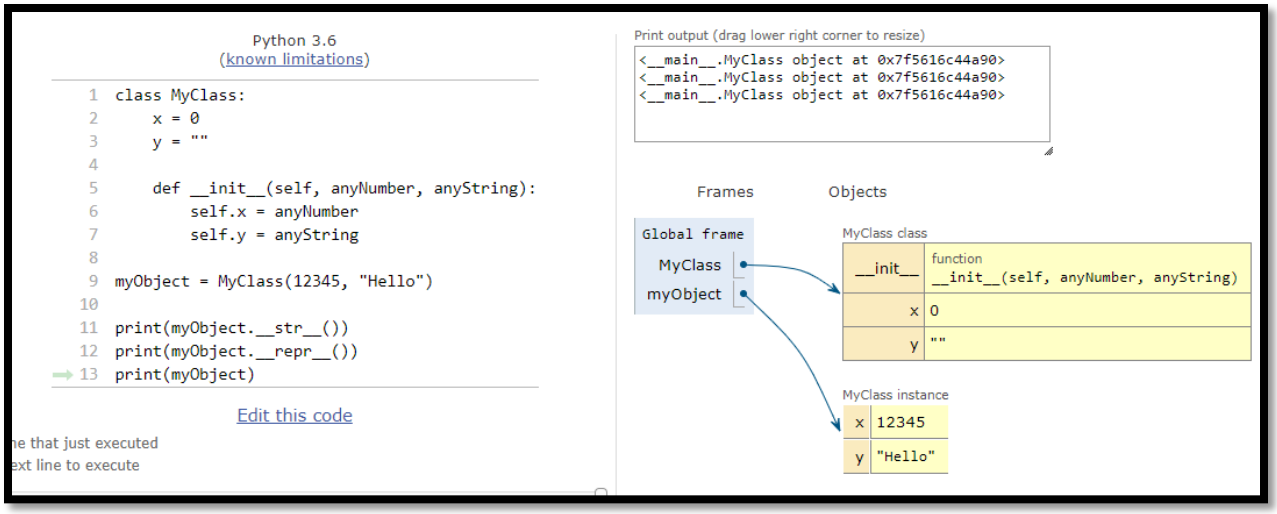


Figure 6.8.4.4 (d) __str__

6.9 Applications of OOPs

6.9.1 Stack and its implementation

Stack is a linear data structure which stores the similar type of elements in it. Some important points about stack are:

- Stack allows the addition of new elements and deletion of elements from the index called ***Top of Stack***.
- Stack works on the principle of Last in First Out, or First In Last Out; this is up to the indexing which a programmer is assuming.
- The addition of the element in the stack is called PUSH(), and the deletion of an element from the stack is known as POP(). The PUSH() and POP() operations and the Top of the stack (alias TOS or TOP) are shown in figure below.

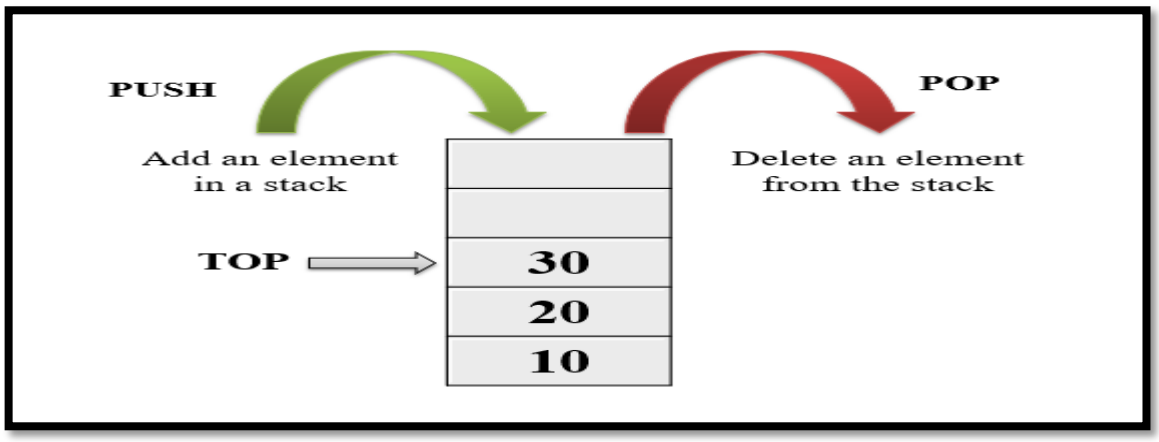


Figure 6.9.1(a): Stack and its operations

In the above-shown figure, firstly, element 10 was inserted into the empty stack. After that, 20 and 30 were added to the stack. The current position of TOP shows the Top of the stack.

6.9.1.1 Introduction to Stack

Based on the usage of the stack, the stack divided into two categories as shown in figure 6.8.5.1(b).

1. **Stack with fixed size or capacity**

Stack with a fixed number of locations to store the elements in it. Say the size of the stack is five. In this case, the stack can only be able to hold five elements in it. When the five elements are there in the stack, it will show the full stack notification. No new element can be inserted into the stack. This type of stack is also known as a Bounded buffer or bounded stack. In data structures and practical applications, we generally use a stack with a fixed size.

2. **Stack with an infinite size**

Stack without any fixed number of locations, where stack can store any number of elements in it. The size of the stack is not defined. When we are using the stack, its capacity is considered infinite. In this case, the stack can only be empty if no element is there in it. But, full-Stack notification is never seen in this type of stack. This type of stack is also known as Un-Bounded buffer or infinite size stack. Generally, for theoretical purposes, we consider the infinite size stack.

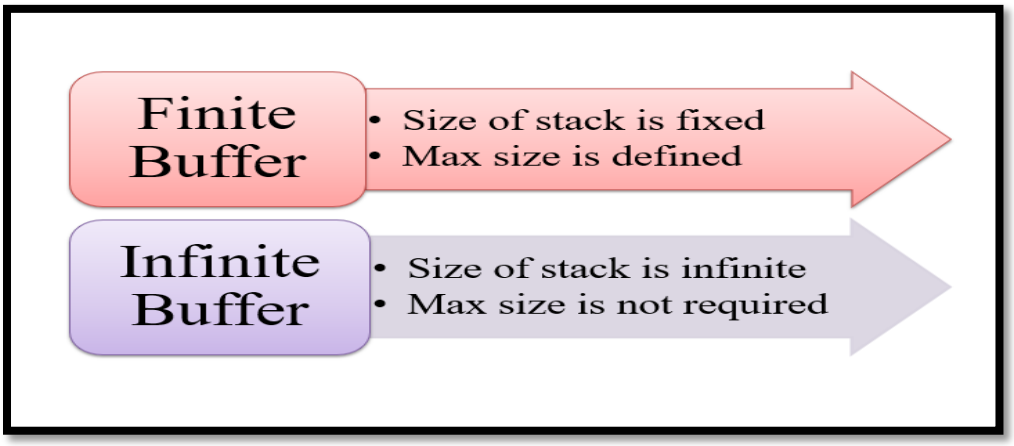


Figure 6.9.1(b): Finite buffer and infinite buffer Stack

6.9.1.2 Applications of the stack

Stack is used in computer systems at various locations. Some of the applications in terms of computer programming is mentioned in the list:

- To solve expressions and parse the grammar and language
- All the tokens of the languages are parsed using stack
- Expression Conversion (Infix to Postfix, Postfix to Prefix, etc.)
- In computer organization, a stack is used as memory and register stack

6.9.1.3 Stack has two main operations

- **PUSH ()** – This is an insertion method in the stack. Whenever an element is inserted into a stack, it is through the PUSH () operation.
- **POP ()** – This is a deletion method in the stack. Removal of an element from the stack is through the POP () operation.

To code, the behavior of the stack in python, functions associated with the stack need to understand first. The list of stack function is mentioned below and shown in figure 6.8.5.1(c)

- **isEmpty()** – Returns whether the stack is empty or not
- **isFull()** – Returns whether the stack is full or not
- **max size()** – Returns the size of the stack; limit
- **top()** – Returns a reference to the topmost element of the stack
- **push(x)** – Adds the element 'x' at the top of the stack
- **pop()** – Deletes the topmost element of the stack
- **peek()** – Returns the top most element of the stack

