**Encapsulation in Python**

Encapsulation in Python describes the concept of **bundling data and methods within a single unit**. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

Implement encapsulation using a class

**Example**:

In this example, we create an Employee class by defining employee attributes such as name and salary as an instance variable and implementing behavior using work() and show() instance methods.

**class** Employee:

# constructor

**def** \_\_init\_\_(self, name, salary, project):

# data members

self.name = name

self.salary = salary

self.project = project

# method

# to display employee's details

**def** show(self):

# accessing public data member

**print**("Name: ", self.name, 'Salary:', self.salary)

# method

**def** work(self):

**print**(self.name, 'is working on', self.project)

# creating object of a class

emp = Employee('Jessa', 8000, 'NLP')

# calling public method of the class

emp.show()

emp.work()

**Output**:

Name: Jessa Salary: 8000

Jessa is working on NLP

Using encapsulation, we can hide an object’s internal representation from the outside. This is called information hiding.

Also, encapsulation allows us to restrict accessing variables and methods directly and prevent accidental data modification by creating private data members and methods within a class.

Encapsulation is a way to can restrict access to methods and variables from outside of class. Whenever we are working with the class and dealing with sensitive data, providing access to all variables used within the class is not a good choice.

For example, Suppose you have an attribute that is not visible from the outside of an object and bundle it with methods that provide read or write access. In that case, you can hide specific information and control access to the object’s internal state. Encapsulation offers a way for us to access the required variable without providing the program full-fledged access to all variables of a class. This mechanism is used to protect the data of an object from other objects.

**Access Modifiers in Python**

Encapsulation can be achieved by declaring the data members and methods of a class either as private or protected. But In Python, we don’t have direct access modifiers like public, private, and protected. We can achieve this by using single **underscore** and **double** **underscores**.

Access modifiers limit access to the variables and methods of a class. Python provides three types of access modifiers private, public, and protected.

* **Public Member**: Accessible anywhere from otside oclass.
* **Private Member**: Accessible within the class
* **Protected Member**: Accessible within the class and its sub-classes

Data hiding using access modifiers

**Public Member**

Public data members are accessible within and outside of a class. All member variables of the class are by default public.

**Example**:

**class** Employee:

# constructor

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

# public data members

self.name = name

self.salary = salary

# public instance methods

**def** show(self):

# accessing public data member

**print**("Name: ", self.name, 'Salary:', self.salary)

# creating object of a class

emp = Employee('Jessa', 10000)

# accessing public data members

**print**("Name: ", emp.name, 'Salary:', emp.salary)

# calling public method of the class

emp.show()

**Output**

Name: Jessa Salary: 10000

Name: Jessa Salary: 10000

**Private Member**

We can protect variables in the class by marking them private. To define a private variable add two underscores as a prefix at the start of a variable name.

Private members are accessible only within the class, and we can’t access them directly from the class objects.

**Example**:

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# creating object of a class

emp = Employee('Jessa', 10000)

# accessing private data members

**print**('Salary:', emp.\_\_salary)

**Output**

AttributeError: 'Employee' object has no attribute '\_\_salary'

In the above example, the salary is a private variable. As you know, we can’t access the private variable from the outside of that class.

We can access private members from outside of a class using the following two approaches

* Create public method to access private members
* Use name mangling

Let’s see each one by one

**Public method to access private members**

**Example**: Access Private member outside of a class using an instance method

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# public instance methods

**def** show(self):

# private members are accessible from a class

**print**("Name: ", self.name, 'Salary:', self.\_\_salary)

# creating object of a class

emp = Employee('Jessa', 10000)

# calling public method of the class

emp.show()

**Output**:

Name: Jessa Salary: 10000

**Name Mangling to access private members**

We can directly access private and protected variables from outside of a class through name mangling. The name mangling is created on an identifier by adding two leading underscores and one trailing underscore, like this \_classname\_\_dataMember, where classname is the current class, and data member is the private variable name.

