**What is Object-Oriented Programming?**

Object Oriented Programming is a way of computer programming using the idea of “objects” to represents data and methods. It is also, an approach used for creating neat and reusable code instead of a redundant one. the program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.

OOP uses the concept of objects and classes. A class can be thought of as a 'blueprint' for objects. These can have their own attributes (characteristics they possess), and methods (actions they perform).

OOP Example

An example of a class is the class Dog. Don't think of it as a specific dog, or your own dog. We're describing what a dog is and can do, in general. Dogs usually have a name and age; these are instance attributes. Dogs can also bark; this is a method.

When you talk about a specific dog, you would have an object in programming: an object is an instantiation of a class. This is the basic principle on which object-oriented programming is based. So my dog Ozzy, for example, belongs to the class Dog. His attributes are name = 'Ozzy' and age = '2'. A different dog will have different attributes.

Difference between Object-Oriented and Procedural Oriented Programming

|  |  |
| --- | --- |
| **Object-Oriented Programming (OOP)** | **Procedural-Oriented Programming (Pop)** |
| It is a bottom-up approach | It is a top-down approach |
| Program is divided into objects | Program is divided into functions |
| Makes use of *Access modifiers*  ‘public’, private’, protected’ | Doesn’t use *Access modifiers* |
| It is more secure | It is less secure |
| Object can move freely within member functions | Data can move freely from function to function within programs |
| It supports inheritance | It does not support inheritance |

**OOP in Python**

Python is a great programming language that supports OOP. You will use it to define a class with attributes and methods, which you will then call. Python offers a number of benefits compared to other programming languages like Java, C++ or R. It's a dynamic language, with high-level data types. This means that development happens much faster than with Java or C++. It does not require the programmer to declare types of variables and arguments. This also makes Python easier to understand and learn for beginners, its code being more readable and intuitive.

**What are Python OOPs Concepts?**

Major OOP (object-oriented programming) concepts in Python include

1. Class
2. Object
3. Method
4. Inheritance
5. Polymorphism
6. Data Abstraction
7. Encapsulation.

**What is Python Class And Object?**

A class is a collection of objects, and an object is defined as an instance of class possessing attributes. The object is an entity that has state and behavior. A class has all the similar attributes, like if we have a class students, then it will only consist of students related data, such as subjects, names, attendance ratio, etc.

By using oop, we can divide our code into many sections known as classes. Each class holds a distinct purpose or usage. For example, if we have created a class named "Books" then all the attributes it possesses should be related to books such as the number of pages, publishing date or price, etc.

There is no limit to the number of classes we can create in a program. Also, one class can be easily accessible by another, and we can also restrict the access of a class so other classes can not use its functions. This concept comes in handy while working on bigger projects. All the employees are given separate tasks to work on the classes they have been assigned. And after they are done with their contribution, the classes can be combined as a whole to form a complete project.

**Instance variable:**

"**Instance variables are the variables for which the value of the variable is different for every instance**."

We can also say that the value is different for every object that we create. Let us dive into some in-depth explanation. When we create a class, we define a few variables along with it. For example, we have created a class of Students, and we have defined a variable age. All the students cannot have the same age in a class, so we have assigned the variable an average age of 16. Now, whenever we use an object to print the value of age, it will show 16. We can try to change the value of age, but it will create a new instance variable for the specific object that we are updating it for, hence defining the value to it.

The code for changing age for a particular object will be something like this:

**Std1.age = 18**

**Class variable:**

**"Class attributes are owned by the class directly, which means that they are not tied to any object or instance."**

Same as in the above example, if we want to change the age for every instance from 16 to 17, then we can do it by using the class variable, which in this case is Student.

**"It is worth noting that updating the value of the class variable will not change it for the instance variables of the objects, such as in the case above."**

The code for changing age using a class variable will be something like this:

**Students.age = 18**

The following are the notable differences between Class (static) and instance variables.

|  |
| --- |
| **Following are the differences between Class and instance variables**. |
| |  |  | | --- | --- | | **Instance variables** | **Class variables** | | When an object is created with the use of the new keyword, instance variables are created. They destroyed when the object is destroyed. | When the program starts, static variables are created and destroyed when the program stops. | | Instance variables can be accessed by calling the variable name inside the class. ***ObjectReference.VariableName*.** | Static variables can be accessed by calling using a class name. ***ClassName.VariableName*.** | | Every instance of the class has its own copy of that variable. Changes made to the variable don't affect the other instances of that class. | There is only one copy of that variable that is shared with all instances of the class. If changes are made to that variable, all other instances will be affected. | |

**The \_\_dict\_\_ attribute**

Every object in Python has an attribute which is denoted by \_\_dict\_\_, it maps the attribute name to its value. This dictionary is used to stores all the attributes defined for the object itself. Following is the syntax of using \_\_dict\_\_:

**object.\_\_dict\_\_**

**A quick review:**

Instance, variables are created only for a specific object. The object can change, create, or update only its instance variables. While in the case of class variables, the variables and values we create or define are set as default for all the objects. The objects cannot change the value or variable in the class by just updating it using object\_name.class\_name .However, it can change the values of their particular instance variables. Making use of class and instance variables can ensure that our code adheres to the DRY (don't repeat yourself) principle to reduce repetition within code.

