Index

List VS Tuple

Both List and Tuple are store collections of items.

List	Tuple
List are mutable	Tuple are immutable
Mutable means we can modify their content by adding, removing elements, and changing their values.We can use methods such as append, extend, insert, remove, pop to modify the list.	Immutable means once they are created, we can not change or modify them. We can not add, remove, modify element after they have been created
Lists are created using square brackets [ ].	Tuple are created using parentheses ( ).
Lists are Slower as compared to tuples. Because of it Mutable and it takes slightly larger memory.	Tuple are faster as compared to list. Because it's immutable.
List have more built-in methods	Tuple are fewer built methods as compared to list.
Can not used as key for dictionary	Can be used as key for dictionary

Which is faster between List or Tuple?

1) **Mutability:**

List: Lists are mutable, meaning their contents can be changed after creation (e.g., elements can be added, removed, or modified).
Tuple: Tuples are immutable, meaning their contents cannot be changed once created.

2) **Memory Usage:**

List: Lists generally consume more memory than tuples because they need to support operations that modify the list.
Tuple: Tuples are more memory-efficient because they are fixed-size and immutable.

3) **Performance:**

Access Time: Both lists and tuples provide O(1) time complexity for accessing elements by index. So, in terms of access speed, they are similar.
Creation Time: Creating a tuple is generally faster than creating a list. This is because tuples are simpler and do not involve the overhead associated with mutability and resizing.

4) **Operations:**

Lists: Lists support a wide range of operations, including adding, removing, and modifying elements, which introduces additional overhead.
Tuples: Tuples support fewer operations due to their immutability, which means fewer operations are required to maintain their integrity.

Why is Tuple Faster?

Immutability: Tuples are immutable, allowing for optimizations that make them faster and more memory-efficient.
Memory Layout: Tuples have a simpler memory layout compared to lists, which need to support dynamic resizing.
Performance: Tuple operations generally incur less overhead due to their fixed size and immutability.

What is Module?

1) A module allows you to organize your python code to be easier to manage and reuse.
2) The file name is the module name with the suffix .py added.
3) Modules help in building large, organized, and scalable codebases.
4) To create a module, we have to write a python script containing classes, functions and variables and save it with the .py extension.
5) Python comes with so many built-in modules. Like:- math, random, time, datetime, os, sys, json, urllib, collections, re, csv, pickle,

How to import modules?

1) We can use functions and variables from modules by importing them into another python script.

Example:-

```
import module
module.fun1("Rahul")
module.pi
```

2) We can import specific functions and variables from a module.

```
from module import fun1, pi
fun1("Rahul")
print(pi)
```

What is a Package?

1) When you have multiple module files organized within a single directory along with an __init__.py file, it is called package.
2) Import:- from package import module1

Recursive Function?

1) A function calls itself.
2) A recursive function is a function that calls itself during its execution.
3) Each recursive call typically solves a smaller instance of the same problem, and the process continues until a base case is reached.
4) The base case is essential to prevent infinite recursion and defines the smallest problem that can be solved.

```
def factorial(n):
    # Base case
    if n == 0 or n == 1:
        return 1
    else:
        # Recursive case
        return n * factorial(n - 1)

result = factorial(5)
print(result)  # Output: 120
```

What is a Suite?

1) A suite refers to a block of code grouped together.
2) It is associated with control flow statements such as if else elif for while and function definition.
3) Indentation is used to define the extent of a suite.

```
if condition:
    # This is the suite for the if statement
    print("Condition is True")
else:
    # This is the suite for the else statement
    print("Condition is False")
```

Data Type

Primary Data types:

1) **Numeric Types**
   int = (Whole Number) = integer_variable = 42
   float = (Decimal Number) = float_variable = 3.14
   complex = (imaginary part) = complex_variable = 1 + 2j

2) **Text Type**
   str = (text in single or double quotes) = string_variable = "Hello, World!"