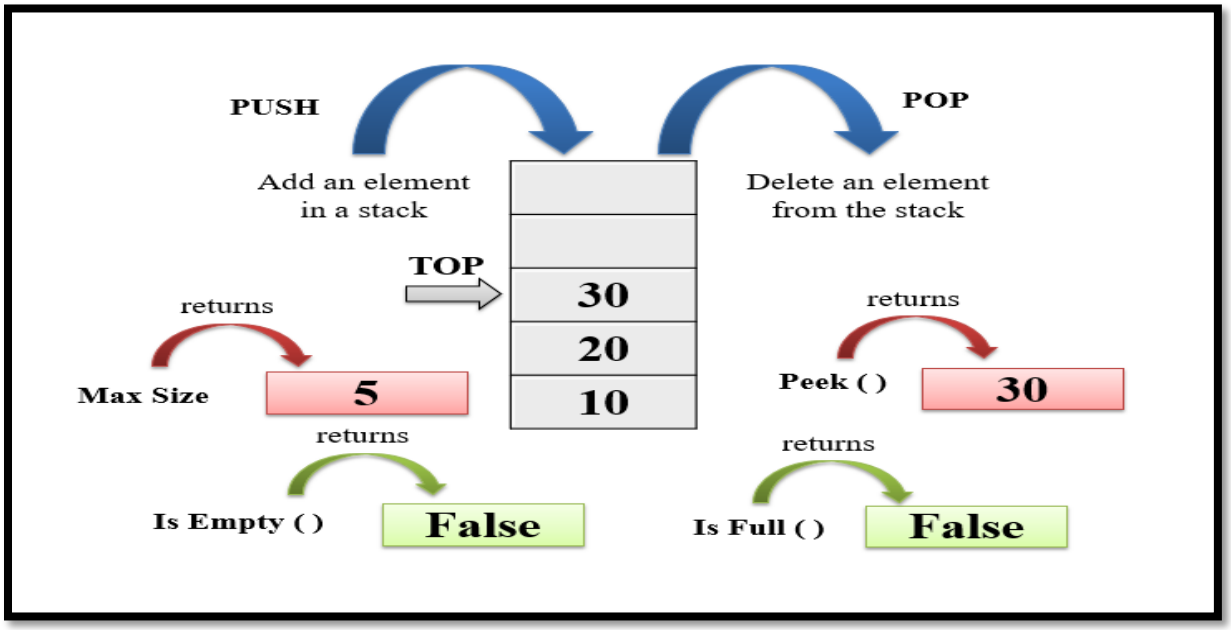


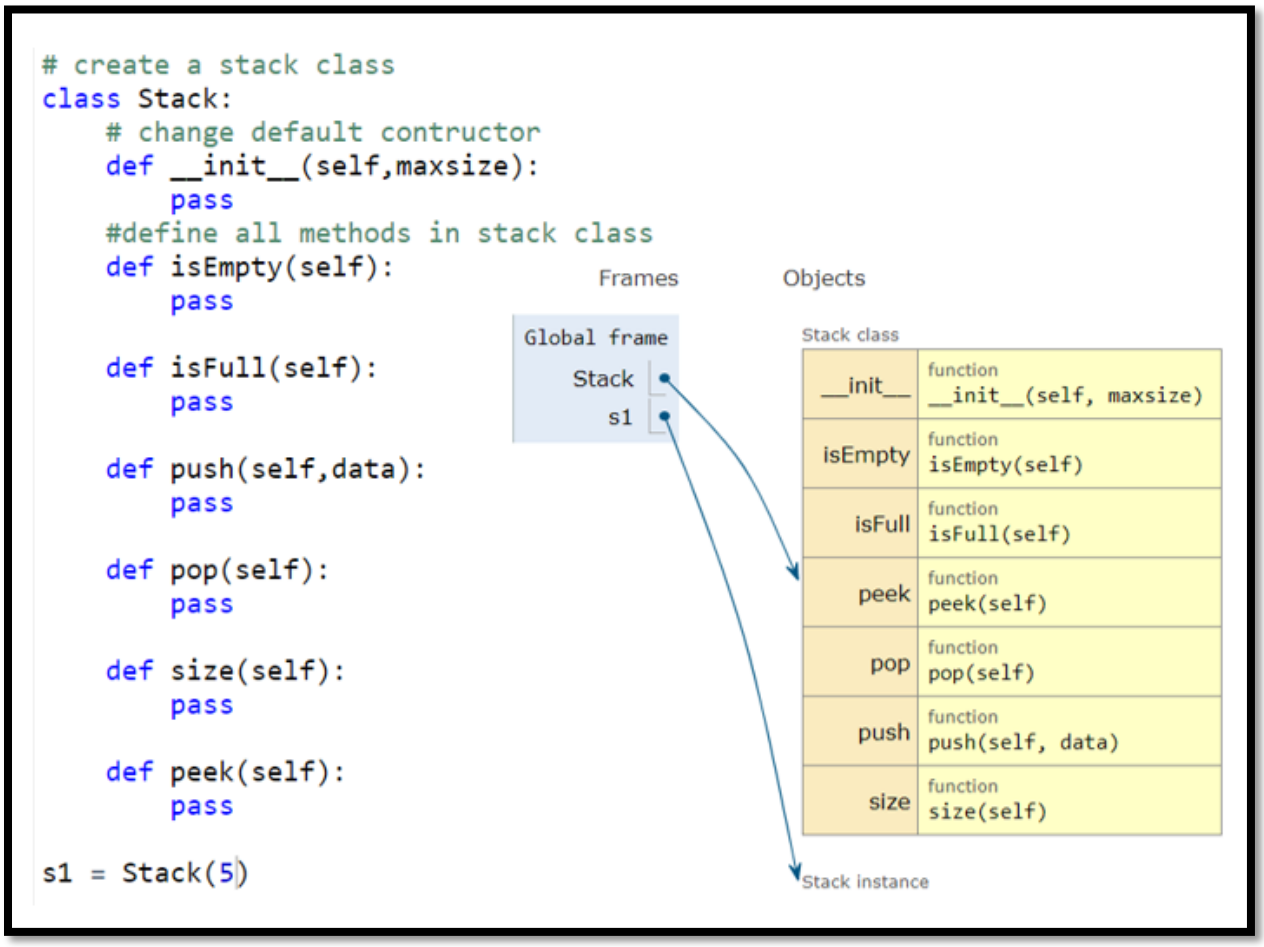
Figure 6.9.1(c): Stack operations and functions

6.9.1.4 Implementation of Stack

We will see the implementation of the finite stack. Step by step process is shown below.

Step 1: Create a class for stack and mention all the methods of the stack. Here,

- the stack approach is LIFO i.e. Last In First Out.
- Attributes used top and Maxsize
- Method / Behaviour of stack class:
    a. Push () - Add at top
    b. Pop ()- remove from top
    c. isEmpty()- Boolean, true when the stack is empty
    d. isFull() - Boolean, true when stack reaches at maxsize
    e. Size () Returns the size of the stack
    f. Peek () - Same as pop, but does not remove the element




```
# create a stack class
class Stack:
    # change default contructor
    def __init__(self,maxsize):
        pass
    #define all methods in stack class
    def isEmpty(self):
        pass

    def isFull(self):
        pass

    def push(self,data):
        pass

    def pop(self):
        pass

    def size(self):
        pass

    def peek(self):
        pass

s1 = Stack(5)
```

Figure 6.8.5.1(d): Stack class and its methods

Step 2: write code for all methods to define the working in stack.

After creating a class, first task is to write the code for constructor. This code decides the behavior of the instance of the respective class. Here, in stack, the **maxsize** is defined as an argument to pass while creating an abject of stack class. By default, value of TOP is –1. It can also be taken as zero, but the zero is an index value where the element can be stored. So, the **self.top** is taken as –1. m**axsize** is the input value at the time of object creation. As it is a fixed size stack, so the dummy list is created in the memory to show all elements of stack are None. The same code is shown in figure 6.8.5.1(e)

```
class Stack:
    def __init__(self,maxsize):
        self.top=-1
        self.maxSize=maxsize
        self.l = [None for i in range(maxsize)]
```

Figure 6.8.5.1(e): Stack class constructor code

Next method in the stack class is to check whether a stack is empty or not. For this, **isEmpty()** method is created. The code is shown in figure 6.8.5.1(f)

```
def isEmpty(self):
    if self.top==-1:
        return True
    else:
        return False
```

Figure 6.8.5.1(f): isEmpty() method

For the finite buffer or fixed size stack, the isFull() method is also required, which notifies the stack is whether full or not. The code of the method is shown in figure 6.8.5.1(g).

```
def isFull(self):
    if self.maxSize-1==self.top:
        return True
    else:
        return False
```

Figure 6.8.5.1(g): isFull() method

push () and pop() are the main operations of the stack. Here, push () method is called to add an element in the stack. While adding an element, the first thing is to check whether a stack is full or not. This condition is also checked in the method before adding an element in stack at TOP of stack. Figure 6.8.5.1(h) shows the code of push () method.

```
def push(self,data):
    if self.isFull():
        print("Stack is Full")
    else:
        self.top+=1
        self.l[self.top]=data
```

Figure 6.8.5.1(h): push () method

pop () method is called to delete or pop an element from the top of the stack. While deleting an element, check whether a stack is empty or not. This condition is called from the isEmpty() method and checked prior to delete an element from stack. Figure 6.8.5.1(i) shows the code of pop () method.

```
def pop(self):
    if self.isEmpty():
        print("Stack is Empty")
    else:
        print("item deleted from Stack : ", self.l[self.top])
        self.l[self.top]=None
        self.top-=1
```

Figure 6.8.5.1(i): pop () method

peek () method shows the top most element of the stack, where the top of stack is pointing. The code of peek () is shown in figure 6.8.5.1(j)

```
def peek(self):
    if self.isEmpty():
        print("stack empty")
    else:
        print("Top item is ", self.l[self.top])
```

Figure 6.8.5.1(j): peek() method

size () method shows the maxsize of the stack as shown in figure 6.8.5.1(k)

```
def size(self):
    print(self.maxSize)
```

Figure 6.8.5.1(k): size () method

Step 3: create an object of stack class, as s1 = stack (5) and perform the push(), pop() and peek() operations on stack object. Figure 6.8.5.1(l) shows the creation of s1 instance of stack class with max size 5.