**Method:**

A method is just like a function, with a **def** keyword and a single parameter in which the name of the object has to be passed. The purpose of the method is to show all the details related to the object in a single go. We choose variables that we want the method to take but do not have to pass all of them as parameters. Instead, we have to set the parameters we want to include in the method during its creation. Using methods make the process simpler and a lot faster.

**Self keyword:**

The self keyword is used in the method to refer to the instance of the current class we are using. The self keyword is passed as a parameter explicitly every time we define a method.

**def read\_number(self):**

**print(self.num)**

**\_\_init\_\_ method:-**

"\_\_init\_\_" is also called as a constructor in object-oriented terminology. Whereas a constructor is defined as:"**Constructor in Python is used to assign values to the variables or data members of a class when an object is created**."

Python treats the constructor differently as compared to C++ and Java. The constructor is a method with a def keyword and parameters, but the purpose of a constructor is to assign values to the instance variables of different objects. We can give the values by accessing each of the variables one by one, but in the case of the constructor, we pass all the values directly as parameters. Self keyword is used to assign value to a constructor too.

We declare a constructor in Python using def keyword,

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

**# body of the constructor**

Here,

The **def** keyword is used to define the function.

The first argument refers to the current object which binds the instance to the **init()** method.

In **init()** method ,arguments are optional. Constructors can be defined with any number of arguments or with no arguments.

For Example:

**class Person:**

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

**self.name = name**

**self.age = age**

**p1 = Person("John", 36)**

**print(p1.name)**

#Output: John

**Types of constructors in Python**

We have two types of constructors in Python.

1. The default constructor is the one that does not take any arguments.
2. Constructor with parameters is known as parameterized constructor.

**Method vs. Function:**

Methods and functions are very similar, yet there are some differences:

1. Methods are explicitly for Object-Oriented programming.
2. The method can only be used by the object that it is called for. In simple terms, for a method, the parameter must be an object.
3. The method can only access the data that is initialized in the class the method is formed in.

**Class method:**

In Object-Oriented programming, there is a new concept of a class method, which we will see today in this lecture. They are very different from static methods as they are limited in their functionality to the class they are built-in. They can be called by using the class name and also can be accessed by using the object.

As we have observed in the previous tutorials, we cannot change the value of a variable defined in the class from outside, using an object. Instead, if we try that, a new instance variable will be created for the class having the value we assigned. But no change will occur in the original value of the variable.

**Syntax:**

**class myClass:**

**@classmethod**

**def myfunc (cls, arg1, arg2, ...):**

....

**myfunc** defines the function that needs to be converted into a class method

returns**: @classmethod** returns a class method for function.

Because the class method only has access to **cls** argument, it cannot modify object instance state. However, class methods can still modify class state that applies to all instances of the class. So a class method automatically recognizes a class, so the only parameter that remains to be passed is the function that needs conversion.

**@classmethod Decorator:**

A **@classmethod** Decorator is a built-in function in Python. It can be applied to any method of the class. We can change the value of variables using this method too.

Python class method is a way to define a function for the Python class. It receives the class as an implicit first argument. Using **@classmethod** decorator, it is possible to create as many constructors for a class that behaves like a factory constructor.

**split():**

Let us have a brief overview of the split() function. What split() does is, it takes a separator as a parameter. If we do not provide any, then the default separator is any whitespace it encounters. Else we can provide any separator to it such as full stop, hash, dash, colon, etc. After separating the string into parts, the split() function stores it into a list in a sequence. For example:

**text = "Python tutorial for absolute beginners."**

**t = text.split()**

**print(t)**

Here, we are not providing it any separator as a parameter, so it will automatically divide, taking whitespace as a separator.

The output will be a list, such as:

**['Python', 'tutorial', 'for', 'absolute', 'beginners.']**

**Example of Class methods - alternative constructor:**

**class Date:**

**def \_\_init\_\_(self, year, month, day):**

**self.year = year**

**self.month = month**

**self.day = day**

**@classmethod**

**def from\_dash(cls,string):**

**return cls(\*string.split("-"))**

**date1=Date.from\_dash("2008-12-5")**

**print(date1.year)**

#Output: 2008

If we want multiple and independent "constructors", we can use class methods. They are usually called factory methods. It does not invoke the default constructor \_\_init\_\_.In the above example, we split the string based on the "-" operator. We first create a class method as a constructor that takes the string and split it based on the specified operator. For this purpose, we use a split() function, which takes the separator as a parameter. This alternative constructor approach is useful when we have to deal with files containing string data separated by a separator.

