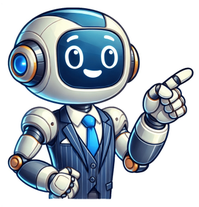
Click Here



Python has been ranked as one of the most popular programming languages in recent years. As a result, the demand for Python developers is multiplying. According to a recent study, more than 9,000 remote Python jobs are currently available. This means that the need for skilled Python developers, who know the language well, is growing rapidly. If you wish to pursue a career as a Python developer you must understand the fundamentals of OOPs concepts in Python with real-time examples. The term Object Oriented Programming (OOP) in relation to Python has attracted much attention over the years in the tech industry. In this blog, we will give you a detailed overview of OOPs in Python and simplify this concept for your better understanding. Object-Oriented Programming (OOP) is a robust model that enables users to link every feature of a program with real-world objects or concepts. It helps to write more organized and reusable codes. For example, we can create a modular code using classes and objects which can be easily tested and debugged. Python is an OOPs language making it ideal for developing various applications. OOP comprises five essential concepts, they are encapsulation, inheritance, polymorphism, abstraction, and classes and objects. They are discussed below in detail: A class is also known as a blueprint. It is used for developing objects. The following are the properties of classes: Classes are created using the class keyword in Python. The main aim of classes is to define a set of properties and methods that an object of that class will have. The instance of a class is referred to as an object. This instance has its own set of properties. It is also capable of performing actions using methods. Benefits of Classes Classes offer several benefits. Some of them are: It allows the encapsulation of data and functions within a single entity. This leads to better readability of the code. Classes allow code reusability. As we can create multiple objects from the same class. They help improve security by limiting access to sensitive data. Here is an example of a class in Python: class Rect: def __init__(self, wd, ht): self.wd = width self.ht = height def A(self): return self.wd * self.ht def P(self): return 2 * (self.wd + self.ht) Encapsulation is the technique of binding data and methods within a single class. It is implemented using access modifiers. Python classes have the following modifiers: These access modifiers determine the accessibility of a classs properties and methods. They prevent unwanted modification outside of the class. Benefits Of Encapsulation Some of the benefits offered by Encapsulation include: Encapsulation prevents unauthorized access to the data. It increases the modularity and reusability of code. It hides the implementation details of the object making it very secure. Here is an example of encapsulation in Python: class Student: def __init__(self, name, age, email): self.__name = name # private attribute self.__age = age # private attribute self.__email = email # private attribute def get_name(self): return self.__name def set_name(self, new_name): self.__name = new_name def get_age(self): return self.__age def set_age(self, new_age): if new_age < 0: print(Age cannot be negative.) else: self.__age = new_age def get_email(self): return self.__email def set_email(self, new_email): self.__email = new_email Deriving a new class from an existing one is known as inheritance. Inheritance has the following properties: The original class is called the parent or base class. The new class derived from it is termed the child or derived class. It has the ability to inherit all the properties of the parent class. Benefits of Inheritance Inheritance offers several benefits, such as: It allows us to reuse code and build on existing classes. It promotes reusability which leads to more robust and less error-prone programs. Allows for easy maintenance of code by creating a hierarchy of classes. You can pursue a Python course if you want to understand inheritance and other Python OOPs concepts easily. Example of Inheritance in Python: class Person(object): # Constructor def __init__(self, name): self.name = name # To get name def getName(self): return self.name # To check if this person is an employee def isEmployee(self): return False# Inherited or Subclass (Note Person in bracket)class Employee(Person): # Here we return true def isEmployee(self): return True# Driver codeemp = Person(Demo1) # An Object of Personprint(emp.getName(), emp.isEmployee())emp = Employee(Demo2) # An Object of Employeeprint(emp.getName(), emp.isEmployee()) Polymorphism allows us to perform a single action in different ways. It allows us to have more than one method with the same name, as long as they have different parameters. Some features of polymorphism are: It allows objects of different classes to be used interchangeably. It helps to reduce the complexity of code by providing a way to abstract away differences between related classes. It enables method overloading and method overriding. Method overloading allows multiple methods with the same name but different parameters to coexist within a single class. It leads to loosely coupled programs that can handle new types without recompilation. Benefits of Polymorphism It provides the following benefits: Polymorphism increases the flexibility of code. It allows objects of different classes to be treated as a part of the same class. It helps in code reusability. Since multiple classes share the same ion names. It allows the same method or function name to be used for different classes. Example of Polymorphism in Python class A: def __init__(self, name): self.name = name def talk(self): passclass Dog(A): def talk(self): print(Wooof!)class Cat(A): def talk(self): print(Meoww!)def make_A_talk(A): A.talk()my_dog = Dog(Rufus)make_A_talk(my_dog)my_cat = Cat(Fluffy)make_A_talk(my_cat) Abstraction allows us to focus on what the object does instead of how it does it. Some of the features of abstraction are: It only showcases the necessary information to the user. The classes with one or more abstract methods are known as Abstract classes. Whereas an abstract method is declared but does not hold an implementation. Benefits of Abstraction Abstraction provides several benefits, such as: It helps in minimizing the complexity of the code written. It makes the code reusable. It enhances security because it hides sensitive information from the user. Example of Abstraction in Python from abc import ABC, abstractmethodclass Polygon(ABC):@abstractmethoddef noofsides(self):passclass Triangle(Polygon):# overriding abstract methoddef noofsides(self):print(I have 3 sides)class Pentagon(Polygon):# overriding abstract methoddef noofsides(self):print(I have 5 sides)class Hexagon(Polygon):# overriding abstract methoddef noofsides(self):print(I have 6 sides)class Quadrilateral(Polygon):# overriding abstract methoddef noofsides(self):print(I have 4 sides)# Driver codeR = Triangle()R.noofsides()K = Quadrilateral()K.noofsides()R = Pentagon()R.noofsides()K = Hexagon()K.noofsides() The major differences between Object Oriented Programming in Python and Procedural Oriented Programming are listed below: Object Oriented ProgrammingProcedural Oriented ProgrammingProgramming MethodObject Oriented Programming is a type of programming that is entirely based on objectsProcedural Oriented Programming is a linear type method of programmingFocusIt focuses on objects and their behavior.It concentrates on procedures or functionsData and FunctionsIt emphasizes on data over functionsIt emphasizes on functions over dataCode OrganizationIt enables the better readability of codeIt can lead to a more complex codeCode ReusabilityIt encourages code reusability and modularityIt has limited code reusability and modularityApproach to ProblemsIt represents real-world entities as objectsIt represents real-world tasks as functionsCode MaintenanceIt enables easy maintenance of code by creating a hierarchy of classesIt can result in code that becomes difficult to maintain as it grows in sizeKey FeaturesIt encourages encapsulation, abstraction, and inheritanceIt does not have features like encapsulation, abstraction, and inheritance Some of the real-time examples of OOPs are: Classes and Objects An example of classes and objects can be a bookstore inventory. To manage your bookstore inventory efficiently, use a class called Book with attributes like title, author, publisher, genre, and price. Create objects of this class for each book in your inventory with their specific attributes. This will help you organize and manage your inventory effectively. Each object represents a single book, making it easy to search for specific titles or authors and update the inventory as needed. Encapsulation Banking System is the perfect example of encapsulation. Banks use encapsulation to protect sensitive customer data. It wraps the private methods or variables. They can only be accessed through public methods with proper authentication. Inheritance Inheritance allows a child class to inherit the features and characteristics of its parent class. For example, an e-commerce website can create a Parent Product class and then derive multiple Child Product classes from it, like books, electronics, and clothes. This can help to reuse the common traits and manners of the Parent Product class. Polymorphism A music app can use polymorphism to play different audio. It will use a class named Player with a method named play(). It will accept different arguments depending on the format type. The application will be able to handle various formats seamlessly. You wont have to create new methods or classes for each format. Abstraction Lets take an example of an online food ordering system. It can use abstraction to hide the complex performance details of a Payment Gateway class from the users. It only needs to showcase the methods and properties necessary for making the payments. We learned what is oops concept in Python and how it is crucial for building robust and scalable applications. These concepts, when used effectively, can help you write efficient and modular code in Python. Now that we have discussed these concepts in detail, start practicing