# Object Oriented Programming

| Index | Object-oriented Programming | Procedural Programming |
|---|---|---|
| 1. | Object-oriented programming is the problem-solving approach and used where computation is done by using objects. | Procedural programming uses a list of instructions to do computation step by step. |
| 2. | It makes the development and maintenance easier. | In procedural programming, It is not easy to maintain the codes when the project becomes lengthy. |
| 3. | It simulates the real world entity. So real-world problems can be easily solved through oops. | It doesn't simulate the real world. It works on step by step instructions divided into small parts called functions. |
| 4. | It provides data hiding. So it is more secure than procedural languages. You cannot access private data from anywhere. | Procedural language doesn't provide any proper way for data binding, so it is less secure. |
| 5. | Example of object-oriented programming languages is C++, Java, .Net, Python, C#, etc. | Example of procedural languages are: C, Fortran, Pascal, VB etc. |

Python is a versatile programming language that supports various programming styles, including object-oriented programming (OOP) through the use of **objects** and **classes**.

An object is any entity that has **attributes** and **behaviors**. For example, a `parrot` is an object. It has

- **attributes** - name, age, color, etc.
- **behavior** - dancing, singing, etc.

Similarly, a **class** is a blueprint for that object.

An object is simply a collection of data (variables) and methods (functions). Similarly, a class is a blueprint for that object.

# Python Classes

A class is considered as a blueprint of objects. We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

Some points on Python class:

- Classes are created by keyword **class**.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

# Define Python Class

We use the `class` keyword to create a class in Python. For example,

```
class ClassName:
    # class definition
```

Here, we have created a class named `ClassName`.

Let's see an example,

```
class Bike:
    name = ""
    gear = 0
```

Here,

- `Bike` - the name of the class
- `name/gear` - variables inside the class with default values `""` and **0** respectively

**Note**: The variables inside a class are called attributes.

# Python Objects

An object is called an instance of a class. For example, suppose `Bike` is a class then we can create objects like `bike1`, `bike2`, etc from the class.

Here's the syntax to create an object.

```
objectName = ClassName()
```

Let's see an example,

```
# create class
class Bike:
    name = ""
    gear = 0

# create objects of class
bike1 = Bike()
```

Here, `bike1` is the object of the class. Now, we can use this object to access the class attributes.

# Access Class Attributes Using Objects

We use the . notation to access the attributes of a class. For example,

```
# modify the name attribute
bike1.name = "Mountain Bike"

# access the gear attribute
bike1.gear
```

Here, we have used `bike1.name` and `bike1.gear` to change and access the value of `name` and `gear` attribute respectively.

```
# define a class
class Bike:
    name = ""
    gear = 0

# create object of class
bike1 = Bike()

# access attributes and assign new values
bike1.gear = 11
bike1.name = "Mountain Bike"

print(f"Name: {bike1.name}, Gears: {bike1.gear} ")
```

In the above example, we have defined the class named `Bike` with two attributes: `name` and `gear`.

We have also created an object `bike1` of the class `Bike`.

Finally, we have accessed and modified the attributes of an object using the . notation.

# Create Multiple Objects of Python Class

We can also create multiple objects from a single class. For example,

```
# define a class
class Employee:
    # define an attribute
    employee_id = 0

# create two objects of the Employee class
employee1 = Employee()
employee2 = Employee()

# access attributes using employee1
employee1.employeeID = 1001
print(f"Employee ID: {employee1.employeeID}")

# access attributes using employee2
employee2.employeeID = 1002
print(f"Employee ID: {employee2.employeeID}")
```

# Python Methods

We can also define a function inside a Python class. A Python Function defined inside a class is called a method. Let's see an example,

```
# create a class
class Room:
    length = 0.0
    breadth = 0.0

    # method to calculate area
    def calculate_area(self):
        print("Area of Room =", self.length * self.breadth)

# create object of Room class
study_room = Room()

# assign values to all the attributes
study_room.length = 42.5
study_room.breadth = 30.8

# access method inside class
study_room.calculate_area()
```

In the above example, we have created a class named `Room` with:

- Attributes: `length` and `breadth`
- **Method**: calculate_area()

Here, we have created an object named `study_room` from the `Room` class. We then used the object to assign values to attributes: `length` and `breadth`.