**Example**: Access private member

**class** Employee:

# constructor

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

# public data member

self.name = name

# private member

self.\_\_salary = salary

# creating object of a class

emp = Employee('Jessa', 10000)

**print**('Name:', emp.name)

# direct access to private member using name mangling

**print**('Salary:', emp.\_Employee\_\_salary)

**Output**

Name: Jessa

Salary: 10000

**Protected Member**

Protected members are accessible within the class and also available to its sub-classes. To define a protected member, prefix the member name with a single underscore \_.

Protected data members are used when you implement inheritance and want to allow data members access to only child classes.

**Example**: Proctecd member in inheritance.

# base class

**class** Company:

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

# Protected member

self.\_project = "NLP"

# child class

**class** Employee(Company):

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

self.name = name

Company.\_\_init\_\_(self)

**def** show(self):

**print**("Employee name :", self.name)

# Accessing protected member in child class

**print**("Working on project :", self.\_project)

c = Employee("Jessa")

c.show()

# Direct access protected data member

**print**('Project:', c.\_project)

**Output**

Employee name : Jessa

Working on project : NLP

Project: NLP

**Getters and Setters in Python**

To implement proper encapsulation in Python, we need to use setters and getters. The primary purpose of using getters and setters in object-oriented programs is to ensure data encapsulation. Use the getter method to access data members and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming languages. The getters and setters methods are often used when:

* When we want to avoid direct access to private variables
* To add validation logic for setting a value

**Example**

**class** Student:

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

# private member

self.name = name

self.\_\_age = age

# getter method

**def** get\_age(self):

**return** self.\_\_age

# setter method

**def** set\_age(self, age):

self.\_\_age = age

stud = Student('Jessa', 14)

# retrieving age using getter

**print**('Name:', stud.name, stud.get\_age())

# changing age using setter

stud.set\_age(16)

# retrieving age using getter

**print**('Name:', stud.name, stud.get\_age())

**Output**

Name: Jessa 14

Name: Jessa 16

Let’s take another example that shows how to use encapsulation to implement information hiding and apply additional validation before changing the values of your object attributes (data member).

**Example**: Information Hiding and conditional logic for setting an object attributes

**class** Student:

**def** \_\_init\_\_(self, name, roll\_no, age):

# private member

self.name = name

# private members to restrict access

# avoid direct data modification

self.\_\_roll\_no = roll\_no

self.\_\_age = age

**def** show(self):

**print**('Student Details:', self.name, self.\_\_roll\_no)

# getter methods

**def** get\_roll\_no(self):

**return** self.\_\_roll\_no

# setter method to modify data member

# condition to allow data modification with rules

**def** set\_roll\_no(self, number):

**if** number > 50:

**print**('Invalid roll no. Please set correct roll number')

**else**:

self.\_\_roll\_no = number

jessa = Student('Jessa', 10, 15)

# before Modify

jessa.show()

# changing roll number using setter

jessa.set\_roll\_no(120)

jessa.set\_roll\_no(25)

jessa.show()

**Output**:

Student Details: Jessa 10

Invalid roll no. Please set correct roll number

Student Details: Jessa 25

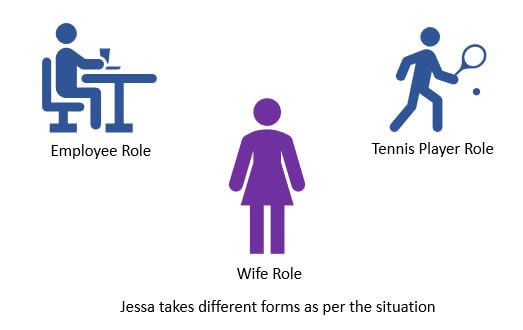
**Advantages of Encapsulation**

* **Security**: The main advantage of using encapsulation is the security of the data. Encapsulation protects an object from unauthorized access. It allows private and protected access levels to prevent accidental data modification.
* **Data Hiding**: The user would not be knowing what is going on behind the scene. They would only be knowing that to modify a data member, call the setter method. To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.
* **Simplicity**: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.
* **Aesthetics**: Bundling data and methods within a class makes code more readable and maintainable

**Polymorphism in Python**

Polymorphism in Python is the ability of an object to take many forms. In simple words, polymorphism allows us to perform the same action in many different ways.