and exploring them and take your programming skills to the next level. Object-Oriented Programming, also known as OOPs concepts in python, is what lets us develop applications using an Object-Oriented approach. It does so by clubbing together similar or related behaviors and properties and converting them into objects. In this article, I will explain the basic concepts of Object-Oriented Programming in Python programming, oop fundamentals, and features of oops. You must know Python programming before you continue. Developers often choose to useOOPs principles in Python programs because it makes code more reusable and easier to work with larger programs. OOP programs prevent you from repeating code because a class can be defined once and reused many times. In this article, you will learn about object-oriented programming (OOP) and its basic ideas, like classes and objects. We will look atobject-oriented programming in Pythonand see how it works. Next, we will discussobject-oriented programming in C++, focusing on its special features. Then, we will exploreobject-oriented programming in Javaand how it uses classes. Finally, we will give some easyobject-oriented programming examplesto help you understand these concepts better. Start your Python journey today with our free Introduction to Python course, designed specifically for beginners! Learning Objectives Understanding the Object Oriented Programming tutorial. Learning and implementing different OOPs principles in python. Table of contents While learning Object-Oriented Programming (oops concepts), I decided to dive into its history to fully know what is oops concept and it turned out to be fascinating. The term Object-Oriented Programming (OOP), also known as OOPs principles in python, was coined by Alan Kay around 1966 while he was at grad school. The language called Simula was the first programming language with the features of Object-oriented programming. It was developed in 1967 for making simulation programs, in which the most important information was called objects. Though OOPs were in the market since the early 1960s it was in the 1990s that OOPs began to grow because of C++. After that, this technique was adopted by various programmers across various programming languages, including Python. Today its application is in almost every field, such as Real-time systems, machine learning, Artificial intelligence, Data Science, expert systems, Client-server systems, Object-oriented databases, and many more. In Python, OOPs stands for Object-Oriented Programming. It is a programming paradigm that focuses on the use of objects and classes to create programs. An object is a group of interrelated variables and functions. These variables are often referred to as properties of the object, and functions are referred to as the behavior of the objects. These objects provide a better and clear structure for the program. Main principles of OOPs in Python are abstraction, encapsulation, inheritance, and polymorphism. Abstraction refers to the ability to hide the implementation details of an object from the user, while still providing a simple and easy-to-use interface. Encapsulation refers to the practice of combining data and methods within a class to protect the data from external interference. Inheritance is a way to create new classes based on existing ones, allowing the new classes to inherit the attributes and methods of the parent class. Polymorphism allows you to treat objects of different classes as if they were of the same class. Consequently, you can write generic code that works with a variety of objects. By using OOPs concepts in Python, developers can create programs that are more modular, flexible, and easier to maintain. OOPs makes it possible to write code that is reusable and scalable, reducing the overall development time and increasing code efficiency. Chekckout this article all about OOPs in Python For example, a car can be an object. If we consider the car as an object, its properties would be its color, model, price, brand, etc. And its behavior/function would be acceleration, slowing down, and gear change. Another example If we consider a dog as an object then its properties would be its color, breed, name, weight, etc., and its behavior/function would be walking, barking, playing, etc. Object-Oriented Programming is famous because it implements real-world entities like objects, hiding, inheritance, etc, in programming. It makes visualization easier because it is close to real-world scenarios. The basic differences between OOP and POP are the following: Structure: POP uses sequential functions, OOP organizes code into objects Scalability: POP suits small tasks, OOP better for larger programs Data security: POP allows universal access, OOP provides encapsulation Code reusability: OOP offers inheritance, POP has limited reusability Abstraction: OOP allows higher level of abstraction through classes Modularity: OOP facilitates better modularity and easier maintenance Flexibility: OOP is more flexible for changes and extensions Real-world modeling: OOP better represents real-world entities and relationships Data and behavior: OOP combines data and methods, POP separates them Polymorphism: OOP supports polymorphism, enhancing code flexibility Language examples: POP (C, Pascal), OOP (Java, Python, C++) Learning curve: POP generally easier to learn, OOP concepts more complex In this section, we will dive deep into the basic concepts of object oriented programming Tutorial. Here is list of OOPs concepts given below: Class Object Method Inheritance Class Constructor Polymorphism Data Abstraction Encapsulation Dynamic Binding Message Passing A straightforward answer to this question is- A class is a collection of objects. Unlike Raw data structures, classes are user-defined data structures that offer more flexibility and functionality. They make the code more manageable. Lets see how to define a class below- class class_name: class body We define a new class with the keyword class following the class_name and colon. And we consider everything you write under this after using indentation as its body. To make this more understandable, lets see an example. Consider the case of a car showroom. You want to store the details of each car. Lets start by defining a class first- class Car: pass Thats it! Note: Ive used the pass statement in place of its body because the main aim was to show how you can define a class and not what it should contain. Even the datatypes in Python, like tuples,dictionaries, etc. are classes. Before going into detail, first, understand objects and instantiation. When we define a class, only the description or a blueprint of the object is created. There is no memory allocation until we create its object. The objector instance contains real data or information. Instantiation is nothing but creating a new object/instance of a class. Lets create the object of the above class we defined- obj1 = Car() And its done! Note that you can change the object name according to your choice. Try printing this object- print(obj1) Since our class was empty, it returns the address where the object is stored i.e. 0x7fc5e677b6d8 You also need to understand the class constructor before moving forward. Until now, we have had an empty class Car, time to fill up our class with the properties of the car. The class constructor assigns values to the data members when you create an object of the class. There can be various properties of a car, such as its name, color, model, brand name, engine power, weight, price, etc. Well choose only a few for understanding purposes. class Car: def __init__(self, name, color): self.name = name self.color = color So, the properties of the car or any other object must be inside a method that we call __init__( ). We also know this __init__() method as the constructor method. We call the constructor method whenever we construct an object of the class. Now lets talk about the parameter of the __init__() method. So, the first parameter of this method has to be self. Then only will the rest of the parameters come. The two statements inside the constructor method are self.name = name self.color = color: This will create new attributes, namely name and color, and then assign the value of the respective parameters to them. The self keyword represents the instance of the class. By using the self keyword, we can access the attributes and methods of the class. It is useful in method definitions and in variable initialization. The self is explicitly used every time we define a method. Note: You can also create attributes outside of this __init__() method. But those attributes will be universal to the whole class, and you will have to assign a value to them. Suppose all the cars in your showroom are Sedan, and instead of specifying it again and again, you can fix the value of car_type as Sedan by creating an attribute outside the __init__(). class Car: car_type = "Sedan" #class attribute def __init__(self, name, color): self.name = name #instance attribute self.color = color #instance attribute Here, Instance attributes refer to the attributes inside the constructor method, i.e., self.name and self.color. And Class attributes refer to the attributes outside the constructor method, i.e., car_type. So far, weve added the properties of the car. Now its time to add some behavior. Methods are the functions we use to describe the behavior of objects. They are also defined inside a class. Look at the following code-Python Code: #You can play around with the properties of the car and also add and remove some properties in the code block down belowclass Car: car_type = "Sedan" def __init__(self, name, mileage): self.name = name self.mileage = mileage def description(self): return f"The {self.name} car gives the mileage of {self.mileage}km/l" def max_speed(self, speed): return f"The {self.name} runs at the maximum speed of {speed}km/hr"obj2 = Car("Honda City",24.1)print(obj2.description())print(obj2.max_speed(150)) We know the methods defined inside a class, other than the constructor method, as instance methods. Furthermore, we have two instance methods here description() and max_speed(). Lets talk about them individually: description()- This method returns a string with the description