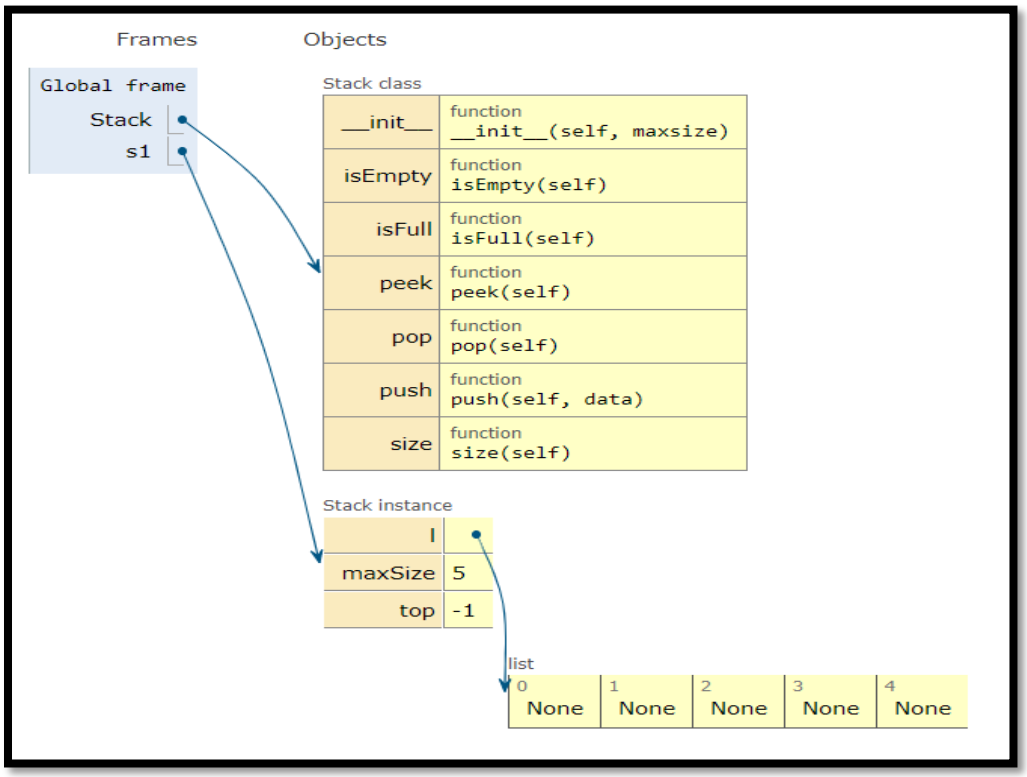


Figure 6.8.5.1(l): Creation of stack with size 5

Figure 6.8.5.1(m) shows the push () command to add integer value 10 in the s1 stack. Figure 6.8.5.1(n) shows the push () Stack operation as per the commands given to the python interpreter.

```
s1 = Stack()
s1.push(10)
```

Figure 6.8.5.1(m): push() operation to add 10 in stack

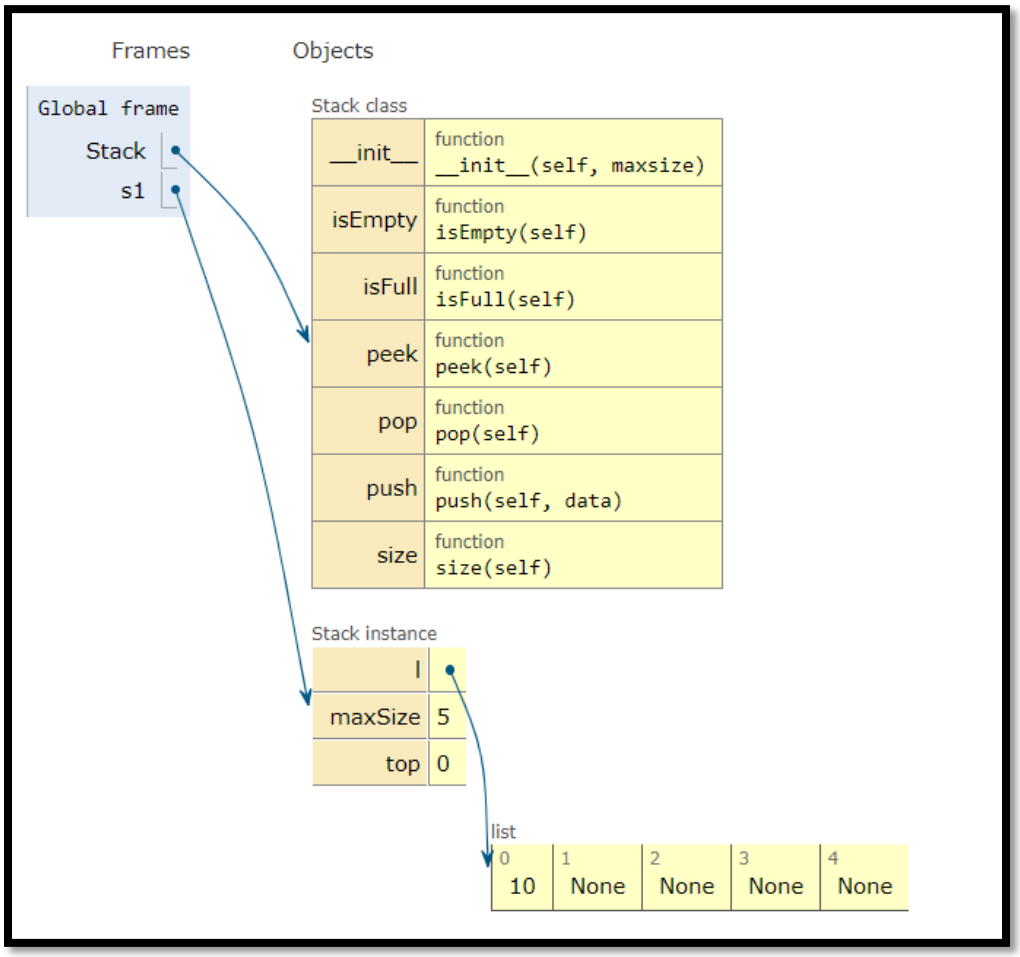


Figure 6.8.5.1(n): push () operation visualization to add 10 in stack

If we continuously add the elements in stack and stack reached to the maxsize, it will not push new elements and shows an Stack full notification on console. Figure 6.8.5.1(o) shows the code and figure 6.8.5.1(p) shows the visualized output of the code.

```
s1 = Stack(5)    # stack created
s1.push(10)      # already added in previous step
s1.push(20)      # added 20, now top is 1
s1.push(30)      # added 30, now top is 2
s1.push(40)      # added 40, now top is 3
s1.push(50)      # added 50, now top is 4
s1.push(60)      # stack full; no space for new items
```

