

What is self in Python?

The self-keyword is **used to represent an instance (object) of the given class.**

By using the “self” we can access the attributes and methods of the class in python. It binds the attributes with the given arguments. The reason you need to use self. is because Python does not use the @ syntax to refer to instance attributes.

What is self in \_\_init\_\_ ?

The self in keyword in Python is **used to all the instances in a class.**

By using the self-keyword, one can easily access all the instances defined within a class, including its methods and attributes. init. \_\_init\_\_ is one of the reserved methods in Python.

In object-oriented programming, it is known as a constructor.

## Class

A class in Python is a category or set of different elements grouped together that share one or more similarities with one another, but yet distinct from other classes via type, quality and kind.

In technical terminology, we can define a class in Python as being a blueprint for individual objects with same or exact behavior.

## Object

An object in Python is one instance of a class and it can be programmed to perform the functions that have been defined in the class.

## Self

The self in keyword in Python is used to all the instances in a class. By using the self keyword, one can easily access all the instances defined within a class, including its methods and attributes.

## init

\_\_init\_\_ is one of the reserved methods in Python. In object oriented programming, it is known as a constructor.

The \_\_init\_\_ method can be called when an object is created from the class, and access is required to initialize the attributes of the class.

```
class Car(object):
```

```
    """
```

```
    blueprint for car
```

```
"""
```

```
def __init__(self, model, color, company, speed_limit):  
    self.color = color  
    self.company = company  
    self.speed_limit = speed_limit  
    self.model = model  
  
def start(self):  
    print("started")  
  
def stop(self):  
    print("stopped")  
  
def accelerate(self):  
    print("accelarating...")  
    print("accelarator functionality here")  
  
def change_gear(self, gear_type):  
    print("gear changed")  
    print(" gear related functionality here")
```

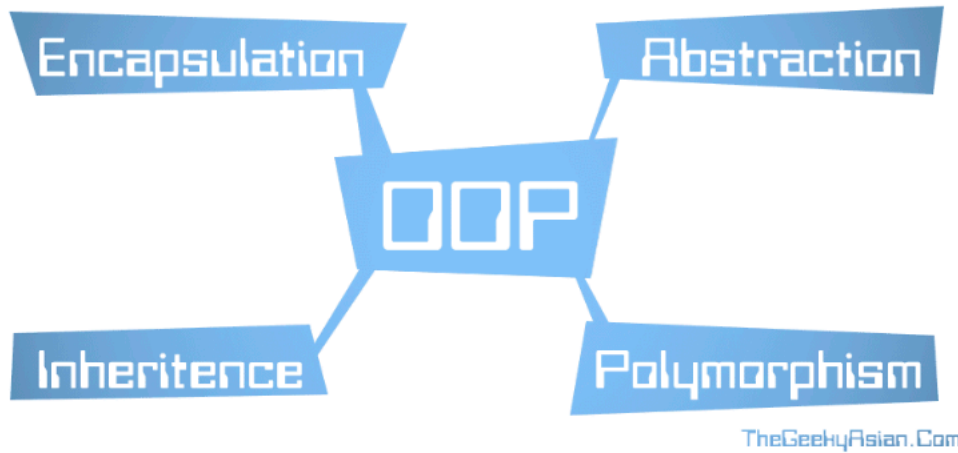
*# Now that we have created the objects, let's move on to create the individual cars in the game.*

```
maruthi_suzuki = Car("ertiga", "black", "suzuki", 60)
```

```
audi = Car("A6", "red", "audi", 80)
```

### **There are 4 pillars of OOP:**

- Encapsulation.
- Inheritance.
- Abstraction.
- Polymorphism.



<https://dev.to/terrythreatt/the-four-principles-of-object-oriented-programming-in-python-1jbi>

Object-oriented or OOP we shall call it is one of several different programming paradigms used in order to structure your code in a way that is easier to follow. It receives its name by defining objects you can interface within your Python programs.

## Objects

Objects purposely represent real-world objects or things like a cat or dog. Python objects have a collection of related properties or behaviors like `meow()` or `bark()`.

