**Q.69. What makes Python object-oriented?**

Python is object-oriented because it follows the Object-Oriented programming paradigm. This is a paradigm that revolves around classes and their instances (objects). With this kind of programming, we have the following features:

* Encapsulation
* Abstraction
* Inheritance
* Polymorphism
* Data hiding

Python is considered an object-oriented programming (OOP) language due to its support for the principles of object-oriented programming. These principles enable developers to structure their code in a way that models real-world entities and their interactions. Here’s what makes Python object-oriented:

### 1. ****Classes and Objects****

* **Classes**: Classes are blueprints for creating objects. They encapsulate data for the object and methods to manipulate that data. In Python, you define a class using the class keyword.

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 am {self.age} years old."

* **Objects**: Objects are instances of classes. They are created using the class definition and can use the methods defined in the class.

person1 = Person("Alice", 30)

print(person1.greet()) # Output: Hello, my name is Alice and I am 30 years old.

**2. Encapsulation**

* **Encapsulation**: This principle involves bundling the data (attributes) and methods that operate on the data into a single unit (class). It also involves restricting direct access to some of the object's components, which is achieved using access modifiers.

class Person:

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

self.\_\_name = name # Private attribute

self.\_\_age = age # Private attribute

def get\_name(self):

return self.\_\_name

def set\_name(self, name):

self.\_\_name = name

**3. Inheritance**

* **Inheritance**: This allows a new class to inherit the attributes and methods of an existing class. It promotes code reuse and establishes a natural hierarchy between classes.

class Employee(Person):

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

super().\_\_init\_\_(name, age)

self.employee\_id = employee\_id

def get\_employee\_id(self):

return self.employee\_id

**4. Polymorphism**

* **Polymorphism**: This principle allows objects of different classes to be treated as objects of a common superclass. It also refers to the ability of different classes to define methods with the same name, but those methods perform different tasks.

def print\_greeting(person):

print(person.greet())

class Student(Person):

def greet(self):

return f"Hi, I'm {self.name}, a student."

student1 = Student("Bob", 22)

print\_greeting(student1) # Output: Hi, I'm Bob, a student.

**5. Abstraction**

* **Abstraction**: This principle involves hiding complex implementation details and showing only the essential features of the object. In Python, this is often achieved through abstract base classes and interfaces, though Python does not enforce strict abstraction.

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def make\_sound(self):

pass

class Dog(Animal):

def make\_sound(self):

return "Woof!"

class Cat(Animal):

def make\_sound(self):

return "Meow!"

**Summary**

Python is object-oriented because it supports:

* **Classes** and **objects**: Fundamental constructs in OOP.
* **Encapsulation**: Bundling data and methods.
* **Inheritance**: Reusing and extending existing classes.
* **Polymorphism**: Using methods in different ways based on the object.
* **Abstraction**: Hiding complex details and exposing only necessary features.

These features enable developers to create programs that are modular, reusable, and easier to maintain.

constructor

In object-oriented programming (OOP), a constructor is a special method used for initializing objects of a class. When an object is created, the constructor is automatically called to set up the initial state of the object by assigning values to its attributes or performing any necessary setup.

**Key Concepts of a Constructor:**

1. **Automatic Invocation:**
   * The constructor is automatically invoked when a new object of a class is instantiated. It’s typically used to set initial values for attributes or perform any setup needed when the object is created.
2. **Constructor Name:**
   * In many programming languages, the constructor has a specific name. In Python, the constructor is named \_\_init\_\_, in Java, it's named the same as the class, and in C++, it's also named after the class.
3. **No Return Type:**
   * Constructors do not have a return type, not even void, because their purpose is to initialize the object, not to return a value.
4. **Types of Constructors:**
   * **Default Constructor:** A constructor that takes no arguments. If no constructor is defined in a class, most languages automatically provide a default constructor.
   * **Parameterized Constructor:** A constructor that takes arguments to initialize the attributes with specific values.

**URL Routing in Django:**

**URL routing** is crucial for defining how your web application responds to different URLs.

In Django, URL routing is handled by the urls.py file, where you define URL patterns that map to specific views.

# urls.py

from django.urls import path

from . import views

urlpatterns = [

path('', views.home, name='home'), # Home page

path('about/', views.about, name='about'), # About page

path('blog/<int:post\_id>/', views.blog\_post, name='blog\_post'), # Dynamic URL with post\_id

path('contact/', views.contact, name='contact'), # Contact page

]

**Explanation:**

* **path('', views.home, name='home'):** This maps the root URL (/) to the home view function. When a user visits the home page, the home function is called.
* **path('about/', views.about, name='about'):** This maps the /about/ URL to the about view function.
* **path('blog/<int:post\_id>/', views.blog\_post, name='blog\_post'):** This defines a dynamic URL where post\_id is a variable part of the URL. The blog\_post function will receive post\_id as an argument.
* **name='blog\_post':** The name parameter gives the URL pattern a name, which can be used in templates or for reverse URL resolution.

Orm

Oop

url routing

mvt

Django advantages

Mvt concept

The Model-View-Template (MVT) is an architectural design pattern used in Django, a popular Python web framework. MVT is similar to the Model-View-Controller (MVC) pattern used in other frameworks but with a Django-specific twist. It separates the application logic into three interconnected components:

1. **Model**
2. **View**
3. **Template**
4.  The **Model** is the data layer of the application. It is responsible for defining the structure of the database, including the schema of the tables, relationships between tables, and constraints. In Django, models are defined as Python classes, where each class represents a database table.
5.  The model handles all the data-related logic, including querying the database, creating, reading, updating, and deleting records (CRUD operations).

**. View:**

* The **View** is the business logic layer of the application. It processes user requests, interacts with the model to retrieve or manipulate data, and determines what data to send back to the user. In Django, a view is usually a Python function or a class-based view (CBV) that takes a web request and returns a web response.
* Views handle the core logic of your application, deciding which models to use and which templates to render.

**Template:**

* The **Template** is the presentation layer of the application. It defines how the data received from the view should be presented to the user. Templates are HTML files with placeholders for dynamic content, allowing you to create the user interface.
* Django templates allow you to include logic, like loops and conditionals, using Django’s template language.
* The **Model** is responsible for the data and database logic.
* The **View** handles the business logic, processing requests and deciding which template to render.
* The **Template** is responsible for the presentation, rendering the HTML with dynamic conten