Classes and Objects in Python

What we will learn?

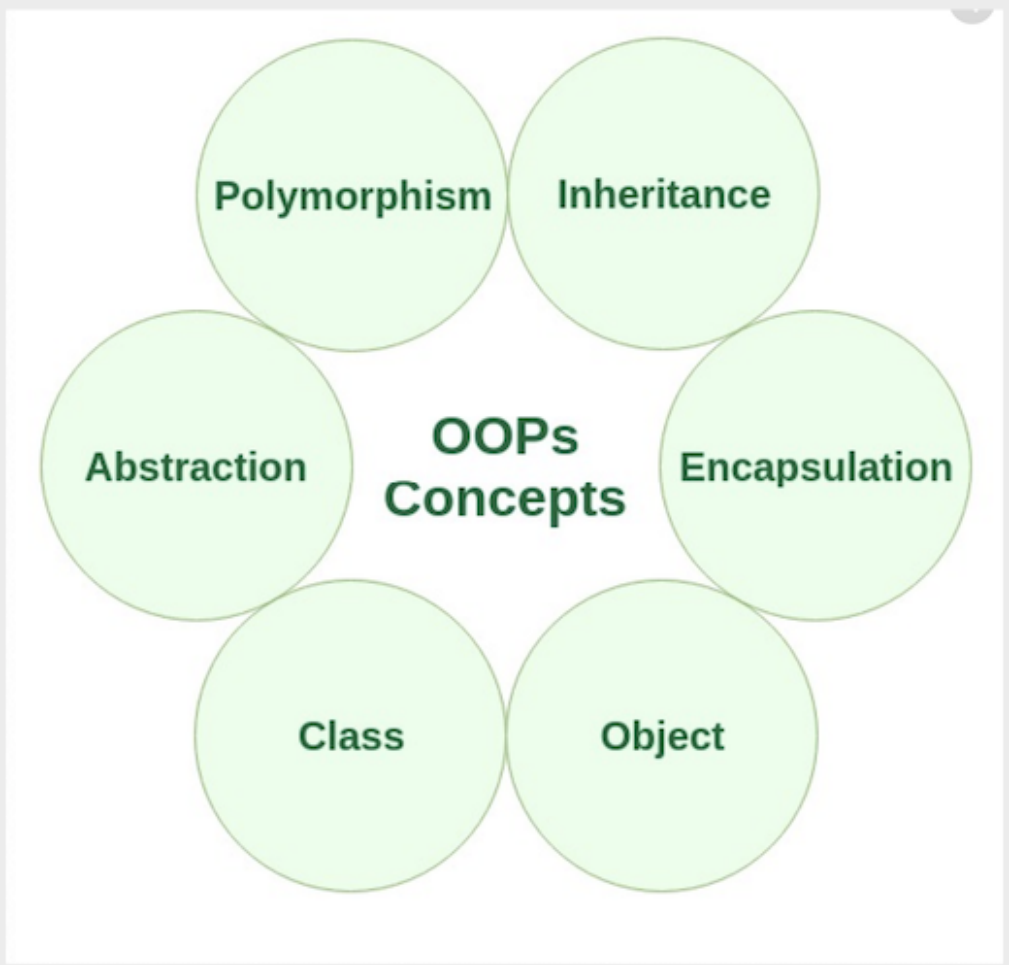
- Introduction to OOPS
    - Classes and Objects
    - Encasulation
    - Abstraction
    - Inheritance
    - Polymorhism
- Classes and Objects
    - Creating Class
    - Creating Object
    - Types of variables
    - Types of methods
    - Passing member of one class to another class
    - Inner Classes

Introduction to OOPS

> Python is a **multi-paradigm** programming language. It supports different programming approaches.