Notice that we have also used the object to call the method inside the class,
**Note:** The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

# Python Constructors

Earlier we assigned a default value to a class attribute,

```
class Bike:
    name = ""
...
# create object
bike1 = Bike()
```

However, we can also initialize values using the constructors. For example,

```
class Bike:

    # constructor function
    def __init__(self, name = ""):
        self.name = name

bike1 = Bike()
```

Here, `__init__()` is the constructor function that is called whenever a new object of that class is instantiated.

The constructor above initializes the value of the `name` attribute. We have used the `self.name` to refer to the `name` attribute of the `bike1` object.

If we use a constructor to initialize values inside a class, we need to pass the corresponding value during the object creation of the class.

```
bike1 = Bike("Mountain Bike")
```

Here, `"Mountain Bike"` is passed to the `name` parameter of `__init__()`.

```python
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

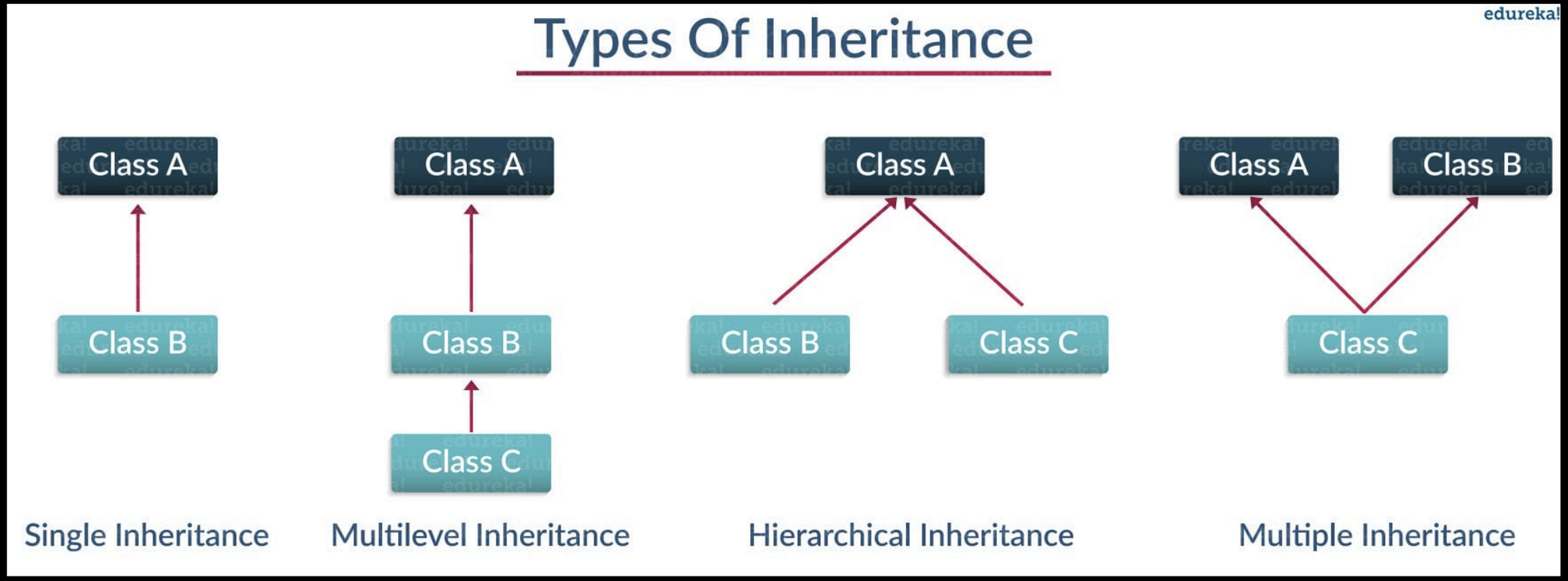
Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog.

# Python Inheritance

Inheritance is a way of creating a new class for using details of an existing class without modifying it.

The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).



## Python Inheritance Syntax

```
# define a superclass
class super_class:
    # attributes and method definition

