Web Development using Python

Recap of Python and managing dependencies

Working with List type:

- If we want to represent a group of values as a single entity, where insertion order is required to preserve and duplicates are allowed, then we should use the **list** data type.

1. insertion order is preserved
2. heterogeneous objects are allowed
3. duplicates are allowed
4. Growable in nature
5. values should be enclosed within square brackets.

Example:

```
list = [10, 10.5, ”masai”, True, 20]
print(list)   # [10, 10.5, “masai”, True, 20]


list = [10,20,30,40]
print(list[0])
print(list[-1])  # prints 40 ,negative index starts from reverse
print(list[8]) # Error list out of the range

list[0] =50

for item in list:
        print(item)
```

- The list is growable in nature. i.e. based on our requirement we can increase or decrease the size.
- A list is an ordered, mutable, heterogeneous collection of elements, where duplicates are also allowed.

Example:

```
list = [10, 20, 30]
list.append(“masai”)
print(list)

list.remove(20)
print(list)


letters = [“a”, “b”, “c”]
zero_list = [0] * 5

combined = letters + zero_list # It will combine both the lists into a single list


numbers = list(range(20))
# It will generate a list from 0 to 19, and the range() function will return an iterable


chars = list("Hello");
print(chars); # ["H", "e", "l", "l", "o"]
```

Getting a sublist from the existing list:

```
chars = list("Hello");

newList= chars[0:2]
print(newList) # ["H", "e"]

numbers = list(range(20))

print(numbers[::2])
#it will print all the even numbers from 0 to 19, ::2 means start:end:step

print(numbers[::-1])
#it will print all the elements from 0 to 19 in reverse order:
```

Unpacking values from the list:

```
numbers = [1,2,3]
```

```
first = numbers[0]
second = numbers[1]
third = numbers[2]
```

- The following code is the same as above, the number of variables must be the same as the number of items inside the list

```
first, second, third = numbers
```

- To unpack only few values from a list:

```
numbers = [1, 2, 3, 4, 4, 5, 2, 1]

first, second, *others = numbers
```

- Here only 2 values will be unpacked inside the **first** and **second** and the rest of the values will be unpacked inside the **others** list.

- To unpack the first and last item:

```
numbers = [1,2,3,4,4,5,2,1]

first, *others,  last = numbers
```

Printing list items with their indexes:

```
letters = ["a","b","c"]

for letter in letters:
    print(letter)
```

- Printing with indexes

```
for letter in enumerate(letters):
    print(letter[0], letter[1])
```

- Here the **enumerate** function will return the **tuple** with the index number in each iteration like (0,"a"), (1,"b")..

- Above example with unpacking concept:

```
for index, letter in enumerates(letters):
    print(index, letter)
```

Adding and removing items from a list:

- Adding an item to the end of the list

```
list.append(item)
```

- Adding an item to the beginning or at any index position:

```
list.insert(index, item)
```

Note: if the position is out of the range then the item will be inserted at the last.

Removing an item from the list:

```
pop() # It will delete an item from the last
pop(index) #It will delete an item from any index
remove(item) # It will remove any item from the list for the first occurence

del list[0]  # It will also delete the item from an index

del list[0:4] # It will delete a range of items

list.clear() # It will remove all items from the list
```

Count the frequency of an item inside the list

```
number_of_occ = list.count(item)
```

Sort a list items:

- To use sort() function, a compulsory list should contain only homogeneous elements. otherwise we will get TypeError

```
numbers = [3,52, 2,1, 4,7]

numbers.sort() # sort the numbers in ascending order
```

```
numbers.sort(reverse=True) # sort a list in descending order
```

Note: The above sort() method will sort an existing list, in order to get the sorted list without affecting the original list we should make use of the **sorted()** function

```
newSortedList= sorted(numbers)
newSortedList= sorted(numbers, reverse=True)

print(numbers)
print(newSortedList)
```

Sorting the complex items like list of tuples:

Example:1

```
items = [

        ("Product1", 10),
        ("Product2", 9),
        ("Product3", 12),
]


def sort_item(item):
        return item[1]
```

- From the above function we need to return the item by using which the sort function will sort the items.
- Then pass this function name to the sort function as a key

```
        items.sort(key=sort_item)
```

Example2: Using lambda expression

```
items.sort(key=lambda item:item[1])
```

Modules in python:

- As our program grows we should also split our code across multiple files, we refer to each file as a Module.
- A group of functions, variables and classes saved to a file, which is nothing but a module.
- Every Python file (.py) acts as a module.
- We should keep related(objects) classes, functions, variables, etc inside a module.

Example:

chitkaramath.py

```
x=888

def add(a,b):
    print("The Sum:",a+b)

def product(a,b):
    print("The Product:",a*b)
```

Here **chitkaramath** module contains one variable and 2 functions

- If we want to use members of a module in our program then we should import that module.

  Syntax:

  ```
  import modulename
  ```

- Now we can access members by using module names.

  ```
  modulename.variable
  modulename.function()
  ```

app.py

```
import chitkaramath
```

```
print(chitkaramath.x)

chitkaramath.add(10, 20)

chitkaramath.product(10, 20)
```

Note: whenever we are using a module in our program, for that module a compiled file will be generated and stored in the hard disk permanently.

Renaming a module at the time of import (module aliasing):

```
import chitkaramath as m
```

- Now we can access members by using alias name **m**

```
m.add(10, 20)
m.product(10, 20)
```

Importing members from a module:

- We can import particular members of the module by using **from ... import**. The main advantage of this is we can access members directly without using module names.