## Classes

OOP in Python is class-based and your objects will be defined with the `class` keyword like the example below:

```
class Cat:  
    pass
```

## Principle 1 - Abstraction

Abstraction is the concept of hiding all the implementation of your class away from anything outside of the class.

```
class Dog:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def bark(self):
        print("woof!")

# we don't care how it works just bark
spot = Dog("spot") #=> spot was adopted.
spot.bark() #=> woof!
```

## Principle 2 - Inheritance

Inheritance is the mechanism for creating a child class that can inherit behavior and properties from a parent(derived) class.

```
class Animal:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def run(self):
        print("running!")

class Dog(Animal):

    def __init__(self):
        super().init

    def bark(self):
        print("woof!")

# new dog behavior inherited from Animal parent class
spot = Dog("spot") #=> spot was adopted.
spot.run() #=> running!
```

## Principle 3 - Encapsulation

Encapsulation is the method of keeping all the state, variables, and methods private unless declared to be public.

```
class Fish:

    def __init__(self):
        self.__size = "big"

    def get_size(self):
        print("I'm a " + self.__size + " fish")

    def set_size(self, new_size):
        self.__size = new_size

# using the getter method
oscar = Fish()
oscar.get_size() #=> I'm a big fish

# change the size
bert = Fish()
bert.__size = "small"
bert.get_size() #=> I'm a big fish

# using setter method
fin = Fish()
fin.set_size("tiny")
fin.get_size() #=> I'm a tiny fish
```

## Principle 4 - Polymorphism

Polymorphism is a way of interfacing with objects and receiving different forms or results.

```
class Animal:

    def __init__(self, name):
        self.name = name
        print(self.name + " was adopted.")

    def run(self):
        print("running!")
```

```
class Turtle(Animal):
```

```
    def __init__(self):  
        super().__init__
```

```
    def run(self):  
        print("running slowly!")
```

# we get back an interesting response

tim = Turtle("tim") #=> tim was adopted.

tim.run() #=> running slowly!

★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★

<https://tutorialsinhand.com/tutorials/python-tutorial/python-oops-concept/four-pillars-of-oop-python.aspx>

## 1. Inheritance in python

In case of real world objects, every element is a specialized within its general group of elements.

For example, if we have a general class "Animal", it can have two specialized classes like "Wild" and "Domestic". Under "Domestic" also we may have multiple other specialized class. Thus "Wild" is a specialized subclass of the parent "Animal" class and the properties of "Wild" class share some properties with its parent class but it also has some unique properties which makes it unique within in parents class. This is what is called "**Inheritance**".

Lets see *example of inheritance in python*:

```
# Creating a Class in Python  
class Animal:  
    def sounds(self):  
        print("This animals makes some sound!")
```

```
# Wild class inherits Animal class  
class Wild(Animal):  
    def asPet(self):  
        print("This animal can't be pet!")
```

```
# Domestic class inherits Animal class  
class Domestic(Animal):  
    def asPet(self):  
        print("This animal can be pet!")
```

```
# Creating object of Wild class  
tiger = Wild()
```

```
tiger.sounds()  
# Output : This animals makes some sound!
```

```
tiger.asPet()  
# Output : This animal can't be pet!
```

As we can see in the example, we specify a class inherits the members of another class by writing the parent class within parenthesis of the child class. And a child class possesses all the methods and variables as defined by the super class. That is why we can use sounds() method in an object of Wild class because the Wild class inherits the properties of the Animal class.

---

## 2. Encapsulation in python

In Python, we can protect the member variables of a class from being accessed by any program outside our class using the principle of Encapsulation.

Encapsulation means binding up of code and data together under 1 wrapper which we already know as "Class" and provide proper access to change or modify the data of the class. We hide the variables by providing one underscore as prefix to the name of the variable.

Let see an *example of encapsulation in python*:

```
# Creating a Class in Python
class Animal:

    # Class Constructor
    def __init__(self,species):
        self._species = species

    def sounds(self):
        print("This animals makes some sound!")
```

```
tiger = Animal("Tiger")
print(tiger._species)
```

# Output : Tiger

In the above code, we see a new method \_\_init\_\_(self,value). This is said to be the class constructor. Every class by default has a constructor but it doesn't initialize any value. To initialize any value of the object, we can overwrite the default constructor and write our own parameterized constructor to set the state of the object. It is to be noted that this method has 2 underscores as prefix and suffix respectively. This is so because 2 underscores as prefix and suffix defines this method is supposed to be called by the Python interpreter and not by any code written by the programmer. The work of the constructor is to initialize an object in the memory and return a reference to it.

Here, the variable "species" has an underscore as prefix. This signifies that this variable is a private variable and should not be accessed directly without the invoking object. Moreover subclass gets access to the parent class's private member variables. Unlike, the previous example, here we instantiate a tiger object by passing an argument to the Animal() constructor which goes into the \_\_init\_\_ method and the object gets created with the value of the private member variable species set to "Tiger".

