Q from U tube interviews

1. What is fundamental data types in Python? and why it is called fundamental?
2. **Integer (int):** Represents whole numbers without any decimal point.
3. **Float (float):** Represents real numbers with a decimal point.
4. **String (str):** Represents sequences of characters enclosed within single quotes (**'**) or double quotes (**"**).
5. **Boolean (bool):** Represents the truth values **True** and **False**.
6. **NoneType (None):** Represents the absence of a value or a null value.

called "fundamental" because they are the basic and essential data types provided by the language itself. They form the foundation upon which more complex data structures and objects are built.

1. interpreted language, high level language ,meaning and its advantage

**interpreted language :-**  the source code is executed line by line, directly by an interpreter program, without separate compilation step The interpreter reads the source code, analyzes it, and executes it directly,

**high level language:-** is designed to be human-readable and easier to understand, write, and maintain compared to low-level languages. It provides abstractions that hide the details of the underlying hardware.

1. difference between for loop and while loop?

operator + is used to add two integers as well as join two strings and merge two lists. It is achievable because ‘+’ operator is overloaded by int class and str class.

Monkey patching in python refers to modifying or updating a piece of code or class or any module at the runtime. In simple words, we can change the behavior or working of a class/ module at the runtime without changing the whole python code.

Python programming language is rich with built-in operators.

Python supports the following types of operators:

* Arithmetic Operators
* Assignment Operators
* Comparison (Relational) Operators
* Logical Operators
* Identity Operators
* Bitwise Operators
* Membership Operators

|  |  |  |
| --- | --- | --- |
| Python Logical Operators | | |
| Symbol | Operator Name | Description |
| or | Logical OR | The condition is true if any of the two operands are non-zero. |
| and | Logical AND | If both the operands are true, then the condition is true. |
| not | Logical NOT | It is used to reverse the logical state of its operand. |

**Identity Operators**

For comparing the memory locations of two objects, identity operators are used.

***WEB SCRAPING***

libraries like BeautifulSoup and Scrapy in Python, utilizing web browser automation tools like Selenium.

**How can you handle dynamic content or JavaScript-rendered websites in web scraping?**

* Answer: To handle dynamic content, you can use browser automation tools like Selenium, which allows you to simulate user interactions and extract data from JavaScript-rendered pages.

**What is the difference between web scraping and web crawling?**

* Answer: Web scraping involves extracting specific data from web pages, while web crawling involves systematically browsing the web to index and gather information from multiple pages or websites.

**DJANGO COMMAND**

1. **<model\_name>.objects.all()**: View all items in a model.
   * This command retrieves all objects (records) from the specified model in the database.
2. **<model\_name>.objects.get()**: Retrieve a single item from a model based on specified criteria.
   * This command retrieves a single object (record) from the specified model that matches the given criteria.
3. **<model\_name>.objects.filter()**: Filter items in a model based on specified criteria.
   * This command retrieves a queryset (collection of objects) from the specified model that match the given criteria.
4. **<model\_name>.objects.create()**: Create a new item in a model.
   * This command creates a new object (record) in the specified model with the provided data.
5. **<model\_instance>.save()**: Save changes to a model instance.
   * This command saves any changes made to the attributes of the specified model instance to the database.
6. **<model\_instance>.delete()**: Delete a model instance from the database.
   * This command deletes the specified model instance from the database.
7. **python manage.py makemigrations**: Create database migration files.
   * This command generates database migration files based on changes made to the models in your Django project.
8. **python manage.py migrate**: Apply database migrations.
   * This command applies database migrations to synchronize the database schema with the current state of the models in your Django project.
9. **python manage.py runserver**: Run the Django development server.
   * This command starts the Django development server, allowing you to test your Django project locally.
10. **python manage.py createsuperuser**: Create a superuser for the Django admin interface.
    * This command prompts you to create a superuser account with administrative privileges for accessing the Django admin interface.

**Question 1: What is Django?**

* high-level Python web framework
* It follows the Model-View-Controller (MVC) architectural pattern, although in Django's case, it's referred to as Model-View-Template (MVT).
* help developers build web applications quickly by providing various built-in features and tools, such as an ORM (Object-Relational Mapping) for interacting with databases, a URL routing system, a template engine, and more.

Question 2: What is the purpose of Django's ORM (Object-Relational Mapping)?

Django's ORM is used to **interact** with databases in a Pythonic way, without directly writing SQL queries.

define database models as Python classes,

Django's ORM takes care of **translating** these Python objects into database tables and performing CRUD (Create, Read, Update, Delete) operations on them

* Django's ORM is a layer of abstraction that allows developers to interact with databases using high-level Python objects rather than writing raw SQL queries.
* Key points about Django's ORM:
  + **Models**: In Django, database tables are represented as Python classes called "models". Each model class corresponds to a database table, and each attribute of the model class represents a database field.
  + **CRUD operations**: Django's ORM provides methods for performing CRUD (Create, Read, Update, Delete) operations on database objects. For example, **Model.objects.create()** is used to create a new object, **Model.objects.all()** retrieves all objects, **Model.objects.filter()** filters objects based on specified criteria, and so on.
  + **Automatic migrations**: Django's ORM includes a built-in migration system that automatically manages changes to the database schema as models are modified. Developers can create migration files to track changes to models and apply migrations to update the database schema accordingly.
  + **Database agnostic**: Django's ORM is database agnostic, meaning it supports multiple database backends, including PostgreSQL, MySQL, SQLite, and Oracle, allowing developers to switch databases without changing their code.

