Python basic tools:

1. Some Boolean conditional statements:

- If true in the condition, then the statement doesn't matter, and output prints that block. Then clarify the other (only else) statement.
- If false in the condition, then the statement doesn't matter and output doesn't print that block. Then clarify the other (only else) statement.

Example 1:

```
if a > b:
    print('A is bigger')
    print(1)
elif b > a:
    print('B is bigger')
else:
    print('They are equal')

Input : a=10, b=20
Output: B is bigger
```

Example 2:

```
if True:
   print('A is bigger')
elif b > a:
   prínt('B is bigger')
if a == b:
   print('They are equal')


Input : a=10, b=20
Output: A is bigger, B is bigger

Input : a=100, b=20
Output: A is bigger
```

Example 3:

```
if False:
   print('A is bigger')
elif b > a:
   print('B is bigger')
else:
   print('They are equal')
```

```
Input : a=100, b=20
Output: They are equal

Input : a=100, b=200
Output: B is bigger
```

2. Loop:

Example 1:

```
val = 1
while val <= 10:
    print(val)
    val =7 val + 1

Output: 1,2,3,4,5,6,7,8,9,10
```

Example 2:

```
1st = [1,2,3,4,5]
for c in 1st:
   print(c)

Or

for c in range(1,6):
   print(c)

Output: 1,2,3,4,5
```

Example 3:

```
i = 1
while i < 6:
print(i)
if i == 3:
break
i += 1

Output: 1,2,3
```

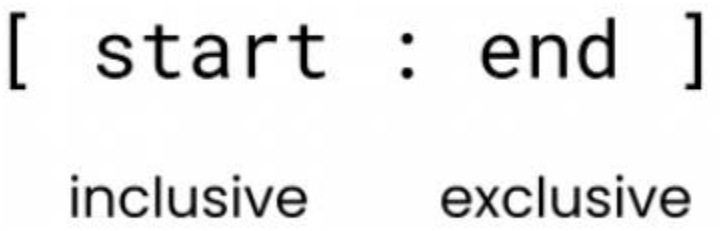
Example 4:

```
i = 0
while i < 6:
i += 1
if i == 3:
continue
print(i)

Output: 1,2,4,5,6
```

Python List Slicing:

- List slicing means selecting multiple elements from our list.
- Iterator: Iterator in python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets.
- List[Initial index : End index : Iterator / IndexJump]
- -1 represents the End index.
- `x = ["a", "b", "c", "d"]. Here, x[1] = x[-3]  #same result!`




```
#Iterate list in specific index
List = [10,20,30,40,50]
print(List[1:4])
```

```
Output: [20,30,40]


# In a list, if we don’t give any value in the initial and end index,
by default it iterates the first value to the last value.
List = [10,20,30,40,50]
print(List[::])

Output: [10,20,30,40,50]


#Fixed the iteration value
List = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(List[::2])

Output: [1, 3, 5, 7, 9]


#Reversing the list
List = [10,20,30,40,50]
print(List[::-1])

Output: [50,40,30,20,10]


#Some technical slicing
List = [10,20,30,40,50]
print(List[10::2])
print(List[1:1:1])
print(List[-1:-1:-1])
print(List[:0:])

Output: []


# Creating new List
List = [1, 2, 3, 4, 5, 6, 7, 8, 9]
List1 = List[:3]+List[7:]
print(List1)
List2 = List[::2]+List[1::2]
print(List2)

Output: [1, 2, 3, 8, 9]
[1, 3, 5, 7, 9, 2, 4, 6, 8]
```

Data Types in Python:

- List is a collection which is ordered and changeable. Allows duplicate members.
- Tuple is a collection which is ordered and unchangeable. Allows duplicate members.
- Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- Dictionary is a collection which is ordered** and changeable. No duplicate members.

List:

- Lists are used to store multiple items in a single variable. In Java, we called it Array, and in Python, we called it List. List items are ordered, changeable, and allow duplicate values. So, If we add new items to a list, the new items will be placed at the end of the list. And we can change, add, and remove items in a list after it has been created. Since lists are indexed, lists can have items with the same value. List items are indexed, the first item has index [0], the second item has index [1] etc. List completely like an array.

- Add: To add an item, use the append() method. Items will be added in the last index.

```
Example: list = ["Rahim", "Karim", "Selim"]
thislist.append("Mohim")
print(thislist)
Output: Rahim, Karim, Selim, Mohim
```

- Insert: To insert a list item at a specified index, use the insert() method.

```
Example:list = ["Rahim", "Karim", "Selim"]
list.insert(1, "Mohim")
print(thislist)
Output: Rahim, Mohim, Karim, Selim
```

- Extend List: To append elements from another list to the current list, use the extend() method.

```
Example: thislist = ["apple", "banana", "cherry"]
thislist2 = ["mango", "pineapple", "papaya"]
thislist.extend(thislist2) = ["mango", "pineapple", "papaya"]
print (thislist)
Output:['apple', 'banana', 'cherry', 'mango', 'pineapple']
```

