Notes from https://www.w3schools.com/django/

1. Understanding Django Apps: Learn how Django organizes applications within a project and how to create new apps using python manage.py startapp <app_name>.
2. Defining Models: Explore how to create database models using Django's ORM (Object-Relational Mapping) by defining classes that inherit from django.db.models.Model.
3. Creating Model Fields: Learn about different model field types such as CharField, IntegerField, DateField, etc., and how to use them to define the structure of your database tables.
4. Admin Interface: Understand how to register models with the Django admin site to make managing data easier, and how to customize the admin interface using admin.py within your app.
5. Migrations: Learn how to create and apply migrations to keep your database schema synchronized with your models using python manage.py makemigrations and python manage.py migrate.
6. Querysets: Explore how to retrieve data from the database using Django's querysets, and learn about methods like filter(), get(), exclude(), annotate(), etc.
7. Creating Views: Understand how to create views to handle HTTP requests using Django's class-based views or function-based views, and how to map URLs to views using urls.py.
8. Templates: Learn about Django's template system and how to create HTML templates using Django's template language to render dynamic content.
9. Template Inheritance: Understand how to use template inheritance to create a base template and extend it in other templates to avoid redundancy.
10. Static Files: Learn how to manage static files (e.g., CSS, JavaScript, images) in Django projects and serve them using the {% static %} template tag.
11. URLs and Namespaces: Explore how to organize URLs using Django's URL dispatcher, and how to use namespaces to avoid naming conflicts.
12. Forms: Understand how to create HTML forms using Django's forms.py, handle form submissions, perform form validation, and render form errors.
13. Form Handling: Learn how to process form data in views, validate it, and save it to the database using Django's form handling mechanisms.
14. Class-Based Views: Dive deeper into class-based views, learning about different types of class-based views such as ListView, DetailView, FormView, etc., and how to use mixins to extend their functionality.
15. User Authentication: Explore Django's built-in authentication system, including how to create login, logout, and registration views, and how to restrict access to certain views to authenticated users.
16. User Authorization: Understand how to implement user authorization by checking permissions and roles using Django's built-in decorators or by creating custom decorators.
17. Middleware: Learn about Django middleware and how to use it to modify requests or responses globally across your project.
18. Signals: Explore Django's signal system, which allows decoupled applications to get notified when certain actions occur elsewhere in the application.

19. Handling Files: Learn how to handle file uploads in Django using forms, and how to store and serve files using Django's built-in file handling capabilities.
20. Internationalization and Localization: Understand how to make your Django application support multiple languages and locales, allowing users to view content in their preferred language.
21. Caching: Explore Django's caching framework and learn how to cache views, template fragments, and database queries to improve performance.
22. Handling Email: Learn how to send email notifications from your Django application using Django's send_mail() function and configuring email settings in settings.py.
23. Custom Management Commands: Understand how to create custom management commands using Django's command-line utility to perform administrative tasks.
24. Testing: Explore how to write unit tests and integration tests for your Django applications using Django's built-in testing framework.
25. Django REST Framework: Get an introduction to Django REST Framework, a powerful toolkit for building Web APIs in Django, and learn how to create APIs for your Django applications.
26. Authentication and Permissions in DRF: Learn about authentication and permission classes provided by Django REST Framework to secure your APIs.
27. Serializers in DRF: Understand how to serialize and deserialize Django model instances into JSON representations using Django REST Framework serializers.
28. Views and ViewSets in DRF: Explore how to define views and viewsets for your APIs using Django REST Framework, and how to wire them up to URLs.
29. Pagination and Filtering in DRF: Learn about pagination and filtering options available in Django REST Framework to manage large datasets and improve API usability.
30. Versioning and Documentation in DRF: Understand how to version your APIs and generate interactive documentation using Django REST Framework's tools like DRF's built-in documentation or third-party packages like Swagger.

Notes from https://www.tutorialspoint.com/beautiful_soup/index.htm

1. Importing BeautifulSoup: Begin by importing the BeautifulSoup library. You can do this with the line from bs4 import BeautifulSoup. This allows you to use BeautifulSoup's functionality in your Python code.
2. Loading HTML: Use the open() function to load an HTML file into Python. For example: with open("example.html") as file:. Then, read the file contents and pass it to BeautifulSoup: soup = BeautifulSoup(file, 'html.parser').
3. Parsing HTML: Once you have loaded the HTML into BeautifulSoup, you can parse it to extract the data you need. BeautifulSoup provides methods like find() and find_all() to locate specific elements in the HTML document based on tags, attributes, or CSS classes.

4. Accessing Tags: You can access tags within the HTML using dot notation or by treating the BeautifulSoup object like a dictionary. For example: soup.title or soup['title'] to access the title tag.
5. Navigating the HTML Tree: BeautifulSoup allows you to navigate the HTML tree using methods like parent, children, descendants, and next_sibling. These methods help you traverse through the structure of the HTML document.
6. Extracting Text: To extract text from HTML elements, you can use the text attribute. For instance, soup.p.text would give you the text within the first paragraph tag.
7. Extracting Attributes: You can extract attributes from HTML tags using dictionary-like syntax. For example, soup.a['href'] would give you the value of the href attribute of the first anchor tag.
8. Searching with Regular Expressions: BeautifulSoup allows you to search for specific patterns within the HTML using regular expressions. You can pass a compiled regular expression object as an argument to the find() or find_all() methods.
9. Modifying HTML: You can modify the HTML document using BeautifulSoup. This includes adding new tags, modifying existing tags, or removing tags altogether. Changes made to the BeautifulSoup object are reflected in the original HTML.
10. Outputting HTML: Once you have manipulated the HTML document, you can output it back to a file or print it to the console using the prettify() method. This method formats the HTML document with proper indentation, making it more readable.