**Question: What is Django Middleware, and how is it used?**

Answer:

* Django middleware is a framework of hooks into Django's request/response processing. It's a lightweight, low-level plugin system for globally altering Django's input or output.
* Key points about Django middleware:
* **Request and response processing**: Middleware operates on requests and responses handled by Django. It allows you to modify request objects before they reach the view layer and response objects before they are returned to the client.
* **Common use cases**: Middleware is commonly used for authentication, session management, CSRF protection, caching, compression, and other cross-cutting concerns.
* **Custom middleware**: Django allows developers to create custom middleware by defining classes that implement the appropriate methods (**\_\_init\_\_**, **\_\_call\_\_**, **process\_request**, **process\_response**, etc.).
* Example usage of Django middleware:

pythonCopy code

class MyMiddleware: def \_\_init\_\_(self, get\_response): self.get\_response = get\_response def \_\_call\_\_(self, request): # Code to be executed for each request before the view (optional) response = self.get\_response(request) # Code to be executed for each response before returning to the client (optional) return response

In this example, a custom middleware class **MyMiddleware** is defined. It implements the **\_\_init\_\_** and **\_\_call\_\_** methods to define initialization and request/response processing logic, respectively.

Question: What is Django's **Authentication system**, and how is it used?

Answer:

* Django's authentication system provides built-in support for user authentication, authorization, and session management in web applications.
* Key points about Django's authentication system:
  + **User model**: Django provides a built-in **User** model (**django.contrib.auth.models.User**) that represents a user of the application. The **User** model includes fields for username, password, email, first name, last name, etc.
  + **Authentication backends**: Django supports authentication backends, allowing developers to use different authentication methods (e.g., username/password, OAuth, LDAP) by writing custom authentication backends.
  + **Login and logout views**: Django provides built-in views (**django.contrib.auth.views.LoginView** and **django.contrib.auth.views.LogoutView**) for handling user login and logout functionality. These views can be included in URL patterns and customized as needed.
  + **Authentication middleware**: Django includes authentication middleware (**django.contrib.auth.middleware.AuthenticationMiddleware**) that associates authenticated users with the current request. This middleware adds the **user** attribute to the request object, providing access to the authenticated user's information.
  + **Permissions and authorization**: Django's authentication system includes support for permissions and authorization. Developers can define custom permissions, assign permissions to users or groups, and use decorators (**@login\_required**, **@permission\_required**) to restrict access to views based on user authentication and permissions.

Example usage of Django's authentication system:

DJ CRUD

class StatusViewSet(viewsets.ModelViewSet):

queryset = Status.objects.all()

    serializer\_class = StatusSerializer

    def get\_queryset(self):

        return Status.objects.all()

    def list(self, request, \*args, \*\*kwargs):

        try:

            #search\_query = request.query\_params.get('search', '')

            queryset = self.get\_queryset()

            serializer = self.get\_serializer(queryset, many=True)

            return Response({"success":True, "message":"List retrieved successfully", "data":serializer.data}, status=status.HTTP\_200\_OK)

        except Exception as e:

            return Response({"success":False, "message":"Something went wrong. Please try again later.", "data":None, "errors":str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)

class StatusViewSet(viewsets.ModelViewSet):

* This line defines a Django viewset class named **StatusViewSet**.
* **viewsets.ModelViewSet** is a Django REST Framework class that provides CRUD (Create, Retrieve, Update, Delete) operations for a model. It is a convenience class that combines several mixins and simplifies the creation of CRUD views.
* queryset = Status.objects.all()
* This line sets the initial queryset for the viewset to retrieve all objects of the **Status** model.
* It retrieves all instances of the **Status** model from the database using **Status.objects.all()**.
* serializer\_class = StatusSerializer
* This line specifies the serializer class to be used for serializing and deserializing instances of the **Status** model.
* It associates the **StatusSerializer** class with the **StatusViewSet**, indicating that instances of the **Status** model should be serialized/deserialized using the **StatusSerializer**.
* def get\_queryset(self):
  + This line defines a method named **get\_queryset()** within the **StatusViewSet** class.
  + It overrides the default **get\_queryset()** method provided by **ModelViewSet** to customize the queryset used by the viewset.