Example:

app.py:

```
from chitkaramath import x, add

print(x)
add(10, 20)
product(10, 20) # NameError: name “product is not defined”
```

Various possibilities of import:

- import modulename

- import module1,module2,module3
- import module1 as m
- import module1 as m1,module2 as m2,module3
- from module import member
- from module import member1,member2,memebr3
- from module import memeber1 as x
- from module import *

Note: By default module will be loaded only once even though we are importing multiple times inside a program

Example:

module1.py:

```
print(“This is from the module1”)
```

app.py:

```
import module1
import module1
```

```
import module1
import module1
print(“The is app module”)
```

- In the above program **module1** module will be loaded only once even though we are importing multiple times.

.

Finding members of module by using dir() function:

- Python provides an inbuilt function **dir()** to list out all members of the current module or a specified module.
    - dir(): To list out all members of current module
    - dir(moduleName): To list out all members of specified module

Example:

app.py

```
x =10
y = 20

def f1():
    print(“Welcome”)

print(dir())
```

Output: ['__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'f1', 'x', 'y']

Example 2: To display members of particular module:

app.py

```
import chitkaramath

print(dir(chitkaramath))
```

Output: ['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add', 'product', 'x']

Note: For every module at the time of execution Python interpreter will add some special properties automatically for inter**nal use.**

The Special variable __name__:

- For every Python program , a special variable **__name__** will be added internally.
- This variable stores information regarding whether the program is executed as an individual program or as a module.

- If the program executed as an individual program then the value of this variable is **__main__**
- If the program is executed as a module from some other program then the value of this variable is the **name of the module** where it is defined.
- Hence by using this **__name__** variable we can identify whether the program executed directly or as a module.

Example:

module1.py

```
def f1():
    if __name__ == “__main__”:
        print(“The code is executed as a program”)
    else:
        print(“The code is executed as module from some other program”)

f1()
```

app.py

```
import module1
```

```
module1.f1()
```

Now we can run both files and check the output

Module search path:

- when we try to import a module, lets say:

```
import chitkaramath
```

- it will try to look for the **chitkaramath.py** file inside the current directory, if it does not get this file inside the current directory, it will look for this file inside a bunch of predefined directories/folders which comes along with the python installation.

- Note: with the python installation we get a bunch of predefined modules which we can make use inside our application by importing them.

Example:

app.py

```
import chitkaramath
import sys

print(sys.path)
```

Output:

```
[
'C:\\Users\\ratan\\OneDrive\\Desktop\\py_text',
 'C:\\Users\\ratan\\AppData\\Local\\Programs\\Python\\Python312\\python312.zip',
 'C:\\Users\\ratan\\AppData\\Local\\Programs\\Python\\Python312\\DLLs',
 'C:\\Users\\ratan\\AppData\\Local\\Programs\\Python\\Python312\\Lib',
 'C:\\Users\\ratan\\AppData\\Local\\Programs\\Python\\Python312',
 'C:\\Users\\ratan\\AppData\\Local\\Programs\\Python\\Python312\\Lib\\site-packages'
]
```

math module:

app.py

```
import math

# Find the square root of a number
print(math.sqrt(25))  # 5.0

# Calculate the power of a number
print(math.pow(2, 3))  # 8.0

# Get the value of pi
print(math.pi)  # 3.141592653589793

# Calculate the sine of an angle (in radians)
print(math.sin(math.radians(30)))  # 0.5

# Round a number down to the nearest integer
print(math.floor(7.8))  # 7

# Round a number up to the nearest integer
print(math.ceil(7.2))  # 8

# Find the greatest common divisor (GCD) of two numbers
print(math.gcd(12, 18))  # 6

# Find the least common multiple (LCM) of two numbers (Python 3.9+)
print(math.lcm(12, 18))  # 36
```

random module:

app.py

```
import random
```

```
print(random.random())
 #here we get a floating point number between 0 and 1

print(random.randInt(1,10))
#this method will generate a random number between 1 and 10

print(random.choice([1,2,3,5,8,,15,10]))
# this method will generate a random number from the given list values

print(random.choices([1,2,3,5,8,,15,10], k=2))
# this method will generate 2 random number from the given list, if we provide k(keyword
# value) as 3 then it will generate 3 numbers inside a list
```

With this choices() method we can generate a random password:

```
print(random.choices("abcdefghi", k=4))
// it will generate a random list of 4 character each time


str = "".join(random.choices("abcdefghi", k=4))

str = ",".join(random.choices("abcdefghi", k=4))
#here comma will be joined

print(str)
```

There is a module called "string" inside this we have some attributes like:

```
string.ascii_letters  #it returns all the lowercase and uppercase of english alphabets
string.ascii_lowercase # it returns all the lowercase alphabets
string.ascii_uppercase
string.digits
```