Figure 6.8.5.1(o): Code to add elements in stack

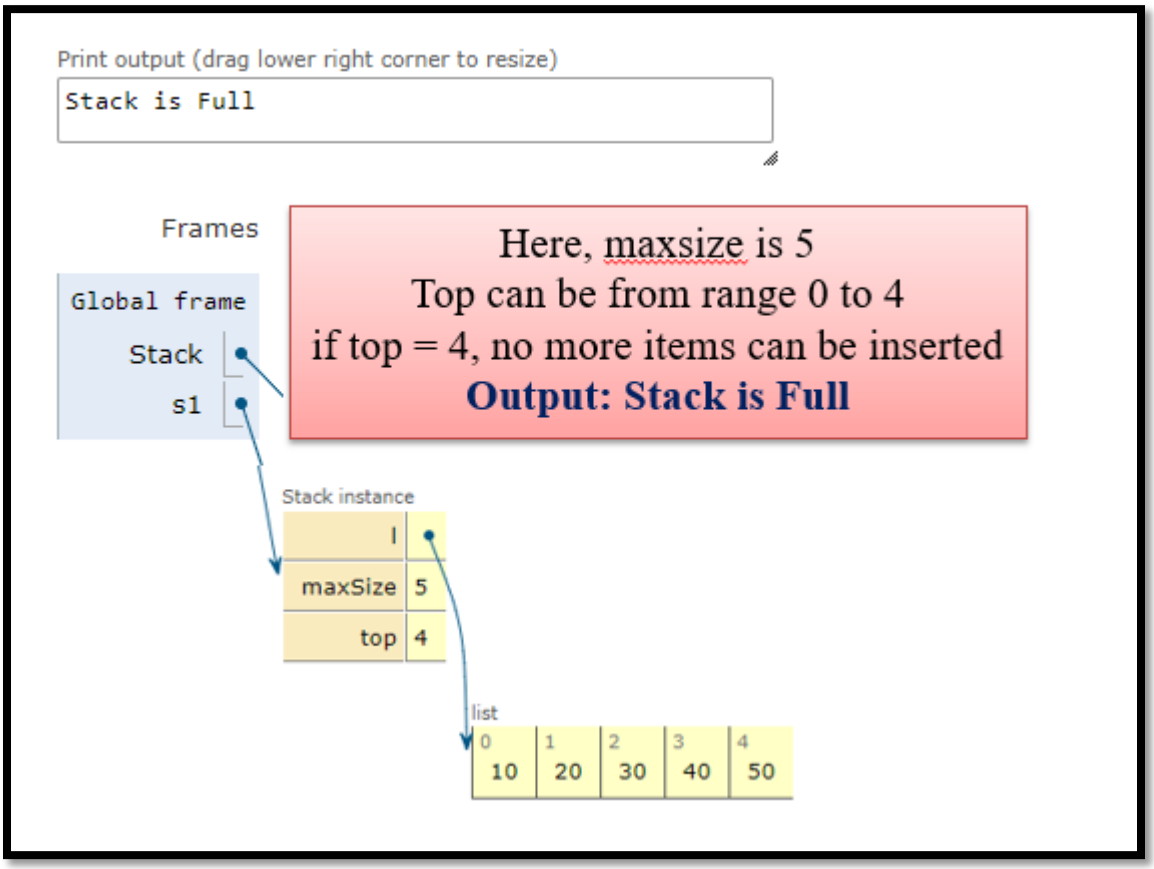


Figure 6.8.5.1(p): Push () operation when stack is full

Figure 6.8.5.1(q) shows the code for pop () and peek () method. Figure 6.8.5.1(r) shows the working of pop () command and peek () command of the respective code on the stack instance s1.

```
# to delete an element, call pop()
s1.pop()     # delete 50, now top is 3
s1.pop()     # delete 40, now top is 2

# peek() shows the top most element of stack
s1.peek()     # shows integer value 30
```

Figure 6.8.5.1(q): code of pop () and peek ()

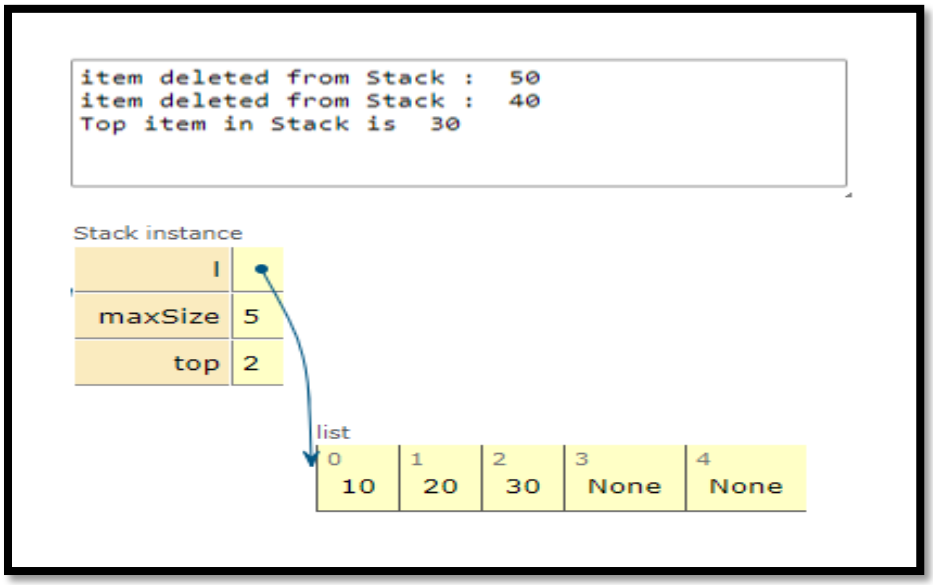


Figure 6.8.5.1(r): shows the working of pop () command and peek () command

If further pop () will be called three times then the stack will reach to the empty state. The notification of empty stack is displayed to the console as output. Figure 6.8.5.1(s) shows the code and figure **6.8.5.1**(t) shows the visualized output

```
s1.pop()     # delete 30, now top is 1
s1.pop()     # delete 20, now top is 0
s1.pop()     # delete 10, now top is -1
s1.pop()     # Stack is Empty, can't delete item
```

Figure 6.8.5.1(s): pop () command till stack is empty

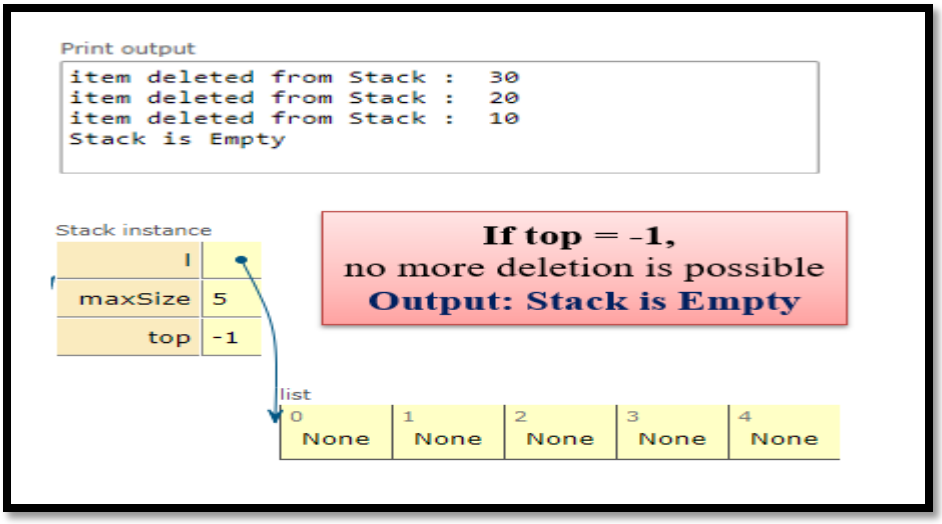


Figure 6.8.5.1(t) visualization of pop () command till stack is empty

6.9.2 Queue and its implementation

In the programming world, data structures are extremely important. They assist us with organizing our data so that it can be used effectively. The Queue is one of the most basic data structures.

6.9.2.1 Introduction to Queue

If you look at the line at the counter of movie theatre, you'll see that the second person will only go to the counter after the first person has finished his or her job. The first person arrives at the counter, followed by the second. This concept is called FIFO (First In/First Out), which is followed by Queue.

Figure 6.8.5.2(a): Illustrate the concept of Queue

6.9.2.1 Queue Operations

The following are the operations in queue:

enqueue(data/object): In the very end, it adds/insert new data to the queue. The backside is referred to as the rear end. *rear ()* : Returns the first element from the end of the queue.

dequeue () : Removes and returns the data from the queue's front. The front is the name given to the side. *front():* Returns the first element from the start of the queue.

IsEmpty: Check if the queue is empty

IsFull: Check if the queue is full

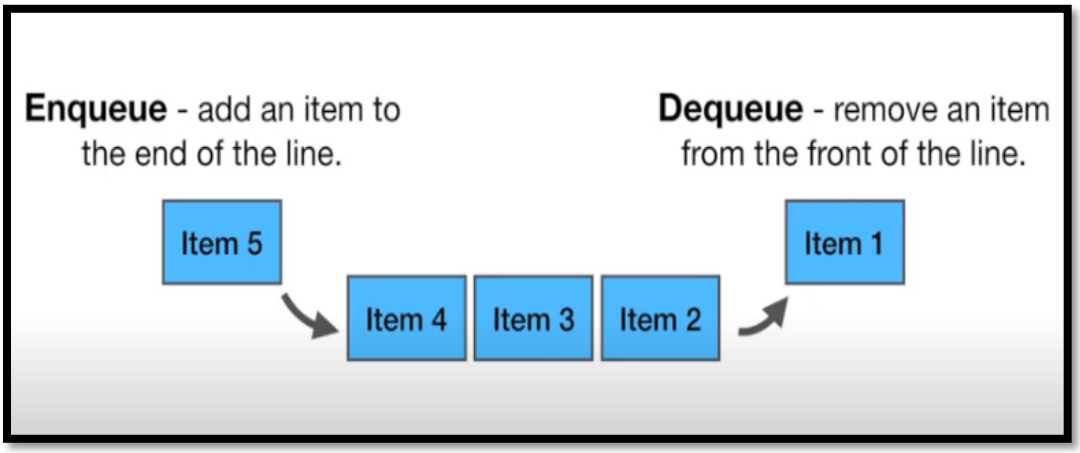


Figure 6.8.5.1(b): Operations of Enqueue and Dequeue

6.9.2.3 Implementation of Queue operations

Step1: Creation of class and Initialization of Queue

```
class Queue:
   # Initialization of Queue
   def __init__(self,maxsize):
      self.rear = None
      self.front = None
      self.maxsize = maxsize
      self.q = [None for _ in range(maxsize)]
```

Figure 6.8.5.1(c): Creation of class and Initialization of Queue

Step2: The method adds some information to the queue (elements). To connect data to the end of the queue.

```
# enqueue() - Put an item into the queue.
def enqueue(self,data):
    if self.full():
        print("Queue is full")
    # When first element added
    elif self.front==None and self.rear==None:
        self.front=self.rear=0
        self.q[self.rear]=data
    else:
        self.rear+=1
        self.q[self.rear]=data
```

