Day1

**OOP**

There are mainly 3 terms in object oriented programming. They are : class object and reference variable.

**Class**

A class is a blueprint or template for creating objects. It defines the attributes (data) and methods (functions) that the objects created from the class will have. For example: a class is a recipe, and the objects as the actual dishes made using that recipe. The essentials that are needed to make the food are reference variables. Inside class we can create variables (attributes, properties) or functions. Functions inside the class are called methods (behaviors, actions).

**Syntax:**

class className:

‘’’doc string’’’ #this is optional

#variables

#methods

**Object**

An object is an instance of a class. When a class is defined, no memory is allocated until an object is created. Objects are the actual data created using the class blueprint.

**Reference Variable**

A reference variable stores the memory address of an object. It refers to the object created in memory. In Python, when we create an object, the variable doesn’t store the object itself but a reference to that object.

Let’s see an example here:

|  |
| --- |
| class student:  '''This class is demo class. This class does nothing'''  def \_\_init\_\_(self, name, roll, marks):  self.name=name  self.roll= roll  self.marks= marks    def info(self):  print("hello my name is: ",self.name)  print("hello my roll no is: ",self.roll)  print("hello my marks is: ",self.marks)    s= student('Bhaskar',23,90) #s is reference variable and student() create the object  s.info()  output:  hello my name is: Bhaskar  hello my roll no is: 23  hello my marks is: 90 |

Important points:

* Objects are instances of classes.
* Reference variables store the address of objects, not the actual object itself.
* Multiple reference variables can point to the same object.
* Python uses automatic memory management (garbage collection), so when no reference variable points to an object, it is deleted from memory.

**Constructor in python**

In Python, a constructor is a special method used for initializing objects when they are created. The constructor is defined using the \_\_init\_\_() method in a class. It is automatically called when a new instance of the class is created, allowing us to set up the initial state of the object.

**Key Points About the Constructor (\_\_init\_\_()):**

* **Initialization**: It initializes the object's attributes.
* **Automatic Call**: It is automatically called whenever an object is instantiated.
* **Optional Arguments**: We can pass arguments to the constructor to initialize the object's attributes

**Syntax**:

class ClassName:

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

# Initialization of attributes

self.attribute1 = value1

self.attribute2 = value2

|  |
| --- |
| class Person:  # Constructor method  def \_\_init\_\_(self, name, age):  self.name = name # Assign the parameter 'name' to the object's attribute 'name'  self.age = age # Assign the parameter 'age' to the object's attribute 'age'  # Method to display information  def display\_info(self):  print(f"Name: {self.name}, Age: {self.age}")  # Creating an object of the Person class  person1 = Person("Bhaskar", 23)  person1.display\_info()  Output:  Name: Bhaskar, Age: 23 |

**Default Constructor (No Parameters):**

If we don't define the \_\_init\_\_() method, Python will provide a default constructor that doesn’t initialize any attributes.

|  |
| --- |
| class Animal:  pass  # Creating an object with the default constructor  dog = Animal() |

**Constructor with Default Values:**

We can also provide default values to parameters in the constructor, which will be used if no arguments are passed when the object is created.

|  |
| --- |
| class Car:  def \_\_init\_\_(self, brand="Toyota", model="Corolla"):  self.brand = brand  self.model = model  def display(self):  print(f"Car: {self.brand} {self.model}")  # Object with default values  car1 = Car()  car1.display()  Output:  Car: Toyota Corolla |

Python doesn’t support constructor overloading like some other programming languages (i.e., we can’t define multiple \_\_init\_\_() methods). When we define multiple \_\_init\_\_() then the most recent one will be called and other are ignored.

**Self-argument**

In Python, the self argument refers to the instance of the class itself and is used to access the attributes and methods of the class. It is a convention in Python to use self as the name of the first parameter in instance methods, although we can use any other name but it's highly recommended to stick with self for clarity and consistency. It represent the current object. First argument of constructor is always self.

**Key Points About self:**

1. **Refers to the Object**: self represents the instance of the class (the object) through which a method is called.
2. **Used in Instance Methods**: It is passed automatically to instance methods and allows the method to access the object’s attributes and other methods.
3. **Not Passed Manually**: When you call an instance method, Python automatically passes the self argument, we don’t need to provide it explicitly.
4. **Mandatory in Instance Methods**: In Python, self must be the first parameter in instance methods.

|  |
| --- |
| class Dog:  def \_\_init\_\_(self, name, breed):  self.name = name # Using self to refer to instance attributes  self.breed = breed  def bark(self):  print(f"{self.name} says bhawwww!") # Accessing the instance attribute using self  dog1 = Dog("kancho", "Golden Retriever")  dog1.bark()  Output:  kancho says bhawwww! |

**Why self is Needed:**

* Without self, we wouldn't be able to reference the object's attributes or methods.
* It allows each instance of the class to maintain its own set of data, preventing attribute conflicts between objects.

Self is a reference to the current instance of the class. It allows methods to access and modify the instance’s attributes and call other instance methods. It's a crucial part of object-oriented programming in Python, making it possible to handle object-specific data.

Day 2

**Types of variables**

In Object-Oriented Programming (OOP), variables are classified based on their scope, lifetime, and accessibility within a class and objects. In Python, the three main types of variables used in OOP are:

1. Instance Variables (Object level variables)
2. Static Variables (Class level variables)
3. Local Variables (Method level variables)
4. **Instance variables**

Instance variables are variables that are unique to each object (instance) of a class. They are defined inside the \_\_init\_\_() method (constructor) or any instance method using self. First argument of instance variable is always self .These variables are accessible only by the instance (object) they belong to. They exist as long as the object exists. Different instances can have different values for these variables.

|  |
| --- |
| class Employee:  def \_\_init\_\_(self, name, salary):  # Instance variables, unique to each object  self.name = name  self.salary = salary  def display\_info(self):  print(f"Name: {self.name}, Salary: {self.salary}")  # Creating two instances of Employee  emp1 = Employee("Bhaskar", 50000)  emp2 = Employee("Kabin", 60000)  # Accessing instance variables  emp1.display\_info()  emp2.display\_info()  Output:  Name: Bhaskar, Salary: 50000  Name: Kabin, Salary: 60000 |

In the example above, name and salary are instance variables, and each object (emp1, emp2) has its own copy of these variables.