**What is Abstraction?**

Abstraction refers to hiding the unnecessary details so that the focus could be on the whole product instead of parts of the project separately. It is a mechanism that represents the important features without including implementation details. Abstraction helps us in partitioning the program into many independent concepts so, we may hide the irrelevant details in the code. It offers the greatest flexibility when using abstract data type objects in different situations.

**Example of Abstraction**

**1.**Let us take the example of a car. It has an engine, tires, windows, steering wheel, etc. All these things combine to forming a car, which is an abstraction, and all the different parts are its layers of abstraction. Now an engine is composed of different parts such as camshaft, valves, oil pan, etc. these flayers the engine is an abstraction. In simple words, abstraction can be achieved by hiding the background details and showing only the necessary ones. In programming, abstraction can not be achieved without Encapsulation.

**2**. Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don’t know the procedure of how the pin is generated or how the verification is done. This is called ‘abstraction’ from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.

An abstract class cannot be instantiated which simply means you cannot create objects for this type of class. It can only be used for inheriting the functionalities.

Example:

**from abc import ABC,abstractmethod**

**class employee(ABC):**

**def emp\_id(self,id,name,age,salary): //Abstraction**

**pass**

**class childemployee1(employee):**

**def emp\_id(self,id):**

**print("emp\_id is 12345")**

**emp1 = childemployee1()**

**emp1.emp\_id(id)**

**Output: emp\_id is 12345**

**Explanation:** As you can see in the above example, we have imported an abstract method and the rest of the program has a parent and a derived class. An object is instantiated for the ‘childemployee’ base class and functionality of abstract is being used.

**Why Abstraction is Important?**

In Python, an abstraction is used to hide the irrelevant data/class in order to reduce the complexity. It also enhances the application efficiency.

**Abstraction classes in Python**

In Python, abstraction can be achieved by using abstract classes and interfaces.

A class that consists of one or more abstract method is called the abstract class. Abstract methods do not contain their implementation. Abstract class can be inherited by the subclass and abstract method gets its definition in the subclass. Abstraction classes are meant to be the blueprint of the other class. An abstract class can be useful when we are designing large functions. An abstract class is also helpful to provide the standard interface for different implementations of components. Python provides the abc module to use the abstraction in the Python program. Let's see the following syntax.

**Syntax**

**from abc import ABC**

**class ClassName(ABC):**

**Abstract Base Classes**

An abstract base class is the common application program of the interface for a set of subclasses. It can be used by the third-party, which will provide the implementations such as with plugins. It is also beneficial when we work with the large code-base hard to remember all the classes.

**Working of the Abstract Classes**

Unlike the other high-level language, Python doesn't provide the abstract class itself. We need to import the abc module, which provides the base for defining Abstract Base classes (ABC). The ABC works by decorating methods of the base class as abstract. It registers concrete classes as the implementation of the abstract base. We use the **@abstractmethod** decorator to define an abstract method or if we don't provide the definition to the method, it automatically becomes the abstract method. Let's understand the following example.

**Example -**

1. # Python program demonstrate
2. # **abstract** base **class** work
3. from abc **import** ABC, abstractmethod
4. **class** Car(ABC):
5. def mileage(self):
6. pass
8. **class** Tesla(Car):
9. def mileage(self):
10. print("The mileage is 30kmph")
11. **class** Suzuki(Car):
12. def mileage(self):
13. print("The mileage is 25kmph ")
14. **class** Duster(Car):
15. def mileage(self):
16. print("The mileage is 24kmph ")
18. **class** Renault(Car):
19. def mileage(self):
20. print("The mileage is 27kmph ")
22. # Driver code
23. t= Tesla ()
24. t.mileage()
26. r = Renault()
27. r.mileage()
29. s = Suzuki()
30. s.mileage()
31. d = Duster()
32. d.mileage()

**Output:**

The mileage is 30kmph

The mileage is 27kmph

The mileage is 25kmph

The mileage is 24kmph

**Explanation -**

In the above code, we have imported the abc module to create the abstract base class. We created the Car class that inherited the ABC class and defined an abstract method named mileage(). We have then inherited the base class from the three different subclasses and implemented the abstract method differently. We created the objects to call the abstract method.

Let's understand another example.

**Example -**

1. # Python program to define
2. # **abstract** **class**
4. from abc **import** ABC
6. **class** Polygon(ABC):
8. # **abstract** method
9. def sides(self):
10. pass
12. **class** Triangle(Polygon):

15. def sides(self):
16. print("Triangle has 3 sides")
18. **class** Pentagon(Polygon):

21. def sides(self):
22. print("Pentagon has 5 sides")
24. **class** Hexagon(Polygon):
26. def sides(self):
27. print("Hexagon has 6 sides")
29. **class** square(Polygon):
31. def sides(self):
32. print("I have 4 sides")
34. # Driver code
35. t = Triangle()
36. t.sides()
38. s = square()
39. s.sides()
41. p = Pentagon()
42. p.sides()
44. k = Hexagon()
45. K.sides()

**Output:**

Triangle has 3 sides

Square has 4 sides

Pentagon has 5 sides

Hexagon has 6 sides

**Explanation -**

In the above code, we have defined the abstract base class named Polygon and we also defined the abstract method. This base class inherited by the various subclasses. We implemented the abstract method in each subclass. We created the object of the subclasses and invoke the sides() method. The hidden implementations for the sides() method inside the each subclass comes into play. The abstract method sides() method, defined in the abstract class, is never invoked.

