Object Oriented Programing: -

Python is multi-paradigm programing language. It supports different approaches. One of the popular approach is to solve a programming problem is by creating objects. This is known as object oriented programming.

The concept of OOP in python focuses on creating reusable code. This concept is called DRY (Don’t repeat yourselves) principle.

Class: -

A class is blueprint for objects. A class can be defined by using class keyword. We can create instances from class. An instance is a specific object created from particular class.

Object: -

An object is an instantiation of class. When class is defined, only description for the object is defined. Therefore, no memory or storage is allocated.

*class* Parrot:

  species = "Bird" #class attribute

  def \_\_init\_\_(self,name,age): #instance attribute

    self.name = name

    self.age = age

blu = Parrot("Blue",5)

print(f"{blu.name} is {blu.age} years old.")

The attributes are the characteristics of an object, which are defined inside the \_\_init\_\_ method of the class. It is the initializer method that is first run as soon as the object is created.

Methods: -

Methods are the functions which are declared inside the body of class. They are used to define the behaviour of an object.

*class* Parrot:

#Instance attribute

  def \_\_init\_\_(self,name,age):

    self.name = name

    self.age = age

#instance methods => These are called an instance method because they are called on an instance object.

  def sings(self,song):

    return f"{self.name} sings a {song} songs."

blu = Parrot("Blue",8)

print(blu.sings("Happy happy hu..."))

Inheritance: -

Inheritance is a way of creating new class by using property of existing class without modifying it. The class which we are created using existing class is called derived class or child class and that existing class is called base class.

*# Base or parent class*

*class* Bird:

  def \_\_init\_\_(self):

    print("Bird is ready")

  def whoIsThis(self):

    print("Bird")

  def swim(self):

    print("swim faster")

#child or derived class

class Penguin(Bird):

  def \_\_init\_\_(slef):

    super().\_\_init\_\_()

    print("Penguin is ready")

  def whoIsThis(self):

    print("Penguin")

  def run(self):

    print("Run faster")

peggy = Penguin()

peggy.whoIsThis()

peggy.swim()

The super() method allows us to run the \_\_init\_\_() method of parent class.

1. Single Inheritance: -

In single inheritance the derived class inherites the features of base class into child or derived class.

2. Multiple Inheritance: -

In multiple inheritance the derived class inherites the features from more than one base class.

3. Multilevel Inheritance: -

The derived class inherites the features from base class. Again the new child class inherites the features from previous derived class and base class as well. This is called multilevel inheritance.

Encapsulation: -

Using OOPs, we can restrict the access of methods and variables. Basically we are preventing the direct modification of data from outside of the class. This is called as Encapsulation.

e.g.

class Computer:

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

self.\_\_maxPrice = 1000

def sell(self):

print(f"The maximum selling price of computer is {self.\_\_maxPrice} Rs. /-.")

def setMaxPrice(self,price):

self.\_\_maxPrice = price

c = Computer()

c.sell()

c.\_\_maxPrice = 500

c.sell()

#using setter method to modify the \_\_maxPrice value

c.setMaxPrice(499)

c.sell()

=> In python the private attributes are defined using single or double underscore as prefix.

Polymorphism: -

Polymorphism means, the methods in parent class and methods in child class have same name is called polymorphism. The method which is written in base class does not suit the child class, in that case we need to re-implement that method in child class. The process of re-implementing the methods is called method overriding.

class Bird:

def intro(self):

print("There are different kind of birds.")

def flight(self):

print("Most of the birds can fly but some can not.")

class Parrot(Bird):

def flight(self):

print("Parrots can fly.")

bird\_obj = Parrot()

bird\_obj.flight()

bird\_obj.intro()

Constructor: -

Class have functions that are defined with double underscore and these function are called special functions. One of the popular function called \_\_init\_\_() method. This special function get called whenever object is instantiated from that class. This special function is called constructor. It is used to initialize all the varibles.

class Person:

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

self.name = name

self.age = age

self.city = city

def getAllData(self):

return f"I am {self.name} from {self.city}. I am {self.age} years olds."

p1 = Person("Joe",25,"Ville")

print(p1.getAllData())

Method resolution order (MRO): -

This concept is basically used in inheritance. It is used to searched the method in classes hierarchy. In python the MRO is from bottom to top and from left to right. This means that, first the method is searched in the object of class. If it is not found, then the method is searched in the immediate super class.

class A:

def method(self):

print("A.method() called")

class B:

pass

class C(B,A):

pass

c = C()

c.method()

Python operator overloading: -

Python airthmatic operators works fine for in-built classes. These same operator behaves differently with different types. Let us suppose to add two number, to merge two list and to concatenate two strings we used plus operator. But for adding two object, this operator raise an typeError. This features in python that allows the same operator to have different meaning according to the context is called operator overloading.

class Point:

def \_\_init\_\_(self,x = 0, y = 0):

self.x = x

self.y = y

p1 = Point(5,4)

p2 = Point(3,7)

print(p1 + p2)

Python overloading operator does not limit to airthmatic operators but it is also applicable for comparison operator.

class Point:

def \_\_init\_\_(self,x=0, y=0):

self.x = x

self.y = y

def \_\_str\_\_(self):

return f"({self.x},{self.y})"

def \_\_lt\_\_(self,other):

selfMag = self.x \*\* 2 + self.y \*\* 2

otherMag = other.x \*\* 2 + other.y \*\* 2

return selfMag < otherMag

p1 = Point(4,5)

p2 = Point(2,3)

print(p1 < p2)

Static methods: -

Static method is a function defined inside class. We don't need to pass the instance of the class to the methods as its first argument.The static method not accepting the object as parameter, so we can not access its attributes and modify the object as well.

class Student:

name = "Joe"

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

self.age = 20

@staticmethod

def toString():

print("student class")

stud = Student()

stud.toString()

Student.toString()

Class methods: -

It is a method that bounds to the class rather than object. It does not require creation of instance of class similar to static method. The difference between class method and static method are

1. Static method knows nothing about class, it deals only with the parameters.

2. Class method works with the class itself since it take class as parameter.

Dunder or Magic methods: -

Dunder methods are those mehods which starts and ends with double underscore. These methods are not invoked by ourself, it is invoked internally from class based on certain actions. Let us suppose we need to add two numbers, for addition of two number we used "+" operator. When python interpreter strikes with + operator it internally calls \_\_add\_\_() method. Thats why these dunder methods are also called as magic methods.

class Employee:

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

self.name = "Swati"

self.salary = 10000

def \_\_str\_\_(self):

return f"name = {self.name} salary = {self.salary}"

e = Employee()

print(e)

Exception Handling: -

Python use try and except keyword to handle the exception. Exception errors are those errors which comes into picture during execution of code.

try :

a = 5

b = "1"

print(a/b)

except:

print("Some error occurr")

print("Out of try and except blocks")

try:

print("Try block")

x = int(input("Enter a number : "))

y = int(input("Enter another number : "))

z = x / y

except :

print("Except zero division block")

print("Division by zero is not accepted")

else:

print("Else block")

print("Division is = ", z)

finally:

print("Final block")

x = 0

y = 0

Generators: -

Generator is a special type of function which allow us to create our own iterator function. It does not return single value, instead, it returns a sequence of values. In generator function, we used yield statement instead of return statement. The differnce between yield and return statement is yield statement pause the execution and stores the value internally but return statement terminates the function.

example:

def square\_of\_sequence(n):

for i in range(1, n+1):

yield i \*\* 2

seq\_sqr = square\_of\_sequence(5)

while True:

try:

print("Recieved on next() :", next(seq\_sqr))

except StopIteration:

break

In the above example square\_of\_sequence is generator function. In this function we have used a yield keyword instead of return. We need to create an iterator object from this generator function. So, here seq\_sqr is a iterator object. We can iterate over this object only once. When interpreter strikes with the yield keyword, it pauses the execution and send the first value of the iterator to the calling function. The execution again resumes when we called next function on the iterator object. The function finally terminates when next function strikes with stopIterationError.