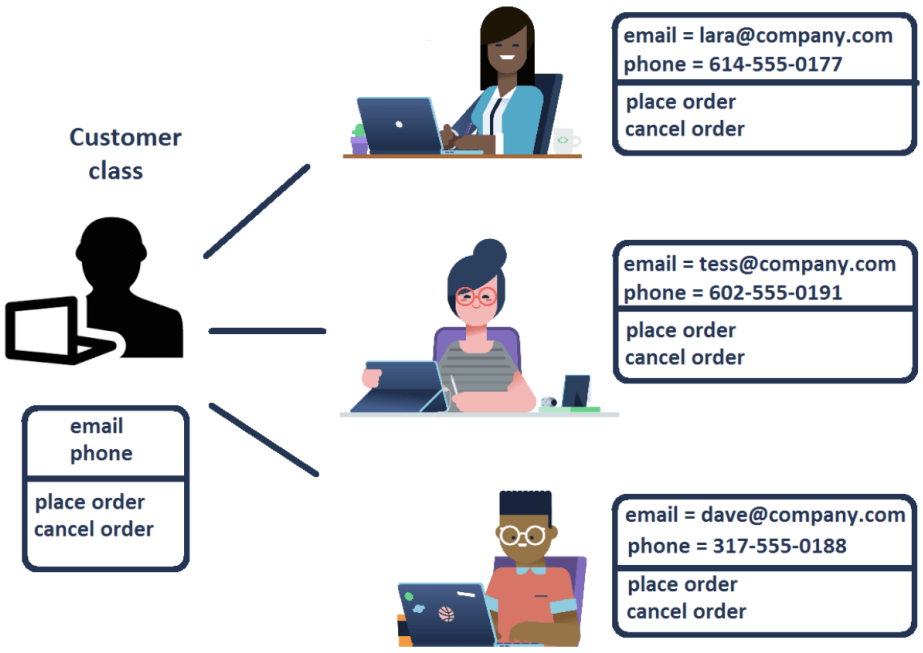
> One of the popular approaches to solve a programming problem is by creating objects. This is known as **Object-Oriented Programming** (OOP).

> Features of object oriented programmig

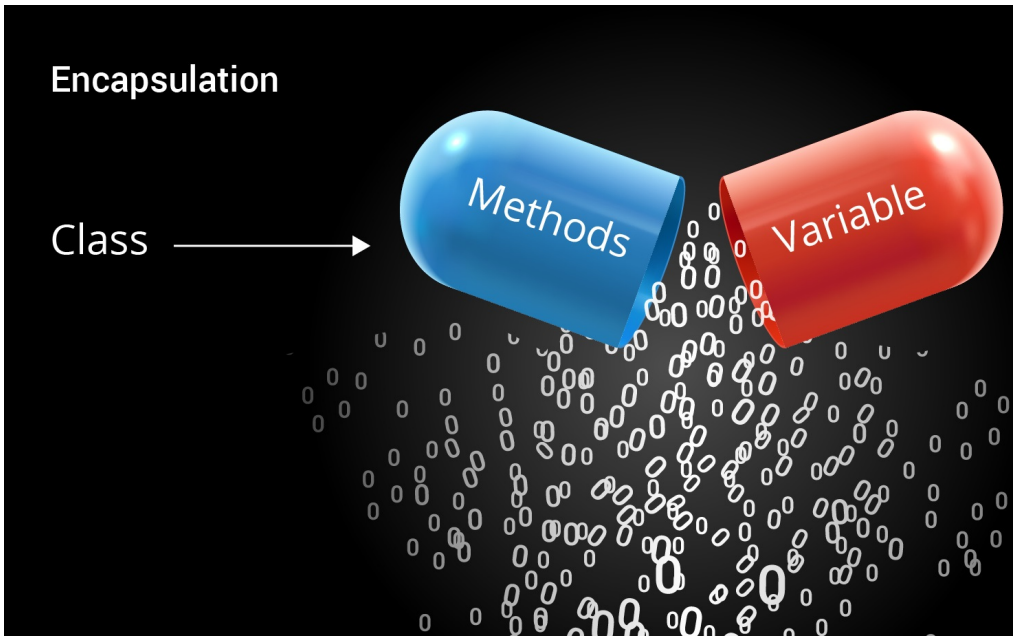


Object is anything that exists in the world and can be distinguished from others. Every object has some **behaviour**, reresented by **attributes and actions**.

A **class** is a **model or blueprint** for creating the objects.