For example, Jessa acts as an employee when she is at the office. However, when she is at home, she acts like a wife. Also, she represents herself differently in different places. Therefore, the same person takes different forms as per the situation.



A person takes different forms

In polymorphism, a method can **process objects differently depending on the class type or data type**. Let’s see simple examples to understand it better.

**Polymorphism in Built-in function len()**

The built-in function len() calculates the length of an object depending upon its type. If an object is a string, it returns the count of characters, and If an object is a list, it returns the count of items in a list.

The len() method treats an object as per its class type.

**Example**:

students = ['Emma', 'Jessa', 'Kelly']

school = 'ABC School'

# calculate count

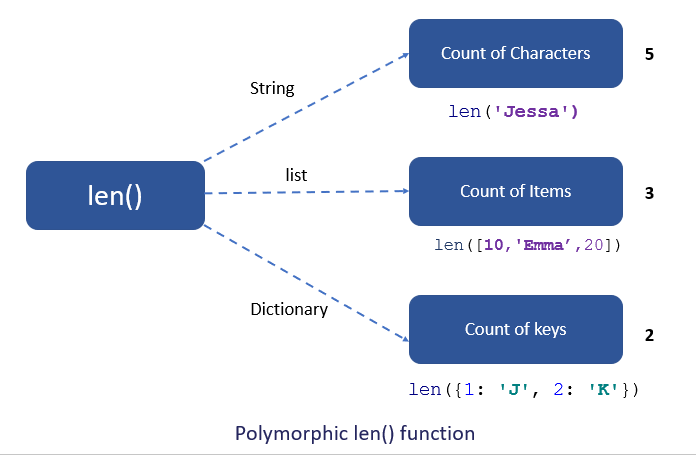
**print**(**len**(students))

**print**(**len**(school))

**Output**

3

10

Polymorphic len() function

**Polymorphism With Inheritance**

Polymorphism is mainly used with inheritance. In **inheritance**, child class inherits the attributes and methods of a parent class. The existing class is called a base class or parent class, and the new class is called a subclass or child class or derived class.

Using **method overriding** polymorphism allows us to defines methods in the child class that have the same name as the methods in the parent class. This **process of re-implementing the inherited method in the child class** is known as Method Overriding.

**Advantage of method overriding**

* It is effective when we want to extend the functionality by altering the inherited method. Or the method inherited from the parent class doesn’t fulfill the need of a child class, so we need to re-implement the same method in the child class in a different way.
* Method overriding is useful when a parent class has multiple child classes, and one of that child class wants to redefine the method. The other child classes can use the parent class method. Due to this, we don’t need to modification the parent class code

In polymorphism, **Python first checks the object’s class type and executes the appropriate method** when we call the method. For example, If you create the Car object, then Python calls the speed() method from a Car class.

Let’s see how it works with the help of an example.

**Example: Method Overriding**

In this example, we have a vehicle class as a parent and a ‘Car’ and ‘Truck’ as its sub-class. But each vehicle can have a different seating capacity, speed, etc., so we can have the same instance method name in each class but with a different implementation. Using this code can be extended and easily maintained over time.Polymorphism with Inheritance

**class** Vehicle:

**def** \_\_init\_\_(self, name, color, price):

self.name = name

self.color = color

self.price = price

**def** show(self):

**print**('Details:', self.name, self.color, self.price)

**def** max\_speed(self):

**print**('Vehicle max speed is 150')

**def** change\_gear(self):

**print**('Vehicle change 6 gear')

# inherit from vehicle class

**class** Car(Vehicle):

**def** max\_speed(self):

**print**('Car max speed is 240')

**def** change\_gear(self):

**print**('Car change 7 gear')

# Car Object

car = Car('Car x1', 'Red', 20000)

car.show()