Exception Handling: -

Syntax errors occurs when parser detects the incorrect statement. Exception error occurs when syntactically correct python code results in an error. e.g ZeroDivisionError, IndexError etc. The try and except block we can used to catch and handle the exceptions. As we know the syntactically correct code in python results in an error, it crashes the entire application if we are not handle that exception error. To avoid this kind of things we need to handle the exception errors.



example:

while True:

try:

age = int(input("Enter your age: "))

10/age

raise ValueError("Stop working")

except ZeroDivisionError:

print("Age must be greater than zero.")

else:

print("Thank you !")

break

finally:

print("I am done with it.")



If no exception is encountered in try block then and only then else block will execute. Finally block always executed regardless of try and except block.

\*\*Method overloading: -

Overloading means the ability of the function or operator that behaves in different way according to the parameter that are passed to the function or based on operands that the operator acts on. The advantage of overloading method is that, instead of writing multiple methods that differs slightly, we can write one method and overload it. It also improves the code clearity and eliminates the complexity. The process of calling the same method in different ways is called method overloading.

example:

class Greet:

def hello(self, name=None):

if name != None:

print(f"Hello {name}")

else:

print("Hello ")

# create instance of the class Greet

obj1 = Greet()

# one way of calling method without parameter

obj1.hello()

# Another way of calling method with parameter

obj1.hello("Ravindra")

1. Method overloading is compiled time polymorphism where as method overriding is run time polymorphism.

2. Inheritance may or may not be required in method overloading. In method overriding inheritance is always required.

3. With the help of method overloading we can add more to the behavour to the method. In method overriding we can change the behaviour of the existing methods.

Method overriding: -

Let's suppose we have inheritance in our code. We have base/parent class and child class. In the base class we have one method, that method we can access by child class. But whatever functionality doing by this base class method that we don't need for child class. In this case we need to re-implement the same method in child class as well. This process of re-implementing the method is called method overriding.

Example:

class Animal:

def walk(self):

print("This is base class")

class Dog(Animal):

def walk(self):

print("This is child class")

d = Dog()

d.walk()

Prerequisites required for method overriding:

1. Inheritance must be present. Method overriding should not be done within the class. We need to derived a child class from base class.

Operator overloading: -

Operator overloading means to provide the extended meaning beyond the predefined operational meaning. For example we used "+"operator to add two numbers, to concatenates two string or for merging two lists. Now as we can see the same operator behaves differently for object of different classes. This process is known as operator overloading.

Let us suppose we need to add two objects. For this we used "+" operator. When we add these objects using "+" operator it throws an error. Because python interpreter don't know how to add two objects using "+" operator. So, we need to define the function using the operator. This process is called as operator overloading. Python provides some special method or magic method for performing the operator overloading, which is automatically get invoked when it is associated with that operator.

example:

class Complex:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_add\_\_(self, U):

return self.x + U.x, self.y + U.y

obj1 = Complex(23, 21)

obj2 = Complex(21, 20)

print(obj1 + obj2)

Modules: -

The logical seperation of code is called modules. We used modules to break down the larger programs into small managable and organizied files. With the help of this we can provide reusabilty of code. Python has tons of standard modules which are present in the lib directory.

example:

# this is myModule module

person = {

"name": "Joe",

"age": 28,

"city": "Pune"

}

import myModule # importing myModule module

print(myModule)

print(myModule.person["age"])

A folder wich contains the modules is called package.

Built-in Modules: -

A] random => returns a sequence of random numbers.

1. random.random() => returns a random float number from 0 to 1.