3) **Sequence Type**
   list = (Mutable,ordered collection of item) = list_variable = [1, 2, 3, 4]
   tuple = (Immutable,ordered collection of item) = tuple_variable = (1, 2, 3, 4)
   range = (sequence of number) = range_variable = range(5)

4) **Set Types**
   set = (unordered collection of unique item) = set_variable = {1, 2, 3, 4}
   frozenset = (immutable set) = frozenset_variable = frozenset({1, 2, 3, 4})

5) **Mapping Type**
   dict = (collection of key-value) = dictionary_variable = {"key1": "val1", "key2": "val2"}

6) **Boolean Type**
   bool = (True or False) = boolean_variable = True

7) **None Type**
   None = (absence of value) = none_variable = None

Slice Operator?

1) The slice operator is used to extract a portion of sequence from string, list, tuple.
2) The syntax of slice operator is start:stop:step
3) Start: Start is the index which slice is begin(Inclusive)
4) Stop: Stop is the index at which slice ends.(Exclusive)
5) Step: Step is the step between elements in the slice. By Default it is 1.

```
str1 = "Rahul Gupta"
print(str1[1:4]) #ahu

str1 = "Hello World"
```

```
print(str1[::2]) # HloWrd
```

What is Class?

1) A Class is a blueprint of an object.
2) Classes define the attribute and behavior of the objects.
3) __init__ method initializes object attributes
4) The __init__ method is a special method in class used as a constructor.
5) It is called automatically when you create a new instance of a class.
6) Once a class is defined, we can create multiple instances of the class.

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says woof!")

# Creating an instance (object) of the Dog class
my_dog = Dog("Buddy", 3)

# Accessing attributes
print(my_dog.name)  # Output: Buddy
print(my_dog.age)   # Output: 3

# Calling methods
my_dog.bark()  # Output: Buddy says woof!
```

What is an Object in class?

1) An object is an instance of a class in object-oriented programming.
2) Each object has a unique identity, type, and value, distinguishing it from other objects.
3) An object is any entity that has attributes and behaviors. For example, a parrot is an object. It has **attributes** - name, age, color, etc. **behavior** - dancing, singing, etc.

```
class College():
    def __init__(self, name):
```

```
        self.name=name
    def print_data(self):
        print("Name is ",self.name)

# Creating instances of the College class
obj = College("Rahul") # creating object
obj.print_data()

# Accessing attributes and calling methods of the instances
print(obj.name)
```

What is the __init__ function?

1) __init__ stands for initialize'.
2) __init__ is a special method, also known as constructor in python.
3) It is automatically called when an object is created from class.
4) It is used to initialize the attribute of the object.

What is Self ?

Self is used to represent the instance of class.

1) It is the first parameter of methods(Function) in a class and refers to the instance of class.
2) By convention, this parameter is named self, but you could name something else.
3) Self is to create and access instance variables (attribute).

Pass statement

1) The pass statement is the no-operation statement.
2) It serves as a placeholder where syntactically some code is required but no action is desired.
3) In this situation, if you want to write a function in future. So easily you can pass a statement with a function.

```
if condition:
    # Some code will go here
else:
    pass  # Placeholder for future code
```

Regular Expression (RegEx) ?

1) Regular expressions in python are a powerful tool for pattern matching and string manipulation.
2) RegEx can be used to check if a string contains the specified search pattern.
3) Python provides **re** module, which contains functions for working with regular expressions.

```
import re

str1="Rahul is a good boy."
x = re.search("Rahul", str1)
print(x) # <re.Match object; span=(0, 5), match='Rahul'>
```

Refer:- https://www.w3schools.com/python/python_regex.asp

What is the difference between append vs extend ?

Append	Extend
Add a single element at the end of the list.	Add multiple element at the end of list
Accept a single element as an argument.	Accept an iterable (list, tuple, string) as an argument.
lst1 = [3,6,2] lst1.append([8,9]) print(lst1) # [3, 6, 2, [8, 9]]	lst1 = [3,6,2] lst1.extend([8,9]) print(lst1) # [3, 6, 2, 8, 9]

What is Decorator ?

1) Decorator is a special kind of function that adds extra functionality to another function.
2) Decorator can be used to wrap or enhance the functionality of functions without modifying their code directly.
3) Decorator function takes another function as an argument.

Simple syntax of decorator is @my_decorator

```
def my_decorator(fun):
    def wrapper():
        print("Before")
        fun()
        print("After")
    return wrapper