---

### 3. Polymorphism in python

The word "**Polymorphism**" has been derived from Greek literature meaning "having many forms". It generally emphasized on the fact of common interface for general set of activities.

An *example of polymorphism in python* is given below.

```
# Creating a Class in Python
class Dog:

    def makeSound(self):
        print("Dogs bark!")

class Tiger:

    def makeSound(self):
        print("Tigers growl!")

# common interface
def check_makeSound(animal):
    animal.makeSound()

myDog = Dog()
myTiger = Tiger()

check_makeSound(myDog) # Output : Dogs bark!
check_makeSound(myTiger) # Output : Tigers growl!
```

In the above example, we have 2 classes Dog and Tiger which share the same method "makeSound". But what they print are different. So we just created an interface via a function that will take in an object and will call its respective "makeSound" method. This we have implemented Polymorphism. For the same method name, what they do are different and we have successfully implemented it.

---

### 4. Abstraction in python

The concept of abstraction in O.O.P. is mainly utilized to create a blueprint of a class.

Like we just define what member methods should be present in the class and not the way how the class implements them. The abstract class just defines what methods should be present in the classes which inherits it and not what the method does. The activity of the method is left on the subclass to decide which inherits the abstract class.

By default, Python doesn't provide abstraction. There is a **module named "abc" (Abstract Base Class)** which helps us to realize the concept of abstraction in Python.

```
# Abstraction
from abc import ABC, abstractmethod

class Book(ABC):

    # Abstract method
    def book_name(self):
        pass
```



```

class Python(Book):

    # Overwrite Abstract method
    def book_name(self):
        print("This is a Python programming book!")

class Java(Book):

    # Overwrite Abstract method
    def book_name(self):
        print("This is a Java programming book!")

myBook = Java()
myBook.book_name()

# Output : This is a Java programming book!

```

An abstract class can define both an abstract method and a concrete method. But a proper class should define concrete methods only. If it inherits any abstract class then the concrete class should define the body of the abstract method by overwriting abstract method's body.

★★

## CLASSES AND OBJECTS

### 1. What is a class ?

A class is logical grouping of attributes(variables) and methods(functions)

Syntax:

```

class ClassName:

    # class body

```

Example:

```

class Employee:

    # class body

```

## 2. What is an object ?

Object is an instance of a class that has access to all the attributes and methods of that class

Syntax:

```
objectName = ClassName()
```

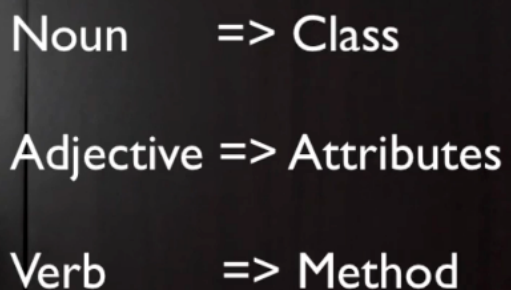
Example:

```
employee = Employee()
```

# The object employee now has access to all the attributes and methods of the class Employee. You will be learning more about attributes and methods in the next section.

## 3. What is object instantiation ?

The process of creation of an object of a class is called instantiation



```
Noun    => Class  
Adjective => Attributes  
Verb     => Method
```

# ATTRIBUTES AND METHODS

## 1. What are data members ?

Data members are attributes declared within a class. They are properties that further define a class.

There are two types of attributes: class attributes and instance attributes.

## 2. What is a class attribute ?

An attribute that is common across all instances of a class is called a class attribute.

Class attributes are accessed by using class name as the prefix.

Syntax:

```
class className:  
    classAttribute = value
```

Example:

```
class Employee:  
    # This attribute is common across all instances of this class  
  
    numberOfEmployees = 0  
  
    employeeOne = Employee()  
    employeeTwo = Employee()  
  
    Employee.numberOfEmployees += 1
```

```
print(employeeOne.numberOfEmployees)
print(employeeTwo.numberOfEmployees)
```

# Output => 1, 1

# Since numberOfEmployees was created as a class attribute, we incremented it to 1 by making use of its class name. This value has been reflected across objects employeeOne and employeeTwo.

### 3. What is an instance attribute ?

An attribute that is specific to each instance of a class is called an instance attribute. Instance attributes are accessed within an instance method by making use of the self object.

Syntax:

```
class className:

    def methodName(self):
        self.instanceAttribute = value
```

Example:

```
class Employee:

    def employeeDetails(self, name):

        # name is the instance attribute
        self.name = name
```

## 4. What is the self-parameter ?

Every instance method accepts has a default parameter that is being accepted. By convention, this parameter is named self.