1. **Static variables**

Static variables are variables that are shared across all instances of a class. Unlike instance variables, class variables have the same value for every object created from a class, unless explicitly changed. They are often used when a property is common to all objects. Shared among all instances of the class, meaning any change made to the class variable will affect all instances.

They exist as long as the class exists. It is accessed either by using the class name (ClassName.variable) or through an instance of the class.

|  |
| --- |
| class Employee:  company\_name = "MeroShare"  def \_\_init\_\_(self, name, salary):  # Instance variables  self.name = name  self.salary = salary  def display\_info(self):  print(f"Name: {self.name}, Salary: {self.salary}, Company: {Employee.company\_name}")  # Creating two instances  emp1 = Employee("Bhaskar", 50000)  emp2 = Employee("Kabin", 60000)  # Accessing class and instance variables  emp1.display\_info()  emp2.display\_info()  Output:  Name: Bhaskar, Salary: 50000, Company: MeroShare  Name: Kabin, Salary: 60000, Company: MeroShare  🡨---------------------------------------------------------🡪  # Changing the class variable  Employee.company\_name = "CodeWorks"  # Both instances reflect the change  emp1.display\_info()  emp2.display\_info()    Output:  Name: Bhaskar, Salary: 50000, Company: CodeWorks  Name: Kabin, Salary: 60000, Company: CodeWorks |

In this example, company\_name is a class variable that is shared by all instances. Changing the class variable affects both emp1 and emp2 since the variable is shared across the class.

1. **Local variables**

Local variables are variables that are defined within methods and are only accessible inside those methods. They are created when the method is called and destroyed when the method returns. Local variables are not accessible outside the method in which they are defined. They are defined inside methods, without self. They exist only during the execution of the method and are destroyed once the method completes. They can be accessed directly within the method, no self is needed. It is used to store temporary data or perform operations within a method.

|  |
| --- |
| class Employee:  def \_\_init\_\_(self, name, salary):  self.name = name # Instance variable  self.salary = salary # Instance variable  def calculate\_bonus(self, percentage):  # Local variable  bonus = self.salary \* percentage / 100 # Temporary, method-specific variable  return bonus  emp = Employee("Bhaskar", 50000)  print(emp.calculate\_bonus(10))  Output:  5000 |

In this example, bonus is a local variable inside the calculate\_bonus method. It only exists while the method is executing and is not accessible outside the method.

**Types of methods**

In Object-Oriented Programming (OOP), methods are functions defined within a class to define the behavior of objects. In Python, there are three main types of methods based on how they interact with class and object data.

1. Instance Methods
2. Class Methods
3. Static Methods
4. **Instance Methods**

Instance methods are the most common type of methods in Python. They work on instance variables (specific to an object) and usually interact with the data stored within the instance of the class. To access instance variables, these methods require self as the first parameter.

Instance variables are always declared by using reference variables.

**Characteristics:**

* **Definition**: Inside the class, with self as the first parameter.
* **Scope**: They operate on the instance of the class.
* **Access**: They can access and modify instance variables via self and also call other instance methods.
* **Purpose**: To manipulate object-specific data, like updating object attributes or performing operations based on instance-specific data.

|  |
| --- |
| class Employee:  def \_\_init\_\_(self, name, salary):  self.name = name # Instance variable  self.salary = salary # Instance variable  def display\_info(self):  # Instance method accessing instance variables  print(f"Name: {self.name}, Salary: {self.salary}")  # Creating an instance of Employee  emp = Employee("Bhaskar", 50000)  emp.display\_info()  Output:  Name: Bhaskar, Salary: 50000 |

Here, display\_info() is an instance method. It uses self to access the instance variables name and salary.

Instance methods work with instance variables. They always have self as the first parameter to refer to the instance. They can modify the state of the object and access other instance methods.

1. **Class Methods**

Class methods are methods that are bound to the class and not the object instance. They take cls as the first parameter, which refers to the class itself rather than an instance. Class methods can access and modify class variables, but they cannot modify instance variables directly. They are defined using the @classmethod decorator.

**Characteristics:**

* + **Definition**: Inside the class, with cls as the first parameter.
  + **Scope**: They operate on the class rather than the instance.
  + **Access**: They can access class variables and modify the class's state. However, they cannot access or modify instance variables.
  + **Purpose**: To perform operations that apply to the class as a whole, rather than specific instances.

|  |
| --- |
| class Employee:  company\_name = "TechCorp" # Class variable  def \_\_init\_\_(self, name, salary):  self.name = name  self.salary = salary  @classmethod  def change\_company(cls, new\_name):  # Class method modifying a class variable  cls.company\_name = new\_name  # Creating an instance of Employee  emp = Employee("Bhaskar", 50000)  print(emp.company\_name)  Output:  TechCorp  # Calling the class method to modify the class variable  Employee.change\_company("CodeWorks")  print(emp.company\_name)  Output:  CodeWorks |

In this example, change\_company() is a class method that modifies the class variable company\_name. It doesn't deal with specific instances but rather modifies the class's shared state.

Class methods work with class variables. It uses the @classmethod decorator and take cls as the first argument to refer to the class itself. It can modify class-level state and affect all instances of the class.

1. **Static Methods**

Static methods are methods that don't require access to instance variables or class variables. They are independent of both class and instance. Static methods are used for utility functions that don't need to interact with the class or its instances. They are defined using the @staticmethod decorator. They are basically used for temporary use.

**Characteristics:**

* **Definition:** Inside the class, without self or cls as parameters.
* **Scope:** They do not operate on the instance or the class.
* **Access:** They cannot access or modify instance variables or class variables. Static methods are entirely self-contained.
* **Purpose:** To perform actions or calculations that don’t depend on instance or class-level data. They are utility functions, often used for performing generic tasks.

|  |
| --- |
| class MathOperations:  @staticmethod  def add(a, b):  return a + b  @staticmethod  def subtract(a, b):  return a - b  # Calling static methods  print(MathOperations.add(10, 5))  print(MathOperations.subtract(10, 5))  Output:  15  5 |

In this example, add() and subtract() are static methods. They don't require access to any instance or class-level data and work solely with the provided arguments.