Figure 6.8.5.1(d): Adds some elements to the queue

Step3: The method deletes the information from the queue (elements).

```
# dequeue() - Remove and return an item from the queue.
def dequeue(self):
    if self.empty():
        print("Queue is empty")
        return None
    elif self.front==self.rear:
        item = self.q[self.front]
        self.q[self.front]=None
        self.front=self.rear=None
    else:
        item = self.q[self.front]
        self.q[self.front]=None
        self.front+=1
    return item
```

Figure 6.8.5.1(e): Delete an element from the queue

Step4: Add supporting methods

```
# empty() function return True if queue is empty otherwise False
def empty(self):
    if self.rear == None and self.front == None:
        return True
    else:
        return False
# full () function return True when value of rear, reaches to maxsize-1
def full(self):
    if self.rear == self.maxsize-1:
        return True
    else:
        return False
```

Figure 6.8.5.1(e): Some supporting functions

Step5: Creation of object with inserting and deletion

```
q1 = Queue(5)
q1.enqueue(10)
q1.enqueue(160)
q1.enqueue(106)
q1.dequeue()
q1.dequeue()
```

Figure 6.8.5.1(f): Creation of object with inserting and deletion

S

Output of Insertion

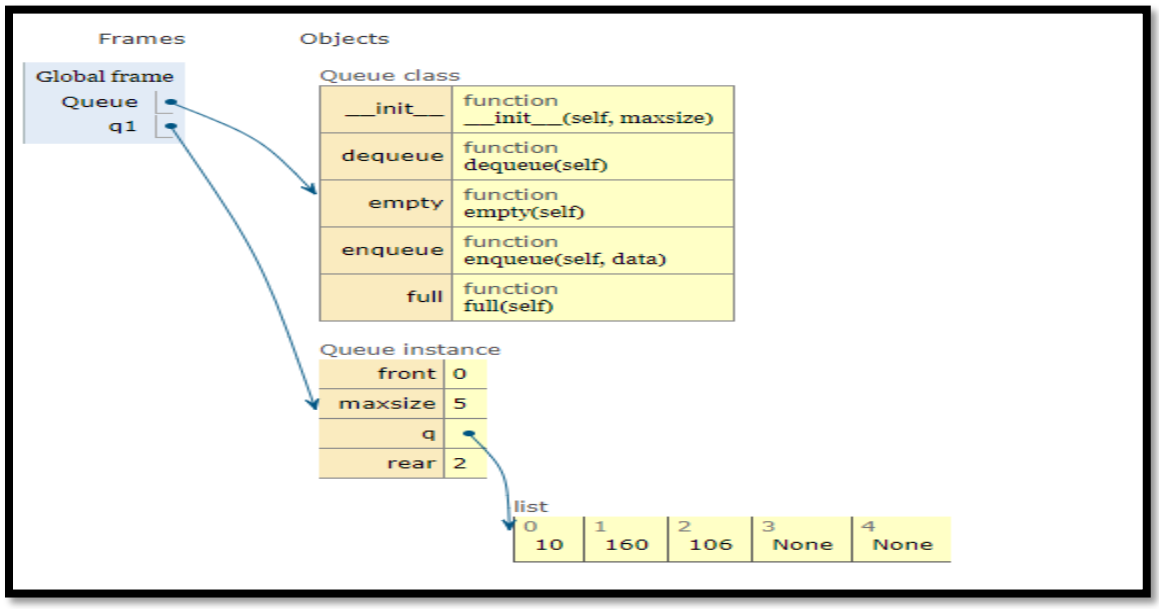


Figure 6.8.5.1(g): Visualization of insertion in queue

Output of deletion

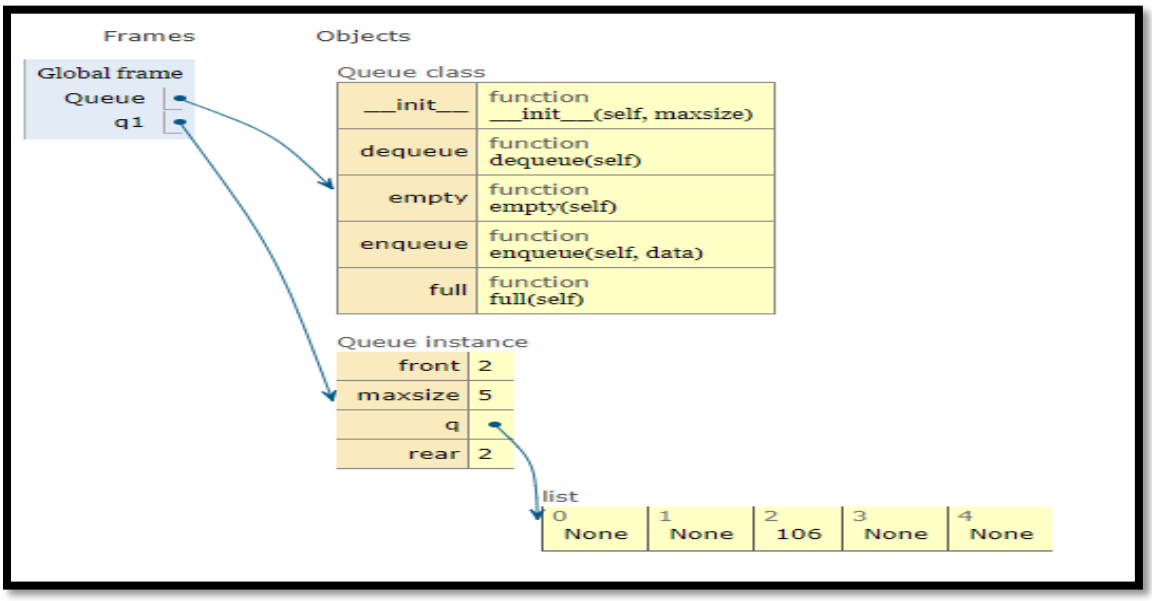


Figure 6.8.5.1(h): Visualization of deletion in queue

6.9.3 LinkedList and its implementation

6.9.3.1 Introduction to Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using reference as shown in the below image:

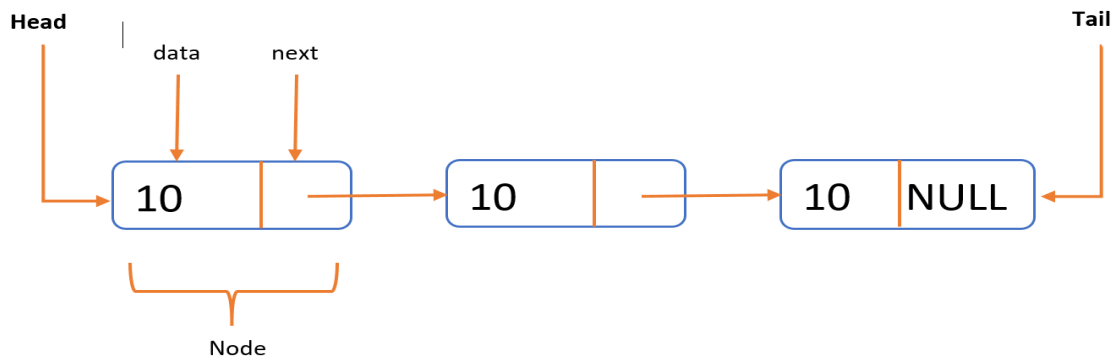


Figure 6.9.3.1:

Node - A simple type of node is one that only has a link to the next node and data. Next field is used to store reference of next adjacent node and data field contains data.