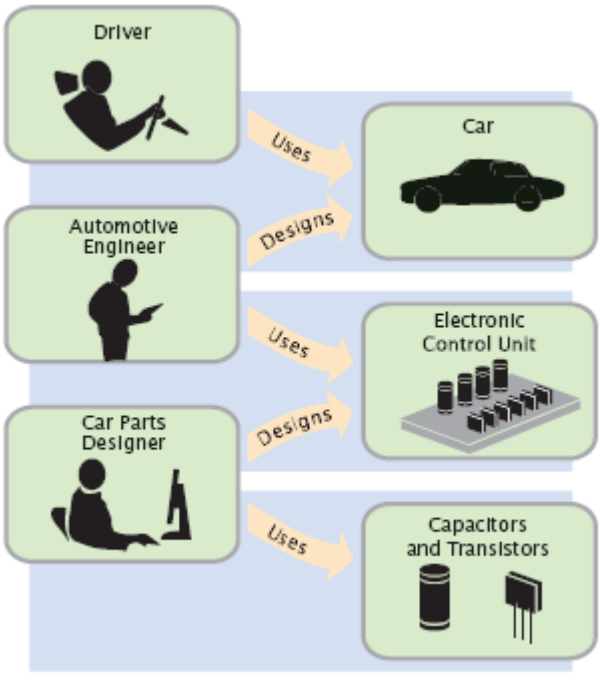
Encapsulation is a mechanism where the data(variables) and the code (methods) that act on the data are **binded** together. Example classes.



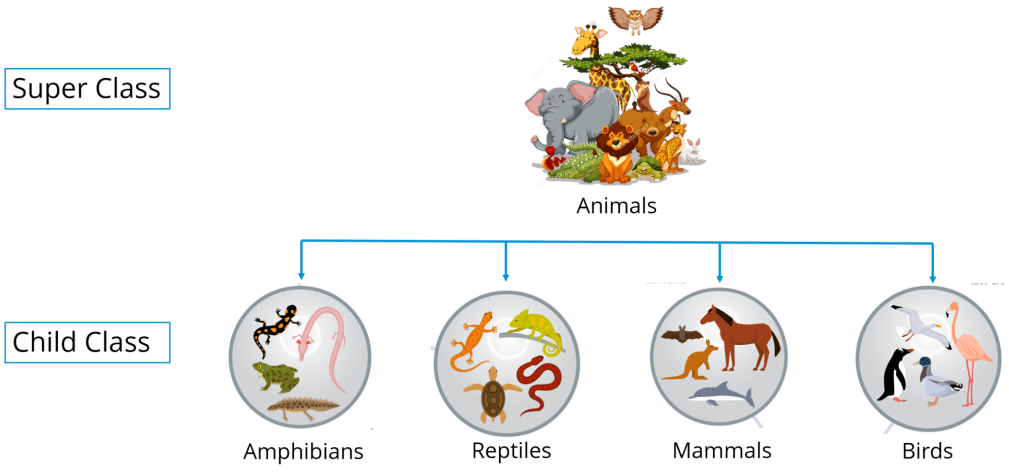
Encapsulation **isolates** member of a class from the members of another class. The reason is when objects are created, each objects share different memory, hence will not be any overwriting of data. That indicates programmer can use same name for the members of two different classes.

Abstraction is the process of **hiding** the real implementation of an application from the user and emphasizing only on usage of it.

Through the process of abstraction in Python, a programmer can hide all the irrelevant data/process of an application in order to **reduce complexity and increase efficiency.**



Inheritance is creating new classes from existing classes, so that the new classes will acquire all the features of the existing classes.



Polymorphism, word comes from two greek words, **poly means many and morphos means forms.**

> Polymorphisn represents the ability to assume different form. In programming, if object or method is behaving differenly in different context, it is called polymorphic in nature.

> > 

Classes and Object

Class is a model to create objects. This indicates class well have attributes and actions. Attributes are represented by variables and actions are per formed by methods.

> ## Creating a class

```
#Syntax
class classname(object):
  '''DOCSTRING'''
  attributes
  def __init__(self):
    pass
```

```
  def __method1__():
    pass
```

> A class is created with the keyword ***class*** followed by classname.

> After the classname, **object** is written inside the paranthesis. Object represnts base class for all classes in python. Writing word object is optional.

> Just like functions, first statement inside class can be a **docstring**, helful for creating documentation hence it is optional.

> **Attributes** are variables to store data.

> **init()** is a special method to initialize variable. This method has double underscores before and after, this indicates that the method is internally defined and we cannot call this method explicitly.

> **self** is the variable that refers to current class instance. When we create an instance of a class, separate memory block is allocated in the heap and that memory location is by default stored in self.

> **method()** are intended to process variables

```
#EXAMPLE1 - An Emlopyee class

class Employee: # or class Employee(object):
  def __init__(self):
    self.name = 'ABC'
    self.eid = 111


  #Every instance method must have self as first parameter
  def display(self):
    print("Name of employee is {} and employee ID is {}".format(self.name,self.eid))
```

Creating Object

> To use functionalities of a class, we should create an instance or object of the class.