- Remove:

```
Example: thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
Output:['apple', 'cherry']
```

- Remove Specific Index: The pop() method or del key removes the specified index.

```
Example:thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
Output: ["apple", "cherry"]

Or,
#Deleting specific index:
thislist = ["apple", "banana", "cherry"]
del thislist[1]
print(thislist)
Output: ["apple", "cherry"]

#Delete the entire list(also delete the list completely):
thislist = ["apple", "banana", "cherry"]
del thislist

#The clear() method empties the list. The list still remains, but
it has no content
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
Output: []
```

- Copy a List: Make a copy of a list with the copy() method.

```
Example: thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
Output: ['apple', 'banana', 'cherry']
```

- Join Two Lists:

```
Example: list1 = [“a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
Output:[‘a’, ‘b’, ‘c’, 1, 2, 3]

Or,

Example: list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
Output:[‘a’, ‘b’, ‘c’, 1, 2, 3]
```

So,

append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Tuple:

- Tuple items are ordered, unchangeable, and allow duplicate values. It means that we cannot change, add or remove items after the tuple has been created.

```
thistuple=("apple","banana","cherry", "apple","cherry")
print(thistuple)
#iterating tuple items (2)
```

```
print(thistuple[1])
#negatively iterating tuple items (-4 included,-1 excluded) (3)
print(thistuple[-4:-1])
#Convert tuple into a list (4)
y = list(thistuple)
#Add or remove items in tuple (First convert the tuple into a
list, add "orange", and convert it back into a tuple) (5)
lst = list(thistuple)
lst.append("orange")
thistuple = tuple(lst)
#Add two tuple (6)
z = ("mango",)
thistuple += z
print(thistuple)
#iterating tuple by loop (7)
thistuple1 = ("apple", "banana", "cherry")
for x in thistuple1:
  print(x)
#or
thistuple1 = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple1):
  print(thistuple1[i])
  i = i + 1
```

```
Output:1.('apple','banana','cherry','apple','cherry')
2. banana
3.(banana, cherry, apple)
4.['apple','banana','cherry','apple','cherry']
5.('apple','banana','cherry','apple','cherry','orange')
6.('apple','banana','cherry','apple','cherry','orange','mango')
7.('apple','banana','cherry')
```

Dictionary:

- Dictionaries store values in key:value pairs. It is ordered, changeable, and do not allow duplicates.

```
thisdict = {
```

```
  "Bangladesh": "Dhaka",
  "India": "Delhi",
  "Year" : 1971
 }
print(thisdict)
print(thisdict["India"])
print(len(thisdict))
print(type(thisdict["year"]))
#printing the keys or values of dictionary (5)
x = thisdict.keys()
print(x)
x = thisdict.values()
print(x)
#update or change the value (7)
thisdict["year"] = 2021
print(thisdict)
#add the keys and values in dictionary (8)
thisdict["color"] = "red"
print(thisdict)
#remove the keys and values of dictionary (9)
thisdict.pop("India") or del thisdict["India"]
print(thisdict)
#clear or remove all key and values of dictionary (10)
thisdict.clear()
print(thisdict)
#copy value of dictionary (11)
mydict = thisdict.copy()
print(mydict)

#Print all key names by loop (12)
for x in thisdict:
  print(x)
#Or,
for x in thisdict.keys():
  print(x)

#Print all values (13)
for x in thisdict:
  print(thisdict[x])
#Or,
for x in thisdict.values():
```

```
  print(x)

#Print all keys,values and others (14)
for x,y in thisdict.items():
  print(x,y)
#Dictionary over dictionary (Nested Dictionary) (15)
d = {
 1:{
   'a':1,
   'b':2
  }
}
print(d)
```

```
Output:1.{‘Bangladesh’: ‘Dhaka’, ‘India’: ‘Delhi’, ‘year’: 1971}
2.Delhi
3.3
4.Int
5.dict_keys([‘Bangladesh’, ‘India’, ‘year’])
6.dict_values([‘Dhaka’, ‘Delhi’, 1971])
7.{‘Bangladesh’: ‘Dhaka’, ‘India’: ‘Delhi’, ‘year’: 2021}
8.{‘Bangladesh’: ‘Dhaka’, ‘India’: ‘Delhi’, ‘year’: 2021, ‘color’:
‘red’}
9.{‘Bangladesh’: ‘Dhaka’, ‘year’: 2021, ‘color’: ‘red’}
10.{}
11.{‘Bangladesh’: ‘Dhaka’, ‘year’: 2021, ‘color’: ‘red’}
12.Bangladesh, year, color
13.Dhaka, 2021, red
14.Bangladesh Dhaka
   year 2021
   color red
15.{1: {'a': 1, 'b': 2}}
```

Function:

- A function is a block of code which only runs when it is called. In Python, a function is defined using the def keyword.

```
#Calling a Function (1)
def my_name():
```

```
  print("Aman")
my_name()
#passing a value as a parameter (2)
def my_name(k):
  print("My name is "  +k )
my_name("Aman")
#Default Parameter Value (If value is not passed in the
parameter, then by default value will be the main value) (3)
def my_location(state = "Sylhet"):
  print("I am from " + state)
my_location("Dhaka")
my_location()
my_location("Brahmanbaria")
#return statement (4)
def sum(x):
  return 5 + x
print(sum(3))
print(sum(5))
```

```
Output:1. Aman
2. My name is Aman
3. I am from Dhaka
   I am from Sylhet
   I am from Brahmanbaria
4. 8
   10
```

Python Classes and Object:

- Every object has a unique/different address to store. One object isn't dependent on another object because ram location, memory location and their work is totally different. They are like two different buildings at a different place. But every object is dependent on the class. Because class is a building sample of construction. In every building, there are some specific things which are the same from building samples or class. It is also called that object is the subsection of class.
- Class value is the same for all objects. If we call a class value from any object, it works equally for all objects.
- All classes have a by-default function called __init__(), which is always executed when the class is being initiated. The __init__() function is called automatically every time, so it is better to set or pass some parameter in this function.

- In Python, method doesn't support overloading but supports overwriting.
- Every object has an address and any change in object means change in address.

```
#Creating Object (1)
class MyClass:
  x = 5
p1 = MyClass()
print(p1.x)


#default constructor in every python (initially auto setup) (2)
#here, like self is basically a by default object
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

emp1 = Person("Khan", 34)
emp2 = Person("Aman", 22)

print(emp1.name, emp2.name)
print(emp1.age, emp2.age)


#creating object methods (3)
class Person:
  def __init__(self, name, age):
    self.name = name
     self.age = age

  def myfunc(person1):
    print("Hello my name is " + person1.name)
    print("My age is " + person1.age)

p1 = Person("Aman", "22")
p1.myfunc()
p1.myfunc()


#
```

```
Output: 1. 5
2. Khan Aman
   34 22
3. Hello my name is Aman
   My age is 22
4.
5.
```

Python Inheritance:

- Parent class is the father class, which is called the base class. Child class is the inheritance class that inherits from the parent class and holds the parent class as a parameter.

```
#vehicle parent class, car child class, move method, car1 object
class Vehicle:
    def _init_(self, engine, number_of_wheels,numberofseats):
        self.engine = engine
        self.number_of_wheels = number_of_wheels
        self.numberofseats = numberofseats
    def move(self):
        print('Vehicle is moving')
class Car(Vehicle):
    def _init_(self, engine, number_of_wheels, numberofseats,
milePerHour):
        Vehicle._init_(self, engine, number_of_wheels, numberofseats)
        self.milePerHour = milePerHour
    def move(self):
        print('Car is moving')
```

```
car1 = Car('dsfsdf','A','5',20)
print(car1.move())
```

Server:

A server is a computer or system that provides resources, data, services, or programs to other computers, known as clients, over a network. If a computer shares resources with client machines they are considered servers. Server can be a physical machine, a virtual machine or a software. A physical server is simply a computer that is used to run server software. A virtual server is a virtual representation of a physical server. Like a physical server, a virtual server includes its own operating system and applications and virtual servers that might be running on the physical server.
Every server has an IP and Port where connection is bound or connected.
We know, a server requires two software components: an operating system and an application. The operating system acts as a platform for running the server application. A server operating system, such as Windows Server or Linux, acts as the platform that enables applications to run.

Request and response time depends on efficient API logic (Time complexity), means logical portion and load balancing depending on the Server, means managing capability.

Types of Servers:

- Web servers: A computer program that serves requested HTML pages or files. Here, a web browser acts as the client.
- File servers: File servers are useful for sharing files across a network. With a file server, the client passes requests for file records over the network to the file server.