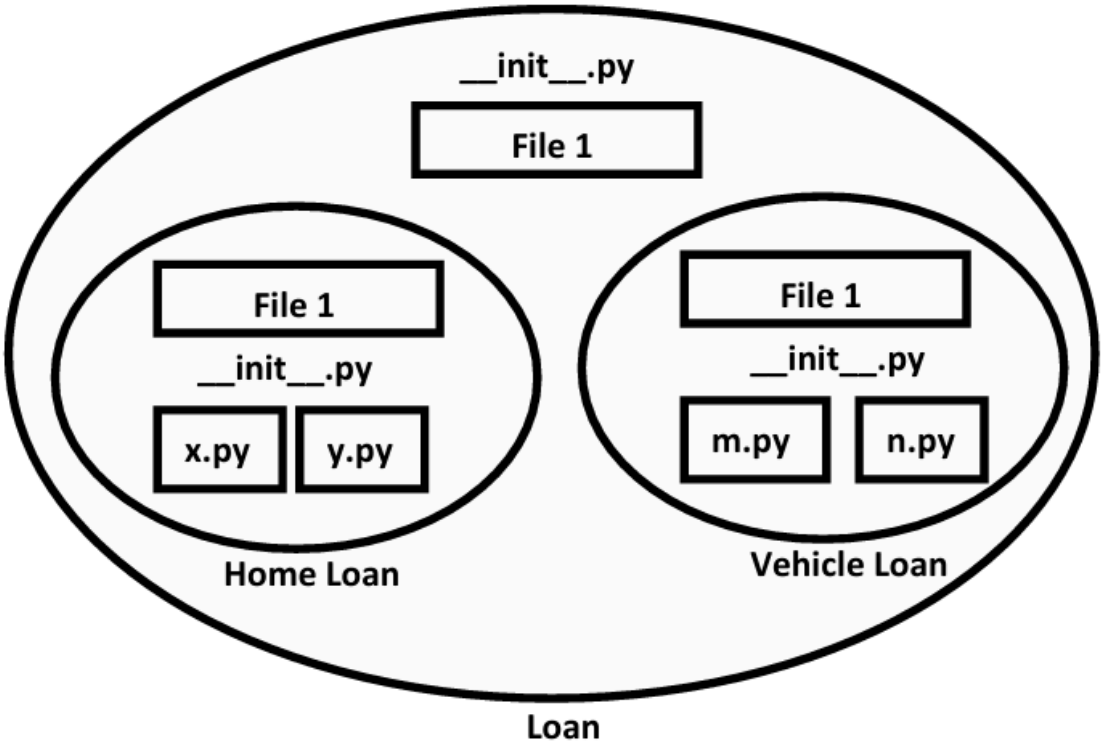
app.py:

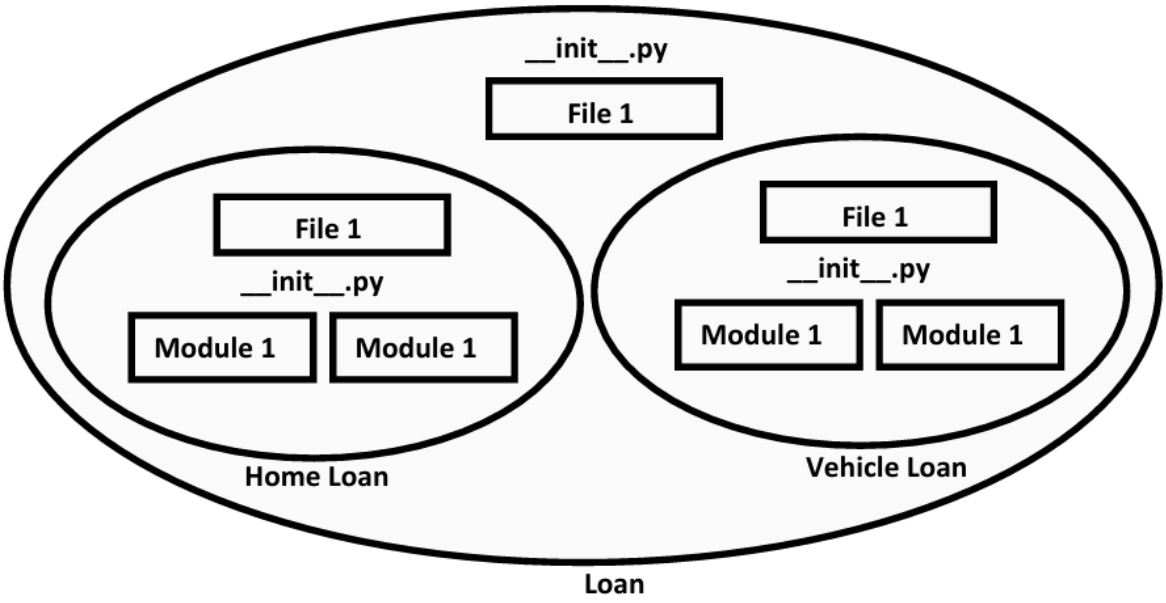
```
import string
import random

str = "".join(random.choices(string.ascii_letters + string.digits, k=4))

print(str)
```

```
# Shuffle a list
colors = ['red', 'blue', 'green']
random.shuffle(colors)
print(colors)  # output ['green', 'blue', 'red']
```

Packages:

- It is an encapsulation mechanism to group related modules into a single unit.
- package is nothing but a folder or directory which represents a collection of Python modules.
- Any folder or directory containing **__init__**.py file,is considered as a Python package.This file can be empty.
- A package can contain sub packages also.





The main advantages of packages are:

1. We can resolve naming conflicts
2. We can identify our components uniquely
3. It improves modularity of the application

Example1: app.py:

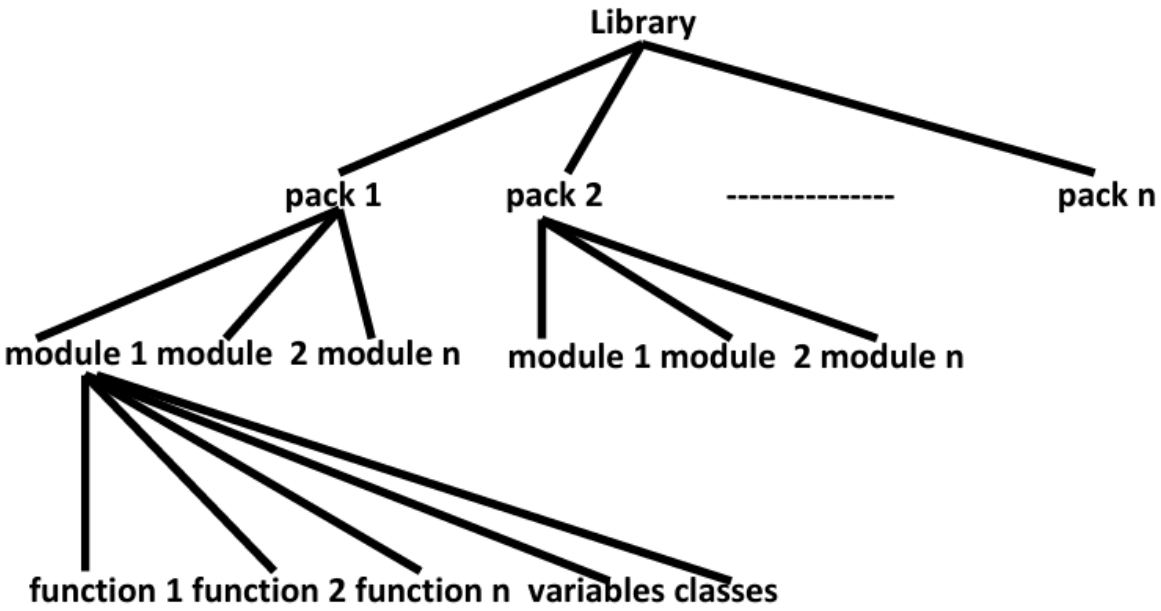
```
import pack1.module1

pack1.module1.f1()
```

Example2: app.py:

```
from pack1.module1 import f1

f1()
```

Libraries:

- It is the collection of packages



Python package index: Pypi

- With the python installation, we get some built in standard libraries which have a bunch of packages, but in these standard libraries which comes along with python installation, everything is not there which are potentially required for developing a real time large application.

- To get some extra features we need some external libraries in addition to these inbuilt standard libraries.

- Here Pypi (Python package index) comes to the picture.

- The Pypi is like npm or node package manager in nodejs environment or maven repositories in Java

- It is basically a repository for python packages built by different developers.

  https://pypi.org/

- This repository has lots of python packages for almost every type of our requirement.
- Let's say you want to work with the pdf file inside our application, just search the word pdf inside the website search box.
- Here not all the packages will work fine, some of them are buggy, we can search google like best packages to work with pdf.

The pip tool:

- To install a package from the Pypi repository we need a tool called **pip**:
- This **pip** tool comes along with the python installation, but it is developed independently so sometimes we need to update it as well.
- We may see a yellow warning message to the console updating the pip, while installing any package with the command to upgrade the pip.
- To check the version of **pip** tool, type the following command inside the terminal

```
pip --version
// or inside the mac os
pip3 --version
```

- To upgrade the pip version

```
pip install --upgrade pip
```

- Let's try to install the "**requests**" package which is used to send the http request from the python application.

Syntax:

```
pip install <package-name>
```

Example:

```
pip install requests
```

- To list out all the packages installed inside our system:

  pip list

Output: it will show all the packages installed inside our system default environment with version

```
Package           Version
----------------- ----------
certifi           2024.12.14
charset-normalizer 3.4.1
distlib           0.3.9
filelock          3.16.1
idna              3.10
packaging         24.2
pip               24.3.1
pipenv            2024.4.0
platformdirs      4.3.6
requests          2.32.3
setuptools        75.8.0
urllib3           2.3.0
virtualenv        20.28.1
```

- Here for the **"requests"** package the version **2.32.3** is also called semantic versioning, Which has 3 parts.

  1. major version
  2. minor version
  3. patches or bug fixes

- We can go to the particular package related website inside the Pypi website and visit the release history

- Some version may become incompatible with some other applications

- installing the earlier version or exact version of any package:

  pip install requests==2.29.0

- It will uninstall the already installed version and install the specified version

- We can also make uses of wildcards for example

pip install requests==2.29.*
#latest compatible version for 2.9 higher patch version

pip install requests==2.*
#It will install the higher minor version

- To uninstall a package:

  pip uninstall requests

- To get the location and other info for a installed package

  pip show <package-name>

  Example

  pip show requests

  Output:

  Name: requests
  Version:2.29.0
  Summary: Python HTTP for Humans.
  Home-page: http://python-requests.org
  Author: Kenneth Reitz
  Author-email: me@kennethreitz.com
  License: Apache 2.0
  Location: C:\Users\ratan\AppData\Local\Programs\Python\Python312\Lib\site-packages
  Requires:
  Required-by:

Note: Inside our default python environment we can not have multiple version installed for a particular package

- After installing the packages we can import them as a module inside the our application,
- We can read their documentation for more info

Example:

app.py:

```
import requests

response = requests.get("https://google.com")

print(response)
```

Virtual Environment in Python:

- Just now we have installed a package called "**requests**" whose version is: **2.32.3**
- But let's say we have another project here we want to make use of an earlier version of this package

- Inside the **default Python environment** we can not have multiple version of any of the packages, we can have only a single version of a package

- To solve this problem we need to create an isolated virtual environment for each project and install these dependencies into the isolated virtual environments.

- A virtual environment in Python is a self-contained directory that contains a Python installation and a set of libraries. It allows developers to manage dependencies for different projects independently, avoiding conflicts between them.

- Inside a computer, we can create multiple Python virtual environments, each configured with different versions of specific dependencies.

Why Use a Virtual Environment?

1. **Isolation**: Ensures that dependencies for one project do not interfere with others.
2. **Reproducibility**: Makes it easier to share and maintain consistent environments across different systems.
3. **Version Management:** Allows using different versions of libraries or Python for different projects.