**Points to Remember**

Below are the points which we should remember about the abstract base class in Python.

1. An Abstract class can contain the both method normal and abstract method.
2. An Abstract cannot be instantiated; we cannot create objects for the abstract class.

Abstraction is essential to hide the core functionality from the users. We have covered the all the basic concepts of Abstraction in Python.

**What is Encapsulation?**

Encapsulation means hiding under layers. When working with classes and handling sensitive data, global access to all the variables used in the program is not secure. In Encapsulation, the internal representation of an object is generally hidden from outside to secure the data. It improves the maintainability of an application and helps the developers to organize the code better. OR,

When working with classes and dealing with sensitive data, providing global access to all the variables used within the program is not a good choice. Encapsulation offers a way for us to access the required variables without providing the program full-fledged access to any of those variables.

Updating, modifying, or deleting data from variables can be done through the use of methods that are defined specifically for the purpose. The benefit of using this approach to programming is improved control over the input data and better security.

**What is Encapsulation in Python**?

The concept of encapsulation is the same in all object-oriented programming languages. The difference is seen when the concepts are applied to particular languages.

Compared to languages like Java that offer access modifiers (public or private) for variables and methods, Python provides access to all the variables and methods globally.

Check the below demonstration of how variables can easily be accessed.

**class Person:**

**def \_\_init\_\_(self, name, age=0):**

**self.name = name**

**self.age = age**

**def display(self):**

**print(self.name)**

**print(self.age)**

**person = Person('Dev', 30)**

**#accessing using class method**

**person.display()**

**#accessing directly from outside**

**print(person.name)**

**print(person.age)**

**Output**

Dev

30

Dev

30

Since we do not have access modifiers in Python, we will use a few different methods to control the access of variables within a Python program.

**Methods to Control Access**

There are multiple methods that are offered by Python to limit variable and method access across the program. Let’s go over the methods in detail.

**Using Single Underscore**

A common Python programming convention to identify a private variable is by prefixing it with an underscore. Now, this doesn’t really make any difference on the compiler side of things. The variable is still accessible as usual. But being a convention that programmers have picked up on, it tells other programmers that the variables or methods have to be used only within the scope of the class.

See the below example:

**class Person:**

**def \_\_init\_\_(self, name, age=0):**

**self.name = name**

**self.\_age = age**

**def display(self):**

**print(self.name)**

**print(self.\_age)**

**person = Person('Dev', 30)**

**#accessing using class method**

**person.display()**

**#accessing directly from outside**

**print(person.name)**

**print(person.\_age)**

Output

Dev

30

Dev

30

It’s clear that the variable access is unchanged. But can we do anything to really make it private? Let’s have a look further.

**Using Double Underscores**

If you want to make class members i.e. methods and variables private, then you should prefix them with double underscores. But Python offers some sort of support to the private modifier. This mechanism is called Name mangling. With this, it is still possible to access the class members from outside it.

**Name Mangling**

In Python, any identifier with \_\_Var is rewritten by a python interpreter as \_Classname\_\_Var, and the class name remains as the present class name. This mechanism of changing names is called Name Mangling in Python.

In the below example, in Class person, the age variable is changed and it’s prefixed by leading double underscores.

**class Person:**

**def \_\_init\_\_(self, name, age=0):**

**self.name = name**

**self.\_\_age = age**

**def display(self):**

**print(self.name)**

**print(self.\_\_age)**

**person = Person('Dev', 30)**

**#accessing using class method**

**person.display()**

**#accessing directly from outside**

**print('Trying to access variables from outside the class ')**

**print(person.name)**

**print(person.\_\_age)**

Output

Dev

30

Trying to access variables from outside the class

Dev

Traceback (most recent call last):

File "Person.py", line 16, in <module>

print(person.\_\_age)

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

You can observe that variables are still be accessed using methods, which is a part of the class. But you cannot access age directly from outside, as it is a private variable.

**Using Getter and Setter methods to access private variables**

If you want to access and change the private variables, accessor (getter) methods and mutators(setter methods) should be used, as they are part of Class.

**class Person:**

**def \_\_init\_\_(self, name, age=0):**

**self.name = name**

**self.\_\_age = age**

**def display(self):**

**print(self.name)**

**print(self.\_\_age)**

**def getAge(self):**

**print(self.\_\_age)**

**def setAge(self, age):**

**self.\_\_age = age**

**person = Person('Dev', 30)**

**#accessing using class method**

**person.display()**

**#changing age using setter**

**person.setAge(35)**

**person.getAge()**

Output

Dev

30

35

**Example of Encapsulation**

We can take an example of a capsule in which the medicine is encapsulated. We have often used examples of bigger projects in which many programmers contribute according to the task assigned to them. In the end, the whole project is done by joining the contribution of each participant. Well, this is what Encapsulation aims to achieve.