@my_decorator
def hello():
    print("Hello")
hello()
```

Used in our code:

```
def exception_handler(func):
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            print(f"Exception in {func.__name__}: {e}")
            return None
    return wrapper

@exception_handler
def fun1():
    print("fun1")
    lst=[1,2,3]
    for i in range(len(lst)+1):
        print(lst[i])
fun1()
```

Write a decorator function that calculate the execution time of any function
import time

def execution_time(func):

```
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_duration = end_time - start_time
        print(f"Execution time of {func.__name__}: {execution_duration:.6f} seconds")
        return result
    return wrapper

# Example usage
@execution_time
def example_function(n):
    total = 0
    for i in range(n):
        total += i
    return total

example_function(1000000)
```

What is the purpose of Decorator ?

1) **Code Reusability:**

Definition: Decorators allow you to define reusable functionality that can be applied to multiple functions or methods. This helps to avoid code duplication and adhere to the DRY (Don't Repeat Yourself) principle.
Example: Logging, authentication, and timing can be implemented once and applied to different functions.

2) **Separation of Concerns:**

Definition: Decorators help to separate the core logic of a function from auxiliary tasks like logging, access control, or caching. This improves code organization and readability.
Example: You can separate the logic of a function from its logging behavior, making both the function and the logging more manageable.

3) **Enhanced Readability:**

Definition: By using decorators, you can keep the core function clean and focused on its primary purpose, while the decorator handles additional responsibilities.
Example: Applying a decorator to manage access control or validation makes the main function simpler and easier to understand.

```
import time

def calculate_time(func):
    def wrapper():
        start_time = time.time()
        func()
        end_time = time.time()
        print("Time Taken = ", end_time-start_time)
    return wrapper

@calculate_time
def sample():
    print("sample function")
    time.sleep(2)

sample()
```

```
sample function
Time Taken =  2.000176191329956
```

Why use Decorator ?

1) **Code Reusability**: Avoid duplication by applying the same functionality to multiple functions.
2) **Separation of Concerns**: Keep the core logic clean by isolating cross-cutting concerns.
3) **Enhanced Readability**: Simplify the main logic of functions by handling additional responsibilities through decorators.
4) **Aspect-Oriented Programming**: Inject behaviors like logging, caching, or error handling without modifying the original functions.
5) **Function Modification**: Dynamically alter function behavior to add features like caching or retry logic.
6) **Method Wrapping in Classes**: Enhance or modify class methods with additional behavior.

Difference between Generator and Iterator ?

Iterator:-

An **iterator** is an object that enables a programmer to traverse through all the elements of a collection (like a list or a tuple) without needing to know the underlying data structure.

```
lst1=[5,1,8]
lst_iter = iter(lst1)
print(next(lst_iter))
print(next(lst_iter))
print(next(lst_iter))
```

Generator:-

A **Generator** in Python is a function that returns an iterator using the Yield keyword.
it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a Python generator function.
A **generator** is a special type of iterator that is defined using a function rather than a class. It allows you to iterate over a sequence of values but without the overhead of storing the entire sequence in memory.

```
def fibo():
    first = 0
    second = 1
    while True:
        yield first
        third = first + second
        first, second = second, third

fib_result = fibo()
print(next(fib_result))
print(next(fib_result))
print(next(fib_result))
print(next(fib_result))
```

```
for i in range(20):
   print(next(fib_result))
```

```
def square_generator():
    for i in range(1, 6):
        yield i*i
res = square_generator()
print(next(res))
print(next(res))
```

What is Interface (`@abstractmethod`)`or abstract class` ?

1) Interface is a set of methods that a class must implement without specifying the exact behavior of those methods.
2) It acts as a blueprint for other classes, ensuring that they follow a certain structure.
3) In python, interface can be implemented using Abstract Base Classes from the abc module.
4) interface can also mean how you interact with external services or libraries. Like aws sqs queue manager

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Bark")
```

What is Abstract Class ?

1) Abstract classes in Python are used to define a blueprint for other classes. They allow you to create a set of methods that must be created within any child classes built from the abstract class.
2) An abstract class can include abstract methods as well as concrete methods.
3) An abstract base class is a class that cannot be directly used to create objects; Instead, it serves as a blueprint for other classes.
4) In Python, Abstract base classes are defined using the abc module, and they typically include one or more abstract methods.