# calls methods from Car class

car.max\_speed()

car.change\_gear()

# Vehicle Object

vehicle = Vehicle('Truck x1', 'white', 75000)

vehicle.show()

# calls method from a Vehicle class

vehicle.max\_speed()

vehicle.change\_gear()

**Output**:

Details: Car x1 Red 20000

Car max speed is 240

Car change 7 gear

Details: Truck x1 white 75000

Vehicle max speed is 150

Vehicle change 6 gear

As you can see, due to polymorphism, the Python interpreter recognizes that the max\_speed() and change\_gear() methods are overridden for the car object. So, it uses the one defined in the child class (Car)

On the other hand, the show() method isn’t overridden in the Car class, so it is used from the Vehicle class.

**Overrride Built-in Functions**

In Python, we can change the default behavior of the built-in functions. For example, we can change or extend the built-in functions such as len(), abs(), or divmod() by redefining them in our class. Let’s see the example.

**Example**

In this example, we will redefine the function len()

**class** Shopping:

**def** \_\_init\_\_(self, basket, buyer):

self.basket = **list**(basket)

self.buyer = buyer

**def** \_\_len\_\_(self):

**print**('Redefine length')

count = **len**(self.basket)

# count total items in a different way

# pair of shoes and shir+pant

**return** count \* 2

shopping = Shopping(['Shoes', 'dress'], 'Jessa')

**print**(**len**(shopping))

**Output**

Redefine length

4

**Polymorphism In Class methods**

Polymorphism with class methods is useful when we group different objects having the same method. we can add them to a list or a tuple, and we don’t need to check the object type before calling their methods. Instead, Python will check object type at runtime and call the correct method. Thus, we can call the methods without being concerned about which class type each object is. We assume that these methods exist in each class.

Python allows different classes to have methods with the same name.

* Let’s design a different class in the same way by adding the same methods in two or more classes.
* Next, create an object of each class
* Next, add all objects in a tuple.
* In the end, iterate the tuple using a for loop and call methods of a object without checking its class.

**Example**

In the below example, fuel\_type() and max\_speed() are the instance methods created in both classes.

**class** Ferrari:

**def** fuel\_type(self):

**print**("Petrol")

**def** max\_speed(self):

**print**("Max speed 350")

**class** BMW:

**def** fuel\_type(self):

**print**("Diesel")

**def** max\_speed(self):

**print**("Max speed is 240")

ferrari = Ferrari()

bmw = BMW()

# iterate objects of same type

**for** car **in** (ferrari, bmw):

# call methods without checking class of object

car.fuel\_type()

car.max\_speed()

**Output**

Petrol

Max speed 350

Diesel

Max speed is 240

As you can see, we have created two classes Ferrari and BMW. They have the same instance method names fuel\_type() and max\_speed(). However, we have not linked both the classes nor have we used inheritance.

We packed two different objects into a tuple and iterate through it using a car variable. It is possible due to polymorphism because we have added the same method in both classes Python first checks the object’s class type and executes the method present in its class.

**Polymorphism with Function and Objects**

We can create polymorphism with a function that can take any object as a parameter and execute its method without checking its class type. Using this, we can call object actions using the same function instead of repeating method calls.

**Example**

**class** Ferrari:

**def** fuel\_type(self):

**print**("Petrol")

**def** max\_speed(self):

**print**("Max speed 350")

**class** BMW:

**def** fuel\_type(self):

**print**("Diesel")

**def** max\_speed(self):

**print**("Max speed is 240")

# normal function

**def** car\_details(obj):

obj.fuel\_type()

obj.max\_speed()

ferrari = Ferrari()

bmw = BMW()

car\_details(ferrari)

car\_details(bmw)

**Output**

Petrol

Max speed 350

Diesel

Max speed is 240

**Polymorphism In Built-in Methods**

The word polymorphism is taken from the Greek words poly (many) and morphism (forms). It means a **method can process objects differently depending on the class type or data type.**

The built-in function reversed(obj) returns the iterable by reversing the given object. For example, if you pass a string to it, it will reverse it. But if you pass a list of strings to it, it will return the iterable by reversing the order of elements (it will not reverse the individual string).