6.9.3.2 Implementation of Singly Liked List

Node Creation –

A node consists of two attributes data and next so we can define node using class

```
# Node Class
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

Figure 6.9.3.2(a):

The next attribute is initialized to None, while creating a new node and before adding to existing linked list.

6.9.3.3 Singly Linked List Attributes and Methods

A singly linked list can be defined as attributes and behaviour. Each singly linked list has following attributes and behaviour (Methods).

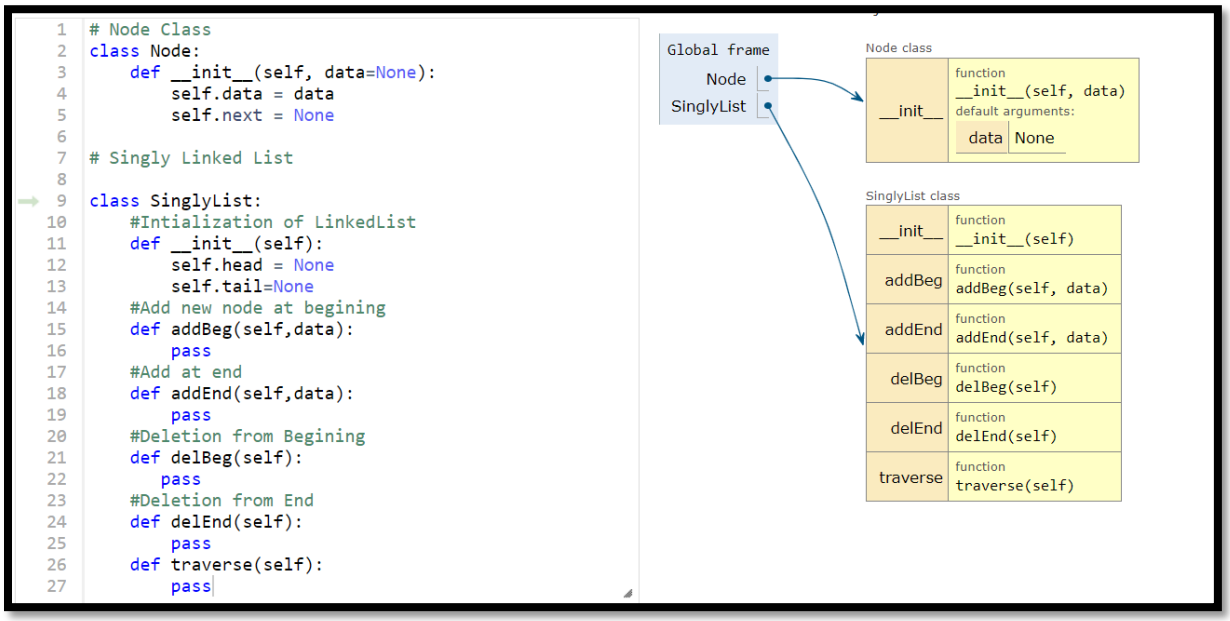
Attributes – Head and Tail

Head attribute is used to store reference of first node and

Tail attribute is used to store reference of last node.

Methods -

- **AddBeg () -** Add new node as first node in a given list
- **AddEnd ()** - Add new node as a last node in a given list
- **AddPos ()** - Add new node after or before given position in a linked list
- **DelBeg ()** - Always delete first node from a given list
- **DelEnd ()** - Always delete last node from a given list
- **DelPos ()** - Delete given pos node
- **Traverse ()** - go through all node from left to right
- And many more.

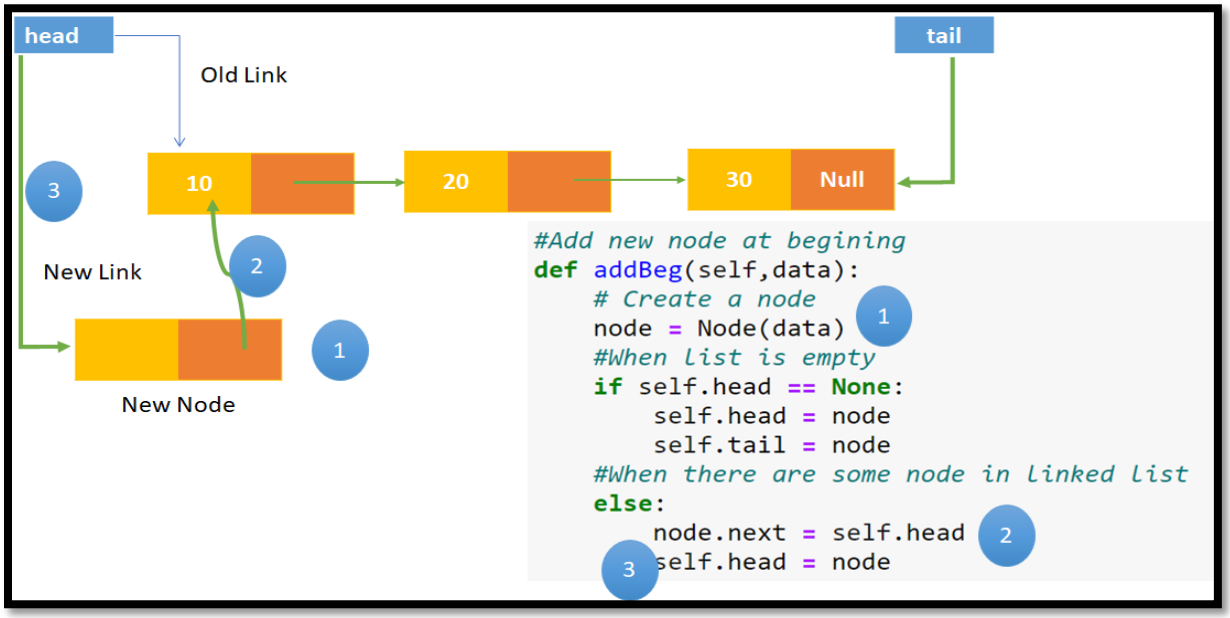


```
1  # Node Class
2  class Node:
3      def __init__(self, data=None):
4          self.data = data
5          self.next = None
6
7  # Singly Linked List
8
9  class SinglyList:
10     #Intialization of LinkedList
11     def __init__(self):
12         self.head = None
13         self.tail=None
14     #Add new node at begining
15     def addBeg(self,data):
16         pass
17     #Add at end
18     def addEnd(self,data):
19         pass
20     #Deletion from Begining
21     def delBeg(self):
22        pass
23     #Deletion from End
24     def delEnd(self):
25         pass
26     def traverse(self):
27         pass
```

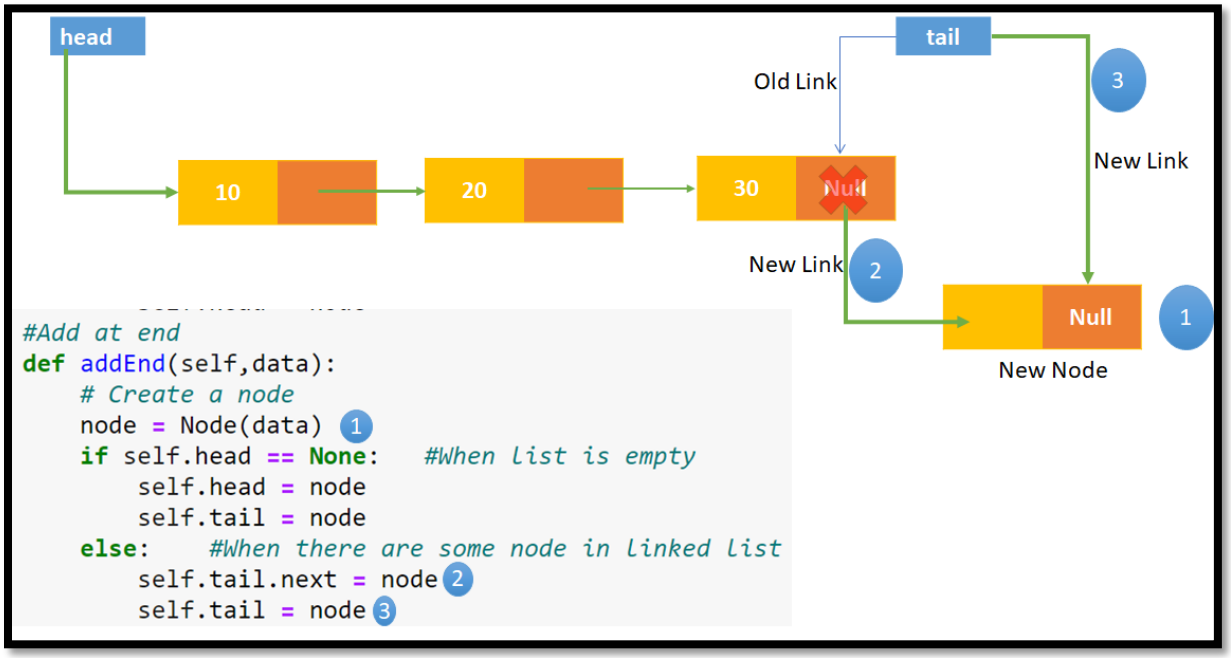

A basic structure of Singly list with Blank method definition.

In the above figure we initialize all attributes and methods are blank. Now let's define each method in details.

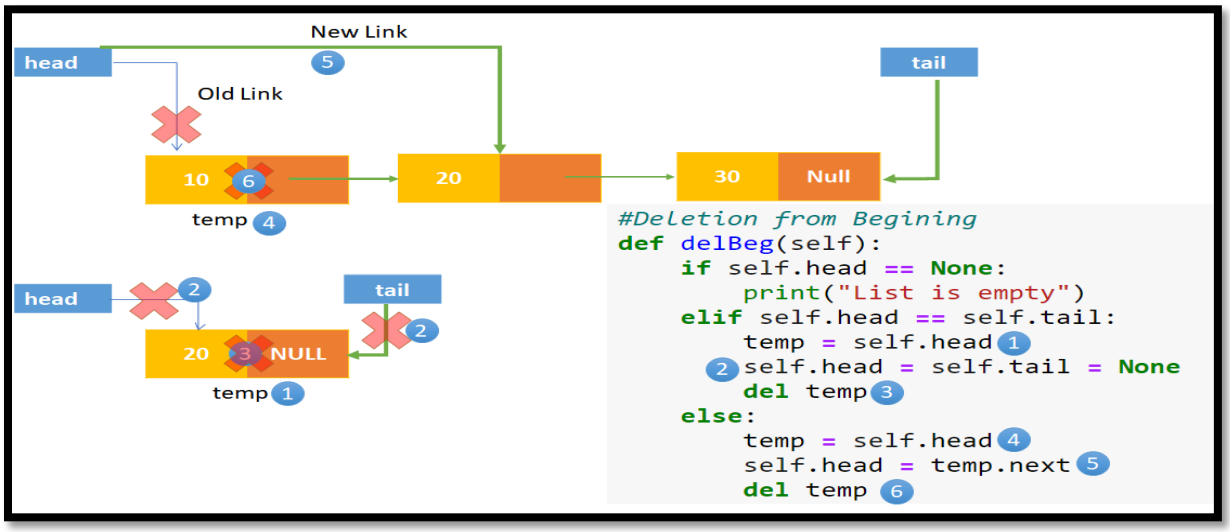
addBeg() Method



addEnd() Method

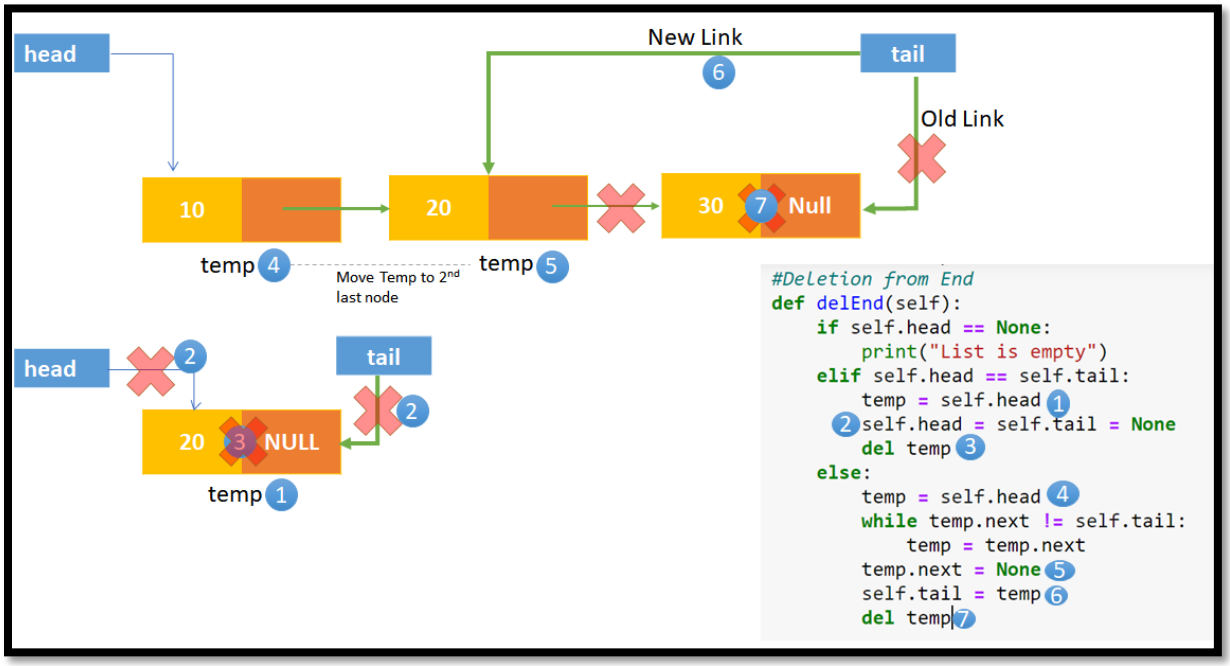


delBeg() Method