Static methods are independent of class and instance data. They use the @staticmethod decorator and don’t take self or cls as parameters. Ideal for utility functions or helper methods that don’t rely on the state of the class or objects.

* **Instance methods** are the most common and are used to manipulate the data unique to each object (instance).
* **Class methods** allow us to manipulate class-level data and are useful when we need to work with class variables or the class itself.
* **Static methods** are utility functions that neither rely on class nor instance data, providing a way to perform operations that don't depend on the state of an object or the class.

These three types of methods provide flexibility in how you interact with class and instance data, making OOP in Python more powerful and modular.

A banking system Program:

|  |
| --- |
| import sys  class Customer:      '''this is Bank system'''      bankname='ACEM'      def \_\_init\_\_(self,name, balance=0.0):          self.name= name          self.balance= balance        def deposit(self, amt):          self.balance= self.balance+ amt          print('New balance after deposit is: ', self.balance)        def withdraw(self,amt):          if amt>self.balance:              print('Insuffient balance. Please deposit first')              sys.exit()          else:              self.balance= self.balance-amt              print('New balance after withdrawl is: ', self.balance)  print()  print('#'\*50)  print("Welcome to ACEM Bank")  print('#'\*50)  name= input('Enter your name: ')  c= Customer(name)  while True:      print('d- Deposit \n w- Withdrawal \n e- Exit')      option = input('Enter your option: ')      if option.lower() == 'd':          amt= int(input('Enter the amount that you want to deposit: '))          c.deposit(amt)      elif option.lower() == 'w':          amt= int(input('Enter the amount that you want to withdraw: '))          c.withdraw(amt)      elif option.lower() == 'e':          print('Thank you for using our service.')          sys.exit()      else:          print('Invalid option. Please select correct option')  output:  ##################################################  Welcome to ACEM Bank  ##################################################  Enter your name: Bhaskar  d- Deposit  w- Withdrawal  e- Exit  Enter your option: d  Enter the amount that you want to deposit: 5000  New balance after deposit is: 5000.0  d- Deposit  w- Withdrawal  e- Exit  Enter your option: w  Enter the amount that you want to withdraw: 2000  New balance after withdrawl is: 3000.0  d- Deposit  w- Withdrawal  e- Exit  Enter your option: e  Thank you for using our service. |

Day 3

Another example:

|  |
| --- |
| class student:      def \_\_init\_\_(self, name , marks):          self.name= name          self.marks= marks        def display(self):          print('hi ',self.name)          print('your marks is ',self.marks)        def grade(self):          if self.marks>=60:              print('fist division')          elif self.marks>=50:              print('second division')          else:              print('failed')    n= int(input('enter number of stds: '))  for i in range(n):      name= input('enter your name: ')      marks= int(input('enter your marks: '))      s= student(name, marks)      s.display()      s.grade()      print()  output:  enter number of stds: 3  enter your name: bhaskar  enter your marks: 99  hi bhaskar  your marks is 99  fist division  enter your name: kabin  enter your marks: 96  hi kabin  your marks is 96  fist division  enter your name: ram  enter your marks: 21  hi ram  your marks is 21  failed |

We have 2 special instance method:

* setter()/mutator
* getter()/accessor

In Python, getters and setters are special instance methods used to access and modify the values of instance variables (attributes) of a class. They provide controlled access to an object's attributes, ensuring encapsulation and data protection, which are key principles in object-oriented programming (OOP).

**Getter (accessor)**

The getter method, also known as an accessor, allows reading or accessing the value of a private or protected instance variable. It provides a controlled way to retrieve the value of an attribute without directly exposing it to the user.

It helps maintain encapsulation by hiding the actual data from external access.

**Setter (Mutator)**

The setter method, also known as a mutator, allows modifying or setting the value of a private or protected instance variable. It provides a controlled way to update the value of an attribute while ensuring that only valid data is set.

It is used to validate or modify the data before assigning it to an attribute.

**Why Use Getters and Setters?**

* **Encapsulation**: By using getters and setters, the internal state (attributes) of an object is hidden from the outside world. Direct access to attributes is avoided, ensuring that the internal structure can be changed without affecting the external code that uses the object.
* **Validation**: Setters can validate or enforce constraints on the values being assigned to the attributes, ensuring that only valid data is set.
* **Control**: Getters and setters allow for greater control over how instance variables are accessed and modified, making the code more robust and flexible.

Here is an example code for getter and setter:

|  |
| --- |
| class student:      def setName(self,name):          self.name= name        def getName(self):          return self.name        def setMarks(self,marks):          self.marks= marks        def getMarks(self):          return self.marks  n= int(input('enter number of stds: '))  for i in range(n):      s= student()      name= input('enter your name: ')      s.setName(name)      marks= int(input('enter your marks: '))      s.setMarks(marks)      print('hi ',s.getName())      print('your marks is ',s.getMarks())      print()  output:  enter number of stds: 3  enter your name: bhaskar  enter your marks: 98  hi bhaskar  your marks is 98  enter your name: kabin  enter your marks: 78  hi kabin  your marks is 78  enter your name: dipesh  enter your marks: 97  hi dipesh  your marks is 97 |

Here, the class student contains two **getter** and two **setter** methods to manage the attributes name and marks of a student. The **setter** methods (setName() and setMarks()) are used to assign values to the instance variables self.name and self.marks. For example, when s.setName(name) is called, it sets the student's name to the provided input. Similarly, s.setMarks(marks) sets the student's marks. The **getter** methods (getName() and getMarks()) are used to retrieve the values of these instance variables. For instance, s.getName() returns the value of self.name, allowing the program to access and print the student’s name. This approach provides encapsulation, ensuring that the attributes name and marks are accessed and modified only through controlled methods, rather than directly, which maintains the integrity of the data.

**Classmethod**

The @classmethod decorator in Python is used to define a method that belongs to the class rather than to an instance of the class. A **class method** can be called on the class itself, and it takes the class as its first argument (conventionally named cls), rather than an instance (which is what the self argument is used for in instance methods).

Class methods are useful when we need to access or modify class-level data (i.e., data shared across all instances of the class) or when we need to create alternative constructors.