1. return Status.objects.all()
   * This line returns the queryset containing all instances of the **Status** model.
   * It ensures that whenever the **get\_queryset()** method is called, it returns all instances of the **Status** model, regardless of any additional filtering.
2. def list(self, request, \*args, \*\*kwargs):
   * This line defines a method named **list()** within the **StatusViewSet** class.
   * It overrides the default **list()** method provided by **ModelViewSet** to customize the behavior of listing objects.
3. queryset = self.get\_queryset()
   * This line retrieves the queryset containing all instances of the **Status** model using the **get\_queryset()** method defined earlier.
   * It ensures that the queryset used for listing objects is consistent with the queryset used for other operations.
4. serializer = self.get\_serializer(queryset, many=True)
   * This line initializes a serializer instance to serialize the queryset obtained in the previous step.
   * It uses the **get\_serializer()** method provided by **ModelViewSet** to create a serializer instance for the queryset, indicating that it contains multiple instances (**many=True**).
5. return Response({"success":True, "message":"List retrieved successfully", "data":serializer.data}, status=status.HTTP\_200\_OK)
   * This line returns a success response containing the serialized data obtained from the queryset.
   * It includes a success message, the serialized data, and a status code indicating success (HTTP 200 OK).
6. except Exception as e:
   * This line starts an exception handling block to catch any exceptions that occur during the execution of the **list()** method.
7. return Response({"success":False, "message":"Something went wrong. Please try again later.", "data":None, "errors":str(e)}, status=status.HTTP\_500\_INTERNAL\_SERVER\_ERROR)
   * This line returns an error response if an exception occurs during the execution of the **list()** method.
   * It includes an error message, details of the exception (**str(e)**), and a status code indicating an internal server error (HTTP 500 Internal Server Error).

How Can You Combine Multiple QuerySets In A View?

combined\_queryset = queryset1.union(queryset2)

**Using Raw SQL Queries**:

combined\_queryset = Model.objects.raw('SELECT \* FROM table1 UNION SELECT \* FROM table2')

Explain Django Architecture? Also Explain Model, Template And Views.

Django follows a Model-View-Template (MVT) architecture, which is similar to the Model-View-Controller (MVC) pattern but with some differences. Let's break down each component of Django's architecture:

1. **Model**:
   * In Django, a model is a Python class that represents a database table. It defines the structure and behavior of the data stored in the database.
   * Models are typically defined in the **models.py** module of Django apps using Django's built-in ORM (Object-Relational Mapping) system.
   * Each attribute of the model class represents a database field, and each instance of the model class represents a row in the corresponding database table.
   * Example:

from django.db import models

class MyModel(models.Model):

    name = models.CharField(max\_length=100)

    age = models.IntegerField()

**View**:

* In Django, a view is a Python function or class that receives HTTP requests and returns HTTP responses.
* Views are responsible for processing user input, querying the database, and generating output to be sent back to the client.
* Views can perform various tasks such as rendering HTML templates, serializing data for APIs, handling form submissions, and interacting with models.
* Views are typically defined in the **views.py** module of Django apps and are mapped to URLs using URL patterns.
* Example:
* from django.http import HttpResponse
* def my\_view(request):
* return HttpResponse("Hello, world!")

**Template**:

* In Django, a template is an HTML file that contains static content as well as placeholders and template tags.
* Templates are responsible for generating the final output to be sent to the client's browser.
* Template tags are special syntax provided by Django that allow you to insert dynamic content, loop over data, conditionally display content, and include other templates.
* Templates are typically stored in the **templates** directory of Django apps and can be rendered using Django's template engine.
* Example:
* <!DOCTYPE html>
* <html>
* <head>
* <title>Hello, {{ name }}</title>
* </head>
* <body>
* <h1>Hello, {{ name }}</h1>
* </body>
* </html>

In summary, Django's architecture consists of models, views, and templates working together to handle requests, interact with the database, and generate dynamic content for web applications. Models represent the data, views handle the request-response cycle, and templates define the presentation layer.

Why Is Django Called A Loosely Coupled Framework?

because its components are designed to be independent and modular, allowing developers to use them interchangeably and customize them according to their needs.

**Modularity**: Django's components, such as models, views, and templates, are designed to be separate and independent. This allows developers to work on different parts of the application independently without affecting other components.

**Flexibility**: Developers can choose to use only the components they need and replace or extend them as required. For example, Django's ORM can be swapped out for a different database library, and its template engine can be replaced with another one.

**Reusability**: Django's components are designed to be reusable across different projects. For example, developers can create custom template tags or model mixins and use them in multiple projects without modification.

front-end developers and backend developers can work simultaneously on the project

What do you mean by the CSRF Token?

CSRF stands for Cross-Site Request Forgery, which is a type of attack where an attacker performing actions on a website without user’s consent. To prevent CSRF attacks, Django includes a CSRF protection mechanism that relies on a CSRF token.

A CSRF token is a unique, random value generated by Django and included in each form rendered on a website. When a user submits a form, the CSRF token is also submitted as part of the form data. The server then verifies that the submitted CSRF token matches the one associated with the user's session. If the tokens match, the request is considered legitimate, and the server processes it. If they don't match, the server rejects the request, preventing CSRF attacks.