Abstraction and Encapsulation are fundamental concepts of OOP. What Encapsulation does is it takes all the worry away from the user, providing him with just the product that he required, irrespective of the way it is formed. Abstraction focuses on the working of the object instead of the how part, while Encapsulation is all about hiding the way or method of working and just providing the working model.

Classes can be a perfect example of abstraction as each member of a team is given a separate class to work on, to develop a bigger project. A person working in a class only knows his job. While Encapsulation can be said as hiding the code from the normal users by making a front end through which the user can interact through the software, without having any direct access to the code.

**Benefits of Encapsulation in Python**

Encapsulation not only ensures better data flow but also protects the data from outside sources. The concept of encapsulation makes the code self-sufficient. It is very helpful in the implementation level, as it prioritizes the ‘how’ type questions, leaving behind the complexities. You should hide the data in the unit to make encapsulation easy and also to secure the data.

**What is the need for Encapsulation in Python**

The following reasons show why developers find the Encapsulation handy and why the Object-Oriented concept is outclassing many programming languages.

1. Encapsulation helps in achieving the well-defined interaction in every application.
2. The Object-Oriented concept focuses on the reusability of code in Python. (DRY – Don’t Repeat Yourself).
3. The applications can be securely maintained.
4. It ensures the flexibility of the code through a proper code organization.
5. It promotes a smooth experience for the users without exposing any back-end complexities.
6. It improves the readability of the code. Any changes in one part of the code will not disturb another.
7. Encapsulation ensures data protection and avoids the access of data accidentally. The protected data can be accessed with the methods discussed above.

Encapsulation in Python is, the data is hidden outside the object definition. It enables developers to develop user-friendly experience. This is also helpful in securing data from breaches, as the code is highly secured and cannot be accessed by outside sources.

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| Abstraction is used to solves the problem and issues that arise at the design stage. | Encapsulation is used to solves the problem and issue that arise at the implementation stage. |
| Abstraction focuses on what the object does instead of how the details are implemented. | Encapsulation focuses on hiding the code and data into a single unit to secure the data from the outside world. |
| Abstraction can be implemented by using Interface and Abstract Class. | Encapsulation can be implemented using Access Modifiers (Public, Protected, and Private.) |
| Its application is during the design level. | Its application is during the Implementation level. |

**Python Inheritance**

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the following syntax to inherit a base class into the derived class.

**Single Inheritance:**

Python Inheritance

**Syntax**

class derived-class(base class):

<class-suite>

**Single Inheritance**

When a child class inherits only a single parent class.

class Parent:

def func1(self):

print("this is function one")

class Child(Parent):

def func2(self):

print(" this is function 2 ")

ob = Child()

ob.func1()

ob.func2()

A class can inherit multiple classes by mentioning all of them inside the bracket. Consider the following syntax.

**Syntax**

class derive-class(<base class 1>, <base class 2>, ..... <base class n>):

<class - suite>

**Python Multiple inheritance**

"In multiple inheritance a class is derived from more than one class i.e. multiple base classes. The child class in this case has features of both the parent classes."

As the name implies, python's multiple inheritance is when a class inherits from more than one class.This concept is very similar to multilevel inheritance, which also is our next topic of this course. It is also nearly the same as a single level inheritance because it contains all of the same functionality, except for the number of base classes.

While using the concept of multiple inheritance, the order of placing the base classes is very important. Let us clear the concept using an example. Suppose we have a child class named Child1, and it has two base classes, named Base1 and Base2.

#### **Example:**

class Base1:

def func1(self):

print("this is Base1 class")

class Base2:

def func2(self):

print("this is Base2 class")

class Child(Base1 , Base2):

def func3(self):

print("this is Base3 class")

obj = Child()

obj.func1()

obj.func2()

obj.func3()

**Output:**

this is Base1 class

this is Base2 class

this is Base3 class

Now, when we are looking for some attribute, let it be a constructor. Then the program will search the current class i.e., the Child1 class first. If it does not find it in the Child1, it will look in the base class that is present at the leftmost side, which is Base1. After that, the program will start moving from left to right in a sequential manner, hence searching the Base2 class at the end. We should always give attention to the ordering of the base classes because it helps us a lot when multiple classes contain the same methods and also in method overriding.

**Method Overriding:**

Override means having two methods that have the same name. They may perform same tasks or different tasks. In python, when the same method defined in the parent class is also defined in the child class, the process is known as Method overriding. This is also true when multiple classes have the same method and are linked together somehow.

There are few rules for Method overriding that should be followed:

1. The name of the child method should be the same as parents.
2. Inheritance should be there, and we need to derive a child class from a parent class
3. Both of their parameters should be the same.