# inheritance
class sub_class(super_class):
    # attributes and method of super_class
    # attributes and method of sub_class
```

Here, we are inheriting the `sub_class` class from the `super_class` class.

## Create a Parent Class

Any class can be a parent class, so the syntax is the same as creating any other class:

Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

## Create a Child Class

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

Create a class named **Student**, which will inherit the properties and methods from the **Person** class:

```
class Student(Person):
  pass
```

**Note:** Use the `pass` keyword when you do not want to add any other properties or methods to the class.

Now the Student class has the same properties and methods as the Person class.

Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

# Add the __init__() Function

So far we have created a child class that inherits the properties and methods from its parent.

We want to add the `__init__()` function to the child class (instead of the `pass` keyword).

**Note:** The `__init__()` function is called automatically every time the class is being used to create a new object.

Add the `__init__()` function to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname):
    #add properties etc.
```

When you add the `__init__()` function, the child class will no longer inherit the parent's `__init__()` function.

**Note:** The child's `__init__()` function **overrides** the inheritance of the parent's `__init__()` function.

To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the __init__() function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function

## Use the super() Function

Python also has a `super()` function that will make the child class inherit all the methods and properties from its parent:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

By using the `super()` function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

## Add Properties

Add a **year** parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

# Add Methods

Add a method called `welcome` to the `Student` class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of",
self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be **overridden**.

```
# base class
class Animal:

    def eat(self):
        print( "I can eat!")

    def sleep(self):
        print("I can sleep!")

# derived class
class Dog(Animal):

    def bark(self):
        print("I can bark! Woof woof!!")

# Create object of the Dog class
dog1 = Dog()

# Calling members of the base class
dog1.eat()
dog1.sleep()

# Calling member of the derived class
dog1.bark();
```

Here, `dog1` (the object of derived class `Dog`) can access members of the base class Animal. It's because `Dog` is inherited from `Animal`.

```
# Calling members of the Animal class
dog1.eat()
dog1.sleep()
```

# Method Overriding in Python Inheritance

However, what if the same method is present in both the superclass and subclass?

In this case, the method in the subclass overrides the method in the superclass. This concept is known as method overriding in Python.

```
class Animal:
    # attributes and method of the parent class
    name = ""

    def eat(self):
        print("I can eat")

# inherit from Animal
class Dog(Animal):

    # override eat() method
    def eat(self):
        print("I like to eat bones")

# create an object of the subclass
labrador = Dog()

# call the eat() method on the labrador object
labrador.eat()
```

## Uses of Inheritance

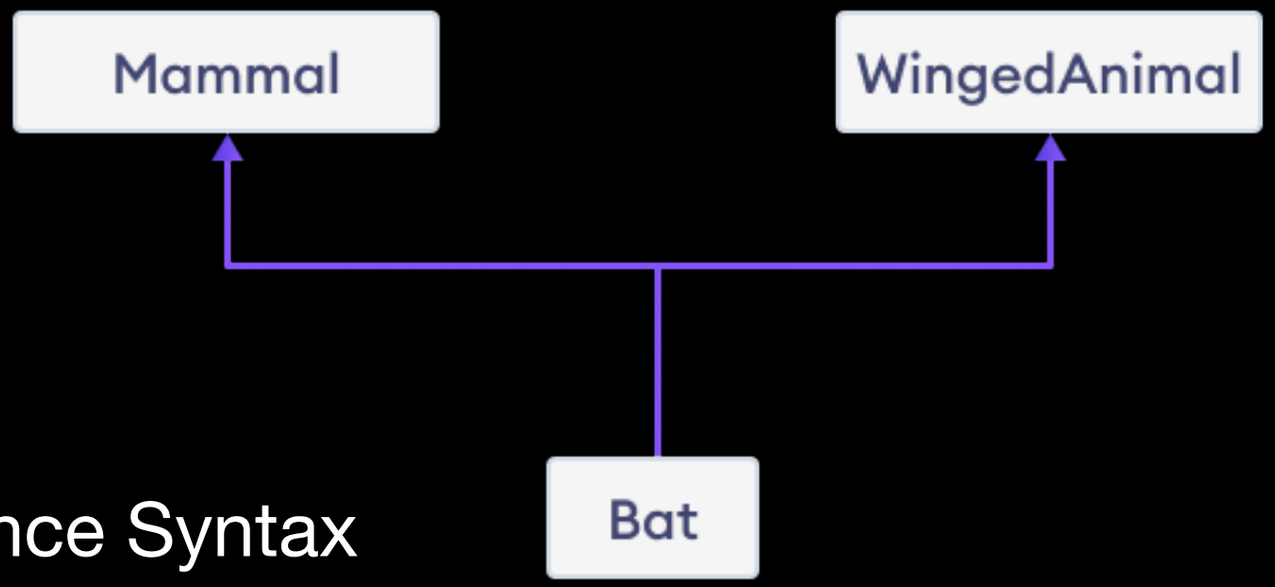
1. Since a child class can inherit all the functionalities of the parent's class, this allows code reusability.