2. random.randrange() => returns the random number between the given range

3. random.randint() => returns the random number between the given range

example:

import random

print(random.randint(1, 10))

print(random.choice([1, 2, 3, 4, 5]))

my\_list = [1, 2, 3, 4, 5, 6]

random.shuffle(my\_list)

print(my\_list)

B] sys => It provides functions and variables that we can used to manipulate the different parts of the python runtime environment.

example:

import sys

print(sys)

sys.argv

print("You entered: ", sys.argv[1], sys.argv[2], sys.argv[3])

Python package index: -

Python has a rich standard library that we can used immidately just importing it. But sometimes, you need a module that is not available in standard libray that we can get from python package index (pypi). To install this we can use command like pip install. This is a repositary that is written by third party and that we can use in our program just by installing it.

To create a virtual environment ==> python -m venv ven

Virtual environment: -

let us suppose in the current project we are using a library of latest version but in another project we need same library of older or some other version, in that case virtual env came into picture.

File I/O: -

Till now we are getting data from terminal and writing back to the terminal. This data is no longer available for us, it is volatile. If we need to store the data permenantly, here comes the concept of File I/O.

To open the file, python gives a function Open(<file name>, <mode>). Open function opens the file and return that file as object. This file is by default in read mode.

To read the open file, we used read() function on the returned object. To read line by line, we used readline.

Alway close the file after doing operation. There are two types of files text and binary.

example:

my\_file = open("test.txt")

print(my\_file)

print(my\_file.read()) # cursor is at the end of the file

my\_file.seek(0) # to keep the cursor at the 0th index

print(my\_file.read())

my\_file.seek(0)

print(my\_file.read())

# file read only once

print(my\_file.readline()) # to read only single line from file

print(my\_file.readline())

print(my\_file.readline())

print(my\_file.readlines()) # list of all lines

my\_file.close()

Access Mode of files: -

1. read ==> "r"

It opens the file only to read. The pointer exist at the begining of the file.

2. "r+" ==> It opens the files to both read and write mode. The file pointer is at the begining of the file. Does not create new file if not present with the same name.

3. "w" ==> It opens the file to write mode only. It overites the content in the file. If file does not exist with same name, it creates new file.

4. "a" ==> It opens the file only in the append mode. The pointer is present at the end of the previously written file. If file does not exist, it creates a new one.

example:

# with the "with" keyword no need to close the files.

with open("test.txt", mode="r+") as my\_file: # read and write r+

text = my\_file.write(":)")

print(text)

with open("test.txt", mode="a") as my\_file: # append

text = my\_file.write(":)")

print(text)

# with open("sad.txt", mode="r") as my\_file: # throw an error file not found

# text = my\_file.write(":(")

# print(text)

with open("sad.txt", mode="w") as my\_file: # throw an error file not found

text = my\_file.write(":(")

print(text)

Regular expression: -

It is sequence of charecter that defines search pattern. Python has module "re" to work with regular expression.

1. re.findall() ==> returns a list of all matches.

example:

import re

string = "search inside this of this text please"

pattern = re.compile("this")

# a = re.search("this", string)

a = pattern.search(string) #if find returns object if not returns None

b = pattern.findall(string)

c = pattern.fullmatch(string)

print(b)

print(a.span()) # returns tuples of start and end

print(a.start()) # returns the starting index

print(a.end()) # returns end index

print(a.group()) # returns the match string

print(c)

Email validation:

import re

pattern = re.compile(r"(^[a-zA-Z0-9+\_.-]+@[a-zA-Z0-9.-]+$)")

email = input("enter your email: ")

# If matches it return the object if not returns None

res = pattern.search(email)

if res:

print("Entered email is correct!")

else:

print("Entered email is incorrect!")

Password validation:

import re

pattern = re.compile(r"[a-zA-Z0-9@#$&!]{8,}\d")

password = input("Enter a password: ")

res = pattern.fullmatch(password)

if res:

print("Login successfully!")

else:

print("Password is incorrect.")