Let us see how a built-in method process objects having different data types.

**Example**:

students = ['Emma', 'Jessa', 'Kelly']

school = 'ABC School'

**print**('Reverse string')

**for** i **in** **reversed**('PYnative'):

**print**(i, end=' ')

**print**('\nReverse list')

**for** i **in** **reversed**(['Emma', 'Jessa', 'Kelly']):

**print**(i, end=' ')

**Output**:

Reverse string

e v i t a n Y P

Reverse list

Kelly Jessa Emma

**Method Overloading**

The process of calling the same method with different parameters is known as method overloading. Python does not support method overloading. Python considers only the latest defined method even if you overload the method. Python will raise a TypeError if you overload the method.

**Example**

**def** addition(a, b):

c = a + b

**print**(c)

**def** addition(a, b, c):

d = a + b + c

**print**(d)

# the below line shows an error

# addition(4, 5)

# This line will call the second product method

addition(3, 7, 5)

To overcome the above problem, we can use different ways to achieve the method overloading. In Python, to overload the class method, we need to write the method’s logic so that different code executes inside the function depending on the parameter passes.

For example, the built-in function range() takes three parameters and produce different result depending upon the number of parameters passed to it.

**Example**:

**for** i **in** **range**(5): **print**(i, end=', ')

**print**()

**for** i **in** **range**(5, 10): **print**(i, end=', ')

**print**()

**for** i **in** **range**(2, 12, 2): **print**(i, end=', ')

**Output**:

0, 1, 2, 3, 4,

5, 6, 7, 8, 9,

2, 4, 6, 8, 10,

Let’s assume we have an area() method to calculate the area of a square and rectangle. The method will calculate the area depending upon the number of parameters passed to it.

* If one parameter is passed, then the area of a square is calculated
* If two parameters are passed, then the area of a rectangle is calculated.

**Example**: User-defined polymorphic method

**class** Shape:

# function with two default parameters

**def** area(self, a, b=0):

**if** b > 0:

**print**('Area of Rectangle is:', a \* b)

**else**:

**print**('Area of Square is:', a \*\* 2)

square = Shape()

square.area(5)

rectangle = Shape()

rectangle.area(5, 3)

**Output**:

Area of Square is: 25

Area of Rectangle is: 15

**Operator Overloading in Python**

Operator overloading means changing the default behavior of an operator depending on the operands (values) that we use. In other words, we can use the same operator for multiple purposes.

For example, the + operator will perform an arithmetic addition operation when used with numbers. Likewise, it will perform concatenation when used with strings.

The operator + is used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

**Example**:

# add 2 numbers

**print**(100 + 200)

# concatenate two strings

**print**('Jess' + 'Roy')

# merger two list

**print**([10, 20, 30] + ['jessa', 'emma', 'kelly'])

**Output**:

300

JessRoy

[10, 20, 30, 'jessa', 'emma', 'kelly']

**Overloading + operator for custom objects**

Suppose we have two objects, and we want to add these two objects with a binary + operator. However, it will throw an error if we perform addition because the compiler doesn’t add two objects. See the following example for more details.

**Example**:

**class** Book:

**def** \_\_init\_\_(self, pages):

self.pages = pages

# creating two objects

b1 = Book(400)

b2 = Book(300)

# add two objects

**print**(b1 + b2)

**Output**

TypeError: unsupported operand type(s) for +: 'Book' and 'Book'

We can overload + operator to work with custom objects also. Python provides some special or magic function that is automatically invoked when associated with that particular operator.