2. Once a functionality is developed, you can simply inherit it. No need to reinvent the wheel. This allows for cleaner code and easier to maintain.

3. Since you can also add your own functionalities in the child class, you can inherit only the useful functionalities and define other required features.

# Multiple Inheritance

For example, A class `Bat` is derived from superclasses `Mammal` and `WingedAnimal`. It makes sense because bat is a mammal as well as a winged animal.




## Multiple Inheritance Syntax

```
class SuperClass1:
    # features of SuperClass1

class SuperClass2:
    # features of SuperClass2

class MultiDerived(SuperClass1, SuperClass2):
    # features of SuperClass1 + SuperClass2 + MultiDerived class
```

```python
class Mammal:
    def mammal_info(self):
        print("Mammals can give direct birth.")

class WingedAnimal:
    def winged_animal_info(self):
        print("Winged animals can flap.")

class Bat(Mammal, WingedAnimal):
    pass

# create an object of Bat class
b1 = Bat()

b1.mammal_info()
b1.winged_animal_info()
```

In the above example, the `Bat` class is derived from two super classes: `Mammal` and `WingedAnimal`. Notice the statements,

Here, we are using `b1` (object of `Bat`) to access `mammal_info()` and `winged_animal_info()` methods of the `Mammal` and the `WingedAnimal` class respectively.

# Polymorphism

The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

Polymorphism in addition operator

We know that the + operator is used extensively in Python programs. But, it does not have a single usage.

For integer data types, + operator is used to perform arithmetic addition operation.

```python
num1 = 1
num2 = 2
print(num1+num2)
```

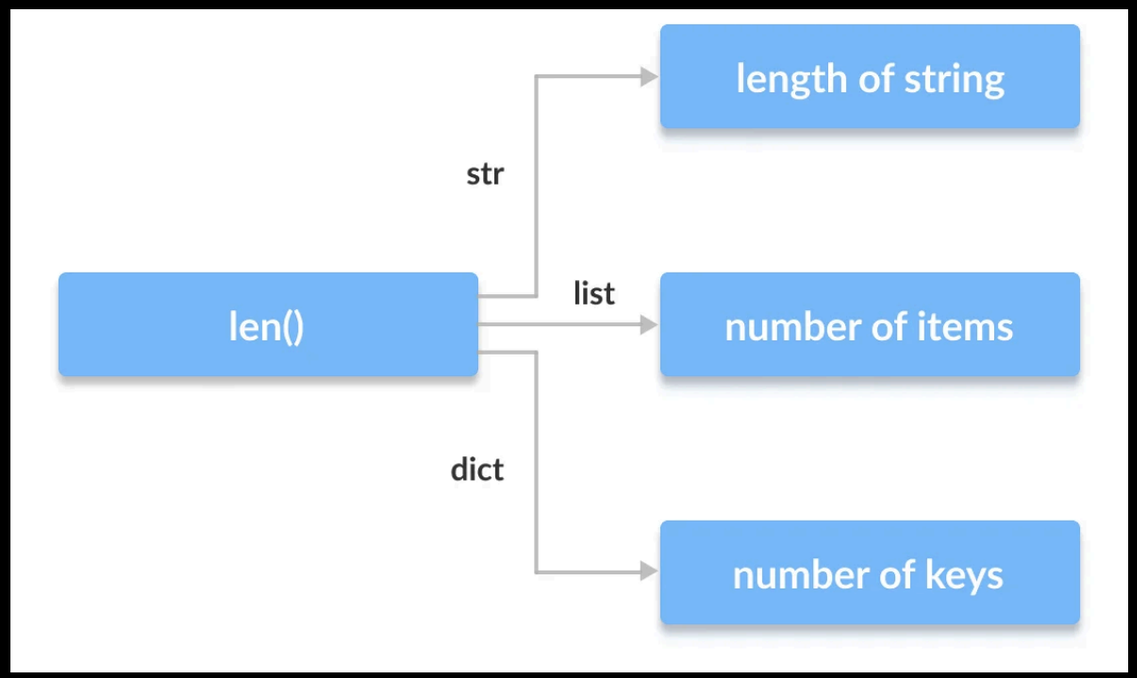
Similarly, for string data types, `+` operator is used to perform concatenation.

```python
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

Here, we can see that a single operator **+** has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

One such function is the **len()** function. It can run with many data types in Python. Let's look at some example use cases of the function.