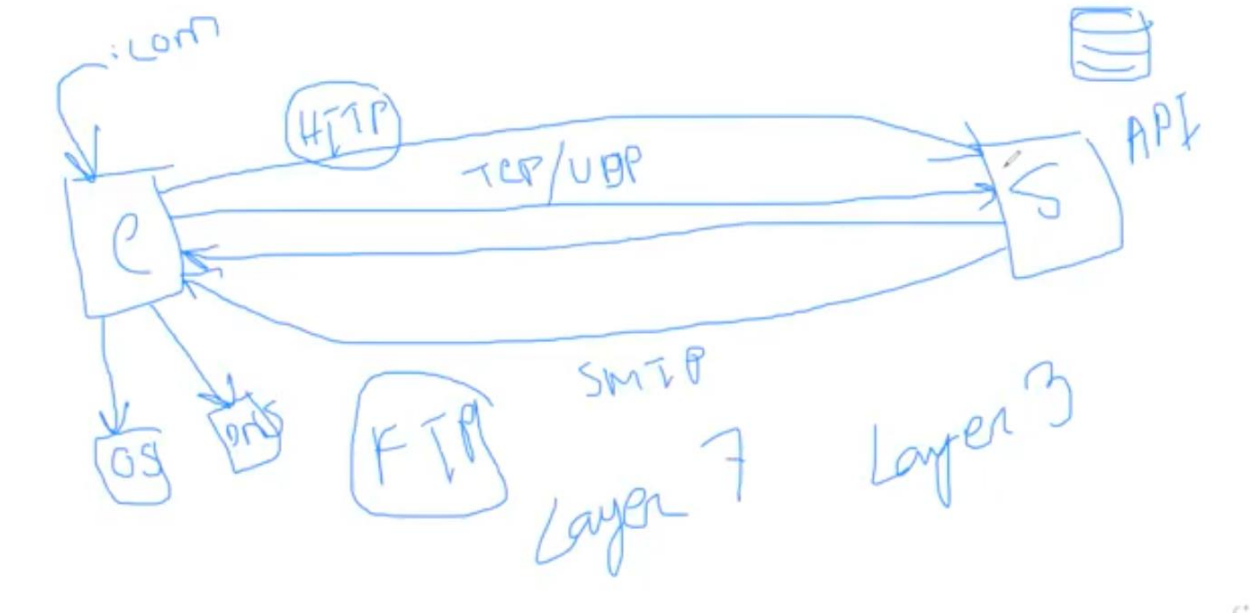
Types of internet protocol stack:

- Application Layer (Layer 07): HTTP (Hypertext Transfer Protocol), FTP (File Transfer Protocol), SMTP (Simple mail transfer protocol), DNS etc
- Transport Layer: TCP(Transmission Control Protocol): Netflix, Youtube (like pre-uploading video) etc
  UDP(User Datagram Protocol): Live match etc
- Network Layer: IP (Internet protocol)
- Link Layer: 802.11, PPP, ATM etc

Difference between TCP and UDP:

- TCP is a connection-oriented protocol and UDP is a Datagram-oriented protocol (connectionless protocol).

- In TCP, an acknowledgment segment is present because packets are waiting for the receiver. But There is no acknowledgment segment in UDP because packets deliver data constantly. Packets are careless about the receiver.
- TCP is comparatively slower than UDP. UDP is a much faster, simpler, and efficient protocol. But retransmission of lost data packets is only possible with TCP. So, it's clear any business using virtual communications can benefit greatly from UDP.
- TCP is used by HTTP, HTTPs, FTP, SMTP etc and UDP is used by DNS etc.



A three-way handshake is a method used in a TCP/IP network to create a connection between a local host/client and server.

API (Application Programming Interface):

API is a set of programming code.
When we use an application on our mobile phone, the application connects to the Internet and sends data to a server. Then the server receives the data, analyzes it, takes the appropriate actions, and delivers it back to our phone. Then the application interprets that data and displays the information. The API can send data in various formats, like JSON, XML, plain text, HTML, and whatever format the client wants.

The main work of API is, at first server accept the client HTTP request. Then server forward the HTTP request to a program named API. And API is a set of programming code, that's why it knows how to handle requests and which response should return to the server.

JSON (JavaScript Object Notation):

JSON is used to represent data on a server. The JSON type is like a python dictionary. It's easy to read by humans, and easy for machines/applications to understand.

REST (Representational State Transfer):

REST APIs are a type of web-service API or architectural style that uses the HTTP protocol and JSON as the data format. REST APIs simplify development with fewer resources and need less security.

RPC (Remote Procedure Call):

GraphQL:

GraphQL API design decides what problems it intends to solve and allows us to get the exact information we want with the fewest amount of server requests.

Curl:

Using HTTP Methods for RESTful Services:

[source: https://www.restapitutorial.com/lessons/httpmethods.html]

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Requests:

```
import requests
response=requests.get('https://api.github.com/users/sadmanamin/repos')
print(response.json()) #1
print(type(response.json())) #2

repos = response.json ()
for repo in repos:
    print(repo['name'], repo['html_url']) #3
```

```
Output:
1. It prints the whole file in JSON format.
2. It prints that this JSON file type is actually a list (In
this list, there are dictionary like every repository is a
dictionary)
3. It prints all the repository names and their html url.
```