Unlike instance methods that take self as the first parameter (which refers to the instance), class methods take cls (which refers to the class). Class methods can access and modify class-level data, and are often used for operations that apply to the class as a whole, rather than individual objects.

|  |
| --- |
| class ACEM:      department=4      @classmethod      def work(cls, name):          print(f'Acem has {cls.department} departments')  ACEM.work('computer')       #classmethod so access through class name  Output:  Acem has 4 departments |

**Inner classes**

In Python, an inner class (or nested class) is a class that is defined inside another class. Inner classes are useful when we want to logically group classes that are only used in one place or when we want to encapsulate the functionality of the inner class within the outer class. Inner classes can help in organizing code and enforcing encapsulation by limiting the scope of the inner class.

Inner classes are often used to model components or logical groupings that are closely related to the outer class. The inner class can access the outer class's members (if needed) and vice versa, provided they have the right scope and permissions. To create an instance of an inner class, we first need to instantiate the outer class, and then the inner class.

|  |
| --- |
| class outer:      def \_\_init\_\_(self):          print("outer class")      class inner:          def \_\_init\_\_(self):              print('inner classes')          def m(self):              print('inner class method')  o= outer()  i= o.inner()  i.m()  Output:  outer class  inner classes  inner class method |

|  |
| --- |
| class outer:      def \_\_init\_\_(self):          print("outer object is created")      class inner:          def \_\_init\_\_(self):              print('inner object is created')          class InnerInner:              def \_\_init\_\_(self):                  print("innerinner object is created")              def m(self):                  print('inner inner class')    o= outer()  i= o.inner()  ii= i.InnerInner()  ii.m()  output:  outer object is created  inner object is created  innerinner object is created  inner inner class |

Day 4

**Garbage collection/ Garbage collector**

Garbage collection is the process of identifying and deallocating memory that is no longer reachable by any part of a program. It helps in freeing up memory that was allocated to objects that are no longer in use, making it available for other objects.

Without garbage collection, programs could suffer from memory leaks, where unused objects remain in memory, leading to inefficient memory usage and potentially causing the program to run out of memory over time.

Those object that have zero reference variable then they are useless object and are eligible for garbage collection.

The garbage collector in Python is a part of the runtime system that handles memory cleanup. It runs automatically in the background to free memory used by unreachable objects. However, Python also provides a module called gc that allows developers to interact with and control the garbage collection process manually. We can enable or disable the garbage collector. When there is sufficient amount of storage then only garbage collector is disabled but it is strongly not recommended to disable garbage collector.

|  |
| --- |
| import gc  # Disable automatic garbage collection  gc.disable()  # Force garbage collection manually  gc.collect()  # Enable garbage collection  gc.enable() |

**Destructor**

A **destructor** is a special method that is called when an object is destroyed or goes out of scope. Its primary purpose is to perform cleanup operations, such as releasing resources like memory, file handles, or database connections, before the object is completely removed from memory.

In Python, destructors are defined using the \_\_del\_\_() method, which is automatically called when an object is about to be destroyed by the garbage collector.

Destructor does not destroy object, garbage collector destroy it. But destructor deallocate the initialize memory, database or network.

**When is the Destructor Called?**

The \_\_del\_\_() method is invoked under these circumstances:

* **When the object’s reference count reaches zero**: This happens when there are no variables or objects referencing the instance.
* **When the program ends**: If the object still exists at the end of the program, the destructor will be called before the program exits.

However, it's important to note that Python's garbage collector handles the cleanup of objects, and \_\_del\_\_() is not always guaranteed to run immediately when an object goes out of scope.

In python per user constructor and per user destructor is called

|  |
| --- |
| class test:      def \_\_init\_\_(self):          print("this is constructor")      def \_\_del\_\_(self):          print("this is destrutor")    l=[test(),test(),test()]  output:  this is constructor  this is constructor  this is constructor  this is destrutor  this is destrutor  this is destrutor |

**Difference between constructor and destructor**:

|  |  |  |
| --- | --- | --- |
| **Features** | **Constructor** | **Destructor** |
| Purpose | Initializes a new object and allocates memory for it. | Cleans up and deallocates resources when an object is destroyed. |
| Method name | \_\_init\_\_(self) | \_\_del\_\_(self) |
| Called when | Automatically called when an object is created. | Automatically called when an object is about to be destroyed or goes out of scope. |
| Arguments | Can accept arguments to initialize the object’s attributes. | Does not take any arguments (except self). |
| Action | Sets up the initial state of the object (e.g., initializing attributes). | Cleans up any resources the object is using (e.g., closing files, releasing memory). |
| Number of Times Called | Called once when the object is created. | Can be called multiple times, but typically occurs once when the object is destroyed. |

**Using Members of one class inside another class**

We have 2 possibilities to use variable of one class in another. They are:

* HAS-A relation (composition)
* IS-A relation (inheritance)

**Composition method (HAS-A)**

Making a big object by combining many small object then it is called composition object. The Has-A relationship represents a composition or aggregation relationship between objects. In this relationship, one class contains another class as a member, implying that the class has another object as part of its attributes. This does not imply inheritance but rather an association where one object "owns" or "uses" another object.

A class has a reference to another class or object. This relationship is modeled using composition or aggregation.

|  |
| --- |
| class engine:      a= 10      def \_\_init\_\_(self):          self.b= 20      def m1(self):          print("This is engine. ")    class car:      def \_\_init\_\_(self):          self.engine= engine()      def m2(self):          print('Car object using engine object')          print(self.engine.a)          self.engine.m1()          print(self.engine.b)  c= car()  c.m2()  output:  Car object using engine object  10  This is engine.  20 |

Another example:

|  |
| --- |
| class car:      def \_\_init\_\_(self,name,model,color):          self.name= name          self.model= model          self.color= color      def getinfo(self):          print(f'Car name : {self.name}\n Car model: {self.model}\n Car color: {self.color}')    class employee:      def \_\_init\_\_(self,ename, eno, car):          self.ename= ename          self.eno=eno          self.car= car      def empinfo(self):          print("Employee name: ",self.ename)          print("Employee number: ",self.eno)          print("Employee car information: ")          self.car.getinfo()    c= car('Tesla','V2','Red')  e= employee('Bhaskar',23,c)  e.empinfo()  output:  Employee name: Bhaskar  Employee number: 23  Employee car information:  Car name : Tesla  Car model: V2  Car color: Red |

Day 5

**Inheritance (IS-A relationship)**

The process of creating new classes based on some existing classes. In Python, an IS-A relationship refers to inheritance in Object-Oriented Programming (OOP), where one class (the child or derived class) inherits attributes and behaviors (methods) from another class (the parent or base class). This establishes a hierarchical relationship, indicating that the child class "is a" type of the parent class. In Python, inheritance allows the child class to inherit methods and properties of the parent class, making it a subtype of the parent class.