Steps to create a virtual Environment:

Step1: Create a Virtual Environment

- Use the `venv` module to create a virtual environment:

  For window:
  python -m venv myenv
  For Mac:
  python3 -m venv myenv

- Here, `myenv` is the name of the virtual environment directory/folder.
- The above command will create a folder called `myenv` inside the current directory.
- inside this `myenv` folder, there will be some folders and an important configuration file called **pyvenv.cfg** created.

  pyenv.cfg:
  home: is the location of our python interpreter for this virtual environment
  version: the version of the python interpreter for this virtual environment

  Lib/site-package folder: Here the 3rd party packages will be installed for this virtual environment.

Step 2: Activate the Virtual Environment

- After creating this virtual environment, we need to **activate** this virtual environment

- For this, inside the **Scripts** folder there are some commands are available:
    - On Windows:
        myenv\Scripts\activate – for powershell terminal
        source myenv\Scripts\activate – for bash terminal
    - On macOS/Linux:
        source myenv/bin/activate

- After activation, the command prompt will show the environment name, e.g., `(myenv)`.

Step 3: Install the required version of Packages

- Once the virtual environment is active, install packages using `pip`:

    pip install requests==2.29.0

Step 4: Deactivate the Virtual Environment

- To exit the virtual environment, run:

    deactivate

Example: Using a Virtual Environment

Scenario:

You have two projects:

- Project A requires **requests.2.29.0**
- .
- Project B requires **requests.2.32.2**

Steps:

1. Create 2 Virtual Environments for both the projects:

    python -m venv project_a_env

    python -m venv project_b_env

2. Activate and Install Specific Versions:

    For Project A:

    project_a_env\Scripts\activate #To Activate

    pip install requests==2.29.0 # Install requests 2.29.0

    pip list # list out all the packages inside this env

    deactivate #To Deactivate

    For Project B:

    project_b_env\Scripts\activate #To Activate

    pip install requests # Install requests latest version

    pip list # list out all the packages inside this env

    deactivate #To Deactivate

Inside the powershell: disable the execution policy to activate the virtual environment:

Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned

- For more info about the virtual environment for python visit the official documentation.

Freezing Dependencies:

- To share your project's dependencies, freeze them into a requirements file:

  pip freeze > requirements.txt

- This creates a `requirements.txt` file with a list of installed packages and their versions.

Installing from `requirements.txt`

- To recreate the environment on another system:
  1. Create and activate a new virtual environment.
  2. Install dependencies from the file:

     pip install -r requirements.txt

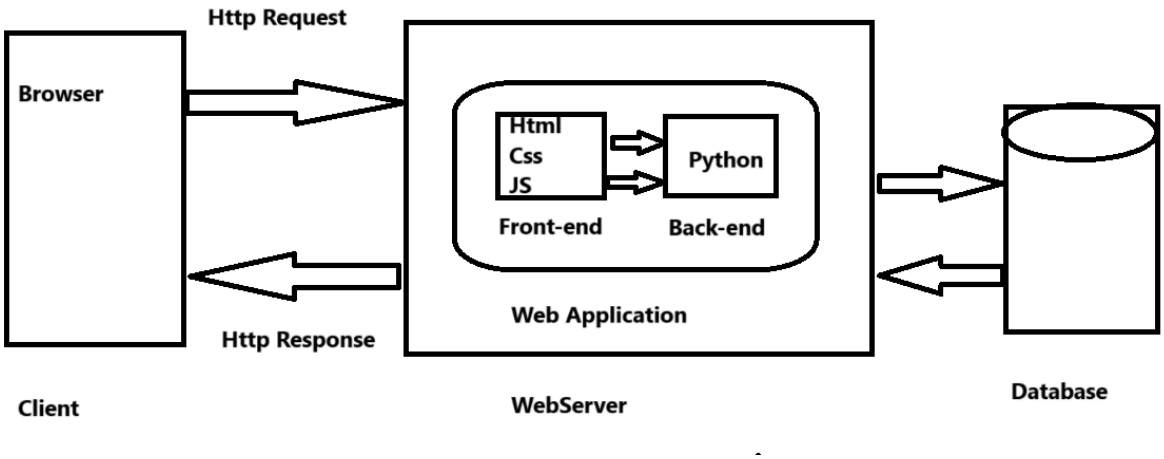
Important notes about the Virtual Environments:

- Use the .gitignore file to exclude the myenv folder. to upload the project inside the GitHub.
- Always activate the virtual environment before running your Python scripts or installing packages.

Introduction to Web Application

- A Web Application is a software program that runs on a web server and is accessed by users through a web browser using the internet. Unlike traditional desktop applications, web applications do not need to be installed on a user's device, making them accessible from anywhere with an internet connection.

Web Application Architecture:



1. **Client/Browser Communication:**
    - The client/browser communicates with the server using the HTTP protocol. This communication involves sending HTTP requests (e.g., GET, POST) and receiving HTTP responses from the server.
2. **Server-Side Web Application:**
    - Inside the server, the web application resides. This application can be divided into two parts:
        - Front-End Application:
            - Handles the presentation layer, responsible for delivering static and dynamic content (e.g., HTML, CSS, JavaScript).
        - Back-End Application:
            - Handles the logic and business operations of the application.
            - Processes incoming requests, communicates with external APIs, and performs necessary operations.
            - systems Frequently interacts with the Database (DB) to retrieve, store, or modify data.
3. **Back-End to Database Communication:**
    - The back-end application communicates with the database (DB) using SQL queries.
    - This interaction ensures the persistence and retrieval of data requested by the client.