In this case, the child method will run, and the reason for which, we have discussed in the paragraph above, related to ordering. Multiple inheritance is based on the same concept on which the single inheritance is based on i.e., DRY (do not repeat yourself). Multiple inheritance makes it easier to inherit methods and attributes from base classes that implement the functionality. When done right, we can reuse the code without having to copy-and-paste a similar code to implement interfaces.

Multiple inheritance and Multilevel inheritance are very similar concepts. If you have a complete understanding of multiple inheritance than understanding, multilevel will take no time. The minimum number of classes for Multilevel inheritance is the same as for multiple inheritance i.e., three.

**What is Multilevel Inheritance in Python?**

In multilevel inheritance, a class that is already derived from another class is derived by a third class. So in this way, the third class has all the other two former classes' features and functionalities. The syntax looks something like this:

class Parent1:

pass

class Derived1(Parent1):

pass

class Derived2(Derived1):

pass

Now let us dive into the priority brought by the ordering of the class. Suppose that we are looking for a constructor or a value for any variable. Our program will first check our current class i.e., Derived2, for it. If nothing found, then it will move towards Derived1 and in order at last towards Parent1 until it finds the constructor or variable in the way.

If we have the same method or variable in the base and derived class, then the order we discussed above will be followed, and the method will be overridden. Else, if the child class does not contain the same method, then the derived1 class method will be followed by the sequence defined in the paragraph above.

**Rules for Method overriding:-**

There are few rules for Method overriding that should be followed:

1. The name of the child method should be the same as parents.
2. Inheritance should be there, and we need to derive a child class from a parent class
3. Both of their parameters should be the same.

|  |
| --- |
| **Multiple inheritance VS. Multilevel inheritance** |
| |  |  | | --- | --- | | **Multiple inheritance** | **Multilevel inheritance** | | * Multiple Inheritance is where a class inherits from more than one base class. * Sometimes,multiple Inheritance makes the system more complex,that’s why it is not widely used. * Multiple Inheritance has two class levels; these are base class and derived class. | * In multilevel inheritance, we inherits from a derived class, making that derived class a base class for a new class. * Multilevel Inheritance is widely used. It is easier to handle code when using multilevel inheritance. * Multilevel Inheritance has three class levels, which are base class, intermediate class, and derived class. | |

**Advantages of Inheritance**

1. It reduces code redundancy.
2. Multilevel inheritance provides code reusability.
3. Using multilevel inheritance, code is easy to manage, and it supports code extensibility by overriding the base class functionality within child classes

**Public, Private & Protected Access Specifiers**

Access modifiers are used for the restrictions of access any other class has on the particular class and its variables and methods. In other words, access modifiers decide whether other classes can use the variables or functions of a specific class or not. The arrangement of private and protected access variables or method ensures the principle of data encapsulation. In Python, there are three types of access modifiers.

1. Public Access Modifier
2. Protected Access Modifier
3. Private Access Modifier

**Public Access Modifier:**

In public, all the functions, variables, methods can be used publicly. Meaning, every other class can access them easily without any restriction. Public members are generally methods declared in a class that is accessible from outside the class. Any ordinary class is by default, a public class. So, all the classes we had made till now in the previous tutorials were all public by default.

**Example of public access modifier:**

**class employee:**

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

**self.name=name**

**self.age=age**

**Protected Access Modifier:**

In case of a protected class, its members and functions can only be accessed by the classes derived from it, i.e. its child class or classes. No other environment is permitted to access it. To declare a class as protected, we use a single underscore “\_” sign before the data members of the class.

**Example of protected access modifier:**

**class employee:**

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

**self.\_name=name # protected attribute**

**Private Access Modifier:**

In the case of private access modifiers, the variables and functions can only be accessed within the class. The private restriction level is the highest for any class. To declare a class as private, we use a double underscore “\_­\_” sign before the data members of the class. Here is a suggestion not to try to access private variables from outside the class, because it will result in an AttributeError.

**Example of private access modifier:**

**class employee:**

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

**self.\_\_name=name # private attribute**

**self.\_\_age=age # private attribute**

**self.\_age=age # protected attribute**

**Name mangling in Python:**

Python does not have any strict rules when it comes to public, protected or private, like java. So, to protect us from using the private attribute in any other class, Python does name mangling, which means that every member with a double underscore will be changed to \_object.\_class\_\_variable when trying to call using an object. The purpose of this is to warn a user, so he does not use any private class variable or function by mistake without realizing its states.

The use of single underscore and double underscore is just a way of name mangling because Python does not take the public, private and protected terms much seriously so we have to use our naming conventions by putting single or double underscore to let the fellow programmers know which class they can access or which they can't.

**Polymorphism in Python**

Polymorphism means having vivid or different forms. In the programming world, Polymorphism refers to the ability of the function with the same name to carry different functionality altogether. It creates a structure that can use many forms of objects.

This permits functions/arguments to use entities of different types at different times.

In object-oriented programming, Polymorphism allows a particular object referring to a particular class to be used in a similar fashion as if it was a different object referring to altogether a different class.

**Example 1: Polymorphism in addition operator**

We know that the + operator is used extensively in Python programs. But, it does not have a single usage.