**Key Features of IS-A Relationship (Inheritance):**

1. **Reusability**: The child class can reuse code from the parent class, promoting code reuse.
2. **Extensibility**: The child class can extend or override the functionality of the parent class.
3. **Hierarchy**: The IS-A relationship creates a hierarchy where the child class is a specialized form of the parent class.
4. We always create the object of child class.

|  |
| --- |
| class P:      def m(self):          print('this is m method of class P')    class C(P):      def m1(self):          print('this is m1 metho of class C')  c= C()  c.m()  c.m1()  output:  this is m method of class P  this is m1 metho of class C |

|  |
| --- |
| class P:      a=10      #instance method      def \_\_init\_\_(self):          self.b=20      def m1(self):          print('this is instance method.')      @classmethod      def m2(self):          print('this is class method')      @staticmethod      def m3():          print('this is static method')  class C(P):      pass  c= C()  print(c.a)  c.m1()  c.m2()  c.m3()  output:  10  this is instance method.  this is class method  this is static method |

|  |
| --- |
| class person:      def \_\_init\_\_(self,name,age):          self.name=name          self.age=age      def eat(self):          print("Person eat momo")    class employee(person):      def \_\_init\_\_(self, name, age,eno,esal):          super().\_\_init\_\_(name,age)          self.eno=eno          self.esal=esal      def work(self):          print("Employee can work")      def empinfo(self):          print("employee name: ",self.name)          print("employee age: ",self.age)          print("employee employee number: ",self.eno)          print("employee salary: ",self.esal)  e= employee('Ram',23,'e01',50000)  e.eat()  e.work()  e.empinfo()  output:  Person eat momo  Employee can work  employee name: Ram  employee age: 23  employee employee number: e01  employee salary: 50000 |

IS-A and HAS-A relationship in a single example.

|  |
| --- |
| class car:      def \_\_init\_\_(self,name,model,color):          self.name= name          self.model= model          self.color= color      def getinfo(self):          print(f'Car name : {self.name}\n Car model: {self.model}\n Car color: {self.color}')    class person:      def \_\_init\_\_(self,name,age):          self.name=name          self.age=age      def eatanddrink(self):          print("Person eat momo and drink coke")  class employee(person):      def \_\_init\_\_(self, name, age,eno,esal,car):          super().\_\_init\_\_(name, age)          self.eno=eno          self.esal=esal          self.car=car      def work(self):          print('employee can work')      def empinfo(self):          print("employee name: ",self.name)          print("employee age: ",self.age)          print("employee employee number: ",self.eno)          print("employee salary: ",self.esal)          print()          print('car details is:')          self.car.getinfo()  c=car('Tesla','V2','Black')  e= employee('Bhaskar',23,'e01',100000,c)  e.empinfo()  e.work()  e.eatanddrink()  output:  employee name: Bhaskar  employee age: 23  employee employee number: e01  employee salary: 100000  car details is:  Car name : Tesla  Car model: V2  Car color: Black  employee can work  Person eat momo and drink coke |

Composition and aggregation

Types of inheritance

* Single inheritance
* Multi-level inheritance
* Hierarchical inheritance
* Multiple inheritance
* Hybrid inheritance
* Cyclic inheritance

Single inheritance

|  |
| --- |
| class P:      def m1(self):          print('Parent class')    class C(P):      def m2(self):          print('child class')    c= C()  c.m1()  c.m2()  output:  Parent class  child class |

Multilevel Inheritance

|  |
| --- |
| class P:      def m1(self):          print('Parent class')    class C(P):      def m2(self):          print('child class')  class CC(C):      def m3(self):          print("Child child class")    cc=CC()  cc.m1()  cc.m2()  cc.m3()  output:  Parent class  child class  Child child class |

Hierarchical inheritance

|  |
| --- |
| class P:      def m1(self):          print('Parent class')    class C1(P):      def m2(self):          print('First child class')  class C2(P):      def m3(self):          print("Second child class")  c=C1()  c.m1()  c.m2()  cc=C2()  cc.m1()  cc.m3()  output:  Parent class  First child class  Parent class  Second child class |

Multiple Inheritance

|  |
| --- |
| class P1:      def m1(self):          print('First Parent class')    class P2():      def m2(self):          print('2nd Parent class')  class C(P1,P2):      def m3(self):          print(" child class")  c=C()  c.m1()  c.m2()  c.m3()  output:  First Parent class  2nd Parent class  child class |

Hybrid Inheritance

**MRO (Method Resolution Order)**

In hybrid inheritance the method resolution order is based on MRO algorithm also called as C3 algorithm. Samuele Pedroni created this algorithm.

If c1 c2 c3 are the parent classes of the class X then MRo(X)= x+ merge(MRO(c1), MRO(c2) ,MRO(c3)….c1c2c3).

Head= c1 is head

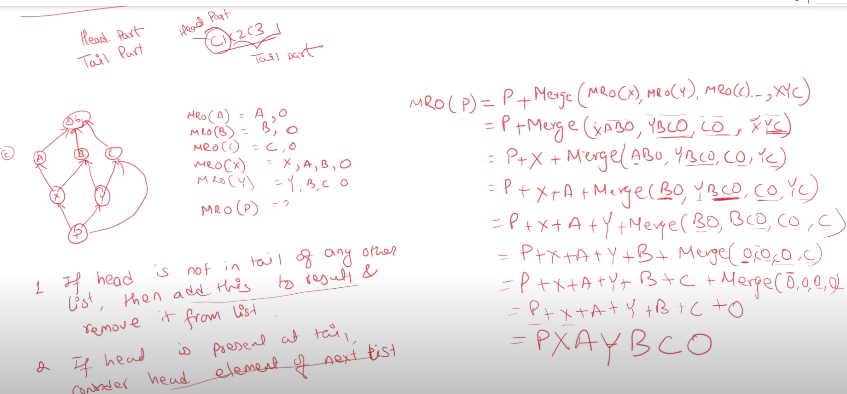
Tail= other except c1 is tail.