```
#Deletion from Begining
def delBeg(self):
    if self.head == None:
        print("List is empty")
    elif self.head == self.tail:
        temp = self.head 1
      2 self.head = self.tail = None
        del temp 3
    else:
        temp = self.head 4
        self.head = temp.next 5
        del temp 6
```

delEnd() Method




```
#Deletion from End
def delEnd(self):
    if self.head == None:
        print("List is empty")
    elif self.head == self.tail:
        temp = self.head 1
      2 self.head = self.tail = None
        del temp 3
    else:
        temp = self.head 4
        while temp.next != self.tail:
            temp = temp.next
        temp.next = None 5
        self.tail = temp 6
        del temp 7
```

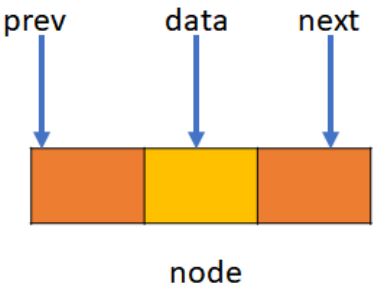
traverse() Method

```
def traverse(self):
    temp = self.head # temp point to first node
    while temp != None:    # Loop until temp reaches last node
        if temp.next != None:
            print(temp.data,end = "->")
        else:
            print(temp.data)
        temp = temp.next     # Move temp to next node
```

6.9.3.4 Implementation of Doubly Liked List

Node Creation:

A node consists of three attributes, data, next and prev. Date attributes hold info, next attributes hold reference of next node and prev attributes hold reference of previous node.




```
# Node class for doubly Linked List
class node:
    def __init__(self,data=None):
        self.data = data
        self.next = None
        self.prev = None
```

Node Class definition

6.9.3.5 Doubly Linked List Attributes and Methods

A doubly linked list can be defined as attributes and behaviour. Each doubly linked list has following attributes and behaviour (Methods).

Attributes – Head and Tail

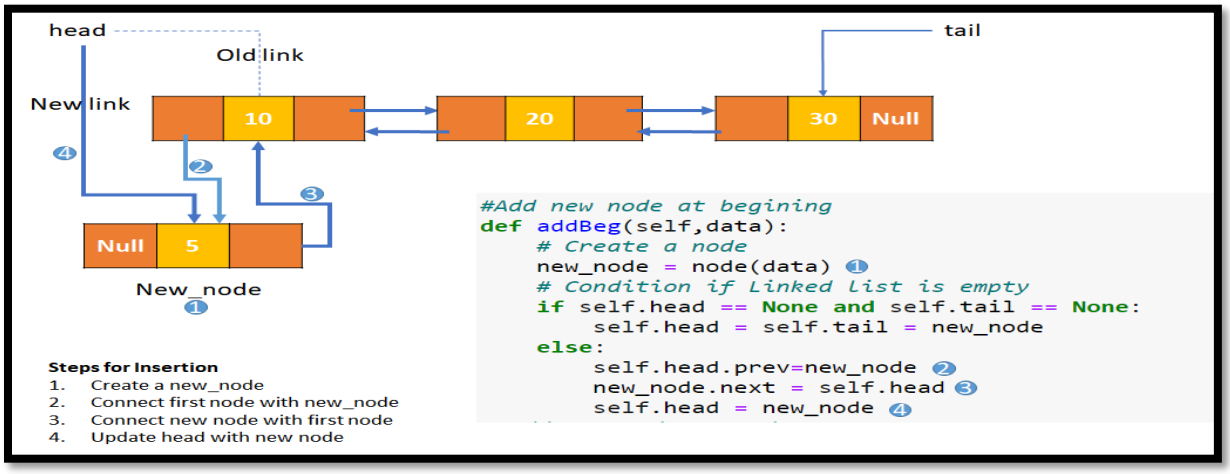
Head attribute is used to store reference of first node and **Tail** attribute is used to store reference of last node.

Methods -

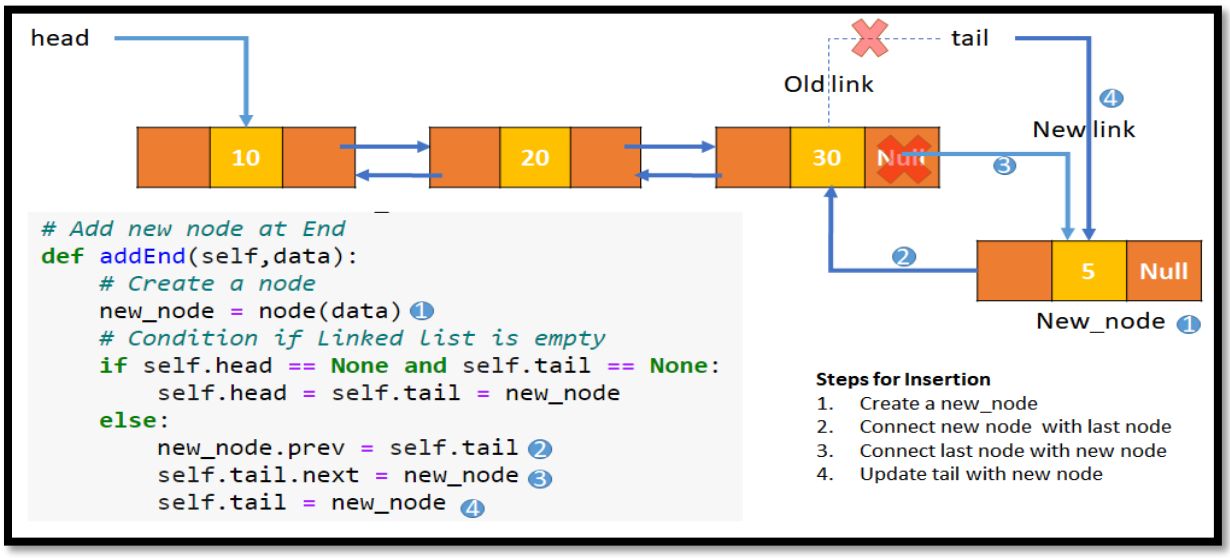
- **AddBeg () -** Add new node as first node in a given list
- **AddEnd ()** - Add new node as a last node in a given list
- **AddPos ()** - Add new node after or before given position in a linked list
- **DelBeg ()** - Always delete first node from a given list
- **DelEnd ()** - Always delete last node from a given list
- **DelPos ()** - Delete given pos node
- **Traverse ()** - go through all node from left to right
- And many more.