For example, when we use the + operator, the magic method \_\_add\_\_() is automatically invoked. Internally + operator is implemented by using \_\_add\_\_() method. We have to override this method in our class if you want to add two custom objects.

**Example**:

**class** Book:

**def** \_\_init\_\_(self, pages):

self.pages = pages

# Overloading + operator with magic method

**def** \_\_add\_\_(self, other):

**return** self.pages + other.pages

b1 = Book(400)

b2 = Book(300)

**print**("Total number of pages: ", b1 + b2)

**Output**

Total number of pages: 700

**Overloading the \* Operator**

The \* operator is used to perform the multiplication. Let’s see how to overload it to calculate the salary of an employee for a specific period. Internally \* operator is implemented by using the \_\_mul\_\_() method.

**Example**:

**class** Employee:

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

self.name = name

self.salary = salary

**def** \_\_mul\_\_(self, timesheet):

**print**('Worked for', timesheet.days, 'days')

# calculate salary

**return** self.salary \* timesheet.days

**class** TimeSheet:

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

self.name = name

self.days = days

emp = Employee("Jessa", 800)

timesheet = TimeSheet("Jessa", 50)

**print**("salary is: ", emp \* timesheet)

**Output**

Wroked for 50 days

salary is: 40000

**Magic Methods**

In Python, there are different magic methods available to perform overloading operations. The below table shows the magic methods names to overload the mathematical operator, assignment operator, and relational operators in Python.

| **Operator Name** | **Symbol** | **Magic method** |
| --- | --- | --- |
| Addition | + | \_\_add\_\_(self, other) |
| Subtraction | - | \_\_sub\_\_(self, other) |
| Multiplication | \* | \_\_mul**\_\_**(self, other) |
| Division | / | \_\_div\_\_(self, other) |
| Floor Division | // | \_\_floordiv\_\_(self,other) |
| Modulus | % | \_\_mod\_\_(self, other) |
| Power | \*\* | \_\_pow\_\_(self, other) |
| Increment | += | \_\_iadd\_\_(self, other) |
| Decrement | -= | \_\_isub\_\_(self, other) |
| Product | \*= | \_\_imul\_\_(self, other) |
| Division | /+ | \_\_idiv\_\_(self, other) |
| Modulus | %= | \_\_imod\_\_(self, other) |
| Power | \*\*= | \_\_ipow\_\_(self, other) |
| Less than | < | \_\_lt\_\_(self, other) |
| Greater than | > | \_\_gt\_\_(self, other) |
| Less than or equal to | <= | \_\_le\_\_(self, other) |
| Greater than or equal to | >= | \_\_ge\_\_(self, other) |
| Equal to | == | \_\_eq\_\_(self, other) |
| Not equal | != | \_\_ne\_\_(self, other) |

magic methods

**Inheritance in Python**

**Types Of Inheritance**

In Python, based upon the number of child and parent classes involved, there are five types of inheritance. The type of inheritance are listed below:

1. Single inheritance
2. Multiple Inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Now let’s see each in detail with an example.

**Single Inheritance**

In single inheritance, a child class inherits from a single-parent class. Here is one child class and one parent class.

Python Single Inheritance

**Example**

Let’s create one parent class called ClassOne and one child class called ClassTwo to implement single inheritance.

# Base class

**class** Vehicle:

**def** Vehicle\_info(self):

**print**('Inside Vehicle class')

# Child class

**class** Car(Vehicle):

**def** car\_info(self):

**print**('Inside Car class')

# Create object of Car

car = Car()

# access Vehicle's info using car object

car.Vehicle\_info()

car.car\_info()

**Output**

Inside Vehicle class

Inside Car class

**Multiple Inheritance**

In multiple inheritance, one child class can inherit from multiple parent classes. So here is one child class and multiple parent classes.