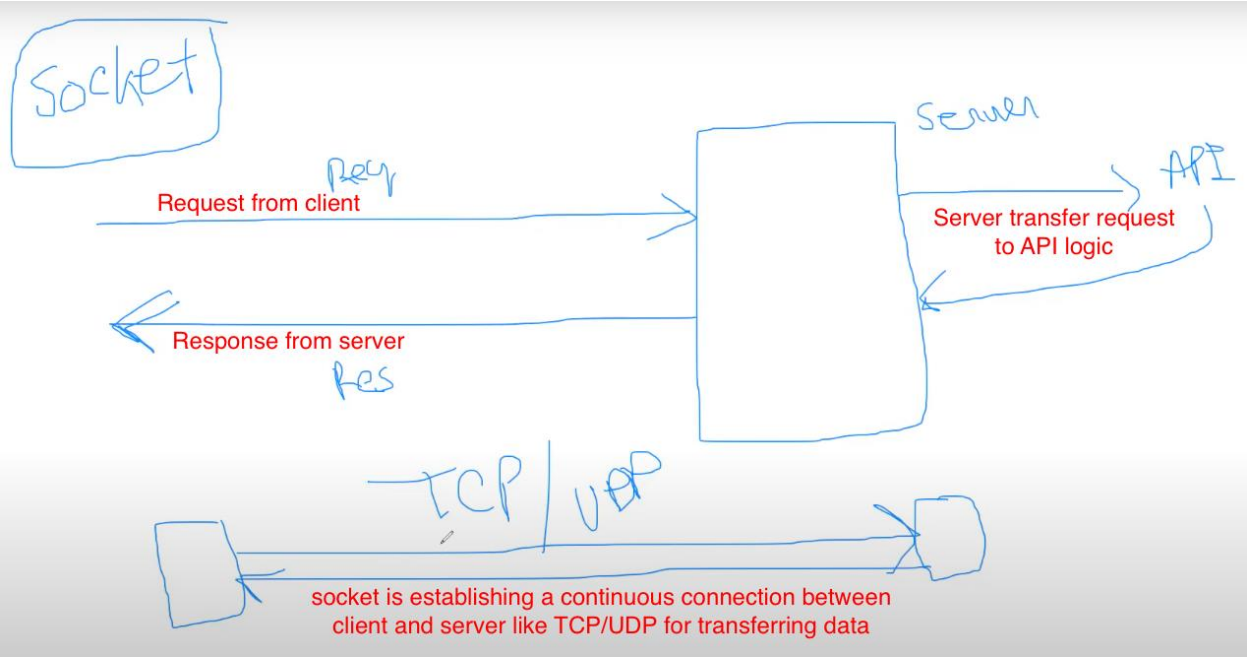
Socket:

The client creates a socket and then tries to connect to the server socket. When a connection is established, data is transferred. Each socket has a specific address like an IP address and a port number.
The main work of socket is establishing a continuous connection like TCP/UDP and client and server is connected or transferring data by this TCP/UDP connection.

Some important socket function:

Create()	To create a socket
Bind()	It's a socket identification like when a socket has both an IP address and a port number it is said to be 'bound to a port', or 'bound to an address'. A bound socket can receive data because it has a complete address. The process of allocating a port number to a socket is called 'binding'.
Listen()	Ready to receive a connection
Connect()	Ready to send data from 1st device act as a sender

Accept()	Confirmation, it is like accepting to receive a call from a sender
Write()	To send data
Read()	To receive data
Close()	To close a connection



```
#connecting client socket to server socket by TCP/UDP connection
#Socket programming is started by making a simple socket with
importing the socket library.
import socket
HOST, PORT ='', 8888

#this call results in a stream socket with the TCP protocol providing
the underlying communication.(AF_INET means that's want to establish a
TCP connection which refers to the address-family ipv4. The
SOCK_STREAM means connection-oriented TCP protocol.)
listen_socket socket.socket (socket.AF_INET, socket.SOCK_STREAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

#here, bind is working to connect between client socket and
server socket via HOST and PORT address
listen_socket.bind ( (HOST, PORT))
listen_socket.listen(1)
print (f'Serving HTTP on port {PORT} ...')
while True:
    client_connection, client_address = listen_socket.accept ()
    request_data = client_connection.recv(1024)
    print (request_data.decode ('utf-8'))
    http_response = b"""\
HTTP/1.1 200 OK
                                        I
Hello, World!
    client_connection.sendall(http_response)
    client_connection.close()


#Represent the same code by function
import socket
SERVER_ADDRESS (HOST, PORT) = '', 8888
def handle_client (client_connection):
    request_data = client_connection.recv(1024)
    print(request_data.decode('utf-8'))

   http_response b"""\
HTTP/1.1 200 OK

Hello, World!
```

```
    client_connection.sendall(http_response)
    client_connection.close()


def serve_forever():
    listen_socket = socket.socket (socket. AF_INET,socket.SOCK_STREAM)
    listen_socket.setsockopt (socket.SOL_SOCKET,
socket.S0_REUSEADDR, 1)
    listen_socket.bind (SERVER_ADDRESS)
    listen_socket.listen(1)
    print(f'Serving HTTP on port {PORT} ...')

    while True:
       client_connection, client_address listen_socket.accept ()
       handle_client(client_connection)

if __name__= "__main__":
    serve_forever()
```

The Code parameter of server takes one of following values:

- 400 − Bad Request
- 401 − Unauthenticated
- 403 − Forbidden
- 404 − Not Found
- 406 − Not Acceptable
- 415 − Unsupported Media Type
- 429 − Many Requests

os.fork():

os.fork() method in Python is used to create a child process. Basically, one task is called one process in CPU. For every process, one PID (Process IDentifier) is assigned in the CPU system and the task is working.

When the os.fork() method is called, in the CPU, two processes are presented and besides two processes assigned two completely different PID (Process IDentifier). One for the parent process, which is the carbon copy of the past process and another for the child process, which is a newly created process. Here, fork means two different identical copies of address base one for child one for the parent.