5) Abstract methods are declared using the @abstractmethod decorator.
6) @abstractmethod decorator is used to define methods that must be implemented by any concrete subclass.

Example:
In this example using the ABC module in Python, I've defined an abstract base class called ButtonFunctionClass. This class includes abstract methods such as textSize, textColor, and btnColor, which serve as blueprints for other classes. Both the LoginButton and SubmitButton classes inherit from ButtonFunctionClass, obligating them to implement these abstract methods. The LoginButton class defines specific implementations for textSize, textColor, and btnColor, returning values like '20px', 'white', and 'black', respectively. Similarly, the SubmitButton class provides its own implementations, returning values such as '18px', 'black', and 'green'. This structure ensures that any class inheriting from ButtonFunctionClass must adhere to the interface defined by its abstract methods, promoting consistency and enabling polymorphic behavior across different button types.

```
from abc import ABC, abstractmethod

class ButtonFunctionClass(ABC):

    @abstractmethod
    def textSize(self):
        pass

    @abstractmethod
    def textColor(self):
        pass

    @abstractmethod
    def btnColor(self):
        pass

class LoginButton(ButtonFunctionClass):

    def textSize(self):
        return "20px"
```

```
    def textColor(self):
        return "white"

    def btnColor(self):
        return "black"

class SubmitButton(ButtonFunctionClass):

    def textSize(self):
        return "18px"

    def textColor(self):
        return "black"

    def btnColor(self):
        return "green"

ob1 = LoginButton()
ob1_text_size = ob1.textSize()
ob1_text_color = ob1.textColor()
ob1_btn_color = ob1.btnColor()
print("ob1_text_size = ", ob1_text_size)
print("ob1_text_color = ", ob1_text_color)
print("ob1_btn_color = ", ob1_btn_color)

ob2 = SubmitButton()
ob2_text_size = ob2.textSize()
ob2_text_color = ob2.textColor()
ob2_btn_color = ob2.btnColor()
print("ob2_text_size = ", ob2_text_size)
print("ob2_text_color = ", ob2_text_color)
print("ob2_btn_color = ", ob2_btn_color)
```

Why use abstract class ?

1) **Enforcing a Contract**: Abstract classes enforce a contract for subclasses. By defining an abstract class with abstract methods, you ensure that any subclass

implements these methods. This is particularly useful in large projects or frameworks where you want to ensure that derived classes conform to a certain interface or set of behaviors.

2) **Providing Default Behavior**: While abstract methods force subclasses to implement specific methods, abstract classes can also provide concrete methods (methods with actual code). This allows you to provide default behavior that can be shared among all subclasses while still enforcing certain methods to be implemented by each subclass.
3) **Preventing Instantiation**: Abstract classes cannot be instantiated directly. This means you can prevent creating instances of a class that is meant to only provide a structure or shared functionality. For example, if you have a base class `Animal`, you may not want to instantiate `Animal` directly, but rather instantiate specific animals like `Dog` or `Cat` that inherit from `Animal`.
4) **Promoting Code Reuse**: Abstract classes can help promote code reuse. If there is a common functionality or method that can be shared among subclasses, you can define it in the abstract class. Subclasses can then reuse this code without having to duplicate it.
5) **Defining a Clear Interface**: Abstract classes are a way to define a clear interface for a set of subclasses. They provide a blueprint for what methods a class should implement, which helps in maintaining consistency across different parts of a codebase. This can be particularly useful in collaborative environments where multiple developers are working on different parts of a system.

What are Access Specifiers ? (ENCAPSULATION)

1. Public Members

Definition: Members (attributes and methods) that are meant to be accessible from outside the class.
Naming Convention: No underscore prefix (_).
Access: Can be accessed freely from outside the class.

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def display_info(self):
        print(f"Name = {self.name}, Marks = {self.marks}")

obj = Student("Rahul", 99)
obj.display_info()
```

2) Protected Members

Definition: Members that are intended to be accessed only within the class and its subclasses.
Naming Convention: Single underscore prefix (_).
Access: Should not be accessed directly outside the class; however, this is only a convention and not enforced by Python.