```
# Class definition of doubly Linked list
class doublyLinkedList:
    #Intialization of LinkedList
    def __init__(self):
        self.head = None
        self.tail = None

    #Add new node at begining
    def addBeg(self,data):
        pass

    # Add new node at End
    def addEnd(self,data):
        pass

    # Delete node from beg
    def delBeg(self):
        pass

    # Delete node from end
    def delEnd(self):
        pass

     # Traversing
    def traverse(self):
        pass
```

addBeg() Method -

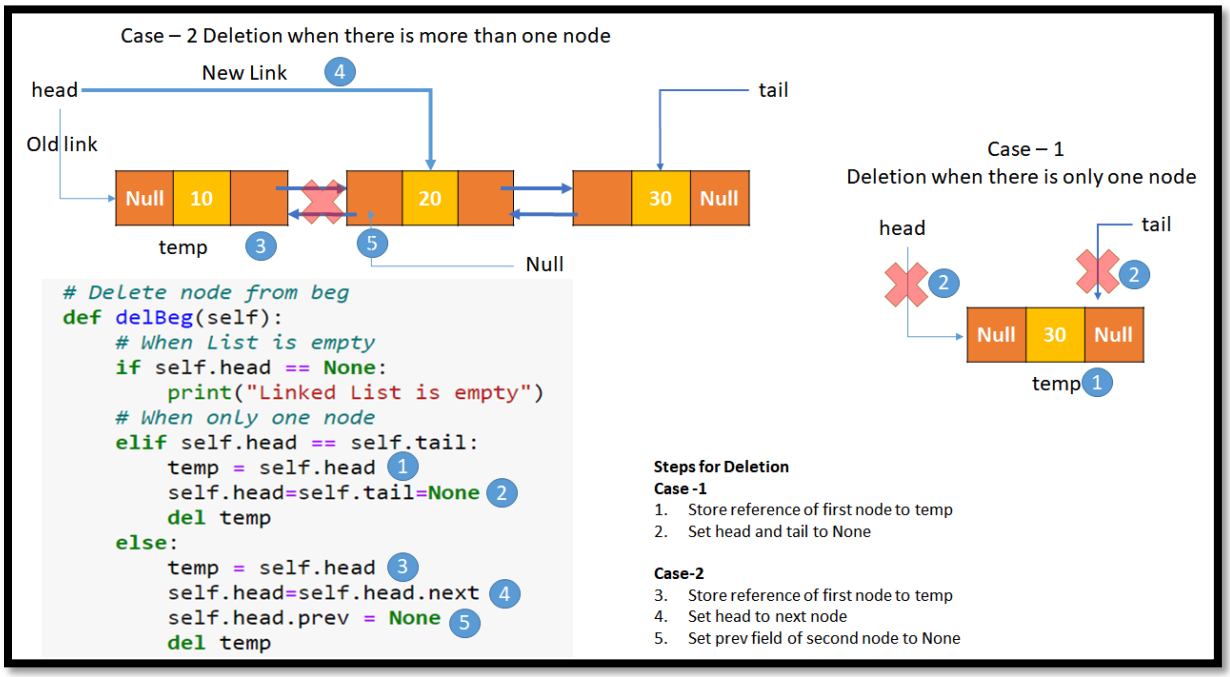


addEnd() Method -



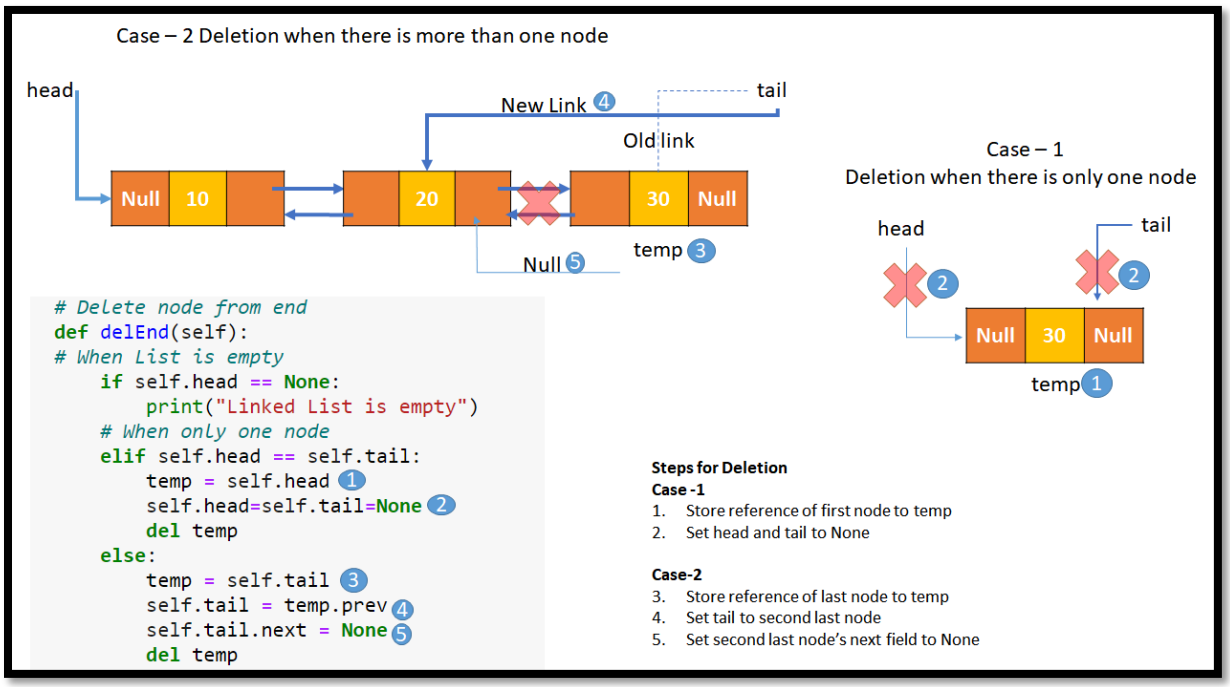

```
# Add new node at End
def addEnd(self,data):
    # Create a node
    new_node = node(data) 1
    # Condition if Linked List is empty
    if self.head == None and self.tail == None:
        self.head = self.tail = new_node
    else:
        new_node.prev = self.tail 2
        self.tail.next = new_node 3
        self.tail = new_node 4
```

delBeg() Method-




```
# Delete node from beg
def delBeg(self):
    # When List is empty
    if self.head == None:
        print("Linked List is empty")
    # When only one node
    elif self.head == self.tail:
        temp = self.head 1
        self.head=self.tail=None 2
        del temp
    else:
        temp = self.head 3
        self.head=self.head.next 4
        self.head.prev = None 5
        del temp
```

delEnd() Method() -




```
# Delete node from end
def delEnd(self):
# When List is empty
    if self.head == None:
        print("Linked List is empty")
    # When only one node
    elif self.head == self.tail:
        temp = self.head 1
        self.head=self.tail=None 2
        del temp
    else:
        temp = self.tail 3
        self.tail = temp.prev 4
        self.tail.next = None 5
        del temp
```

Traverse () Method -

```
 # Traversing
def traverse(self):
    temp = self.head
    while temp!=None:
        if temp.next!=None:
            print(temp.data,end="->")
        else:
            print(temp.data)
        temp=temp.next
```

References:

1. Head-First Python, 2nd edition by Paul Barry (O'Reilly, 2016).
2. Python Crash Course, 2nd Edition: A Hands-On, Project-Based Introduction to Programming Paperback – 3 May 2019.
3. https://www.oreilly.com/library/view/learning-python-5th/9781449355722/
4. https://docs.python.org/3/distributing/index.html.
5. PCEP (Certified Entry-Level Python Programmer) - https://pythoninstitute.org/certification/pcep-certification-entry-level/
6. PCAP (Certified Associate in Python Programming) - https://pythoninstitute.org/certification/pcap-certification-associate/
7. https://python-course.eu/
8. Python Data Structures | Coursera University of Michigan - Python for Data Structure https://www.coursera.org/learn/python-
9. https://www.udemy.com/course/complete-python-bootcamp
10. https://www.geeksforgeeks.org/
11. https://infytq.infosys.com/toc/lex_auth_012540961624342528106 1 (Programming Fundamentals Using Python).
12. https://www.askpython.com/python/oops
13. https://www.netjstech.com/
14. https://techvidvan.com/tutorials/category/python/