If the result of os.fork() is zero, then we are working with the child. Otherwise, if we are working for the parent, and the return value is the PID (Process IDentifier) of the child. This method returns an integer value representing the child's process id in the parent process while 0 in the child process.

```
#By while loop condition, one new connection task is forwarded only in the child process and here the parent process only passes the connection to the child process.
while True:
  client_connection, client_address = listen_socket.accept()
  pid = os.fork()
  if pid == 0:
     listen_socket.close()
     handle_client(client_connection)
     client_connection.close()
     os._exit()
  else
     client_connection.close()
```

Chat system via server command line (using socket):

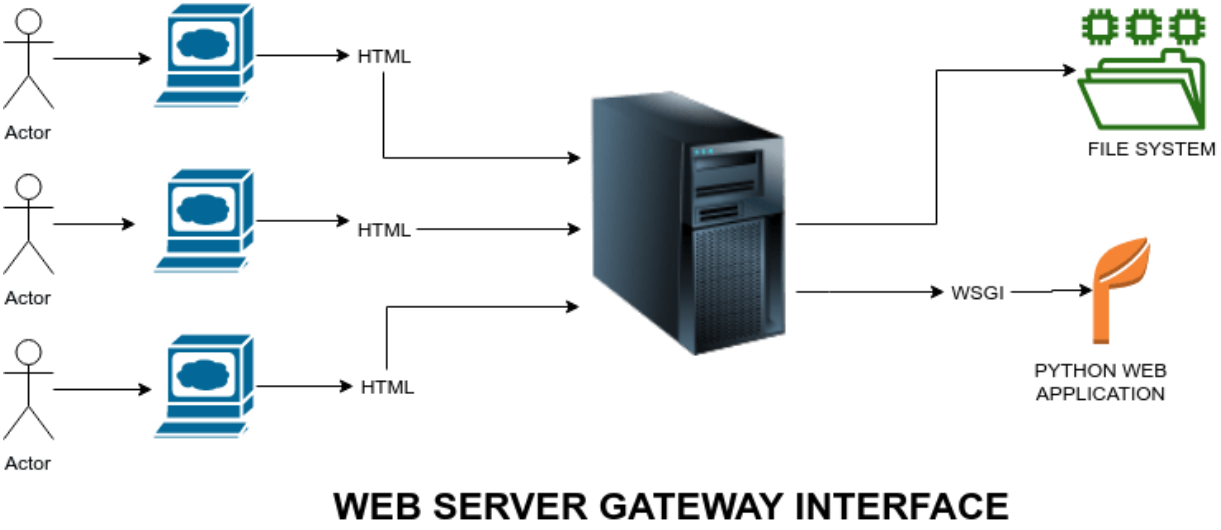
Flask:

Flask is a web framework written in Python. Flask provides tools, libraries and technologies that allow us to build a web application. This web application can be some web pages, a blog, a wiki or go as big as a web-based application or a commercial website.

Flask virtual environment create: python3 -m venv flask_env
Activating the virtual environment: source flask_env/bin/activate (For linux)
flask_env/Script/bin/activate (For windows)

WSGI (Web Server Gateway Interface):

It is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. It is used to forward requests from a web server (such as Apache or NGINX) to a backend Python web application or framework. From there, responses are then passed back to the web server to reply to the client.



Requests are sent from the client's browser to the server. Then WSGI forwards the request to the web server python app, which then returns the completed request back to the webserver and on to the browser.

Gunicorn (Green Unicorn):

Gunicorn is a pure Python HTTP server for WSGI applications. It allows us to run any Python application concurrently by running multiple Python processes within a single dyno. It provides a perfect balance of performance, flexibility, and configuration simplicity.

@ decorator:

```
#if a functions assigns in another variable
def addition(x):
   return x+x
plus = addition
print(addition)
print(plus)
print(plus(2))

Output:
<function addition at 0x7fcc8fecf670>
<function addition at 0x7fcc8fecf670>
4


#defining one function to another function (Nested function)
def house():
   print("This is my house")
   def room():
       print("This is my room")
   room()
house()

Output:
This is my house
This is my room


#passing a function as argument
def add(number):
    return number + 1
def function2(function,number):
    print(function)
    return function(number)

print(add)
```

```
print(function2(add,10))

Output:
<function add at 0x7fcc8fecf5e0>
<function add at 0x7fcc8fecf5e0>
11
```

App routing:

App routing means mapping a specific function with a specific url so that all the logic for that url is written. This is done through @app.route decorator.