Flow Overview:

1. **Client/Browser** sends an HTTP request to the **server**.
2. **Front-End Application** (on the server):
    - Processes the request (if static content is involved).
    - Passes the request to the back-end for dynamic or complex operations.
3. **Back-End Application**:
    - Executes business logic.
    - Communicates with the **database** (if necessary) to fetch or manipulate data.
4. **Response**:
    - The server sends an HTTP response back to the **client/browser**, which then renders the page or data for the user.

A **Web Application** is typically categorized into two main parts:

1. **Client-Side Application**
2. **Server-Side Application**

While both parts reside on the **web server**, their execution environments differ:

- The **Client-Side Application** is **executed on the client machine** (e.g., in a web browser).
- The **Server-Side Application** is **executed on the server machine**.

1. Client-Side Application

- The client-side application focuses on the **user interface (UI)** and presentation layer of the web application.
- It includes the visual components and interactions that users see and interact with directly.

  Execution:

  - Executed in the user's browser or device.

  Technologies:

  - **Languages**:

    - HTML (HyperText Markup Language)
    - CSS (Cascading Style Sheets)
    - JavaScript (for interactivity)
- **Frameworks/Libraries**:
    - React.js, Angular, Vue.js, Svelte, Bootstrap.

Examples:

- Interactive forms, animations, dynamic content updates, or Single Page Applications (SPAs).

2. Server-Side Application

- The server-side application handles the **business logic, data processing**, and communication with databases or external services.
- It is responsible for generating dynamic responses and managing backend operations.

Execution:

- Executed on the server.

Technologies:

- **Languages**:
    - Python (e.g., Flask, Django)
    - Java (e.g., Spring, Servlets)
    - PHP (e.g., Laravel, Symfony)
    - JavaScript (e.g., Node.js)
    - Ruby (e.g., Ruby on Rails)
    - .NET (e.g., ASP.NET)
- **Databases**:
    - Relational: MySQL, PostgreSQL, SQL Server.
    - NoSQL: MongoDB, Redis.

Examples:

- Handling user authentication, processing form data, querying databases, and generating API responses.

Http Protocol:

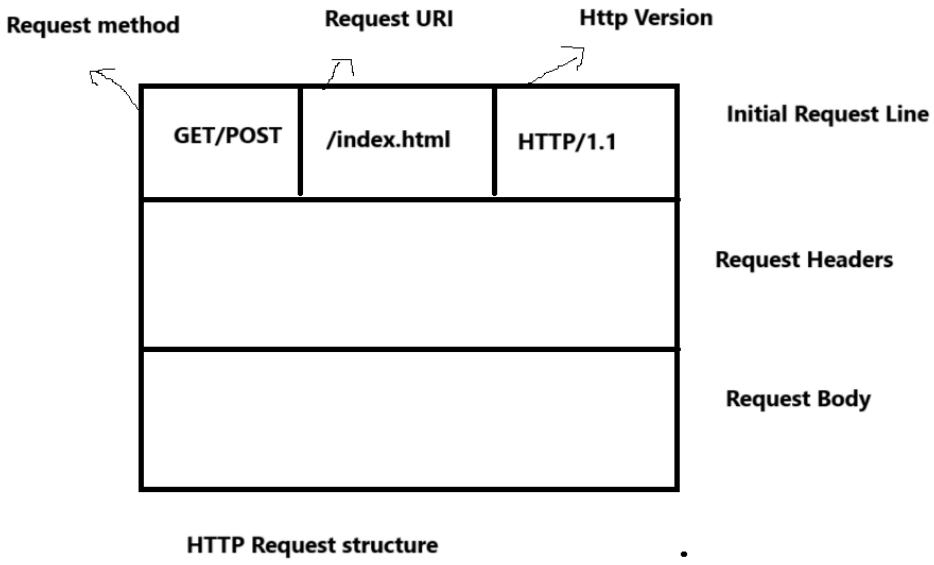
What is HTTP?

- HTTP (**HyperText Transfer Protocol**) is a set of rules and guidelines that enable communication between an **HTTP client** (e.g. web browser) and an **HTTP server** (e.g. web server software).
- It defines how requests and responses are formatted and transmitted over the internet.
- It is a stateless protocol, meaning each request-response cycle is independent of previous ones.
- HTTP operates over TCP/IP and is used to transfer data such as HTML documents, images, videos, and other web resources.

How HTTP Works

1. **HTTP Request:**
    - Generated by the client to initiate communication with the server.
    - Contains details such as the requested resource, headers, and optional data.
2. **HTTP Response:**
    - Generated by the server in response to a client's request.
    - Contains status codes, headers, and optionally, the requested content or error information.

Http Request Structure:



HTTP Request structure

An HTTP request contains the following components:

Request Line:

Specifies the HTTP method, the URL, and the protocol version.

```
GET /index.html HTTP/1.1
```