For integer data types, + operator is used to perform arithmetic addition operation.

**num1 = 1**

**num2 = 2**

**print(num1+num2)**

Hence, the above program outputs **3.**

Similarly, for string data types, + operator is used to perform concatenation.

**str1 = "Python"**

**str2 = "Programming"**

**print(str1+" "+str2)**

As a result, the above program outputs **Python Programming.**

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

**Function Polymorphism in Python**

There are some functions in Python which are compatible to run with multiple data types.

One such function is the len() function. It can run with many data types in Python. Let's look at some example use cases of the function.

**Example 2: Polymorphic len() function**

**print(len("Programiz"))**

**print(len(["Python", "Java", "C"]))**

**print(len({"Name": "John", "Address": "Nepal"}))**

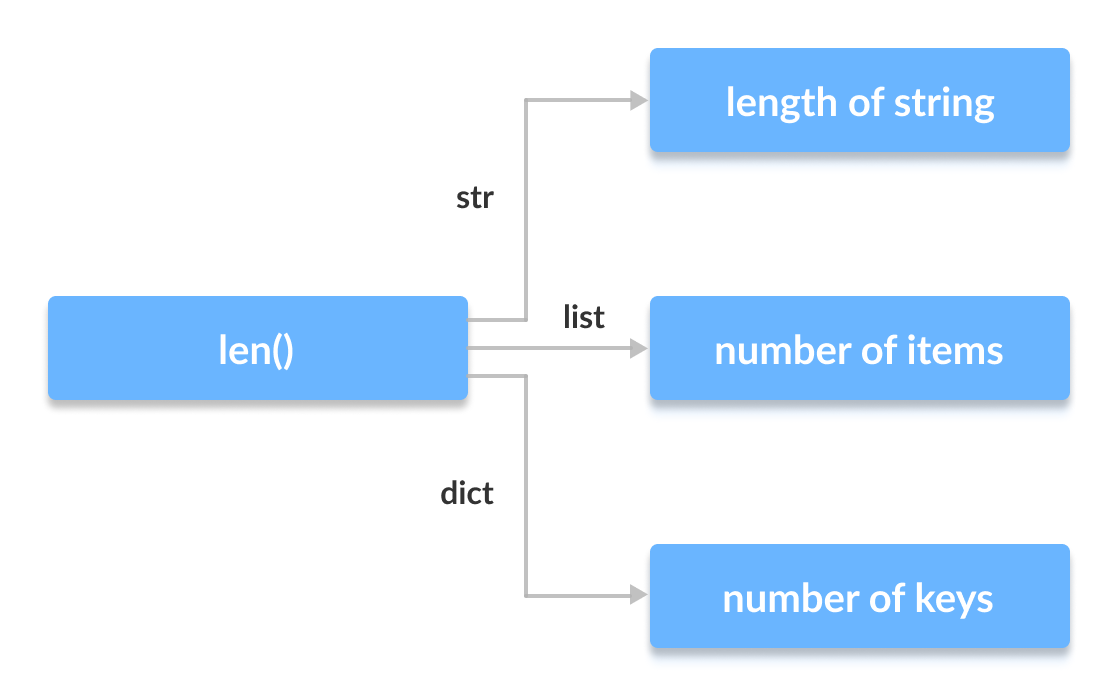
Output

9

3

2

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the len() function. However, we can see that it returns specific information about specific data types.



**Class Polymorphism in Python**

Polymorphism is a very important concept in Object-Oriented Programming.

We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

We can then later generalize calling these methods by disregarding the object we are working with. Let's look at an example:

**Example 3: Polymorphism in Class Methods**

class Cat:

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

self.name = name

self.age = age

def info(self):

print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

def make\_sound(self):

print("Meow")

class Dog:

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

self.name = name

self.age = age

def info(self):

print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

def make\_sound(self):

print("Bark")

cat1 = Cat("Kitty", 2.5)

dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):

animal.make\_sound()

animal.info()

animal.make\_sound()

Output

**Meow**

**I am a cat. My name is Kitty. I am 2.5 years old.**

**Meow**

**Bark**

**I am a dog. My name is Fluffy. I am 4 years old.**

**Bark**

**Polymorphism and Inheritance**

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as Method Overriding.

Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

Let's look at an example:

Example 4: Method Overriding

from math import pi

class Shape:

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

self.name = name

def area(self):

pass

def fact(self):

return "I am a two-dimensional shape."