of the car, such as the name and its mileage. This method has no additional parameter. This method uses instance attributes. max_speed()- This method has one additional parameter and returns a string displaying the cars name and speed. Notice that the additional parameter speed is not using the self keyword. Since speed is not an instance variable, we dont use the self keyword as its prefix. Lets create an object for the class described above. obj2 = Car("Honda City",24.1)print(obj2.description())print(obj2.max_speed(150)) We created an object of the class Car and passed the required arguments. To access the instance methods, we use object_name.method_name(). The method description() didnt have any additional parameter, so we did not pass any argument while calling it. The method max_speed() has one additional parameter, so we passed one argument while calling it. Note: Three important things to remember are- You can create any number of objects in a class. If the method requires n parameters and you do not pass the same number of arguments, then an error will occur. The order of the arguments matters. Lets look at these, one by one: class Car:def __init__(self, name, mileage): self.name = nameself.mileage = mileage def max_speed(self, speed): return f"The {self.name} runs at the maximum speed of {speed}km/hr" Honda = Car("Honda City",21.4)print(Honda.max_speed(150))Skoda = Car("Skoda Octavia",13)print(Skoda.max_speed(210)) class Car: def __init__(self, name, mileage): self.name = name self.mileage = mileage Honda = Car("Honda City")print(Honda) Since we did not provide the second argument, we got this error. class Car: def __init__(self, name, mileage): self.name = nameself.mileage = mileage def description(self):return f"The {self.name} car gives the mileage of {self.mileage}km/l" Honda = Car(24.1,"Honda City")print(Honda.description()) Messed up! Notice we changed the order of arguments. There are four fundamental concepts of Object-oriented programming Inheritance, Encapsulation, Polymorphism, and Data abstraction. It is essential to know about all of these in order to understand OOPs. Till now, weve covered the basics of OOPs concepts in python; lets dive in further. Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as the Parent class. And the class that inherits the properties from the parent class is the Child class. The interesting thing is, along with the inherited properties and methods, a child class can have its own properties and methods. How to inherit a parent class? Use the following syntax: class parent_class:body of parent classclass child_class(parent_class):body of child class Lets see the implementation: class Car: #parent class def __init__(self, name, mileage): self.name = nameself.mileage = mileage def description(self):return f"The {self.name} car gives the mileage of {self.mileage}km/l"class BMW(Car): #child class passclass Audi(Car): #child class def audi_desc(self): return "This is the description method of class Audi." obj1 = BMW("BMW 7-series",39.53)print(obj1.description())obj2 = Audi("Audi A8 L",14)print(obj2.description())print(obj2.audi_desc()) We have created two child classes, namely BMW and Audi, who have inherited the methods and properties of the parent class Car. We have provided no additional features and methods in the class BMW. Whereas one additional method inside the class Audi. Notice how the instance method description() of the parent class is accessible by the objects of child classes with the help of obj1.description() and obj2.description(). The separate method of class Audi is also accessible using obj2.audi_desc(). Encapsulation, as I mentioned in the initial part of the article, is a way to ensure security. Basically, it hides the data from the access of outsiders. Such as, if an organization wants to protect an object/information from unwanted access by clients or any unauthorized person, then encapsulation is the way to ensure this. You can declare methods or attributes as protected by using a single underscore (_) before their names, like _self.name or def _method(). Both lines indicate that the attribute and method are protected, accessible only within the class and subclasses, but can be accessed by class methods and objects. Though Python uses _ just as a coding convention, it tells that you should use these attributes/methods within the scope of the class. But you can still access the variables and methods which are defined as protected, as usual. Now to actually prevent the access of attributes/methods from outside the scope of a class, you can use private members. In order to declare the attributes/method as private members, use double underscore ( ) in the prefix. Such as self.name or def __method(); Both of these lines tell that the attribute and method are private and access is not possible from outside the class. classcar: def__init__(self,name,mileage): self._name = name #protected variableself.mileage=mileage defdescription(self):returnf"The{self._name}cargivesthemileageof{self.mileage}km/l" obj = car("BMW 7-series",39.53)#accessing protected variable via class methodprint(obj.description())#accessing protected variable directly from outsideprint(obj._name)print(obj.mileage) Notice how we accessed the protected variable without any error. It is clear that access to the variable is still public. Let us see how encapsulation works Class Car: def __init__(self, name, mileage): self.__name = name #private variableself.mileage = mileage def description(self):return f"The {self.__name} car gives the mileage of {self.mileage}km/l" obj = Car("BMW 7-series",39.53)#accessing private variable via class methodprint(obj.description())#accessing private variable directly from outsideprint(obj.mileage)print(obj.__name) When we tried accessing the private variable using the description() method, we encountered no error. But when we tried accessing the private variable directly outside the class, then Python gave us an error stating: car object has no attribute __name. You can still access this attribute directly using its mangled name.Name manglingis a mechanism we use for accessing the class members from outside. The Python interpreter rewrites any identifier with __var as _ClassName__var. And using this you can access the class member from outside as well. class Car: def __init__(self, name, mileage): self.__name = name #private variableself.mileage = mileage def description(self):return f"The {self.__name} car gives the mileage of {self.mileage}km/l" obj = Car("BMW 7-series",39.53)#accessing private variable via class methodprint(obj.description())#accessing private variable directly from outsideprint(obj.mileage)print(obj._Car__name) #mangled name Note: that the mangling rules design mostly avoids accidents. But it is still possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger. This is a Greek word. If we break the term Polymorphism, we get poly-many and morph-forms. So Polymorphism means having many forms. In OOPs principles it refers to the functions having the same names but carrying different functionalities. class Audi: defdescription(self):print("ThisthedescriptionfunctionofclassAUDI.")class BMW: defdescription(self):print("ThisthedescriptionfunctionofclassBMW.") audi = Audi()bmw=BMW()for car in (audi,bmw): car.description() When you call the function using the object audi, it calls the function of class Audi. Similarly, when you call it using the object bmw, it calls the function of class BMW. We use Abstraction for hiding the internal details or implementations of a function and showing its functionalities only. This is similar to the way you know how to drive a car without knowing the background mechanism. Or you know how to turn on or off a light using a switch, but you dont know what is happening behind the socket. Any class with at least one abstract function is an abstract class. To create an abstraction class, first, you need to import the ABC class fromabcmodule. This lets you create abstract methods inside it. ABC stands for Abstract Base Class. from abc import ABCclass abs_class(ABC): Body of the class The important thing is you cannot create an object for the abstract class with the abstract method. For example- from abc import ABC, abstractmethodclass Car(ABC): def __init__(self,name): self.name = name @abstractmethod def price(self,x): pass obj = Car("Honda City") Now the question is, how do we use this abstraction exactly? The answer is, by using inheritance. from abc import ABC, abstractmethodclass Car(ABC): def __init__(self,name): self.name = name def description(self): print("This the description function of class car.") @abstractmethod def price(self,x): passclass new(Car): def price(self,x): print(f"The {self.name}'s price is {x} lakhs.") obj=new("HondaCity")obj.description()obj.price(25) Car is the abstract class that inherits from theABCclass from the abc module. Notice how I have an abstract method (price()) and a concrete method (description()) in the abstract class. This is because the abstract class can include both of these kinds of functions, but a normal class cannot. The other class inheriting from this abstract class isnew(). This method defines the abstract method (price()), which is how we use abstract functions. After the user creates objects from thenew()class and invokes the price() method, the definitions for the price method inside thenew()class comes into play. These definitions are hidden from the user. The Abstract method is just providing a declaration. The child classes need to provide the definition. But when the description() method is called for the object of the new() class, i.e,obj, the Cars description() method is invoked since it is not an abstract method. Dynamic binding (or late binding) means deciding which method to call when the program runs, not when the code is written. This makes your code more flexible because the exact method to run is chosen based on the object type at runtime. Example: class Car:def start_engine(self):raise NotImplementedError("Subclass must implement this method")class Honda(Car):def start_engine(self):return "The Honda engine has started."class BMW(Car):def start_engine(self):return "The BMW engine has started."def start_car_engine(car):print(car.start_engine())honda_car = Honda()bmw_car = BMW()start_car_engine(honda_car) # Output: The Honda engine has started.start_car_engine(bmw_car) # Output: The BMW engine has started. In this example, the start_engine() method is chosen based on whether the object is a Honda or a BMW when the program runs. This shows dynamic binding in action because the method that gets called depends on the actual object type at runtime. Message passing is how objects talk to each other in object-oriented programming. One object sends a message (calls a method) to another object to make it do something or to get some information. class Car:def __init__(self, brand):self.brand = branddef start_engine(self):print(f"The {self.brand} engine has started.")class Driver:def __init__(self, name):self.name = namedef drive(self, car):print(f"{self.name} is driving the car.")car.start_engine()car = Car("Toyota")driver = Driver("John")driver.drive(car) In this example, the Driver object sends a message to the Car object by calling its start_engine method through the drive method. This shows how objects interact by sending messages to each other. Weve now covered the basic concepts of Object-Oriented Programming (OOP) in this Python tutorial. In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. OOPs, concepts in python, aim to implement real-world entities like inheritance, polymorphisms, encapsulation, etc., in the programming. Here are some articles you can read to learn more about Python: Key Takeaways We learned the difference between Object-oriented programming and procedure-oriented programming. The two main aspects of OOPs are class and object. The four principles of OOPs are Inheritance, Polymorphism, Encapsulation, and Abstraction. Q1. What is the OOP concept? A. The OOP concept revolves around designing software using objects, which represent real-world entities. Moreover, it organizes code by grouping related properties and behaviors into these objects. Q2. What is object-oriented programming by example? A. In object-oriented programming, you create classes to define objects. For example, you might define a Car class with attributes like color and model, and methods like drive() and stop(). Then, you instantiate objects such as myCar from this class. Q3. What are the four basics of OOP? A. The four basics of OOP include encapsulation, inheritance, polymorphism, and abstraction. First, encapsulation bundles data and methods. Next, inheritance allows new classes to derive from existing ones. Then, polymorphism enables objects to be treated as instances of their parent class. Finally, abstraction simplifies complex systems by modeling classes appropriate to the problem. Q4. What is OOPs and its features? A. OOPs, or Object-Oriented Programming System, includes features like encapsulation, inheritance, polymorphism, and abstraction. Additionally, these features help you create modular, reusable, and scalable code, making software development more efficient and manageable. OOP is an abbreviation that stands for Object-oriented programming paradigm. It is defined as a programming model that uses the concept of objects which refers to real-world entities with state and behavior. This chapter helps you become an expert in using object-oriented programming support in Python language.Python is a programming language that supports object-oriented programming. This makes it simple to create and use classes and objects. If you do not have any prior experience with object-oriented programming, you are at the right place. Let's start by discussing a small introduction of Object-Oriented Programming (OOP) to help you.Procedural Oriented ApproachEarly programming languages developed in 50s and 60s are recognized as procedural (or procedure oriented) languages.A computer program describes procedure of performing certain task by writing a series of instructions in a logical order. Logic of a more complex program is broken down into smaller but independent and reusable blocks of statements called functions.Every function is written in such a way that it can interface with other functions in the program. Data belonging to a function can be easily shared with other in the form of arguments, and called function can return its result back to calling function.Prominent problems related to procedural approach are as follows Its top-down approach makes the program difficult to maintain.It uses a lot of global data items, which is undesired. Too many global data items would increase memory overhead.It gives more importance to process and doesn't consider data of same importance and takes it for granted, thereby it moves freely through the program.Movement of data across functions is unrestricted. In real-life scenario where there is unambiguous association of a function with data it is expected to process. In the real world, we deal with and process objects, such as student, employee, invoice, car, etc. Objects are not only data and not only functions, but combination of both. Each real-world object has attributes and behavior associated with it.AttributesName, class, subjects, marks, etc., of studentName, designation, department, salary, etc., of employeeInvoice number, customer, product code and name, price and quantity, etc., in an invoiceRegistration number, owner, company, brand, horsepower, speed, etc., of carEach attribute will have a value associated with it. Attribute is equivalent to data.BehaviorProcessing attributes associated with an object.Compute percentage of student's marksCalculate incentives payable to employeeApply GST to invoice valueMeasure speed of carBehavior is equivalent to function. In real life, attributes and behavior are not independent of each other, rather they co-exist.The most important feature of object-oriented approach is defining attributes and their functionality as a single unit called class. It serves as a blueprint for all objects having similar attributes and behavior.In OOP, class defines what are the attributes its object has, and how is its behavior. Object, on the other hand, is an instance of the class.Principles of OOPs ConceptsObject-oriented programming paradigm is characterized by the following principles ClassObjectEncapsulationInheritancePolymorphismClass & ObjectA class is an user-defined prototype for an object that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation.An object refers to an instance of a certain class. For example, an object named obj that belongs to a class Circle is an instance of that class. A unique instance of a data structure that is defined by its class. An object comprises both data members (class variables and instance variables) and methods.ExampleThe below example illustrates how to create a class and its object in Python.# defining classclass Smartphone: # constructor def __init__(self, device, brand): self.device = device self.brand = brand # method of the class def description(self): return f"{self.device} of {self.brand} supports Android 14"# creating object of the classphoneObj = Smartphone("Smartphone", "Samsung")print(phoneObj.description()) On executing the above code, it will display the following output Smartphone of Samsung supports Android 14EncapsulationData members of class are available for processing to functions defined within the class only. Functions of class on the other hand are accessible from outside class context. So object data is hidden from environment that is external to class. Class function (also called method) encapsulates object data so that unwarranted access to it is prevented.ExampleIn this example, we are using the concept of encapsulation to set the price of desktop.class Desktop: def __init__(self): self.__max_price = 25000 def sell(self): return f"Selling Price: {self.__max_price}" def set_max_price(self, price): if price > self.__max_price: self.__max_price = price# ObjectdesktopObj = Desktop()print(desktopObj.sell()) # modifying the price directlydesktopObj.__max_price = 35000print(desktopObj.sell()) # modifying the price using setter functiondesktopObj.set_max_price(35000)print(desktopObj.sell()) When the above code is executed, it produces the following result Selling Price: 25000Selling Price: 25000Selling Price: 35000InheritanceA software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called base or parent class, while new class is called child or sub class.Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.SyntaxDerived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name class SubClassName (ParentClass1[, ParentClass2, ...]): 'Optional class documentation string' class_suiteExampleThe following example demonstrates the concept of Inheritance in Python #!/usr/bin/python# define parent classclass Parent: parentAttr = 100 def __init__(self): print ("Calling parent constructor") def parentMethod(self): print ("Calling parent method") def setAttr(self, attr): Parent.parentAttr = attr def getAttr(self): print ("Parent attribute :", Parent.parentAttr)# define child classclass Child(Parent): def __init__(self): print ("Calling child constructor") def childMethod(self): print ("Calling child method")# instance of childc = Child() # child calls its method c.childMethod() # calls parent's method c.parentMethod() # again call parent's method c.setAttr(200) # again call parent's method c.getAttr() When the above code is executed, it produces the following result Calling child constructorCalling child methodCalling parent methodParent attribute : 200Similar way, you can drive a class from multiple parent classes as follows class A: # define your class A.....class B: # define your class B.....class C(A, B): # subclass of A and B.....You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.The issubclass(sub, sup) boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.The isinstance(obj, Class) boolean function returns true if obj is an instance of class Class or is an instance of a subclass of ClassPolymorphismPolymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.ExampleIn this example, we are overriding the parent's method.