Python Multiple Inheritance

**Example**

# Parent class 1

**class** Person:

**def** person\_info(self, name, age):

**print**('Inside Person class')

**print**('Name:', name, 'Age:', age)

# Parent class 2

**class** Company:

**def** company\_info(self, company\_name, location):

**print**('Inside Company class')

**print**('Name:', company\_name, 'location:', location)

# Child class

**class** Employee(Person, Company):

**def** Employee\_info(self, salary, skill):

**print**('Inside Employee class')

**print**('Salary:', salary, 'Skill:', skill)

# Create object of Employee

emp = Employee()

# access data

emp.person\_info('Jessa', 28)

emp.company\_info('Google', 'Atlanta')

emp.Employee\_info(12000, 'Machine Learning')

**Output**

Inside Person class

Name: Jessa Age: 28

Inside Company class

Name: Google location: Atlanta

Inside Employee class

Salary: 12000 Skill: Machine Learning

In the above example, we created two parent classes Person and Company respectively. Then we create one child called Employee which inherit from Person and Company classes.

**Multilevel inheritance**

In multilevel inheritance, a class inherits from a child class or derived class. Suppose three classes A, B, C. A is the superclass, B is the child class of A, C is the child class of B. In other words, we can say a **chain of classes** is **called multilevel inheritance.**

Python Multilevel Inheritance

**Example**

# Base class

**class** Vehicle:

**def** Vehicle\_info(self):

**print**('Inside Vehicle class')

# Child class

**class** Car(Vehicle):

**def** car\_info(self):

**print**('Inside Car class')

# Child class

**class** SportsCar(Car):

**def** sports\_car\_info(self):

**print**('Inside SportsCar class')

# Create object of SportsCar

s\_car = SportsCar()

# access Vehicle's and Car info using SportsCar object

s\_car.Vehicle\_info()

s\_car.car\_info()

s\_car.sports\_car\_info()

**Output**

Inside Vehicle class

Inside Car class

Inside SportsCar class

In the above example, we can see there are three classes named Vehicle, Car, SportsCar. Vehicle is the superclass, Car is a child of Vehicle, SportsCar is a child of Car. So we can see the **chaining of classes**.

**Hierarchical Inheritance**

In Hierarchical inheritance, more than one child class is derived from a single parent class. In other words, we can say one parent class and multiple child classes.

Python hierarchical inheritance

**Example**

Let’s create ‘Vehicle’ as a parent class and two child class ‘Car’ and ‘Truck’ as a parent class.

**class** Vehicle:

**def** info(self):

**print**("This is Vehicle")

**class** Car(Vehicle):

**def** car\_info(self, name):

**print**("Car name is:", name)

**class** Truck(Vehicle):

**def** truck\_info(self, name):

**print**("Truck name is:", name)

obj1 = Car()

obj1.info()

obj1.car\_info('BMW')

obj2 = Truck()

obj2.info()

obj2.truck\_info('Ford')

**Output**

This is Vehicle

Car name is: BMW

This is Vehicle

Truck name is: Ford

**Hybrid Inheritance**

When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.

Python hybrid inheritance

**Example**

**class** Vehicle:

**def** vehicle\_info(self):

**print**("Inside Vehicle class")

**class** Car(Vehicle):

**def** car\_info(self):

**print**("Inside Car class")

**class** Truck(Vehicle):

**def** truck\_info(self):

**print**("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car

**class** SportsCar(Car, Vehicle):

**def** sports\_car\_info(self):

**print**("Inside SportsCar class")

# create object

s\_car = SportsCar()

s\_car.vehicle\_info()

s\_car.car\_info()

s\_car.sports\_car\_info()

**Note**: In the above example,**hierarchical**and**multiple** inheritance exists. Here we created, parent class Vehicle and two child classes named Car and Truck this is hierarchical inheritance.

Another is SportsCar inherit from two parent classes named Car and Vehicle. This is multiple inheritance.

**Python super() function**

When a class inherits all properties and behavior from the parent class is called inheritance. In such a case, the inherited class is a subclass and the latter class is the parent class.