```
class Student:
    def __init__(self, name, marks):
        self._name = name
        self._marks = marks

    def display_info(self):
        print(f"Name = {self._name}, Marks = {self._marks}")

obj = Student("Rahul", 99)
obj.display_info()
print(obj._name)
```

3. Private Members

Definition: Members that are meant to be accessed only within the class itself, not outside or in subclasses.
Naming Convention: Double underscore prefix (__).
Access: Python performs name mangling to make it harder (but not impossible) to access these members from outside the class.

```
class Student:
    def __init__(self, name, marks):
        self.__name = name
        self.__marks = marks

    def display_info(self):
        print(f"Name = {self.__name}, Marks = {self.__marks}")

obj = Student("Rahul", 99)
obj.display_info()
print(obj.__name)
```

```
Name = Rahul, Marks = 99
AttributeError: 'Student' object has no attribute '__name'
```

What is Exception handling ?

1) Exception handling is a mechanism that allows developers to manage and respond to errors or unexpected events (known as exceptions) that occur during the execution of a program.
2) Exception handling is performed using the try, except, else, and finally blocks.
3) These constructs allow you to handle errors gracefully and ensure certain code runs regardless of whether an exception occurs.

Exception Handling Syntax
Here's a basic overview of each block:

try: This block contains the code that may potentially raise an exception.
except: This block catches and handles the exception if one is raised in the try block.
else: This block executes if no exceptions are raised in the try block.
finally: This block always executes, regardless of whether an exception occurred or not. It is typically used for cleanup actions (like closing files or releasing resources).

```
try:
    # Code that may raise an exception
    result = 10 / 2
    print("Inside try block, Result:", result)
except ZeroDivisionError:
    # This block will execute if a ZeroDivisionError is raised
    print("Division by zero is not allowed.")
except TypeError:
    # This block will execute if a TypeError is raised
    print("Type error occurred.")
else:
    # This block will execute if no exceptions are raised in the
try block
    print("No exceptions occurred. Division was successful.")
finally:
```

```
    # This block always executes, regardless of whether an
exception occurred
    print("This is the finally block. It always executes.")

# Output:
# Inside try block, Result: 5.0
# No exceptions occurred. Division was successful.
# This is the finally block. It always executes.
```

Where To Use Finally?

1) The **finally** keyword in Python is used to define a block of code that will always be executed.
2) When working with resources like files, network connections, or database connections, it is crucial to ensure these resources are released or closed properly, even if an error occurs. The finally block is perfect for this purpose.

Difference between Raise VS Exception?

raise: A statement used to manually trigger an exception in your code, providing control over error handling and flow.
Syntax: `raise ValueError("string")`

Exceptions: Objects that represent errors or unusual conditions in a program, automatically or manually raised, and handled using try and except blocks.

```
def check_positive_number(num):
    if num<0:
        raise ValueError("Number must be Positive")
    else:
        print(f"{num} is valid!!!")

check_positive_number(-1) #ValueError: Number must be Positive

try:
    check_positive_number(-1)
except Exception as e:
```

```
    print(f"Error = {e}")
```

How to call an API in python ?

1) To call an API in Python, you typically use the requests library, which is a popular and easy-to-use HTTP library.
2) The requests library allows you to send HTTP requests and handle responses easily.

```
import requests

url = "https://jsonplaceholder.typicode.com/users"

response = requests.get(url)
if response.status_code == 200:
    data = response.json()
    print("Name:", data[0]["name"])
else:
    print("Failed to retrieve data. Status code:",
response.status_code)
```

OOPs Concepts

1) **Encapsulation**: Keeps data safe from outside interference and misuse by hiding internal details and requiring all interactions to be through well-defined methods.
2) **Abstraction**: Simplifies complex systems by focusing on essential characteristics and hiding unnecessary details.
3) **Inheritance**: Facilitates code reuse and establishes a hierarchical relationship between classes.
4) **Polymorphism**: Allows different classes to be treated as instances of the same class through a common interface, promoting flexibility and extensibility in the codebase.

Dict vs Json

Dict	Json
dict is a built-in data type that represents a collection of key-value pairs.	JSON is a lightweight data interchange format that is used for representing

	structured data as text.
It is used for storing data in a way that allows for fast retrieval based on keys.	It is language-independent and is often used for data exchange between a server and a web application.
Python dictionaries are used to store and manage data within a Python program.	JSON is commonly used in web APIs to transmit data between a client and server.
Keys can be of any immutable type (e.g., strings, numbers, tuples.	Keys must be strings (enclosed in double quotes).
You can convert a Python dictionary to a JSON `json.dumps(my_dict)`	You can convert a JSON string to a Python dictionary `json.loads(json_string)`

List Comprehension

List comprehension is an efficient way to create lists in Python.
Basic Syntax
[expression for item in iterable if condition]