# define parent classclass Parent: def myMethod(self): print ("Calling parent method")# define child classclass Child(Parent): def myMethod(self): print ("Calling child method")# instance of childc = Child()# child calls overridden method c.myMethod() When the above code is executed, it produces the following result Calling child methodBase Overloading Methods in PythonFollowing table lists some generic functionality that you can override in your own classes Sr.No.Method, Description & Sample Call1__init__ (self [,args...])Constructor (with any optional arguments)Sample Call : obj = className(args)2__del__(self)Destructor, deletes an objectSample Call : del obj3__repr__(self)Evaluable string representationSample Call : repr(obj)4__str__(self)Printable string representationSample Call : str(obj)5__cmp__ (self, x)Object comparisonSample Call : cmp(obj, x)Overloading Operators in PythonSuppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.You could, however, define the __add__ method in your class to perform vector addition and then the plus operator would behave as per expectation Exampleclass Vector: def __init__(self, a, b): self.a = a self.b = b def __str__(self): return 'Vector (%d, %d)' % (self.a, self.b) def __add__(self,other): return Vector(self.a + other.a, self.b + other.b)v1 = Vector(2,10)v2 = Vector(5,-2)print (v1 + v2)When the above code is executed, it produces the following result Vector(7,8)Image SourceObject-Oriented Programming (OOP) is a method of designing applications and programs using objects. It simplifies software development and maintenance by providing major concepts such as classes, objects, inheritance, and polymorphism. In this article, we will explore these concepts in detail using Python, a popular object-oriented programming language.Object-Oriented Programming BasicsPython, a flexible and widely-used programming language, provides strong support for OOP concepts, making it an excellent choice for both beginners and experienced programmers.What is OOP?OOP is an approach to programming that structures a software system around objects rather than functions and logic. These objects represent real-world entities and are instances of classes, which can be thought of as blueprints for creating objects. The main aim of OOP is to increase the flexibility and maintainability of programs.Core Concepts of OOPTo understand OOP, it is crucial to grasp its four core concepts: encapsulation, abstraction, inheritance, and polymorphism.Encapsulation: Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, called a class. This helps protect the data from outside interference and misuse by restricting access to certain components. In Python, encapsulation is implemented through access modifiers: public, protected, and private.class Car: def __init__(self, make, model, year): self.make = make # public attribute self._model = model # protected attribute self.__year = year # private attribute def get_year(self): return self.__yearmy_car = Car("Toyota", "Corolla", 2020)print(my_car.make) # Accessing public attributeprint(my_car._model) # Accessing protected attributeprint(my_car.get_year()) # Accessing private attribute through a methodAbstraction: Abstraction simplifies complex reality by modeling classes appropriate to the problem. It hides the complex implementation details and exposes only the necessary parts. This allows the programmer to manage complexity by focusing on the high-level operations rather than the low-level details.from abc import ABC, abstractmethodclass Animal(ABC): @abstractmethod def make_sound(self): passclass Dog(Animal): def make_sound(self): return "Bark"class Cat(Animal): def make_sound(self): return "Meow"animals = [Dog(), Cat()]for animal in animals: print(animal.make_sound())Inheritance: Inheritance allows a class to inherit attributes and methods from another class, promoting code reusability. The class that is inherited from is called the superclass, and the class that inherits is called the subclass.class Person: def __init__(self, name, age): self.name = name self.age = age def greet(self): return f"Hello, my name is {self.name}."class Student(Person): def __init__(self, name, age, student_id): super().__init__(name, age) self.student_id = student_id def get_id(self): return f"My student ID is {self.student_id}."student = Student("Kaitlyn", 21, "S12345")print(student.greet())print(student.get_id())Polymorphism: Polymorphism allows methods to perform different tasks based on the object they are acting upon. It can be achieved through method overriding, where a subclass provides a specific implementation of a method that is already defined in its superclass.class Shape: def area(self): passclass Rectangle(Shape): def __init__(self, width, height): self.width = width self.height = height def area(self): return self.width * self.heightclass Circle(Shape): def __init__(self, radius): self.radius = radius def area(self): return 3.14 * self.radius * self.radiusshapes = [Rectangle(3, 4), Circle(5)]for shape in shapes: print(shape.area())Benefits of OOPOOP offers several benefits that make it a popular choice for software development:Modularity: Code is organized into distinct classes and objects, making it easier to manage and understand.Reusability: Once a class is written, it can be reused to create multiple objects, reducing redundancy.Scalability: New functionalities can be added by creating new classes or modifying existing ones without affecting other parts of the program.Maintainability: The modular nature of OOP makes it easier to troubleshoot and update code.Understanding these core concepts and benefits of OOP is important for developing efficient and maintainable software. In the following sections, we will be going deeper into each of these concepts with practical examples in Python.Classes and ObjectsClasses and objects are fundamental concepts in Object-Oriented Programming (OOP). Understanding these concepts is key to mastering OOP and leveraging its benefits in Python.ClassesA class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have. Classes encapsulate data for the object and define behaviors that operate on that data.To define a class in Python, use the class keyword followed by the class name and a colon. Inside the class, define attributes and methods that the class should have. Let's look at an example:class Dog: # Class attribute species = "Canis familiaris" # Initializer / Instance attributes def __init__(self, name, age): self.name = name self.age = age # Instance method def description(self): return f"{self.name} is {self.age} years old." # Another instance method def speak(self, sound): return f"{self.name} says {sound}"In this example, the Dog class has a class attribute species and instance attributes name and age. It also has two instance methods: description and speak.Class attributes are shared by all instances of the class.Instance attributes are unique to each instance of the class.Methods define behaviors for the objects created from the class.ObjectsAn object is an instance of a class. Creating an object from a class is known as instantiation. Each object can have its own unique values for instance attributes, while sharing the class attributes and methods defined by the class.To create an object, use the class name followed by parentheses, optionally passing arguments if the class initializer requires them. Lets create some objects from the Dog class:# Instantiate the Dog classbuddy = Dog("Buddy", 9)lucy = Dog("Lucy", 3)# Accessing instance attributesprint(buddy.name) # Output: Buddyprint(lucy.age) # Output: 3# Accessing class attributeprint(buddy.species) # Output: Canis familiarisprint(lucy.species) # Output: Canis familiaris# Calling instance methodsprint(buddy.description()) # Output: Buddy is 9 years old.print(lucy.speak("Woof Woof")) # Output: Lucy says Woof WoofIn this example:buddy and lucy are objects (instances) of the Dog class.Each object has its own values for the name and age attributes.Both objects share the species class attribute.Instance methods like description and speak can be called on each object.Initializers in ClassesThe __init__ method is a special method called an initializer. It is automatically called when an object is created. The purpose of the initializer is to set up the initial state of the object by assigning values to the instance attributes.class Car: def __init__(self, make, model, year): self.make = make self.model = model self.year = year def description(self): return f"{self.year} {self.make} {self.model}"# Create instances of the Car classcar1 = Car("Toyota", "Corolla", 2020)car2 = Car("Honda", "Civic", 2019)print(car1.description()) # Output: 2020 Toyota Corollaprint(car2.description()) # Output: 2019 Honda CivicIn this example, the __init__ method initializes each Car object with the make, model, and year attributes.Methods in ClassesMethods are functions defined inside a class that describe the behaviors of the objects. There are three types of methods in Python classes:Instance methods: Operate on the instance attributes of the object.Class methods: Operate on the class itself. They use the @classmethod decorator and take cls as the first parameter.Static methods: Do not operate on the class or its instances. They use the @staticmethod decorator and do not take self or cls as parameters.class Circle: pi = 3.14159 # Class attribute def __init__(self, radius): self.radius = radius # Instance attribute def area(self): return Circle.pi * self.radius * self.radius @classmethod def change_pi(cls, new_pi): cls.pi = new_pi @staticmethod def info(): return "This is a Circle class"# Create an instance of Circlecircle = Circle(5)print(circle.area()) # Output: 78.53975# Call class methodCircle.change_pi(3.14)print(circle.area()) # Output: 78.5# Call static methodprint(Circle.info()) # Output: This is a Circle classIn this example:area is an instance method that calculates the area of the circle.change_pi is a class method that modifies the class attribute pi.info is a static method that provides information about the class.By understanding classes and objects, you can create strong and reusable code structures. In the next section, we will go into inheritance, which allows us to build upon existing classes to create new ones.InheritanceInheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows one class to inherit attributes and methods from another class. This promotes code reusability and helps in creating a hierarchical relationship between classes. Inheritance enables the creation of a new class based on an existing class.What is Inheritance?Inheritance allows a new class, called the subclass or derived class, to inherit attributes and methods from an existing class, called the superclass or base class. The subclass can add new attributes and methods, or modify the existing ones, enhancing or specializing the functionality of the superclass.Types of InheritancePython supports different types of inheritance, including:Single Inheritance: A subclass inherits from one superclass.Multiple Inheritance: A subclass inherits from multiple superclasses.Multilevel Inheritance: A subclass inherits from another subclass.Hierarchical Inheritance: Multiple subclasses inherit from a single superclass.Hybrid Inheritance: A combination of two or more types of inheritance.Single InheritanceIn single inheritance, a subclass inherits from one superclass. This is the simplest form of inheritance.class Animal: def __init__(self, name): self.name = name def eat(self): return f"{self.name} is eating."class Dog(Animal): def __init__(self, name, breed): super().