In child class, we can refer to parent class by using the super() function. The super function returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.

**Benefits of using the super() function.**

1. We are not required to remember or specify the parent class name to access its methods.
2. We can use the super() function in both **single** and **multiple inheritances**.
3. The super() function support code **reusability** as there is no need to write the entire function

**Example**

**class** Company:

**def** company\_name(self):

**return** 'Google'

**class** Employee(Company):

**def** info(self):

# Calling the superclass method using super()function

c\_name = **super**().company\_name()

**print**("Jessa works at", c\_name)

# Creating object of child class

emp = Employee()

emp.info()

**Output**:

Jessa works at Google

In the above example, we create a parent class Company and child class Employee. In Employee class, we call the parent class method by using a super() function.

**issubclass()**

In Python, we can verify whether a particular class is a subclass of another class. For this purpose, we can use Python built-in function issubclass(). This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False.

**Syntax**

**issubclass**(**class**, classinfo)

Where,

* class: class to be checked.
* classinfo: a class, type, or a tuple of classes or data types.

**Example**

**class** Company:

**def** fun1(self):

**print**("Inside parent class")

**class** Employee(Company):

**def** fun2(self):

**print**("Inside child class.")

**class** Player:

**def** fun3(self):

**print**("Inside Player class.")

# Result True

**print**(**issubclass**(Employee, Company))

# Result False

**print**(**issubclass**(Employee, **list**))

# Result False

**print**(**issubclass**(Player, Company))

# Result True

**print**(**issubclass**(Employee, (**list**, Company)))

# Result True

**print**(**issubclass**(Company, (**list**, Company)))

Also, see Python isinstance().

**Method Overriding**

In inheritance, all members available in the parent class are by default available in the child class. If the child class does not satisfy with parent class implementation, then the child class is allowed to redefine that method by extending additional functions in the child class. This concept is called **method overriding**.

When a child class method has the same name, same parameters, and same return type as a method in its superclass, then the method in the child is said to **override** the method in the parent class.

Python method overriding

**Example**

**class** Vehicle:

**def** max\_speed(self):

**print**("max speed is 100 Km/Hour")

**class** Car(Vehicle):

# overridden the implementation of Vehicle class

**def** max\_speed(self):

**print**("max speed is 200 Km/Hour")

# Creating object of Car class

car = Car()

car.max\_speed()

**Output**:

max speed is 200 Km/Hour

In the above example, we create two classes named Vehicle (Parent class) and Car (Child class). The class Car extends from the class Vehicle so, all properties of the parent class are available in the child class. In addition to that, the child class redefined the method max\_speed().

**Method Resolution Order in Python**

In Python, Method Resolution Order(MRO) is the order by which **Python looks for a method or attribute**. First, the method or attribute is searched within a class, and then it follows the order we specified while inheriting.

This order is also called the Linearization of a class, and a set of rules is called MRO (Method Resolution Order). The **MRO plays an essential role in multiple inheritances as a single method may found in multiple parent classes**.

In multiple inheritance, the following search order is followed.

1. First, it searches in the current parent class if not available, then searches in the parents class specified while inheriting (that is left to right.)
2. We can get the MRO of a class. For this purpose, we can use either the mro attribute or the mro() method.

**Example**

class A:

def process(self):

print(" In class A")

class B(A):

def process(self):

print(" In class B")

class C(B, A):

def process(self):

print(" In class C")

# Creating object of C class

C1 = C()

C1.process()

print(C.mro())

# In class C

# [<class '\_\_main\_\_.C'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.A'>, <class 'object'>]

In the above example, we create three classes named A, B and C. Class B is inherited from A, class C inherits from B and A. When we create an object of the C class and calling the process() method, Python looks for the process() method in the current class in the C class itself.

Then search for parent classes, namely B and A, because C class inherit from B and A. that is, C(B, A) and always search in **left to right manner.**