```
even = [item for item in range(1, 12) if item%2==0]

lst1 = [
        [1,2,3],
        [4,5,6],
        [7,8,9],
    ]
result = [inner_item for item in lst1 for inner_item in item]
```

Dict Comprehension

Dict comprehension in Python is a concise way to create dictionaries.
Basic Syntax
{key_expression: value_expression for item in iterable if condition}

```
even_square = { i:i*i for i in range(20) if i%2==0}
print(even_square)
#{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100, 12: 144, 14: 196,
16: 256, 18: 324}
```

```
lst=["rahul", "raj", "sandeep"]
count = { item:len(item) for item in lst}
print(count)
# {'rahul': 5, 'raj': 3, 'sandeep': 7}
```

Better between Monolithic and Microservice?

Monolithic:

Monolithic architecture is a traditional software development model where all the components of an application are built as a single, unified unit.
A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together.

Feature:

- Single Codebase
- Tight Coupling
- Single Deployment
- Scalability
- Development and maintenance

Advantage of monolithic

1) **Easy deployment** – One executable file or directory makes deployment easier.
2) **Development** – When an application is built with one code base, it is easier to develop.
3) **Performance** – In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices.
4) **Simplified testing** – Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application.
5) **Easy debugging** – With all code located in one place, it's easier to follow a request and find an issue.

Disadvantage of monolithic

1) **Slower development speed** – A large, monolithic application makes development more complex and slower.

2) **Scalability** – You can't scale individual components.
3) **Reliability** – If there's an error in any module, it could affect the entire application's availability.
4) **Barrier to technology adoption** – Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming.
5) **Lack of flexibility** – A monolith is constrained by the technologies already used in the monolith.
6) **Deployment** – A small change to a monolithic application requires the redeployment of the entire monolith.

Microservice:

A microservices architecture breaks down the application into smaller, independent services, each responsible for a specific business capability (e.g., user service, payment service).
independent code bases.
an architectural method that relies on a series of independently deployable services. These services have their own business logic and database with a specific goal. Updating, testing, deployment, and scaling occur within each service.

Advantages

1) **Agility** – Promote agile ways of working with small teams that deploy frequently.
2) **Flexible scaling** – If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure. We are now multi-tenanant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes.
3) **Continuous deployment** – We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
4) **Highly maintainable and testable** – Teams can experiment with new features and roll back if something doesn't work. This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services.

5) **Independently deployable** – Since microservices are individual units they allow for fast and easy independent deployment of individual features.
6) **Technology flexibility** – Microservice architectures allow teams the freedom to select the tools they desire.
7) **High reliability** – You can deploy changes for a specific service, without the threat of bringing down the entire application.

   Happier teams – The Atlassian teams who work with microservices are a lot happier, since they are more autonomous and can build and deploy themselves without waiting weeks for a pull request to be approved.

Disadvantages

1) **Development sprawl** – Microservices add more complexity compared to a monolith architecture, since there are more services in more places created by multiple teams. If development sprawl isn't properly managed, it results in slower development speed and poor operational performance.
2) **Exponential infrastructure costs** – Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more.
3) **Added organizational overhead** – Teams need to add another level of communication and collaboration to coordinate updates and interfaces.
4) **Debugging challenges** – Each microservice has its own set of logs, which makes debugging more complicated. Plus, a single business process can run across multiple machines, further complicating debugging.
5) **Lack of standardization** – Without a common platform, there can be a proliferation of languages, logging standards, and monitoring.
6) **Lack of clear ownership** – As more services are introduced, so are the number of teams running those services. Over time it becomes difficult to know the available services a team can

Monolithic vs Microservice

Monolithic	Microservice
Single codebase with multiple module	Individual services
If changes, then redeploy entire system	If changes, then deploy only updated service.
Hard to scale	Easy to scale
If service fails, the entire application goes down.	If any service fails, the whole application does not go down.
Low complexity	High complexity
Limitation of technology	Flexibility
Require less planning at the start	Require more planning and infrastructure at start
Entire application deployed at single	Each microservice is independent deployed
End to end testing	Independent components need to be tested individually
Need distributed team effort	Team of developer work together

How to optimize the python code ?

1) Use appropriate Data Structure:

List for Ordered collections

Sets for unique, unordered element
Dictionary for key-value pairs

2) Use Built-in Functions and Libraries
Built-in functions are implemented in C and are faster than custom Python code.
For **example**, prefer sum() over a manually written loop. and Libraries like NumPy or pandas.

3) Cache results
If you're performing a costly computation multiple times, store the result and reuse it.

4) Use List Comprehensions
Use list comprehensions instead of loops for simple operations.

5) Minimize function calls
Store function results in variables if used multiple times.

6) Use generators for large datasets
Saves memory by yielding values one at a time

7) Utilize local variables
Accessing local variables is faster than global ones

8) Implement caching
Use @lru_cache decorator for expensive function calls

9) efficient string concatenation
Use join() method instead of += for string concatenation in loops

How do you check function execution time ?