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

return self.name

class Square(Shape):

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

super().\_\_init\_\_("Square")

self.length = length

def area(self):

return self.length\*\*2

def fact(self):

return "Squares have each angle equal to 90 degrees."

class Circle(Shape):

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

super().\_\_init\_\_("Circle")

self.radius = radius

def area(self):

return pi\*self.radius\*\*2

a = Square(4)

b = Circle(7)

print(b)

print(b.fact())

print(a.fact())

print(b.area())

Output

**Circle**

**I am a two-dimensional shape.**

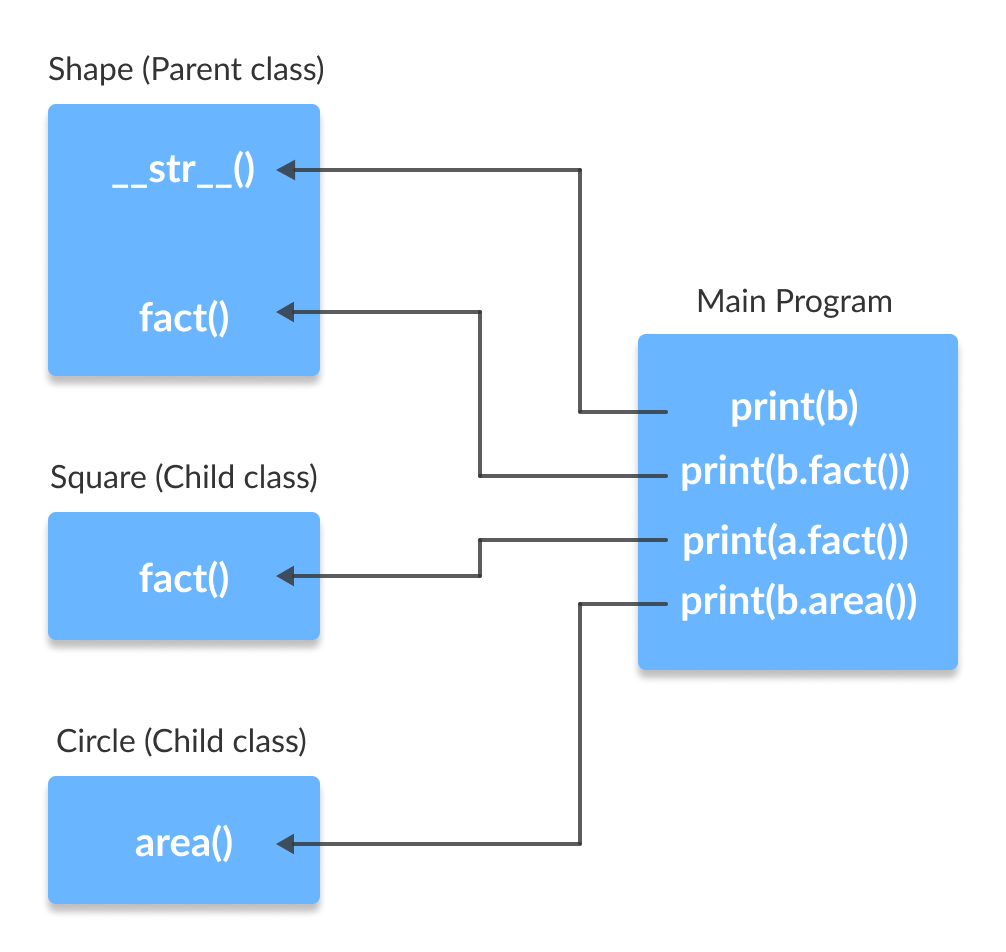
**Squares have each angle equal to 90 degrees.**

**153.93804002589985**

Here, we can see that the methods such as \_\_str\_\_(), which have not been overridden in the child classes, are used from the parent class.

Due to polymorphism, the Python interpreter automatically recognizes that the fact() method for object a(Square class) is overridden. So, it uses the one defined in the child class.

On the other hand, since the fact() method for object b isn't overridden, it is used from the Parent Shape class.



**Note:** Method Overloading, a way to create multiple methods with the same name but different arguments, is not possible in Python.

**Compile-Time Polymorphism or Method Overloading?**

Unlike many other popular object-oriented programming languages such as Java, Python doesn’t support compile-time polymorphism or method overloading. If a class or Python script has multiple methods with the same name, the method defined in the last will override the earlier one.

Python doesn’t use function arguments for method signature, that’s why method overloading is not supported in Python.

**Advantages of Polymorphism**

1. The codes and classes written once can be reused and implemented multiple times.
2. It helps in reducing the coupling between different functionalities and behavior of objects.

**Super() and Overriding In Classes**

when we want to call an already overridden method, then the use of super function comes in. It is a built-in function, so no requirement of any module import statement. What super does is, it allows us to use of the method of our superclass, that in the case of inheritance is the parent class. Syntax of using super() is

**class Parent\_Class(object):**

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

**pass**

**class Child\_Class(Parent\_Class):**

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

**super().\_\_init\_\_()**

super() returns a temporary object of the superclass that then allows you to call that superclass’s methods. The primary use case of super() is to extend the functionality of the inherited method.

We have discussed earlier that in case of method overriding, the previous method could not be called, but super makes an exception, and thus we can partially or completely use the method of the parent class too. We can even use super() to call only a specific variable we used in our overridden method. Calling the superclass built methods with super() saves us from rewriting those methods in our subclass, and allows us to swap out superclasses with minimal code changes.