MRO(X) = X+ merge(MRO(p1), MRO(p2),…..,parent list)

Rules:

If head is not in tail of any other list and add this and remove this from list

If head is not present at tail, consider head element of next list.



**Cyclic Inheritance:**

No programming language support cyclic inheritance. Why to do this kind of inheritance when there is no meaning of this.

Example: Class A(A)

**Super Method:**

It is a built-in method of python. It access to methods and properties of a parent or superclass from a child or subclass.

|  |
| --- |
| class P:      def m1(self):          print("parent method")  class C(P):      def m1(self):          super().m1()          print("child method")  c= C()  c.m1()  output:  parent method  child method |

If the name of method is same in both parent and child class then super method is needed.

|  |
| --- |
| class P:      a= 10      def \_\_init\_\_(self):          self.b=20          print("parent constructor is called")      def m1(self):          print("parent class instance method is called")      @classmethod      def m2(cls):          print("parent class class method is called")      @staticmethod      def m3():          print("parent class static method ")    class C(P):      a=555      def \_\_init\_\_(self):          self.b=666          print("child constructor is called")          super().\_\_init\_\_()  #call the parent class constructor          super().m1()          super().m2()          super().m3()  c= C()  print(c.b)  output:  child constructor is called  parent constructor is called  parent class instance method is called  parent class class method is called  parent class static method  20 |

|  |
| --- |
| class Person:      def \_\_init\_\_(self,name,age):          self.name= name          self.age= age      def display(self):          print('Name:', self.name)          print('Age: ',self.age)  class Student(Person):      def \_\_init\_\_(self, name, age,roll,marks):          super().\_\_init\_\_(name, age)          self.roll=roll          self.marks=marks        def display(self):          super().display()          print('ROll:', self.roll)          print('Marks: ',self.marks)    s= Student('Bhaskar',20,101,99)  s.display()  output:  Name: Bhaskar  Age: 20  ROll: 101  Marks: 99 |

|  |
| --- |
| class A:      def m1(self):          print('A class')    class B(A):      def m1(self):          print('B class')  class C(B):      def m1(self):          print("C class")    class D(C):      def m1(self):          print("D class")    class E(D):      def m1(self):          super().m1()          A.m1(self)          super(C,self).m1()  #Super method of C is called so m1 of B will be printed          print("E class")  e= E()  e.m1()  output:  D class  A class  B class  E class |

**Case I**: From child class we are not allowed to access parent’s class instance variable by super(), compulsory we should use self only. But for accessing class and static variable we can use super() method.

|  |
| --- |
| class P:      a=10      def \_\_init\_\_(self):          self.b=20  class C(P):      def m1(self):          print(super().a)          print(self.b)  c= C()  c.m1()  output:  10  20 |

**Case II**: From child class constructor and instance method, we can access parent class instance, static and class method by using super().

|  |
| --- |
| class P:      def \_\_init\_\_(self):          print('parent class constructor')      # instance method      def m1(self):          print("parent class instance method")      @classmethod      def m2(cls):          print("parent class class method")      @staticmethod      def m3():          print("parent class static method")    class C(P):      def \_\_init\_\_(self):          super().\_\_init\_\_()          super().m1()          super().m2()          super().m3()      def m1(self):          super().\_\_init\_\_()          super().m1()          super().m2()          super().m3()  c= C()  c.m1()  output:  parent class constructor  parent class instance method  parent class class method  parent class static method  parent class constructor  parent class instance method  parent class class method  parent class static method |

**Case III**: From child class class method we cannot access parent class instance methods and constructors by using super directly. But we can access class and static method.

|  |
| --- |
| class P:      def \_\_init\_\_(self):          print('parent class constructor')      # instance method      def m1(self):          print("parent class instance method")      @classmethod      def m2(cls):          print("parent class class method")      @staticmethod      def m3():          print("parent class static method")    class C(P):      @classmethod      def m1(cls):          #we cannot access instance and constructor          # super().\_\_init\_\_()       #we cannnot access          # super().m1()             #we cannot access          super().m2()          super().m3()          super(C, cls).\_\_init\_\_(cls) #it calls constructor from parent class through child class          super(C,cls).m1(cls)  c= C()  c.m1()  output:  parent class constructor  parent class class method  parent class static method  parent class constructor  parent class instance method |

Case IV: In child class static method we are not allowed to use super() generally, but we can use in a special way.

|  |
| --- |
| class P:      def \_\_init\_\_(self):          print('parent class constructor')      # instance method      def m1(self):          print("parent class instance method")      @classmethod      def m2(cls):          print("parent class class method")      @staticmethod      def m3():          print("parent class static method")    class C(P):      @staticmethod      def m1():          super(C,C).m2()          super(C,C).m3()  c= C()  c.m1()  output:  parent class constructor  parent class class method  parent class static method |

Polymorphism

Poly means many and morphe means forms i.e. that type of expression which have many forms.

Example: ‘+’ operator adds 5+5=10 and also it does string concatenation as ‘hello’ +’world’ = helloworld

Overloading:

If one expression has more than one work then it is said to be overloading.

Operator overloading: The work that we do in basic data type, if we do that in object then it is called overloading.

Differ in Number of argument or type of argument then it is said to be method overloading. There is no method overloading in python but we can explicitly overload method.

There is no constructor overloading in python

Overriding

Duck typing: If any animal swims like a duck, walk like a duck, and quack like a duck, then animal🡪 duck. Python concentrate on the behavior of the animal rather than the animal then it is said to be duck typing in python.

Abstract Method

Abstract means not completeness. It only gives the outline. It is a summary or a guide. When we don’t know the complete implementation but later we want to used it then abstract method Is used. We need a decorator to make abstract as @absstractmethod which should be imported from abc model.

|  |
| --- |
| from abc import abstractmethod  class Test:      @abstractmethod      def m1():          pass |

If we have the abstract method in parent class then it should be made complete from every child class that inherit parent class.

