**Python Class and Objects**

A class is a blueprint or a template for creating objects, providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The user-defined objects are created using the class keyword.

Or

A class is a group of related objects those have common properties and common behaviour

or

class is a tool which is used to bind variables & methods

Or

Class is a empty template which can not be used directly.It can be used through the object.

**Creating a Class:**

Let us now create a class using the class keyword

class Details:

name = "Rohan"

age = 20

**Creating an Object:**

Object is the instance of the class used to access the properties of the class Now lets create an object of the class.

obj1 = Details()  
**Printing now**   
class Details:

name = "Rohan"

age = 20

obj1 = Details()

print(obj1.name)

print(obj1.age)

class Person:

name = "name"

occupation = "Software Developer"

networth = 10

def info(self):

print(f"{self.name} is a {self.occupation}")

a = Person()

b = Person()

c = Person()

a.name = "Shubham"

a.occupation = "Accountant"

b.name = "Nitika"

b.occupation = "HR"

# print(a.name, a.occupation)

a.info()

b.info()

c.info()

**self parameter**

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.

It must be provided as the extra parameter inside the method definition.

Example:

class Details:

name = "Rohan"

age = 20

def desc(self):

print("My name is", self.name, "and I'm", self.age, "years old.")

obj1 = Details()

obj1.desc()

**Output:**

My name is Rohan and I'm 20 years old.  
**Constructors**

A constructor is a special method in a class used to create and initialize an object of a class. There are different types of constructors. Constructor is invoked automatically when an object of a class is created.

**Syntax of Constructor**

def \_\_init\_\_(self):

# initializations

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

**Types of Constructors in Python**

1.Parameterized Constructor

2.Default Constructor

**Parameterized Constructor in Python**

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

These arguments can be used inside the class to assign the values to the data members.

Example:

class Details:

def \_\_init\_\_(self, animal, group):

self.animal = animal

self.group = group

obj1 = Details("Crab", "Crustaceans")

print(obj1.animal, "belongs to the", obj1.group, "group.")

Output:

Crab belongs to the Crustaceans group.

**Default Constructor in Python**

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

class Details:

def \_\_init\_\_(self):

print("animal Crab belongs to Crustaceans group")

obj1=Details()

**Output:**

animal Crab belongs to Crustaceans group

class Person:

def \_\_init\_\_(self, name, occ):

print("Hey I am a person")

self.name = name

self.occ = occ

def info(self):

print(f"{self.name} is a {self.occ}")

a = Person("person1", "Developer")

b = Person("Person2", "HR")

a.info()

b.info()

# print(a.name)

# a.name = "person"

# a.occ = "HR"

# a.info()  
  
  
  
**Python Decorators**

Python decorators are a powerful and versatile tool that allow you to modify the behavior of functions and methods. They are a way to extend the functionality of a function or method without modifying its source code.

A decorator is a function that takes another function as an argument and returns a new function that modifies the behavior of the original function. The new function is often referred to as a "decorated" function. The basic syntax for using a decorator is the following:

@decorator\_function

def my\_function():

pass

The @decorator\_function notation is just a shorthand for the following code:

def my\_function():

pass

my\_function = decorator\_function(my\_function)

def greet(fx):

def mfx(\*args, \*\*kwargs):

print("Good Morning")

fx(\*args, \*\*kwargs)

print("Thanks for using this function")

return mfx

@greet

def hello():

print("Hello world")

@greet

def add(a, b):

print(a+b)

# greet(hello)()

hello()

# greet(add)(1, 2)

add(1, 2)

**Inheritance in python**

When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods,this is called as inheritance.

**Python Inheritance Syntax**

class BaseClass:

Body of base class

class DerivedClass(BaseClass):

Body of derived class

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

**Types of inheritance:**

Single inheritance

Multiple inheritance

Multilevel inheritance

Hierarchical Inheritance

Hybrid Inheritance

class Employee:

def \_\_init\_\_(self, name, id):

self.name = name

self.id = id

def showDetails(self):

print(f"The name of Employee: {self.id} is {self.name}")

class Programmer(Employee):

def showLanguage(self):

print("The default langauge is Python")

e1 = Employee("Rohan Das", 400)

e1.showDetails()

e2 = Programmer("Mohan Das", 4100)

e2.showDetails()

e2.showLanguage()

**Access Specifiers/Modifiers**

Access specifiers or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance.