> Creating instance represents alloting memory for storing actual information about the instance.

> > Syntax:
>
> > **instancename = classname()**
>
> > When we create instance, following steps will take place internally.
>
> 1. Block of memory is allocated on heap
> 2. **init**() method is called implicitly.
> 3. The allocated memory address ofthe instance in retured into the variable.
>
> > Variables and methods for an instance can be referenced using **dot operator**.

```
#creating an instance on Employee class
e1 = Employee()

#Referencing instance variable and instance method
print(e1.name)
print(e1.eid)
e1.display()
```

```
ABC
111
Name of employee is ABC and employee ID is 111
```

```
#EXAMPLE2 - An Emlopyee class with parametrized constructor

class Employee:
  def __init__(self, n = 'aaa', id = 000):
    self.name = n
    self.eid = id

  def display(self):
    print("Name of employee is {} and employee ID is {}".format(self.name,self.eid))

e2 = Employee()
e2.display()

e2 = Employee('John',112)   #Contructor Overloading
e2.display()
```

```
Name of employee is aaa and employee ID is 0
Name of employee is John and employee ID is 112
```

Types of Variables

The variables which are written inside the class are of two types:

- Instance Variable
  - Variables whose separate copy is created for every instance
  - Syntax: **instance_name.instance_variable**
- Class Variables
  - Variables whose single copy is available to all instances of a class.
    - If we modify the copy of class variable in an instance, it will be refelected across all instances.

To access class variable, we need class method with cls as first parameter.

To access class variable inside class method, syntax is **cls.variablename** whereas to access outside class, Syntax are

classname.variable

OR

instancename.variable

```
#Example3 -Program to demonstrate type of variables
class Sample:
  x = 10                # class variable

  def __init__(self):   # Constructor
    self.y = 20         # Instance Variable

  @classmethod
  def modifyX(cls):     # Class Method
    cls.x +=1


  def modifyY(self,n):  # Instance Method
    self.y += n

s1 = Sample()           # Creating two instances
s2 = Sample()
```

```
print("Before Modification")
print(s1.x , s2.x)
print(s1.y, s2.y)

print("\nAfter Modifying X")
s1.modifyX()
print(s1.x , s2.x)

print("\nAfter Modifying Y")
s1.modifyY(10)
s2.modifyY(20)
print(s1.y , s2.y)

#we can access class variable using class name instead of instance name
print("\nAccessing using class name")
print(Sample.x)
```

```
Before Modification
10 10
20 20

After Modifying X
11 11

After Modifying Y
30 40

Accessing using class name
11
```

Types of Methods

- **Instance Methods** - These methods are bounded to instances and thy act upon instance variables. They are of two types:
    - **Accessor methods** - Simply reads or access the data. They are generally of the form **getXXXX()**, also known as **getter** methods.
    - **Mutator methods** - Not only reads the data but also modifies it. They are generally of the form **setXXXX()**, also known as **setter methods**.
- **Class Methods**
    - These methods act on class level, written to process class variables.

- These methods are written using **@classmethod decorator** above them.
- By default, first parameter to the class method is **cls** which refers to the class itself.

```
#Example 4 - Program to demonstrate types of methods
class Employee:
   noEmp = 0                          # class variable
   def __init__(self):                #Constructor
    Employee.noEmp += 1

   def setName(self,name):         #Setter methods
     self.name = name

   def setID(self, ID):
     self.ID = ID

   def getName(self):             # Getter Methods
     return self.name

   def getID(self):
     return self.ID

   @classmethod
   def getnoEmp(cls):            # class method
     return cls.noEmp

#outside class
n = int(input('Enter number of Emloyee details to be saved: '))
i = 0
empList = [] # list for storing employee object

while i < n :
  e = Employee()     # creating employee object

  name = input("Enter name of employee %s - "%(i+1))
  e.setName(name)     # Calling instance method

  id = input("Enter ID of employee %s - "%(i+1))
  e.setID(id)        # Calling instance method

  i += 1

  empList.append(e)  # storing single employee object in list

#outside while
print(empList)
```

```
print(empList[0])

print(empList[0].name, empList[0].ID)

print('Total number of emloyees are ',Employee.getnoEmp())
print("EmpID \t Name")
for i in empList:
  print(i.ID,"\t",i.name)
```

```
Enter number of Emloyee details to be saved: 3
Enter name of employee 1 - abc
Enter ID of employee 1 - 1
Enter name of employee 2 - def
Enter ID of employee 2 - 2
Enter name of employee 3 - xyz
Enter ID of employee 3 - 3
[<__main__.Employee object at 0x7ffb9b6820d0>, <__main__.Employee object at 0x7ffb99bae4
<__main__.Employee object at 0x7ffb9b6820d0>
abc 1
Total number of emloyees are  3
EmpID    Name
1        abc
2        def
3        xyz
```