Redirect URL:

Flask class has a redirect() function. When called, it redirects the user to another target location with specific status code.

```
from flask import Flask, redirect, url_for
appFlask = Flask(__name__)

@appFlask.route("/home")
def home():
return redirect(url_for('LoginPage'))

@appFlask.route("/LoginPage")
def LoginPage():
return "You are redirected to Login Page"

if __name__ == "__main__":
appFlask.run(debug=True)
```

```
#create a basic API logic design by flask

from flask import Flask, redirect, url_for

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello World"

@app.route('/home')
def home():
    return "I am from home"
    return redirect(url_for('index'))

@app.route('/user/<name>')
def user(name):
    return "This user name is ”+name




if __name__ =  "__main__":
   app.run(port=5000)
```

Jinja Templates: (Link)

- Jinja Templates are just .html files. It can generate any text-based format, such as Html, XML, CSV, or LaTeX.
- Using keyword arguments in render_template, template variables are substituted into a template when using expression delimiters {{ }}
- For condition or statement delimiters {% %} surround control statements such as if and for loop.

```
#set condition by jinja templates
{% if myname %}
<h1 style="color: Cadetblue;"> Hello World from {{myname}}</h1>
{% else %}
<h1 style="color: Cadetblue;"> Hello World</h1>
{% endif %}
```

- Dictionary: Object containing "key: value" pairs. Keys can be Strings, Numbers (Integers or Floats), or None. Value can be any data type.

Comparison operators:

- == Compares two objects for equality
- != Compares two objects for inequality
- > true if the left-hand side is greater than the right-hand side
- >= true if the left-hand side is greater or equal to the right-hand side
- < true if the left-hand side is lower than the right-hand side
- <= true if the left-hand side is lower or equal to the right-hand side

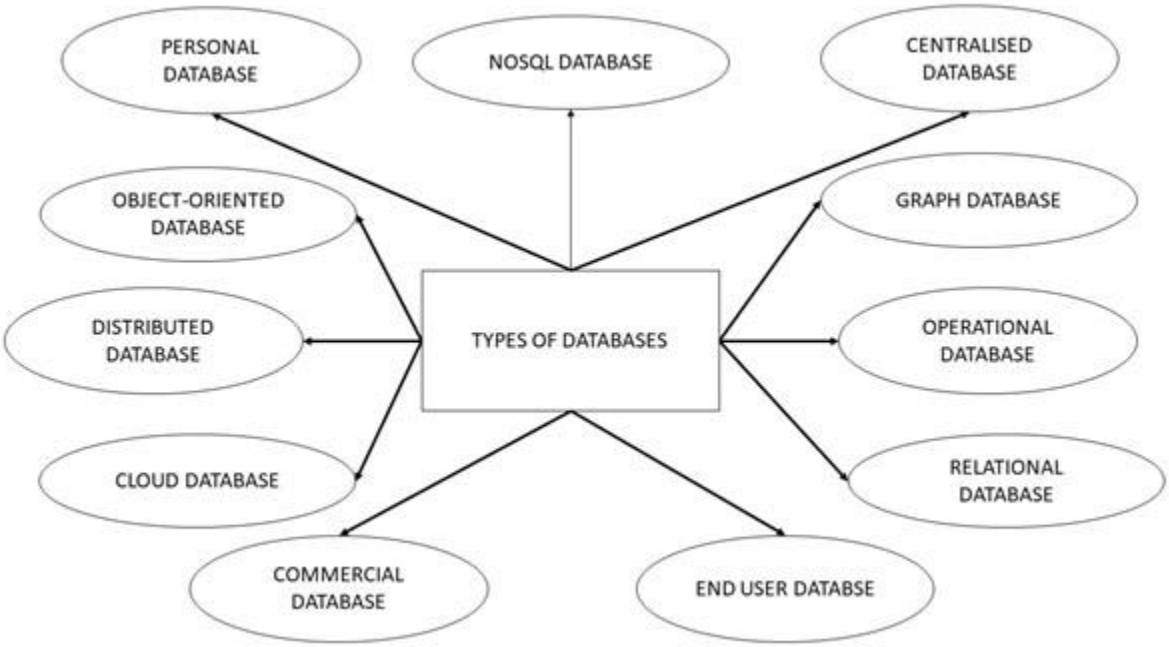
Logic operators in Jinja:

- and Boolean and
- or Boolean or
- not Boolean

```
#like
@app.route('/')
def index():
    return render_template("index.html")
```

Database:

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. For example, a company database may include tables for products, employees, and financial records. Each of these tables would have different fields that are relevant to the information stored in the table. A database is usually controlled by a database management system (DBMS). The data can then be easily accessed, managed, modified, updated, controlled, and organized.



Types of Database:

- Relational databases: The name comes from the way that data is stored in multiple, related tables. Within the tables, data is stored in rows and columns. The relational database management system (RDBMS) is the program that allows us to create, update, and administer a relational database. Structured Query Language (SQL) is the most common language for reading, creating, updating and deleting data. Relational databases are very reliable. They are compliant with ACID (Atomicity, Consistency, Isolation, Durability), which is a standard set of properties for reliable database transactions. Relational databases work well with structured data. Organizations that have a lot of unstructured or semi-structured data should not be considering a relational database.
  Examples: Microsoft SQL Server, Oracle Database, MySQL, PostgreSQL and IBM Db2.