Note: Abstract class with abstract method instantiation is not possible.

|  |
| --- |
| class vehicle(ABC):      @abstractmethod      def noofwheels(self):          pass    class bus(vehicle):      def noofwheels(self):          return 6  b= bus()  print(b.noofwheels())  output:  6 |

|  |
| --- |
| class test:      def m1(self):          print("This is not abstract method")      @abstractmethod      def m2(self):          print("Hello")    class subtest(test):      def m2(self):          print("This is m2 method")    t= subtest()  t.m1()  t.m2()  output:  This is not abstract method  This is m2 method |

Interface

Those classes that have only abstract methods then such class are called interface.

|  |
| --- |
| class A(ABC):      @abstractmethod      def m1(self):          pass      @abstractmethod      def m2(self):          pass      @abstractmethod      def m3(self):          pass  class dipeshimplementation(A):      def m1(self):          print('m1')      def m2(self):          print('m2')      def m3(self):          print('m3')  class biplovimplementation(A):      def m1(self):          print('m1')      def m2(self):          print('m2')      def m3(self):          print('m3')  class nikitaimplementation(A):      def m1(self):          print('m1')      def m2(self):          print('m2')      def m3(self):          print('m3')  d= dipeshimplementation()  d.m1()  d.m2()  d.m3()  output:  m1  m2  m3 |

Public. Protected and Private

There is not this kind of concept in python. But some user use it. By default all the members or variables are public. Private variables are accessed inside class but not through outside class. When we put double underscore (\_\_) infront of a variable then it is private variable and same in private method.

Data hiding

There is no access specifier so python cannot provide that much secure.

Abstraction

It means hiding internal details and show only what a user offers.

Advantage:

* Security
* Modification and enhancement is easy without affecting end user.
* Maintainability

Encapsulation

Collection or placing data (properties) and method (behavior) in a same class then for using it then it is said to be encapsulation.

**3 pillars of OOP**

1. Inheritance: For code reusability and code extend
2. Polymorphism: For flexibility
3. Encapsulation: Date hiding, abstraction, for security

**Exception Handling**

Error: Malfunction or incorrect operation which disturbs the flow of program.

2 types of error:

* Syntax error: Error while typing the code or the grammar of the code. For syntax error programmer is responsible.

|  |
| --- |
| x= int(inptu("enter any number: "))  print(x)  output:  #error as the syntax for input is inptu |

* Runtime error: While executing the program, due to some reason such as user incorrect input, program logic or memory problem, etc if the program could not execute properly then it is said to be runtime error. Another name of runtime error is exception.

|  |
| --- |
| x= int(input("enter any number: "))  print(x)  output:  enter any number: temp #here temp is string and we have asked the input as int so error |

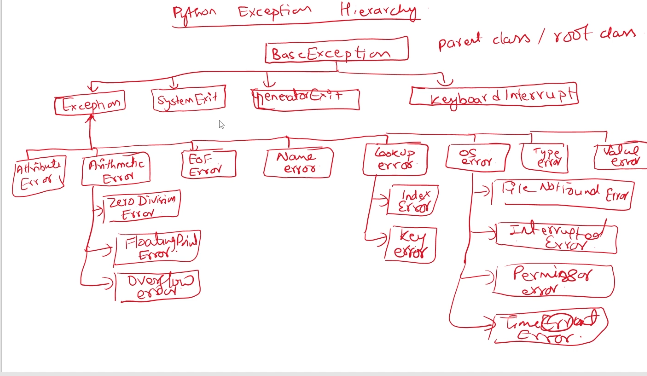
**Exception**: Unwanted or unexpected event that disturbs normal flow of program then it is called exceptions. Exception gives the abnormal termination so we need to handle it.

Finding the alternative of the error or exception is known as exception handling. It does not mean to repair exception only finding the alternative to eliminate abnormal termination.

We use try-except block for handling exception. In try block we write risky code and in except block we write solution for risky code.

Default exception handling:Every exception in python is object.

|  |
| --- |
| print("hello")  print(10/0)  print('hello') #this is not executed  output:  hello  Traceback (most recent call last):  File "e:\intern\class\exception.py", line 2, in <module>  print(10/0)  ~~^~  ZeroDivisionError: division by zero |



Exception handling using try-except

try: Risky code

except: alternative

|  |
| --- |
| print("this is my first line of program")  try:      print(10/0)  except ZeroDivisionError:      print(10/2)  print("this is my last line of program")  output:  this is my first line of program  5.0  this is my last line of program |

|  |
| --- |
| try:      x= int(input("enter 1st number: "))      y= int(input("enter 2nd number: "))      print("Result of division is: ",x/y)  except BaseException as b:      print("type of exception: ", type(b))      print("This is not allowed")      print("class name: ",b.\_\_class\_\_.\_\_name\_\_)  output:  enter 1st number: 1  enter 2nd number: d  type of exception: <class 'ValueError'>  This is not allowed  class name: ValueError |

|  |
| --- |
| try:      x= int(input("enter 1st number: "))      y= int(input("enter 2nd number: "))      print("Result of division is: ",x/y)  except (ZeroDivisionError, ArithmeticError, ValueError) as b:      print("type of exception: ", b)  output:  enter 1st number: 1  enter 2nd number: 0  type of exception: division by zero |

Default except block: It should always be at last part of try block.

|  |
| --- |
| try:      x= int(input("enter 1st number: "))      y= int(input("enter 2nd number: "))      print("Result of division is: ",x/y)  except ZeroDivisionError:      print("type of exception: ")  except:      print('Even valude error is handle')  output:  enter 1st number: 1  enter 2nd number: h  Even valude error is handle |

Various possibilities of except block

* except ZeroDivisionError:
* except (ZeroDivisionError):
* except ZeroDivisionError as m:
* except (ZeroDivisionError) as m:
* except( ZeroDivisionError,ValueError):
* except( ZeroDivisionError,ValueError) as m:
* except:

What we cannot do:

* except( ZeroDivisionError as m):
* except( ZeroDivisionError,ValueError as m):
* except ZeroDivisionError,ValueError:

We also have another block in try- except block .i.e. finally

Finally block is executed in any condition either there is exception or not. Clean up activities are kept in the finally block.

|  |
| --- |
| try:      x= int(input("enter 1st number: "))      y= int(input("enter 2nd number: "))      print("Result of division is: ",x/y)  except ZeroDivisionError as b:      print("type of exception: ",b)  finally:      print('This is cleanup acitivites')  output:  enter 1st number: 1  enter 2nd number: 0  type of exception: division by zero  This is cleanup acitivites |

|  |
| --- |
| try:      print('try')      print(10/0)  except:      print('except')  finally:      print('finally')  output:  try  except  finally |

Finally is always printed except if os.\_exit(0) occurs.

|  |
| --- |
| import os  try:      print('try')      os.\_exit(0)  except:      print('except')  finally:      print('finally')  output:  try |

Finally is used to clean up activities inside try.