```
import time

def some_function():
    # Simulate a delay
    time.sleep(2)
    return "Function Executed"

start_time = time.time()  # Start the timer
result = some_function()
```

```
end_time = time.time()  # End the timer

execution_time = end_time - start_time  # Calculate the time
difference
print(f"Execution time: {execution_time} seconds")
```

What is Concurrency ?

Concurrency refers to the ability of a program to deal with multiple tasks at once.
This doesn't necessarily mean that these tasks are being executed simultaneously, but rather that they can be managed in a way that makes it seem like they are happening at the same time.
Concurrency is useful in I/O-bound tasks, where a program can spend a lot of time waiting (e.g., for data from a network, file system, or database).

Threading:

Python's threading module allows you to run multiple threads (smaller units of a process) concurrently.
However, due to Python's Global Interpreter Lock (GIL), only one thread executes Python bytecode at a time, which can limit the effectiveness of threading for CPU-bound tasks.

Asyncio:

Python's asyncio module is used for writing concurrent code using the async/await syntax.
It is particularly useful for I/O-bound tasks, where the program waits for external resources.
asyncio allows you to run other tasks while waiting.

****Example Using `threading`:****

```
import threading
import time

def worker(num):
    print(f'Worker {num} starting')
    time.sleep(2)
    print(f'Worker {num} finished')
```

```
threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=(i,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Example Using `asyncio`:

```
import asyncio

async def worker(num):
    print(f'Worker {num} starting')
    await asyncio.sleep(2)
    print(f'Worker {num} finished')

async def main():
    tasks = [asyncio.create_task(worker(i)) for i in range(5)]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Parallelism

Parallelism refers to the simultaneous execution of tasks, typically across multiple CPU cores.
It's a type of concurrency that is truly simultaneous.
Parallelism is useful for CPU-bound tasks, where the program can benefit from being split into independent units that can be processed simultaneously on different cores.

Multiprocessing:

The multiprocessing module in Python creates separate memory spaces for each process, allowing true parallel execution. Each process runs independently and in parallel with other processes on separate CPU cores, bypassing the GIL issue.

****Example Using `multiprocessing`:****

```
import multiprocessing
import time

def worker(num):
    print(f'Worker {num} starting')
    time.sleep(2)
    print(f'Worker {num} finished')

if __name__ == '__main__':
    processes = []
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()
```

When to Use Concurrency vs. Parallelism ?

Concurrency:
Best suited for I/O-bound tasks, such as web scraping, database queries, or reading/writing files.
It allows the program to remain responsive and manage multiple I/O operations efficiently.

Parallelism:
Best suited for CPU-bound tasks, such as numerical computations, data processing, or image processing, where splitting the work across multiple CPU cores can reduce execution time.

Combined Example with concurrent.futures

Python's concurrent.futures module provides a high-level interface for both thread-based and process-based parallelism.

ThreadPoolExecutor: Suitable for I/O-bound tasks where threads can handle multiple I/O operations concurrently.

ProcessPoolExecutor: Suitable for CPU-bound tasks where processes can run in parallel on multiple cores.