The self-parameter is used to refer to the attributes of that instance of the class.

Example:

```
class Employee:

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

employee = Employee()

employee.setName('John')
```

# The above statement is inferred as Employee.setName(employee, 'John') where the object employee is taken as the self-argument.

# In the init method when we say self.name, Python actually takes it as employee.name, which is being stored with the value John.

Пример как работает self parameter

```
class Employee:
    def employeeDetails():
        pass

emp = Employee()
emp.employeeDetails()
print(emp.employeeDetails(emp))
```

код выдает ошибку

```
class Employee:
    def employeeDetails(self):
        self.name = 'John'
        print('Name = ', self.name)

emp = Employee()
emp.employeeDetails()
print(Employee.employeeDetails(emp))
```

## 5. What are methods ?

A function within a class that performs a specific action is called a method belonging to that class.

Methods are of two types: static method and instance Method

## 6. What is an instance method ?

A method which can access the instance attributes of a class by making use of self object is called an instance method

Syntax:

```
def methodName(self):
    # method body
```

Example:

```
class Employee:
    # employeeDetails() is the instance method

    def employeeDetails(self, name):
        self.name = name
```

## 7. What is a static method ?

A method that does not have access to any of the instance attributes of a class is called a static method.

Static method uses a decorator `@staticmethod` to indicate this method will not be taking the default self parameter.

Syntax:

```
@staticmethod
# Observe that self is not being declared since this is a
static method

def methodName():
    # method body
```

Example:

```
class Employee:
    numberOfEmployees = 0

    @staticmethod
    def updateNumberOfEmployees():
        Employee.numberOfEmployees += 1

employeeOne = Employee()
employeeTwo = Employee()

employeeOne.updateNumberOfEmployees()

employeeTwo.updateNumberOfEmployees()

print(Employee.numberOfEmployees)

# We have used a static method that updates the class
```

attribute of the class Employee.

```
class Employee:
    def employeeDetails(self):
        self.name = 'Ben'
    @staticmethod
    def welcomeMessage():
        print("Welcome to our organization!")

emp = Employee()
emp.employeeDetails()
print(emp.name)
emp.welcomeMessage()
```

## 8. What are the ways to access the attributes and methods of a class ?

Attributes and methods of a class can be accessed by creating an object and accessing the attributes and objects using class name or object name depending upon the type of attribute followed by the dot operator (.) and the attribute name or method name.

Example:

```
class Employee:
    numberOfEmployees = 0

    def printNumberOfEmployees(self):
        print(Employee.numberOfEmployees)
```

```
employee = Employee()
```

```
# Modify class attribute using dot operator
Employee.numberOfEmployees += 1
```



```
# Call the instance method using dot operator
employee.printNumberOfEmployees()
```

## 9. What is an init method ?

An init method is the constructor of a class that can be used to initialize data members of that class.

It is the first method that is being called on creation of an object.

Syntax:

```
def __init__(self):
    # Initialize the data members of the class
```

Example:

```
class Employee:
```

```
    def __init__(self):
        print("Welcome!")
```

```
employee = Employee()
```

# This would print Welcome! since \_\_init\_\_ method was called on object instantiation.

```
class Employee:
    def enterEmployeeDetails(self):
        self.name = 'Mark'

    def displayEmployeeDetails(self):
        print(self.name)
```

```
empl = Employee()
empl.displayEmployeeDetails() # AttributeError: 'Employee' object has
no attribute 'name'
```

```
class Employee:
    def __init__(self):
        self.name = 'Mark'

    def displayEmployeeDetails(self):
        print(self.name)
```

```
empl = Employee()
empl.displayEmployeeDetails() # Mark
```