Nested try except finally block:

|  |
| --- |
| try:      print("outer try block")      try:          print("Inner try block")          print(10/0)      except ZeroDivisionError:          print("inner except block")      finally:          print("inner finally block")          print(10/0)  except:      print("outer except block")  finally:      print("outer finally block")    output:  outer try block  Inner try block  inner except block  inner finally block  outer finally block |

Else block with try except finally

|  |
| --- |
| try:      print("try block")      print(10/0)  except:      print('except block')  else:      print('else block')  finally:      print("finally block")  output:  try block  except block  finally block |

Types of exception:

1. Pre-defined/Built in exception: Normal exception that we have done above such as ZeroDivisionError, ect.
2. User-defined/Customize exception: User can create their on exception. It is done with the help of raise keyword.

|  |
| --- |
| class TooYoungExcption(Exception):  def \_\_init\_\_(self, args):  self.msg=args    class TooOldExcption(Exception):  def \_\_init\_\_(self, args):  self.msg=args  age= int(input("enter your age: "))  if age>60:  raise TooOldExcption("Oh! Wait some more year to go to heaven")  elif age<18:  raise TooYoungExcption("Still a kiddo child")  else:  print("Work alot man, you need to earn more.")  output:  enter your age: 98  Traceback (most recent call last):  File "e:\intern\class\exception.py", line 63, in <module>  raise TooOldExcption("Oh! Wait some more year to go to heaven")  TooOldExcption: Oh! Wait some more year to go to heaven |

**Assertions**

Bug

When we write a code suppose x but the output occurs y. The deviation between expected output and real output is bug. The process of finding expected output is known as debugging. 95% python developer use print statement for debugging purpose. But there is some problem also while using print statement as print statement is extra in the code which is not good for a long line of code. So python don’t suggest print statement for debugging. Thus python provide assertion statement for debugging purpose.

There are 2 version of assertion:

1. Simple version
2. Augmented version

|  |
| --- |
| def squareit(x):  return x\*\*x  assert squareit(3)==9,"code is ok till now"  print(squareit(2)) #4  print(squareit(3)) #9  print(squareit(4)) #16  output:  Traceback (most recent call last):  File "e:\intern\assertion.py", line 9, in <module>  assert squareit(3)==9,"code is ok till now"  ^^^^^^^^^^^^^^  AssertionError: code is ok till now |

Logging

Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

Generally there are 6 logging level:

1. CRITICAL => 50: Various serious problem to which programmer should take immediately after getting it.
2. ERROR => 40: A little bit less serious than CRITICAL but need to be solved fast. It means there is some error in some part of code.
3. WARNING => 30: Warning message which indicate the programmer to take the message seriously so that in future it should not provide critical error.
4. INFO => 20: Information of a certain flow of message or program or anything.
5. DEBUG => 10: When a problem is debugged while running in which previously there was some bug.
6. NOTSET => 0: When only logging is implemented it is shown

If there is not mentioned about the levels in the logging then by default it is warning and higher levels.

Python has given logging module for implementation of logging. We need log file and logging level for implementing logging. To confiuge log file and logging, basicConfig() function gives this facility .

File handling

When we use data structures like list, tuple, dictionary, etc then the data is temporally stored. When the program is closed then the data also get removed. So to keep the data permanently file handling is used.

Types of file

1. Text file: Those type of file where text data is stored
2. Binary file: Images, videos, etc are stored in binary file.

Opening of file: For doing any type of operation in a file, opening of file is mandatory. Python gives open ( filename, mode) function to open any file.

f= open ( filename, mode)

filename= in which file I am working and mode= in which mode I am opening the file such as ‘r’ for read and so on.

Eg: f= open( ‘abc.txt’, mode= ‘r’)

There are 7 modes in python.

1. r 🡪 read mode: eg: : f= open( ‘abc.txt’, mode= ‘r’). If abc.txt is not found then their comes an error as ‘file not found’. Read mode is default mode
2. w 🡪 write mode: eg : f= open( ‘abc.txt’, mode= ‘w’). If there is no text file present i.e. there is no abc.txt then it will create the file abc.txt. If there is already data present in the text file then it will override the new data.
3. a🡪 append: write mode and append more are exactly same but here the new data is added with old data
4. r+ 🡪 read -> write: It is read then write mode. If text file is not found then their comes an error as ‘file not found’.
5. w+ 🡪 write -> read. It is write then read mode. If there is no text file present then it will create the file and then the file is read.
6. a+ 🡪 append -> read: It is append then read mode. If there is no text file present then it will create the file and then the file is read. The difference in w+ and a+ is that, a+ will add the data if the text file is already present.
7. X 🡪 exclusive: If a file is already exist then it will give FileExistError. It will create a new file every time.

If b is added to the normal modes then it becomes binary mode. Eg: f= open( ‘abc.txt’, mode= ‘rb’).

Every time a file is opened then it needs to be close. Close function is used to close a file and the resources are deallocated that the file holds.

|  |
| --- |
| f= open('abc.txt','r')  print("The file that is opend is: ", f.name)  print('Ths file mode is: ',f.mode)  print('Ths file mode is: ',f.closed)  print('Is file readable? ', f.readable())  print('Is file writable? ', f.writable())  f.close()  print('Ths file mode is: ',f.closed)  output:  The file that is opend is: abc.txt  Ths file mode is: r  Ths file mode is: False  Is file readable? True  Is file writable? False  Ths file mode is: True |

Writing on a file

Reading on a file

Ways to read a file:

f.read(): read total data

f.read(n): read n character from a file

f.readline(): it read one line at a time

f.readline(): it read multiple line at a time

Binary files: Files like text, audio, video are binary files. The modes that that are of in text files, here in binary file also there are same just at last ‘b’ is appended. Eg: for read ‘rb’