__init__(name) self.breed = breed def bark(self): return f"{self.name} is barking."# Create an instance of Dogbuddy = Dog("Buddy", "Golden Retriever")print(buddy.eat()) # Output: Buddy is eating.print(buddy.bark()) # Output: Buddy is barking.In this example, the Dog class inherits from the Animal class. The Dog class adds a new attribute breed and a method bark.Multiple InheritanceIn multiple inheritance, a subclass inherits from more than one superclass. This can be useful for combining functionalities from multiple classes.class Canine: def run(self): return "Running fast!"class Pet: def be_cute(self): return "Being cute!"class Dog(Canine, Pet): def bark(self): return "Woof woof!"# Create an instance of Dogbuddy = Dog()print(buddy.run()) # Output: Running fast!print(buddy.be_cute()) # Output: Being cute!print(buddy.bark()) # Output: Woof woof!In this example, the Dog class inherits from both Canine and Pet classes, combining their functionalities.Multilevel InheritanceIn multilevel inheritance, a subclass inherits from another subclass, forming a chain of inheritance.class Animal: def eat(self): return "Eating..."class Mammal(Animal): def give_birth(self): return "Giving birth to live young."class Dog(Mammal): def bark(self): return "Barking..."# Create an instance of Dogbuddy = Dog()print(buddy.eat()) # Output: Eating...print(buddy.give_birth()) # Output: Giving birth to live young.print(buddy.bark()) # Output: Barking...In this example, the Dog class inherits from the Mammal class, which in turn inherits from the Animal class.Hierarchical InheritanceIn hierarchical inheritance, multiple subclasses inherit from a single superclass.class Animal: def eat(self): return "Eating..."class Dog(Animal): def bark(self): return "Barking..."class Cat(Animal): def meow(self): return "Meowing..."# Create instances of Dog and Catbuddy = Dog()whiskers = Cat()print(buddy.eat()) # Output: Eating...print(buddy.bark()) # Output: Barking...print(whiskers.eat()) # Output: Eating...print(whiskers.meow()) # Output: Meowing...In this example, both Dog and Cat classes inherit from the Animal class.Method OverridingMethod overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. This allows the subclass to modify or extend the behavior of the superclass method.class Animal: def speak(self): return "Animal sound."class Dog(Animal): def speak(self): return "Bark!"class Cat(Animal): def speak(self): return "Meow!"# Create instances of Dog and Catdog = Dog()cat = Cat()print(dog.speak()) # Output: Bark!print(cat.speak()) # Output: Meow!In this example, the Dog and Cat classes override the speak method of the Animal class to provide their specific implementations.Benefits of InheritanceInheritance offers several benefits in software development:Code Reusability: Inheritance promotes code reusability by allowing new classes to reuse the existing code from the superclass.Extensibility: New functionalities can be added to existing classes by creating subclasses without modifying the existing code.Maintainability: Changes made to the superclass are automatically inherited by all its subclasses, making maintenance easier.By grasping and using inheritance, you can create more modular, reusable, and maintainable code. In the next section, we will explore polymorphism, which allows methods to perform different tasks based on the objects they act upon.PolymorphismPolymorphism is a key concept in Object-Oriented Programming (OOP) that allows methods to perform different tasks based on the object they are acting upon. It provides a way to use a single interface to represent different underlying data types. In Python, polymorphism is primarily achieved through method overriding and method overloading.What is Polymorphism?Polymorphism allows objects of different classes to be treated as objects of a common superclass. This is especially useful when you want to perform the same operation in different ways depending on the specific object involved. Polymorphism is implemented through two main mechanisms: method overriding and method overloading.Method OverridingMethod overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows the subclass to modify or extend the behavior of the superclass method. Method overriding is a way to achieve runtime polymorphism.class Animal: def speak(self): return "Some generic animal sound"class Dog(Animal): def speak(self): return "Bark!"class Cat(Animal): def speak(self): return "Meow!"# Create instances of Dog and Catdog = Dog()cat = Cat()# Call the overridden methodsprint(dog.speak()) # Output: Bark!print(cat.speak()) # Output: Meow!In this example, the Dog and Cat classes override the speak method of the Animal class to provide their specific implementations. When the speak method is called on a Dog or Cat object, the corresponding method in the subclass is executed.Method OverloadingPython does not support method overloading in the traditional sense found in other languages like Java or C++. Instead, Python achieves polymorphism through default arguments or variable-length arguments.class Math: def add(self, a, b, c=0): return a + b + c# Create an instance of Mathmath = Math()# Call the add method with different numbers of argumentsprint(math.add(2, 3)) # Output: 5print(math.add(2, 3, 4)) # Output: 9In this example, the add method is defined with a default argument c. This allows the method to be called with two or three arguments, achieving a form of method overloading.Polymorphism with Functions and MethodsPolymorphism is not limited to class methods. It can also be achieved with functions that can take objects of different classes and perform the same operation.class Rectangle: def __init__(self, width, height): self.width = width self.height = height def area(self): return self.width * self.heightclass Circle: def __init__(self, radius): self.radius = radius def area(self): return 3.14 * self.radius * self.radiusdef print_area(shape): print(f"The area is {shape.area()}")# Create instances of Rectangle and Circlerect = Rectangle(3, 4)circ = Circle(5)# Call the print_area function with different objectsprint_area(rect) # Output: The area is 12print_area(circ) # Output: The area is 78.5In this example, the print_area function can take any object that has an area method. When called with a Rectangle or Circle object, it prints the area of the respective shape. This demonstrates polymorphism by allowing the same function to operate on different types of objects.Polymorphism with Abstract Base ClassesAbstract base classes (ABCs) in Python can be used to define a common interface for a group of related classes. ABCs use the abc module and the @abstractmethod decorator to define abstract methods that must be implemented by subclasses.from abc import ABC, abstractmethodclass Animal(ABC): @abstractmethod def speak(self): passclass Dog(Animal): def speak(self): return "Bark!"class Cat(Animal): def speak(self): return "Meow!"# Create instances of Dog and Catdog = Dog()cat = Cat()# Call the speak method on the instancesprint(dog.speak()) # Output: Bark!print(cat.speak()) # Output: Meow!In this example, the Animal class is an abstract base class with an abstract method speak. The Dog and Cat classes inherit from Animal and provide their specific implementations of the speak method. This makes sure that all subclasses of Animal have a speak method, promoting a consistent interface.Benefits of PolymorphismPolymorphism offers several benefits in software development:Flexibility: Code can be written to work with objects of multiple types, making it more flexible and reusable.Extensibility: New classes can be added with little or no modification to existing code, enhancing the extensibility of the system.Maintainability: Polymorphic code is easier to maintain and extend, as changes in the superclass are automatically reflected in the subclasses.Using polymorphism, combined with inheritance and encapsulation, forms the foundation of powerful object-oriented design.ConclusionObject-Oriented Programming (OOP) is an essential concept in modern software development, providing a structured approach to organizing code through classes, objects, inheritance, and polymorphism. By mastering these concepts in Python, you can write more modular, reusable, and maintainable programs. OOP not only simplifies complex systems but also enhances code readability and flexibility, making it a powerful tool for developers.Thank you for reading! If you find this article helpful, please consider highlighting, clapping, responding or connecting with me on Twitter/X as its very appreciated and helps keeps content like this free! Object-oriented programming (OOP) is a style of programming that heavily relies on objects. These objects can have attributes and methods. While attributes store data, methods define behavior.Like many other programming languages, Python supports both OOP and functional programming. However, OOP becomes valuable when writing large-sized and complex programs.In this article, you will learn the benefits of OOP in Python, how to define a class, class and instance attributes, and instance methods. You will also learn the concept of encapsulation and how to implement inheritance between classes in Python.To fully understand