- NoSQL databases: NoSQL is a broad category that includes any database that doesn't use SQL as its primary data access language. These types of databases are also sometimes referred to as non-relational databases. So, these types of databases are great for organizations seeking to store unstructured or semi-structured data. One

advantage of NoSQL databases is that developers can make changes to the database on the fly, without affecting applications that are using the database.
Examples: Apache Cassandra, MongoDB, CouchDB, and CouchBase.

- Object-oriented databases: An object-oriented database is based on object-oriented programming, so data and all of its attributes are tied together as an object. Object-oriented databases are managed by object-oriented database management systems (OODBMS). These databases work well with object-oriented programming languages, such as C++ and Java. Like relational databases, object-oriented databases conform to ACID standards. It is structured data.
  Examples: Wakanda, ObjectStore.

- Personal Database: Data is collected and stored on personal computers which is small and easily manageable. The data is generally used by the same department of an organization and is accessed by a small group of people.
  Examples: G-Suite account, Some types of small organizations database etc.

- Document databases: Document databases, also known as document stores, use JSON-like documents to model data instead of rows and columns. Sometimes referred to as document-oriented databases, document databases are designed to store and manage document-oriented information, also referred to as semi-structured data. Document databases are simple and scalable, making them useful for mobile apps that need fast iterations.
  Examples: MongoDB, Amazon DocumentDB, Apache CouchDB

- Centralised Database: The information(data) is stored at a centralized location and the users from different locations can access this data. This type of database contains application procedures that help the users to access the data even from a remote location. Various kinds of authentication procedures are applied for the verification and validation of end users, likewise, a registration number is provided by the application procedures which keeps a track and record of data usage. The local area office handles this.

- Distributed Database: Just opposite of the centralized database concept, the distributed database has contributions from the common database as well as the information captured by local computers also. The data is not at one place and is distributed at various sites of an organization. These sites are connected to each other with the help of communication links which helps them to access the distributed data easily. You can imagine a distributed database as a one in which various portions of a database are stored in multiple different locations(physical) along with the application procedures which are replicated and distributed among various points in a network. There are two kinds of distributed databases, homogeneous and heterogeneous. The databases which have the same underlying hardware and run over the same operating systems and application procedures are known as homogeneous DDB. All physical locations in a DDB. Whereas, the operating systems, underlying hardware as well as application

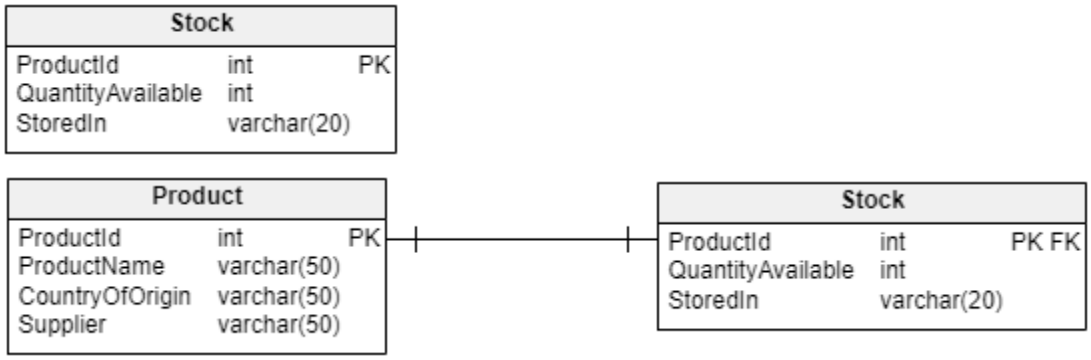
procedures can be different at various sites of a DDB which is known as heterogeneous DDB.

Keys:

Keys are attributes that help us to identify a row(tuple) in a relation(table). They allow us to find the relationship between two tables. Keys help us uniquely identify a row in a table by a combination of one or more columns in that table.

Primary key and Foreign key:

A primary key is a column or a set of columns that uniquely identifies each row in a table. It must obey the UNIQUE and NOT NULL database constraints. And a foreign key is a column or a set of columns linking one table to another. The column or columns appear in both tables, creating a link between the tables. It acts as a cross-reference between two tables as it references the primary key of another table. Every relationship in the database should be supported by a foreign key.



SQL and NoSQL:

SQL databases are relational, NoSQL databases are non-relational. SQL databases use structured query language and have a predefined schema. NoSQL databases have dynamic schemas for unstructured data. SQL databases are table-based, while NoSQL databases are document, key-value, graph, or wide-column stores. SQL databases are better for multi-row transactions, while NoSQL is better for unstructured data like documents or JSON.

ORM (Object-Relational Mapping):

ORM is a technique that lets us query and manipulate data from a database using an object-oriented paradigm. ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures.

SQLAlchemy:

SQLAlchemy is a library that facilitates the communication between Python programs and databases. This library is used as an Object Relational Mapper (ORM) tool that translates Python classes to tables on relational databases and automatically converts function calls to SQL statements.