```python
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```



# Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.

For example, say we have three classes: `Car`, `Boat`, and `Plane`, and they all have a method called `move()`:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747")     #Create a Plane class

for x in (car1, boat1, plane1):
  x.move()
```

A child class inherits all the methods from the parent class. However, in some situations, the method inherited from the parent class doesn't quite fit into the child class. In such cases, you will have to re-implement method in the child class.

```
class Tomato():
    def type(self):
      print("Vegetable")
    def color(self):
      print("Red")
class Apple():
    def type(self):
      print("Fruit")
    def color(self):
      print("Red")
class Test:
   def func(self,obj):
      obj.type()
      obj.color()

obj_tomato = Tomato()
obj_apple = Apple()
func(obj_tomato)
func(obj_apple)
```

# Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called `Vehicle`, and make `Car`, `Boat`, `Plane` child classes of `Vehicle`, the child classes inherits the `Vehicle` methods, but can override them:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

# Encapsulation in Python

Encapsulation is one of the key concepts of object-oriented languages like Python, Java, etc. Encapsulation is used to restrict access to methods and variables. In encapsulation, code and data are wrapped together within a single unit from being modified by accident

```
class Students:
    def __init__(self, name, rank, points):
        self.name = name
        self.rank = rank
        self.points = points

    # custom function
    def demofunc(self):
        print("I am "+self.name)
        print("I got Rank ",+self.rank)

# create 4 objects
st1 = Students("Steve", 1, 100)
st2 = Students("Chris", 2, 90)
st3 = Students("Mark", 3, 76)
st4 = Students("Kate", 4, 60)

# call the functions using the objects created above
st1.demofunc()
st2.demofunc()
st3.demofunc()
st4.demofunc()
```

## Python Access Modifiers

Let us see the access modifiers in Python to understand the concept of Encapsulation and data hiding –

- public
- private
- protected

## The public Access Modifier:

The public member is accessible from inside or outside the class.

## The private Access Modifier

The private member is accessible only inside class. Define a private member by prefixing the member name with two underscores, for example –

## The protected Access Modifier

The protected member is accessible. from inside the class and its sub-class. Define a protected member by prefixing the member name with an underscore, for example –

```python
# Python program to
# demonstrate private members

# Creating a Base class


class Base:
    def __init__(self):
        self.a = "EntriApp"
        self.__c = "EntriApp"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AttributeError as
# private member of base class
# is called inside derived class
```

the protected variable can be accessed out of the class as well as in the derived class (modified too in derived class), it is customary(convention not a rule) to not access the protected out the class body.

Rules must be followed otherwise they produce errors while conventions are necessary for readability and making code similar among developers.

```
# Python program to
# demonstrate protected members

# Creating a base class
class Base:
    def __init__(self):

        # Protected member
        self._a = 2

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling protected member of base class: ",
              self._a)

        # Modify the protected variable:
        self._a = 3
        print("Calling modified protected member outside class: ",
              self._a)


obj1 = Derived()

obj2 = Base()

# Calling protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._a)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._a)
```