this article, you should have the following prerequisites:Basic knowledge of the Python programming language.Familiarity with the concepts of functions and objects in programming.Experience working with a code editor or integrated development environment (IDE).Basic understanding of how to use a command-line terminal or console.Now let's dive in. Object-oriented programming (OOP) offers several benefits when organizing and managing code. By grouping related data and functions into logical classes, OOP promotes code structure and simplifies maintenance, especially as programs grow in size and complexity. The modular approach makes it easier to understand, modify, and reuse code, thereby reducing development time.Another benefit of OOP is its ability to provide a clear and relatable programming style, which can be more helpful for developers. The use of objects and the relationships between them mirror real-world concepts. This makes it easier to reason about code and design complex systems.Finally, OOP's concepts such as encapsulation and inheritance, contribute to code robustness by promoting data protection and code reusability. OOP in Python heavily relies on the concept of class. You can think of a class as a blueprint that is used to create objects. To illustrate this, imagine that you have a blueprint for a speaker. You can use this blueprint to build multiple speakers. Each speaker that is created using the blueprint is an instance of the blueprint. Also, each created speaker has its attributes such as color, model, and name. They will also have their methods showing a certain kind of behavior such as volume up and volume down. To define a class, you have to use the class keyword, provided by Python, then followed by the name of the class and a colon:class ClassName: passNow using our earlier illustration, lets create a class named Speaker. In your code editor, create a file named classes.py and copy the following lines of code into it:class Speaker: passspeaker_one = Speaker()print(speaker_one)In the preceding code, you defined a class named Speaker without any attributes or methods. You used the keyword pass in the class body. In Python, the pass keyword does nothing and is usually used as a placeholder.In line four of the code, you created an instance of the Speaker class and assigned it to the variable speaker_one. This process is also known as instantiating an object from a class. You then printed speaker_one in line five.Now run the code by running the command python classes.py in your terminal. You will get an output similar to this in your terminal:The output shows that speaker_one is an object. 0x10087f8e0 in the output shows the memory address where Python stores the object on your computer.The memory address in your terminal output will be different.You can create several more instances from the Speaker class and they will all be different from one another. To verify this, lets use the equality comparison (\==) operator. In your classes.py file, delete print(speaker_one) and add the following to the bottom of your code:speaker_two = Speaker() if speaker_one == speaker_two: print("speaker one is the same as speaker two")else: print("speaker one is different from speaker two")Now run python classes.py in your terminal. You will get the following output:speaker one is different from speaker two In this section, you'll add attributes to your Speaker class. You can see attributes as data stored within an object. While class attributes are common to all instances created from your class, instance attributes are unique to each instance.Now modify your classes.py file by replacing your code with the following:class Speaker: brand = "Beatpill" def __init__(self, color, model): self.color = color self.model = model You should already be familiar with line one of your new code. In line two, you defined a class attribute, named brand, and assigned it the value Beatpill. Class attributes are variables that belong to the class itself, rather than to individual instances of the class. The effect of class attributes is that all instances you create from your class will inherit and share that class attribute and its value. In this case, every instance you create from your Speaker class will share the same brand value.Line four of your code defines an __init__ method, which takes in three parametersself, color, and model. This method will be called anytime you create a new instance from your Speaker class. The self parameter is a reference to the Speaker class and it's a convention in Python to always have it as the first parameter in a __init__ method. Line five and six set instance attributes, color, and model to your Speaker class.So, every time you create a new instance from your class, you have to provide arguments for your color and model attributes. Not doing this will result in an error.Now add the following to the bottom of your code:speaker_one = Speaker("black", "85XB5")speaker_two = Speaker("red", "Y8F33") print(f"speaker one is {speaker_one.color} while speaker two is {speaker_two.color}")By adding the preceding code, you created two instances of the Speaker classspeaker_one and speaker_two. The arguments in each instance represent the values of their color and model attributes. You then printed a message that displays the color attribute of each object. Now, if you run the command python classes.py in your terminal, you will get the output:speaker one is black while speaker two is redTo see that both instances share the same brand class attribute, modify your print() function like this:print(f"speaker one is {speaker_one.color} while speaker two is {speaker_two.color}", speaker_one.brand, speaker_two.brand, sep="",)Now run python classes.py and you will get the following output:speaker one is black and speaker two is redBeatpillBeatpill In addition to class and instance attributes, classes can also have methods, known as instance methods. Instance methods are functions defined within a class that operate on instances of the class. They use the class and instance attributes to provide behavior and functionality to the objects.To add instance methods, modify your code file as follows:class Speaker: brand = "Beatpill" def __init__(self, color, model): self.color = color self.model = model def power_on(self): print(f"Powering on {self.color} {self.model} speaker.") def power_off(self): print(f"Powering off {self.color} {self.model} speaker.") speaker_one = Speaker("black", "85XB5")speaker_two = Speaker("red", "Y8F33")In the example above, we've added two instance methodspower_on() and power_off(). These methods, just like the __init__ method, take self as the first argument.The power_on() method prints a message indicating that the speaker of the given color and model is being powered on. Similarly, the power_off() method prints a message about powering off the speaker.To call these methods, add the following to the bottom of your file:speaker_one.power_on()speaker_two.power_off()Now run python classes.py in your terminal and you will get the following output:Powering on black 85XB5 speaker.Powering off red Y8F33 speaker. Encapsulation is one of the core concepts of OOP. It refers to the bundling of data (attributes) and methods within a class. Encapsulation provides data protection and control over how the code interacts with an object's internal state.You can achieve encapsulation in Python by defining private attributes and methods within a class. By convention, private attributes and methods are prefixed with a single underscore (_). While Python does not have strict private modifiers like some other

languages, the underscore prefix serves as a warning to other developers not to access or modify the attributes and methods directly from outside the class.Here's an example of encapsulation in Python:class Speaker: brand = "Beatpill" def __init__(self, color, model): self._color = color self._model = model def power_on(self): print(f"Powering on {self._color} {self._model} speaker.") def power_off(self): print(f"Powering off {self._color} {self._model} speaker.") def get_color(self): return self._color def set_color(self, new_color): self._color = new_color speaker_one = Speaker("black", "85XB5")speaker_one.power_on() print(speaker_one._color) print(speaker_one.get_color()) speaker_one.set_color("white")print(speaker_one.get_color())In the preceding example, the color and model attributes are private attributes of the Speaker class. Although it is possible to access and modify these attributes directly from outside the class by using, for example, print(speaker_one._color), this practice is discouraged. Doing so violates encapsulation and can lead to unintended behavior or data corruption.Instead, the class provides getter get_color() and setter set_color() methods to access and modify the private attributes in a controlled manner. These methods act as an interface for interacting with the object's internal state, ensuring data integrity and enabling additional validation.Encapsulation promotes code modularity, maintainability, and data protection by separating the internal state from the public interface (methods). It allows you to change the internal state without affecting the code that uses the class, as long as the public interface remains the same. Inheritance is another core concept of OOP. It allows classes to inherit attributes and methods from other classes. The new class inherits attributes and methods from the existing class, known as the parent or base class. The new class is called the child or derived class.Inheritance promotes code reuse by allowing the child class to inherit and extend the functionality of the parent class. This helps in creating hierarchical relationships between classes and organizing code in a more structured and logical manner.In Python, inheritance is implemented using the following syntax:class DerivedClass(BaseClass): To see how inheritance works, modify your code as follows:class Speaker: brand = "Beatpill" def __init__(self, color, model): self._color = color self._model = model def power_on(self): print(f"Powering on {self._color} {self._model} speaker.") def power_off(self): print(f"Powering off {self._color} {self._model} speaker.") class SmartSpeaker(Speaker): def __init__(self, color, model, voice_assistant): super().