- **Static Methods**

  - Static methods, much like class methods, are methods that are bound to a class rather than its object.
  - They do not require a class instance creation. So, they are not dependent on the state of the object
  - The difference between a static method and a class method is:

    - Static method knows nothing about the class and just deals with the parameters.
    - Class method works with the class since its parameter is always the class itself.

  - These methods are written using **@staticmethod** decorator above them and there is no default first parameter like cls.

```
#Example 5 - Program to demonstrate static methods
```

```
class Square:

  @staticmethod
  def calculate(x):
    return x*x

print("Sqaure of 10 is ", Square.calculate(10)) # static methods are invoked without creating
```

```
Sqaure of 10 is  100
```

Passing member of one class to another class

> It is possible to pass the members of a class to another class.
>
> To pass all members, instance of a class is passed.
>
> In another class, static method is use to access members of the class.

```
#Example 6 - Program to demonstrate passing member of one class to another class
class Emp:
  def __init__(self,id,name,sal):
    self.id = id
    self.name = name
    self.sal = sal

  def display(self):
    print('ID = ',self.id)
    print('Name = ',self.name)
    print('Salary = ',self.sal)
    print('------------')

class AnotherClass:
  #static method to retrieve Emp object and process it
  staticmethod
  def increment(e):
    e.sal += 1000
    e.display()

#outside Another class
e1 = Emp(333,'ABC',5000)
e1.display()
AnotherClass.increment(e1)
```

```
ID =  333
Name =  ABC
Salary =  5000
```

```
------------
ID =  333
Name =  ABC
Salary =  6000
------------
```

Inner Classes

> An inner class or nested class is a defined entirely within the body of another class.

> If an object is created using a class, the object inside the root class can be used. A class can have more than one inner classes, but in general inner classes are avoided.

> **Why Inner Classes?**

> 1. Grouping of two or more classes. Suppose you have two classes Car and Engine. Every Car needs an Engine. But, Engine won't be used without a Car. So, you make the Engine an inner class to the Car. It helps save code.

> 2. Hiding code is another use of Nested classes. You can hide the Nested classes from the outside world.

> 3. It's easy to understand the classes. Classes are closely related here. You don't have to search for the classes in the code. They are all together.

```
#Example 7 - Program to demonstrate nesting of classes
#program to create a dob class inside a person class

class Person:
  def __init__(self):
    self.name = input('Enter your name - ')
    self.dob = self.DateofBirth() #creating instance of inner class

  def display(self):
    print("Name is ",self.name)
    self.dob.display()


  #inner class
  class DateofBirth:
    def __init__(self):
      self.inp = input('Enter date in the format dd-mm-yyyy  ')
      self.date = self.inp.split('-')
```

```
    def display(self):
      print("Date of Birth is {}/{}/{}".format(int(self.date[0]),int(self.date[1]),int(self.d

p = Person()
p.display()
```

```
 Enter your name - abc
 Enter date in the format dd-mm-yyyy  26-03-2021
 Name is  abc
 Date of Birth is 26/3/2021
```

```
import inspect
class A:

    @staticmethod
    def f():
        print('f')
        pass

    @classmethod
    def g(cls):
        pass

    def r():
        print('r')
        pass
    #print('This is class A')

a = A()

print(isinstance(inspect.getattr_static(a, "f"), staticmethod))
print(isinstance(inspect.getattr_static(A, "f"), staticmethod))
print(isinstance(inspect.getattr_static(a, "g"), classmethod))
print(isinstance(inspect.getattr_static(A, "g"), classmethod))
print(isinstance(inspect.getattr_static(a, "r"), classmethod))
print(isinstance(inspect.getattr_static(a, "r"), staticmethod))
print(isinstance(inspect.getattr_static(A, "r"), classmethod))
print(isinstance(inspect.getattr_static(A, "r"), staticmethod))

#A.r()
#a.r()
#A.f()
#a.f()
```

```
True
True
True
True
False
False
```

```
False
False
```