Let us see the each one of access specifiers in detail:

**Types of access specifiers**

1.Public access modifier

2.Private access modifier

3.Protected access modifier

4.Public Access Specifier in Python

All the variables and methods (member functions) in python are by default public. Any instance variable in a class followed by the ‘self’ keyword ie. self.var\_name are public accessed.

Example:

class Student:

# constructor is defined

def \_\_init\_\_(self, age, name):

self.age = age # public variable

self.name = name # public variable

obj = Student(21,"Mohan")

print(obj.age)

print(obj.name)

Output:

21

Mohan

**Private Access Modifier**

By definition, Private members of a class (variables or methods) are those members which are only accessible inside the class. We cannot use private members outside of class.

In Python, there is no strict concept of "private" access modifiers like in some other programming languages. However, a convention has been established to indicate that a variable or method should be considered private by prefixing its name with a double underscore (\_\_). This is known as a "weak internal use indicator" and it is a convention only, not a strict rule. Code outside the class can still access these "private" variables and methods, but it is generally understood that they should not be accessed or modified.

**Example:**

class Student:

def \_\_init\_\_(self, age, name):

self.\_\_age = age # An indication of private variable

def \_\_funName(self): # An indication of private function

self.y = 34

print(self.y)

class Subject(Student):

pass

obj = Student(21,"Mohan")

obj1 = Subject

# calling by object of class Student

print(obj.\_\_age)

print(obj.\_\_funName())

# calling by object of class Subject

print(obj1.\_\_age)

print(obj1.\_\_funName())

Output:

AttributeError: 'student' object has no attribute '\_\_age'

AttributeError: 'student' object has no method '\_\_funName()'

AttributeError: 'subject' object has no attribute '\_\_age'

AttributeError: 'student' object has no method '\_\_funName()'

Private members of a class cannot be accessed or inherited outside of class. If we try to access or to inherit the properties of private members to child class (derived class). Then it will show the error.

**Name mangling**

Name mangling in Python is a technique used to protect class-private and superclass-private attributes from being accidentally overwritten by subclasses. Names of class-private and superclass-private attributes are transformed by the addition of a single leading underscore and a double leading underscore respectively.

class MyClass:

def \_\_init\_\_(self):

self.\_nonmangled\_attribute = "I am a nonmangled attribute"

self.\_\_mangled\_attribute = "I am a mangled attribute"

my\_object = MyClass()

print(my\_object.\_nonmangled\_attribute) # Output: I am a nonmangled attribute

print(my\_object.\_\_mangled\_attribute) # Throws an AttributeError

print(my\_object.\_MyClass\_\_mangled\_attribute) # Output: I am a mangled attribute

In the example above, the attribute \_nonmangled\_attribute is marked as nonmangled by convention, but can still be accessed from outside the class. The attribute \_\_mangled\_attribute is private and its name is "mangled" to \_MyClass\_\_mangled\_attribute, so it can't be accessed directly from outside the class, but you can access it by calling \_MyClass\_\_mangled\_attribute  
**Table of Contents**

Protected Access Modifier

In object-oriented programming (OOP), the term "protected" is used to describe a member (i.e., a method or attribute) of a class that is intended to be accessed only by the class itself and its subclasses. In Python, the convention for indicating that a member is protected is to prefix its name with a single underscore (\_). For example, if a class has a method called \_my\_method, it is indicating that the method should only be accessed by the class itself and its subclasses.

It's important to note that the single underscore is just a naming convention, and does not actually provide any protection or restrict access to the member. The syntax we follow to make any variable protected is to write variable name followed by a single underscore (\_) ie. \_varName.

Example:

**class Student:**

def \_\_init\_\_(self):

self.\_name = "Harry"

def \_funName(self): # protected method

return "CodeWithHarry"

class Subject(Student): #inherited class

pass

obj = Student()

obj1 = Subject()

# calling by object of Student class

print(obj.\_name)

print(obj.\_funName())

# calling by object of Subject class

print(obj1.\_name)

print(obj1.\_funName())

**Output:**

Harry

CodeWithHarry

Harry

CodeWithHarry  
Static methods in Python are methods that belong to a class rather than an instance of the class. They are defined using the @staticmethod decorator and do not have access to the instance of the class (i.e. self). They are called on the class itself, not on an instance of the class. Static methods are often used to create utility functions that don't need access to instance data.

class Math:

@staticmethod

def add(a, b):

return a + b

result = Math.add(1, 2)

print(result) # Output: 3

In this example, the add method is a static method of the Math class. It takes two parameters a and b and returns their sum. The method can be called on the class itself, without the need to create an instance of the class.

class Math:

def \_\_init\_\_(self, num):

self.num = num

def addtonum(self, n):

self.num = self.num +n

@staticmethod

def add(a, b):

return a + b

# result = Math.add(1, 2)

# print(result) # Output: 3

a = Math(5)

print(a.num)

a.addtonum(6)

print(a.num)

print(Math.add(7, 2))

variables

In Python, variables can be defined at the class level or at the instance level. Understanding the difference between these types of variables is crucial for writing efficient and maintainable code.