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

    def displayEmployeeDetails(self):
        print(self.name)
```

```
empl1 = Employee('John')
empl2 = Employee('Diana')
empl1.displayEmployeeDetails()
empl2.displayEmployeeDetails()
```

## Attributes and Methods

What is an attribute?

- An attribute is a property that further defines a class

What is a method?

- A method is a function within a class which can access all the attributes of a class and perform a specific task

What is a class attribute?

- Class attributes are attributes that are shared across all instances of a class
- They are created either as a part of the class or by using `className.attributeName`

What is an instance attribute?

- Instance attributes are attributes that are specific to each instance of a class
- They are created using `objectName.attributeName`

How is self parameter handled?

-The method call `objectName.methodName()` is interpreted as `className.methodName(objectName)` and this parameter is referred to as 'self' in method definition

What is an instance method?

- Method that has the default parameter as the object are static methods. They are used to modify the instance attributes of a class

What is a static method?

- Method that do not modify the instance attributes of a class are called instance method. They can still be used to modify class attributes

What is init() methods?

- `init()` method is the initializer in Python. It is called when an object is instantiated.

- All the attributes of the class should be initialized in this method to make your object a fully initialized object

## **ABSTRACTION AND ENCAPSULATION**

### **1. What is encapsulation ?**

Hiding the implementation details from the end user is called as encapsulation

### **2. What is abstraction ?**

Abstraction is the process of steps followed to achieve encapsulation

### **Abstraction and Encapsulation Problem Statement**

Implement a library management system which will handle the following tasks:

- Customer should be able to display all the books available in the library
- Handle the process when a customer requests to borrow a book
- Update the library collection when the customer returns a book

```
'''  
Class => Library  
Layers of abstraction => display available books, to lend a book, to  
add a book  
  
Class => Customer  
Layers of abstraction => request for a book, return a book  
'''
```

```

class Library:
    def __init__(self, listOfBooks):
        self.availableBooks = listOfBooks

    def displayAvailableBooks(self):
        print("\nAvailable Books\n")
        for book in self.availableBooks:
            print(book)

    def lendBook(self, requestedBook):
        if requestedBook in self.availableBooks:
            print("You have now borrowed the book")
            self.availableBooks.remove(requestedBook)
        else:
            print("Sorry, the books is not available in our list\n")

    def addBook(self, returnedBook):
        self.availableBooks.append(returnedBook)
        print("You have returned the book. Thank you!\n")

class Customer:
    def requestBook(self):
        print("Enter the name of a book you would like to borrow: ")
        self.book = input()
        return self.book

    def returnBook(self):
        print("Enter the name of the book which you are returning: ")
        self.book = input()
        return self.book

library = Library(['Think and Grow Rich', 'Who will cry when you
die', 'For one more day'])

customer = Customer()
while True:
    print("Enter 1 to display the available books")
    print("Enter 2 to request for a book")
    print("Enter 3 to return a book")
    print("Enter 4 to exit")
    userChoise = int(input())
    if userChoise is 1:
        library.displayAvailableBooks()

```

```
elif userChoise is 2:
    requestBook = customer.requestBook()
    library.lendBook(requestBook)
elif userChoise is 3:
    returnBook = customer.returnBook()
    library.addBook(requestBook)
elif userChoise is 4:
    quit()
```

## INHERITANCE

Inheritance: one class acquires properties (methods and fields) of another class!

Why? For reusability

How? Using extends keyword

Subclass (derived class or child class)

Superclass (base class or parent class)

Important points

-----

- Subclass can have it's own methods and fields in addition to Superclass's methods and fields
- Subclass can have only one Superclass. In other words, multiple inheritance is not supported
- Subclass cannot inherit Superclass's constructor, but it can invoke the constructor

Types

-----

Single inheritance

Multilevel inheritance

Hierarchical inheritance

Multiple inheritance (using interface)

super keyword

-----

- Used to differentiate members of Superclass from members of Subclass, if they have same names
- Used to invoke the constructor of Superclass from Subclass

## 1. What is inheritance ?

Inheriting the attributes and methods of a base class into a derived class is called Inheritance.

Syntax:

```
class BaseClass:
    # Body of BaseClass

class DerivedClass(BaseClass):
    # Body of DerivedClass
```

Example:

```
class Shape:
    unitOfMeasurement = 'centimetre'

class Square(Shape):
    def __init__(self):
        # The attribute unitOfMeasurement has been inherited from the class
        Shape to this class Square
        print("Unit of measurement for this square: ", self.unitOfMeasurement)

square = Square()
```

```
class Apple:
    manufacturer = 'Apple Inc.'
    contactWebsite = 'www.apple.com/contact'
```

```

def contactDetails(self):
    print("To contact us, log on to ", self.contactWebsite)

class MacBook(Apple):
    def __init__(self):
        self.yearOfManufacture = 2017

    def manufactureDetails(self):
        print(f"This MacBook was manufactured in the year
{self.yearOfManufacture} by {self.manufacturer}")

macBook = MacBook()
macBook.manufactureDetails()
macBook.contactDetails()

```

## 2. What is multiple inheritance ?

Mechanism in which a derived class inherits from two or more base classes is called a multiple inheritance

Syntax:

```

class baseClassOne:
    # Body of baseClassOne

class baseClassTwo:
    # Body of baseClassTwo

class derivedClass(baseClassOne, baseClassTwo):
    # Body of derived class

```

Example:

```

class OperatingSystem:

```



```
multiTasking = True
```

```
class Apple:  
    website = 'www.apple.com'
```

```
class MacOS(OperatingSystem, Apple):
```

```
    def __init__(self):  
        if self.multiTasking is True:
```

```
# The class MacOS has inherited 'multitasking' attribute from the class  
OperatingSystem and 'website' attribute from the class 'Apple'
```

```
        print("MacOS is a multitasking operating system. Visit {} for more  
        details".format(self.website))
```

```
mac = MacOS()
```

```
class OperatingSystem:  
    multitasking = True
```

```
class Apple:  
    website = 'www.apple.com'
```

```
class MacBook(OperatingSystem, Apple):  
    def __init__(self):  
        if self.multitasking is True:  
            print(f"This is a multi tasking system. Visit  
{self.website} for more details")
```

```
macBook = MacBook()
```

```
*****
```

```
class OperatingSystem:  
    multitasking = True  
    name = 'Mac OS'
```

```

class Apple:
    website = 'www.apple.com'
    name = 'Apple'

class MacBook(OperatingSystem, Apple):
    def __init__(self):

        if self.multitasking is True:

            print(f"This is a multi tasking system. Visit
{self.website} for more details")

            print(f"Name: {self.name}")

macBook = MacBook()

# Output => Name: Mac OS

```

### 3. What is multilevel inheritance ?

Mechanism in which a derived class inherits from a base class which has been derived from another base class is called a multilevel inheritance

Syntax:

```

class BaseClass:
    # Body of baseClass

class DerivedClassOne(BaseClass):
    # Body of DerivedClassOne

class DerivedClassTwo(DerivedClassOne):

    # Body of DerivedClassTwo

```

Example:

```
class Apple:
    website = 'www.apple.com'
```

```
class MacBook(Apple):
    deviceType = 'notebook computer'
```

```
class MacBookPro(MacBook):
    def __init__(self):
        # This class inherits deviceType from the base class MacBook. It also inherits
        website from base class of MacBook, which is Apple.
        print("This is a {}. Visit {} for more
        details".format(self.deviceType,self.website))
```

```
macBook = MacBookPro()
```

```
class MusicalInstruments:
    numberOfMajorKeys = 12
```

```
class StringInstruments(MusicalInstruments):
    typeOfWood = "Tonewood"
```

```
class Guitar(StringInstruments):
    def __init__(self):
        self.numberOfStrings = 6
        print(f"This guitar consists of {self.numberOfStrings}
        strings. "
              f"It is made of {self.typeOfWood} and it can play
        {self.numberOfMajorKeys} keys")
```

```
guitar = Guitar()
```

## 5. What is an abstract base class ?

A base class which contains abstract methods that are to be overridden in its derived class is called an abstract base class. They belong to the abc module.

Example:

```
from abc import ABCMeta, abstractmethod
```

```
class Shape(metaclass = ABCMeta):  
    @abstractmethod  
    def area(self):  
        return 0
```

```
class Square(Shape):  
    def area(self, side)  
        return side * side
```

## 6. What are the naming conventions used for private, protected and public members ?

Private => `__memberName` (e.g. `__`) two underscores in front of the attribute name: `__memberName`

Protected => `_memberName` (e.g. `_`) one underscore in front of the attribute name: `_memberName`

Public => `memberName`

```
# Public =>      memberName  
# Protected =>   _memberName  
# Private =>     __memberName
```

```

class Car:
    numberOfWheels = 4
    _color = 'Black'
    __yearOfManufacture = 2017

class BMW (Car):
    def __init__(self):
        print(f"Protected attribute color: {self._color}")

car = Car()
print(f"Public attribute numberOfWheels: {car.numberOfWheels}")

bmw = BMW()
print(f"Private attribute yearOfManufacture:
{car.__Car__yearOfManufacture}")

```

IF =>

```

print(f"Private attribute yearOfManufacture:
{car.__yearOfManufacture}")
# output: AttributeError: 'Car' object has no attribute
'__yearOfManufacture'

```

## 7. How is name mangling done for private members by Python ?

Name mangling is done by prepending the member name with an underscore and class name.

`__className__memberName`

What is inheritance?

- Deriving the attributes and methods of a base class into a derived class is called inheritance
- The derived class will have access to all the attributes and methods of the base class and it can also have attributes and methods of their own

What is single inheritance?

- When a derived class has just one base class, this called single inheritance

Syntax:

```
class baseClass:
```

```
    # Define attributes and methods
```

```
Class derivedClass(baseClass):
```

```
    # Define attributes and methods of derived class apart from inheriting  
    base class attributes
```

What is multiple inheritance?

- When a derived class inherits from more than one base class, it is a multiple inheritance
- The derived class can have access to attributes and methods of all the base classes that it inherited from

Syntax:

```
class baseClassOne:
```

```
    # Attributes and methods of baseClassOne
```

```
class baseClassTwo:
```

```
    # Attributes and methods of baseClassTwo
```

```
class derivedClass(baseClassOne, baseClassTwo):
```

pass

What is multilevel Inheritance?

- When a derived class of a base class becomes a base class for another derived class, this is called multilevel inheritance
- The final derived class will access to attributes of its family of base classes

Name conventions used for class members:

- Public  $\Rightarrow$  variableName
- Protected member  $\Rightarrow$  \_variableName
- Private member  $\Rightarrow$  \_\_variableName

Name mangling on private members:

$\Rightarrow$  \_\_variableName  $\Rightarrow$  *ClassName*\_\_variableName

## POLYMORPHISM

### 1. What is polymorphism ?

The same interface existing in different forms is called polymorphism

Example :

An addition between two integers  $2 + 2$  return 4 whereas an addition between two strings "Hello" + "World" concatenates it to "Hello World"

## 2. What is operator overloading ?

Redefining how an operator operates its operands is called operator overloading.

Syntax:

```
def __operatorFunction__(operandOne, operandTwo):  
# Define the operation that has to be performed
```

Example:

```
class Square:  
    def __init__(self, side):  
        self.side = side  
  
    def __add__(sideOne, sideTwo):  
        return(sideOne.side + sideTwo.side)  
  
squareOne = Square(10)  
squareTwo = Square(20)  
  
# After overloading __add__ method, squareOne + squareTwo is interpreted as  
Square.__add__(squareOne, squareTwo)  
  
print("Sum of sides of two squares = ", squareOne + squareTwo)
```



### 3. What is overriding ?

Modifying the inherited behavior of methods of a base class in a derived class is called overriding.

Syntax:

```
class BaseClass:
    def methodOne(self):
        # Body of method

class DerivedClass(BaseClass):
    def methodOne(self):
        # Redefine the body of methodOne
```

Example:

```
class Shape:
    def area():
        return 0

class Square(Shape):
    def area(self, side):
        return (side * side)
```

```
class Employee:
    def setNumberOfWorkingHours(self):
        self.numberOfWorkingHours = 40

    def displayNumberOfWorkingHours(self):
        print(self.numberOfWorkingHours)

class Trainee(Employee):
    def setNumberOfWorkingHours(self):
        self.numberOfWorkingHours = 45

    def resetNumberOfWorkingHours(self):
        super().setNumberOfWorkingHours()
```

```

empl = Employee()
empl.setNumberOfWorkingHours()
print('Number of working hours of employee: ', end=' ')
empl.displayNumberOfWorkingHours()

print("\n*****\n")

trainee = Trainee()
trainee.setNumberOfWorkingHours()
print('Number of working hours of trainee: ', end=' ')
trainee.displayNumberOfWorkingHours()

print("\n*****\n")

trainee.resetNumberOfWorkingHours()
print('Number of working hours of trainee after reset: ', end=' ')
trainee.displayNumberOfWorkingHours()

```

#### 4. Why is super() used ?

super() is used to access the methods of base class.

Example:

```

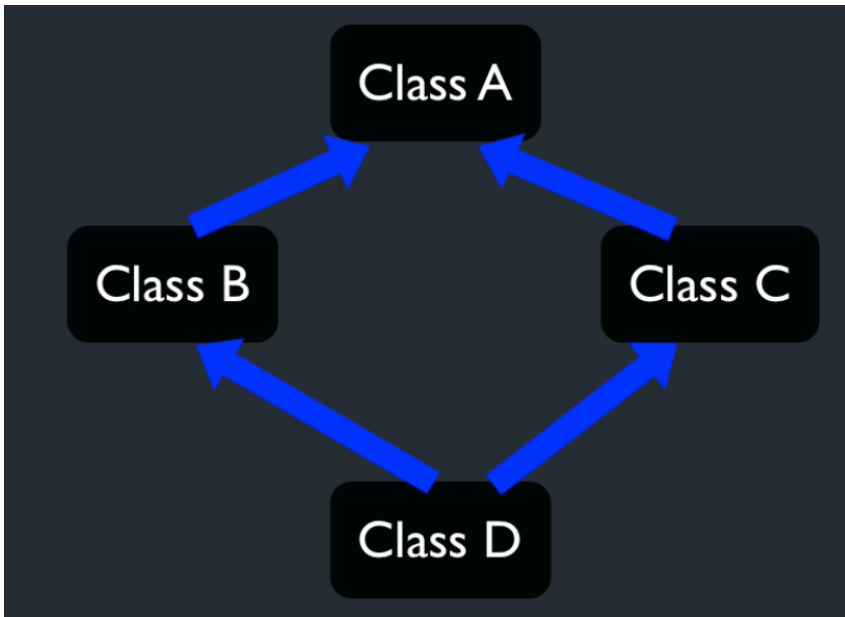
class BaseClass:
    def baseClassMethod():
        print("This is BaseClassOne")

class DerivedClass(BaseClass):
    def __init__(self):
        # calls the base class method

        super().baseClassMethod()

```

## The Diamond Shape Problem in Multiple Inheritance



```
diamond.py
1 class A:
2     def method(self):
3         print("This method belongs to class A")
4     pass
5
6 class B(A):
7     pass
8
9 class C(A):
10    pass
11
12 class D(B, C):
13    pass
14
15 d = D()
```

Case 1: Method will not be overridden in class B and class C

Case 2: Method will be overridden in class B but not in class C

Case 3: Method will be overridden in class C and not in class B

Case 4: Method will be overridden in both class B and class C

```
class A:
    def method(self):
        print("This method belongs to class A")
        pass

class B(A):
    def method(self):
        print("This method belongs to class B")
        pass

class C(A):
    def method(self):
        print("This method belongs to class C")
        pass

class D(C, B):
    pass

d = D()
d.method()
```

```
class A:
    def method(self):
        print('This method belongs to class A')

class B(A):
    def method(self):
        print('This method belongs to class B')

class C(A):
    def method(self):
        print('This method belongs to class C')

class D(C, B):
    def method(self):
        print('This method belongs to class D')

d = D()
d.method()
```

## Overloading an Operator

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

    def __add__(squareOne, squareTwo):
        return (4 * squareOne.side) + (4 * squareTwo.side)

squareOne = Square(10)
squareTwo = Square(10)
print("Sum of sides ob both squares = ", squareTwo + squareOne)
```

```
class Square:
    side = 4

    def area(self):
        print('Area of square: ', self.side * self.side)
```

```
class Rectangle:
    width = 5
    length = 10

    def area(self):
        print('Area of rectangle: ', self.width * self.length)
```

```
square = Square()
rectangle = Rectangle()
```

```
square.area()  
rectangle.area()
```

```
from abc import ABCMeta, abstractmethod
```

```
class Shape(metaclass=ABCMeta):  
    @abstractmethod  
    def area(self):  
        return 0
```

```
class Square(Shape):  
    side = 4  
  
    def area(self):  
        print('Area of square: ', self.side * self.side)
```

```
class Rectangle(Shape):  
    width = 5  
    length = 10  
  
    def area(self):  
        print('Area of rectangle: ', self.width * self.length)
```

```
square = Square()  
rectangle = Rectangle()
```

```
square.area()  
rectangle.area()
```

```
# shape = Shape()      # TypeError: Can't instantiate abstract class  
# Shape with abstract method area
```

## What is Polymorphism?

- The ability of an entity to be able to exist in more than one form

## What is overriding?

- Redefining the behavior of a base class method in a derived class.

What is operator overloading?

- Defining a special method for an operator within your class to handle the operation between the objects of that class is called operator overloading.

What is an abstract base class?

- A base class which consists of abstract methods that should be implemented in its derived class is called an abstract base class

★★

**Parent class** is the class being inherited from, also called base class.

**Child class** is the class that inherits from another class, also called derived class.

## Create a Parent Class

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

### Example

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:

## Example

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

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

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

## Example

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.

## Example

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:

## Example

```
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:

## Example

```
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

### Example

Add a property called `graduationyear` to the `Student` class:

```
class Student(Person):
    def __init__(self, fname, lname):
        super().__init__(fname, lname)
        self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the `Student` class when creating student objects. To do so, add another parameter in the `__init__()` function:

## Example

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

### Example

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)
```