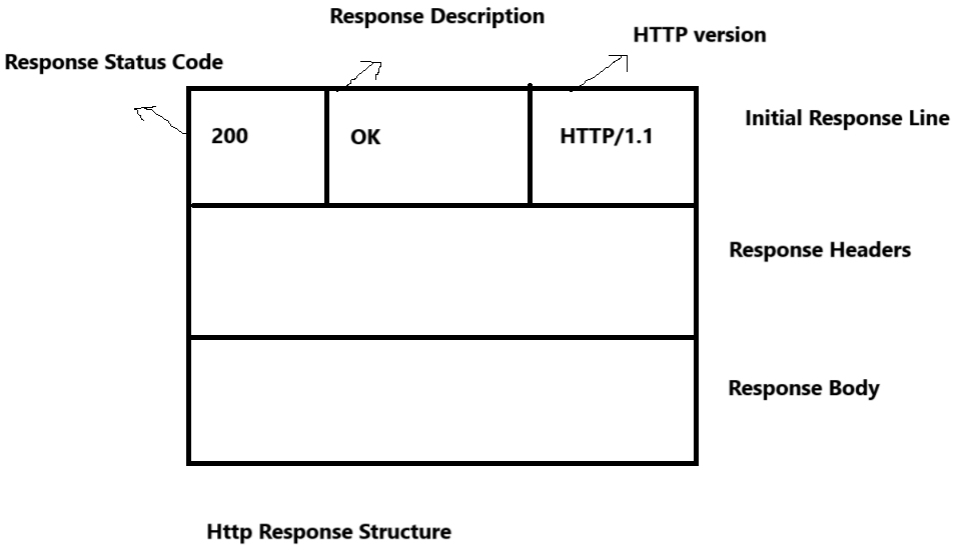
Request Headers:

- Headers are key-value pairs that provide additional information about the request.
- Common headers:
    1. **Date**: The date and time when the request is sent.
    2. **Host**: The IP address or domain of the server.
    3. **User-Agent**: Information about the client (browser/software) making the request.
    4. **Content-Type**: Indicates the format of the request body (e.g., JSON, XML, form-data).
    5. **Connection:** Indicates whether the connection should remain open or closed.
    6. **Cookie:** Includes stored cookies sent by the client for session management.

Request Body

- Contains optional data (e.g., form data, JSON) sent from the client to the server, typically used in POST and PUT requests.

Http Response Structure:



Http Response Structure

An HTTP response contains the following components:

Initial Response Line

- Includes the protocol version, status code, and a status message.

  `HTTP/1.1 200 OK`

Status Codes

- Status codes indicate the result of the HTTP request:
  1. **100-199:** Informational
  2. **200-299:** Success (e.g., `200 OK`, `201 Created`)
  3. **300-399:** Redirection (e.g., `301 Moved Permanently`, `302 Found`)
  4. **400-499:** Client errors (e.g., `400 Bad Request`, `404 Not Found`)

5. **500-599:** Server errors (e.g., `500 Internal Server Error`, `503 Service Unavailable`)

Response Headers

- Headers provide additional metadata about the response.
- Common headers:
    1. **Server**: Details about the web server software.
    2. **Set-Cookie**: Used for setting cookies in the client's browser.
    3. **Last-Modified**: Indicates the last modification date of the resource.
    4. **Refresh:** Specifies the time to refresh the page automatically.
    5. **Content-Type:** Describes the media type of the response body (e.g., `text/html`, `application/json`).

Response Body

- Contains the content requested by the client or error messages if the request fails.

What is a Web Framework?

Web Application Framework or simply Web Framework represents a collection of libraries and modules that enables a web application developer to write applications without having to bother about low-level details such as protocols, thread management etc.

A web framework is a software library or set of tools that helps developers build and manage web applications. It provides standard ways to:

- Handle HTTP requests and responses.
- Manage URL routing.
- Work with databases.
- Render templates to generate dynamic HTML pages.
- Perform common web development tasks efficiently.

Introduction to Flask:

- Flask is a popular web framework for Python that allows you to build web applications quickly and easily. It is lightweight and minimalistic, which makes it perfect for beginners who are new to web development. Here is a beginner-friendly guide to understanding how Flask works and how you can use it to create a simple web application.
- It is classified as a **microframework**, meaning it provides the basic tools for web development without enforcing specific patterns or including unnecessary features.
- Flask is ideal for beginners due to its simplicity and extensive documentation.
- Flask is built upon two powerful components:
    1. **Werkzeug**: A WSGI (Web Server Gateway Interface) toolkit that ensures compatibility with web servers and simplifies request and response handling.
    2. **Jinja2**: A modern and designer-friendly template engine that allows developers to create dynamic HTML pages.

Key Features of Flask

1. **Routing**: In Flask, routing means creating URL patterns and associating them with Python functions. When a user visits a particular URL, the corresponding function is called to handle the request.
2. **Request and Response**: Flask handles HTTP requests (like GET, POST) and sends back HTTP responses. This is how web browsers communicate with the server.
3. **Templates**: Flask uses Jinja2 templating engine, which allows you to generate dynamic HTML pages by embedding Python-like expressions inside HTML files.
4. **Flask App**: The Flask app is the central component of a Flask application. It is the instance that routes requests, handles templates, and manages the application flow.
5. **Lightweight and Minimal:** Only provides essential components, giving developers the freedom to add additional tools as needed.
6. **Built-in Development Server:** Flask includes a debugger and reloader, making development and testing easier.
7. **Extensible:** Supports extensions for additional features like database integration, authentication, etc.
8. **RESTful Request Handling:** Easily handles HTTP methods (GET, POST, PUT, DELETE) for building APIs.

Creating a Flask Application:

- When you develop a Flask application, it's best practice to use a virtual environment. A virtual environment is an isolated environment where you can install dependencies (like Flask) without affecting other Python projects or your system's global Python setup.

Steps to Create a Flask Application with a Virtual Environment

1. **Install Python** (if not already installed):

   Flask requires **Python 3.5 or later**. Check if Python is installed by running:

   ```
   python --version
   ```

   - If Python is not installed, download and install the latest version from the official Python website: https://www.python.org/downloads/.

2. **Create a Project Directory**:

   First, create a directory (folder) where your Flask application will reside.