**Class Variables**

Class variables are defined at the class level and are shared among all instances of the class. They are defined outside of any method and are usually used to store information that is common to all instances of the class. For example, a class variable can be used to store the number of instances of a class that have been created.

class MyClass:

class\_variable = 0

def \_\_init\_\_(self):

MyClass.class\_variable += 1

def print\_class\_variable(self):

print(MyClass.class\_variable)

obj1 = MyClass()

obj2 = MyClass()

obj1.print\_class\_variable() # Output: 2

obj2.print\_class\_variable() # Output: 2

In the example above, the class\_variable is shared among all instances of the class MyClass. When we create new instances of MyClass, the value of class\_variable is incremented. When we call the print\_class\_variable method on obj1 and obj2, we get the same value of class\_variable.

**Instance Variables**

Instance variables are defined at the instance level and are unique to each instance of the class. They are defined inside the init method and are usually used to store information that is specific to each instance of the class. For example, an instance variable can be used to store the name of an employee in a class that represents an employee.

class MyClass:

def \_\_init\_\_(self, name):

self.name = name

def print\_name(self):

print(self.name)

obj1 = MyClass("John")

obj2 = MyClass("Jane")

obj1.print\_name() # Output: John

obj2.print\_name() # Output: Jane

In the example above, each instance of the class MyClass has its own value for the name variable. When we call the print\_name method on obj1 and obj2, we get different values for name.

**Summary**

In summary, class variables are shared among all instances of a class and are used to store information that is common to all instances. Instance variables are unique to each instance of a class and are used to store information that is specific to each instance. Understanding the difference between class variables and instance variables is crucial for writing efficient and maintainable code in Python.

It's also worth noting that, in python, class variables are defined outside of any methods and don't need to be explicitly declared as class variable. They are defined in the class level and can be accessed via classname.varibale\_name or self.class.variable\_name. But instance variables are defined inside the methods and need to be explicitly declared as instance variable by using self.variable\_name.

class Employee:

companyName = "Apple"

noOfEmployees = 0

def \_\_init\_\_(self, name):

self.name = name

self.raise\_amount = 0.02

Employee.noOfEmployees +=1

def showDetails(self):

print(f"The name of the Employee is {self.name} and the raise amount in {self.noOfEmployees} sized {self.companyName} is {self.raise\_amount}")

# Employee.showDetails(emp1)

emp1 = Employee("Harry")

emp1.raise\_amount = 0.3

emp1.companyName = "Apple India"

emp1.showDetails()

Employee.companyName = "Google"

print(Employee.companyName)

emp2 = Employee("Rohan")

Write a Library class with no\_of\_books and books as two instance variables. Write a program to create a library from this Library class and show how you can print all books, add a book and get the number of books using different methods. Show that your program doesnt persist the books after the program is stopped!

class Library:

def \_\_init\_\_(self):

self.noBooks = 0

self.books = []

def addBook(self, book):

self.books.append(book)

self.noBooks = len(self.books)

def showInfo(self):

print(f"The library has {self.noBooks} books. The books are")

for book in self.books:

print(book)

l1 = Library()

l1.addBook("Harry Potter1")

l1.addBook("Harry Potter2")

l1.addBook("Harry Potter3")

l1.showInfo()

**Snake Water Gun**

Snake, Water and Gun is a variation of the children's game "rock-paper-scissors" where players use hand gestures to represent a snake, water, or a gun. The gun beats the snake, the water beats the gun, and the snake beats the water. Write a python program to create a Snake Water Gun game in Python using if-else statements. Do not create any fancy GUI. Use proper functions to check for win

import random

def check(comp, user):

if comp ==user:

return 0

if(comp == 0 and user ==1):

return -1

if(comp == 1 and user ==2):

return -1

if(comp == 2 and user == 0):

return -1

return 1

comp = random.randint(0, 2)

user = int(input("0 for Snake, 1 for water and 2 for Gun:\n"))

score = check(comp, user)

print("You: ", user)

print("Computer: ", comp)

if(score == 0):

print("Its a draw")

elif (score == -1):

print("You Lose")

else:

print("You Won")