__init__(color, model) self._voice_assistant = voice_assistant def say_hello(self): print(f"Hello, I am {self._voice_assistant}") smart_speaker = SmartSpeaker("black", "XYZ123", "Alexa")smart_speaker.power_on() smart_speaker.say_hello()In the preceding code, the SmartSpeaker class is derived from the Speaker class. The SmartSpeaker class inherits the attributes and methods of the Speaker class.The __init__ method of the SmartSpeaker class calls the __init__method of the Speaker class using super().__init__(color, model). This ensures that the inherited _color and _model attributes are properly initialized. Also, the SmartSpeaker class has its own attribute _voice_assistant, and a new method say_hello.Now run python classes.py in your terminal and you will get the following output:Powering on black XYZ123 speaker.Hello, I am Alexa Throughout this article, we highlighted the benefits of Object-Oriented Programming (OOP) and demonstrated how to define classes, create and use instance attributes and methods. We provided practical examples to illustrate the implementation of classes in Python, as well as key OOP concepts such as encapsulation and inheritance.Applying the lessons covered in this article will help you improve your Python programming experience and increase the efficiency and maintainability of your code. //realpython.com/python3-object-oriented-programming/ //docs.python.org/3/tutorial/classes.html //realpython.com/python-double-underscore/ Explore Python oops concepts with our detailed guide, covering classes, objects, inheritance, polymorphism, encapsulation, and abstraction. Master object-oriented programming in Python for more structured and efficient code. Object-oriented programming (OOP) is a programming paradigm that uses objects and classes to design software. Objects are self-contained entities that contain both data and behavior. Classes are blueprints for creating objects. OOP has several advantages over other programming paradigms, including: Reusability:OOP code is more reusable than code written using other paradigms. This is because classes can be reused to create multiple objects. Maintainability:OOP code is easier to maintain than code written using other paradigms. This is because classes encapsulate data and behavior, making it easier to understand and modify the code. Modularity:OOP code is more modular than code written using other paradigms. This is because classes can be used to break down a program into smaller, more manageable pieces. OOP is a programming paradigm that organizes code by modeling real-world entities as objects. Each object has attributes (data) and methods (functions) that operate on the data. Pythons OOP implementation is intuitive and flexible, making it accessible to learners and powerful for professionals. A class in Python is like a blueprint for creating objects. It defines the structure and behavior of objects. Think of it as a recipe for baking cookies; the class is the recipe, and the objects are the cookies. Lets create a simple Person class: class Person: def __init__(self, name, age): self.name = name self.age = age def greet(self): return f"Hello, my name is {self.name} and I'm {self.age} years old."The Person class has attributes name and age. The __init__ method initializes these attributes when an object is created. The greet method returns a greeting message. Now, lets create a Person object: person = Person("Alice", 30)print(person.greet()) # Output: "Hello, my name is Alice and I'm 30 years old." An object is an instance of a class. Its a tangible realization of the class, with its own data and behavior. Create a Person object: person = Person("Bob", 25) person is an object of the Person class, with its unique name and age. A method in Python is a function defined inside a class. It represents the behavior or actions that objects of the class can perform. Weve already seen the greet method in the Person class. When called on a Person object, it returns a greeting message. print(person.greet()) # Output: "Hello, my name is Bob and I'm 25 years old." Inheritance allows a class to inherit attributes and methods from another class. It promotes code reusability and hierarchical structuring. Create a subclass Student that inherits from Person: class Student(Person): def __init__(self, name, age, student_id): super().__init__(name, age) self.student_id = student_id def study(self, subject): return f"{self.name} is studying {subject}."Student inherits attributes and methods from Person and adds its own attribute student_id and method study. Now, create a Student object: student = Student("Charlie", 20, "S12345")print(student.greet()) # Output: "Hello, my name is Charlie and I'm 20 years old."print(student.study("Math")) # Output: "Charlie is studying Math." Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and dynamic behavior. Create a function that works with both Person and Student objects: def introduce(entity): return entity.greet()print(introduce(person)) # Output: "Hello, my name is Bob and I'm 25 years old."print(introduce(student)) # Output: "Hello, my name is Charlie and I'm 20 years old." The introduce function accepts any object with a greet method, demonstrating polymorphism. Data abstraction hides complex implementation details and exposes only necessary functionalities. Classes in Python naturally facilitate abstraction. Imagine a BankAccount class that abstracts the inner workings of a bank: class BankAccount: def __init__(self, balance): self.balance = balance def deposit(self, amount): self.balance += amount def withdraw(self, amount): if amount self.__max_price: self.__max_price = price# ObjectdesktopObj = Desktop()print(desktopObj.sell()) # modifying the price directlydesktopObj.__max_price = 35000print(desktopObj.sell()) # modifying the price using setter functiondesktopObj.set_max_price(35000)print(desktopObj.sell()) When the above code is executed, it produces the following result Selling Price: 25000Selling Price: 25000Selling Price: 35000InheritanceA software modelling approach of OOP enables extending capability of an existing class to build new class instead of building from scratch. In OOP terminology, existing class is called base or parent class, while new class is called child or sub class.Child class inherits data definitions and methods from parent class. This facilitates reuse of features already available. Child class can add few more definitions or redefine a base class function.SyntaxDerived classes are declared much like their parent class; however, a list of base classes to inherit from is given after the class name class SubClassName (ParentClass1[, ParentClass2, ...]): 'Optional class documentation string' class_suiteExampleThe following example demonstrates the concept of Inheritance in Python #!/usr/bin/python# define parent classclass Parent: parentAttr = 100 def __init__(self): print ("Calling parent constructor") def parentMethod(self): print ("Calling parent method") def setAttr(self, attr): Parent.parentAttr = attr def getAttr(self): print ("Parent attribute :", Parent.parentAttr)# define child classclass Child(Parent): def __init__(self): print ("Calling child constructor") def childMethod(self): print ("Calling child method")# instance of childc = Child() # child calls its method c.childMethod() # calls parent's method c.parentMethod() # again call parent's method c.setAttr(200) # again call parent's method c.getAttr() When the above code is executed, it produces the following result Calling child constructorCalling child methodCalling parent methodParent attribute : 200Similar way, you can drive a class from multiple parent classes as follows class A: # define your class A.....class B: # define your class B.....class C(A, B): # subclass of A and B.....You can use issubclass() or isinstance() functions to check a relationships of two classes and instances.The issubclass(sub, sup) boolean function returns true if the given subclass sub is indeed a subclass of the superclass sup.The isinstance(obj, Class) boolean function returns true if obj is an instance of class Class or is an instance of a subclass of ClassPolymorphismPolymorphism is a Greek word meaning having multiple forms. In OOP, polymorphism occurs when each sub class provides its own implementation of an abstract method in base class.You can always override your parent class methods. One reason for overriding parent's methods is because you may want special or different functionality in your subclass.ExampleIn this example, we are overriding the parent's method.# define parent classclass Parent: def myMethod(self): print ("Calling parent method")# define child classclass Child(Parent): def myMethod(self): print ("Calling child method")# instance of childc = Child()# child calls overridden method c.myMethod() When the above code is executed, it produces the following result Calling child methodBase Overloading Methods in PythonFollowing table lists some generic functionality that you can override in your own classes Sr.No.Method, Description & Sample Call1__init__ (self [,args...])Constructor (with any optional arguments)Sample Call : obj = className(args)2__del__(self)Destructor, deletes an objectSample Call : del obj3__repr__(self)Evaluable string representationSample Call : repr(obj)4__str__(self)Printable string representationSample Call : str(obj)5__cmp__ (self, x)Object comparisonSample Call : cmp(obj, x)Overloading Operators in PythonSuppose you have created a Vector class to represent two-dimensional vectors, what happens when you use the plus operator to add them? Most likely Python will yell at you.You could, however, define the __add__ method in your class to perform vector addition and then the plus operator would behave as per expectation Exampleclass Vector: def __init__(self, a, b): self.a = a self.b = b def __str__(self): return 'Vector (%d, %d)' % (self.a, self.b) def __add__(self,other): return Vector(self.a + other.a, self.b + other.b)v1 = Vector(2,10)v2 = Vector(5,-2)print (v1 + v2)When the above code is executed, it produces the following result Vector(7,8)

What are the basic concepts of oops in python. Python oop. Does python have oops concepts. What is oop programming in python. List of oops concepts in python.