   ```
   mkdir my_flask_app
   cd my_flask_app
   ```

3. **Create a Virtual Environment**:

   Use the `venv` module to create a virtual environment. Run the following command in your project directory:

   ```
   python -m venv venv
   ```

   - This will create a directory named `venv` where the virtual environment will reside.

4. **Activate the Virtual Environment**:
   - After creating the virtual environment, you need to activate it.

     On **Windows**, use:

     ```
     venv\Scripts\activate
     ```

On **macOS/Linux**, use:

```
source venv/bin/activate
```

- After activation, your command prompt should change to show the virtual environment is active (e.g., `(venv)` at the beginning of the prompt).

5. **Install Flask**:

   With the virtual environment activated, install Flask using `pip`:

   ```
   pip install Flask
   ```

6. **Create Your Flask Application**:

   Now that Flask is installed, you can start writing your Flask app. Create a new file called `app.py` in the project directory and add the following code:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, Flask!"

# Ensures the app runs only if this file is run directly
# Start the Flask development server with debugging enabled
if __name__ == '__main__':
    app.run(debug=True)
```

7. **Run the Application**:

   To run your Flask app, make sure you're still in the project directory with the virtual environment activated and use the following command:

   ```
   python app.py
   ```

Your Flask app should now be running, and you can visit it in your browser at http://127.0.0.1:5000/.

Note: If we follow the official documentation and omit the following lines from the application:

```
if __name__ == '__main__':
    app.run(debug=True)
```

Then to run the application we need to make use of following command:(Flask CLI)

```
flask run
```

If your application's main file is named something other than `app.py`, you must specify the file name using the `--app` flag. For example:

```
flask --app <app-name> run
```

Example:

```
flask --app app run
```

8. **Deactivate the Virtual Environment**:

   When you're done working on your project, deactivate the virtual environment by running:

   ```
   deactivate
   ```

Routing and URL Handling in Flask

What is Routing?

- Routing is the process of mapping URLs to specific functions in your Flask app.
- Flask uses the @app.route() decorator to define routes.

Basic Routes:

- We just saw how to create a web application with a single page (returned in the form of a string)
- Let's see how we can add multiple routes (multiple pages)
- The key to this is in the **@app.route()** decorator.
- The string parameter passed into the decorator determines the URL extension that will link to the function(**the View function**) to represent the natural routing.
- Currently our homepage or domain is locally represented as http://127.0.0.1:5000/ or http://localhost:5000/
- We use the **@app.route()** decorator to add the URL extensions
    - @app.route("/somepage")
    - http://127.0.0.1:5000/somepage

Example:

app.py:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')  # 127.0.0.1:5000/
def home():
    return "Hello, Flask!"

@app.route("/info")  # 127.0.0.1:5000/info
def info():
    return "<h1>This is the Info page</h1>"

if __name__ == '__main__':
    app.run(debug=True)
```

Here :

```
127.0.0.1:5000/ : Hello Flask!

127.0.0.1:5000/info : info page

127.0.0.1:5000/information : Not Found 404
```

Flask dynamic routing:

- Often we will want URL route extensions to be dynamic based on the situation.
- For Example, we may want a page per user, so that the extension should be in the form:
    - www.site.com/user/unique_user_name
- To achieve this effect we can use dynamic routes.
- Dynamic routing allows for URLs with variables(**Path variables**).
- Dynamic routes have 2 key aspects:
    1. A variable in the route **<variable>**
    2. A parameter passed into the view function

Example:

```
@app.route("/user/<user_name>")
def users(user_name):
    return f"<h1>Welcome, this is a page for {user_name}</h1>"
```

Now we can access each user info page separately:

```
127.0.0.1:5000/Rahul : Welcome, this is a page for Rahul

127.0.0.1:5000/Raj : Welcome, this is a page for Raj
```

Flask Application Debug mode:

- As we code our application, we'll definitely make some mistakes along the way!

- We can set **debug=True** in our application to help us catch errors.
- Debug mode helps identify and fix errors during development. Enabling debug mode provides detailed error messages in the browser.
- Debug mode also gives us access to a console in the browser.

Let's say the following view function:

```
@app.route("/user/<user_name>")
def users(user_name):
    return f"<h1>10the letter of the name: {user_name[10]}</h1>"
```

127.0.0.1:5000/Rahul

- Accessing http://127.0.0.1:5000/user/Rahul would result in an **IndexError** (since "Rahul" does not have 10 letters). Without debug mode, the browser will display an "**Internal Server Error (500).**"
- To enable debug mode, ensure debug=True is set:

```
if __name__ == '__main__':
    app.run(debug=True)
```

- With debug mode enabled, you'll see detailed error tracebacks in the browser, making it easier to debug issues. **However, never enable debug mode in production.**

Flask Routing Exercise:

Problem Statement:

Create a Flask application where a username (email) is taken as a path variable. The application should display different messages based on the domain of the email address.

1. **If the email ends with @gmail.com**:
    - Display: **"Welcome Gmail User: [Name]"**
2. **If the email ends with any other domain**:

- Display: **"Hi Another User: [Name]"**

3. **If no domain is provided (just a name)**:
    - Display: **"Anonymous User: [Name]"**

Solution:

```
from flask import Flask

app = Flask(__name__)

@app.route("/<email>")
def greet_user(email):
    # Check if the email contains '@'
    if "@" in email:
        # list unpacking
        username, domain = email.split("@")
        if domain == "gmail.com":
            return f"Welcome Gmail User: {username.capitalize()}"
        else:
            return f"Hi Another User: {username.capitalize()}"
    else:
        # If no domain is provided
        return f"Anonymous User: {email.capitalize()}"


if __name__ == '__main__':
    app.run(debug=True)
```