```
from concurrent.futures import ThreadPoolExecutor,
ProcessPoolExecutor
import time

def worker(num):
    print(f'Worker {num} starting')
    time.sleep(2)
    print(f'Worker {num} finished')
    return num

# Example with ThreadPoolExecutor for concurrency (I/O-bound tasks)
with ThreadPoolExecutor(max_workers=5) as executor:
    results = list(executor.map(worker, range(5)))

# Example with ProcessPoolExecutor for parallelism (CPU-bound tasks)
with ProcessPoolExecutor(max_workers=5) as executor:
    results = list(executor.map(worker, range(5)))
```

What is Dependency Injection?

Dependency Injection (DI) is a software design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies. Instead of a class creating its own dependencies, they are "injected" from the outside, typically by a framework or through constructor/function parameters.

Why Use Dependency Injection?

- Improves testability (easier to mock dependencies)
- Reduces coupling (classes don't directly create/use other classes)
- Promotes code reuse and flexibility

Example:

```
class Service:
    def serve(self):
        return "Serving"

class Client:
    def __init__(self):
        self.service = Service()  # tightly coupled
```

```
    def do_work(self):
        return self.service.serve()
```

Object Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of objects, which contain data (attributes) and behavior (methods).
It helps in organizing code in a way that's modular, reusable, and easier to maintain. OOP allows us to model real-world entities like cars, users, or employees into code.

Key Concepts of OOP:
Class – A blueprint for creating objects.
Object – An instance of a class.
Encapsulation – Hiding internal data and showing only necessary details.
Inheritance – One class can inherit properties and methods from another.
Polymorphism – The same method name behaves differently in different classes.
Abstraction – Hiding complex implementation and showing only essentials.

Feature of Inheritance [Credit Fair]

Inheritance is a core feature of object-oriented programming that allows a class (child class) to acquire the properties and behaviors (methods) of another class (parent class).

OOPs Concepts in Python

- Class in Python
- Objects in Python
- Polymorphism in Python
- Encapsulation in Python
- Inheritance in Python
- Data Abstraction in Python

Python Class

A class is a collection of objects. Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.

- Attributes are always public and can be accessed using the dot (.) operator. Example: Myclass.Myattribute

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute
```

Explanation:
class Dog: Defines a class named Dog.
species: A class attribute shared by all instances of the class.
__init__ method: Initializes the name and age attributes when a new object is created.

Python Objects

An Object is an instance of a Class. It represents a specific implementation of the class and holds its own data.

An object consists of:
State: It is represented by the attributes and reflects the properties of an object.
Behavior: It is represented by the methods of an object and reflects the response of an object to other objects.
Identity: It gives a unique name to an object and enables one object to interact with other objects.

Creating an object in Python involves instantiating a class to create a new instance of that class. This process is also referred to as object instantiation.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

# Creating an object of the Dog class
dog1 = Dog("Buddy", 3)
print(dog1.name)
print(dog1.species)
```

Explanation:
dog1 = Dog("Buddy", 3): Creates an object of the Dog class with name as "Buddy" and age as 3.
dog1.name: Accesses the instance attribute name of the dog1 object.

dog1.species: Accesses the class attribute species of the dog1 object.

Self Parameter

self parameter is a reference to the current instance of the class. It allows us to access the attributes and methods of the object.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute

dog1 = Dog("Buddy", 3)  # Create an instance of Dog
dog2 = Dog("Charlie", 5)  # Create another instance of Dog

print(dog1.name, dog1.age, dog1.species)  # Access instance and class attributes
print(dog2.name, dog2.age, dog2.species)  # Access instance and class attributes
print(Dog.species)  # Access class attribute directly
```

__init__ Method

__init__ method is the constructor in Python, automatically called when a new object is created. It initializes the attributes of the class.

Class and Instance Variables

In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

Class Variables

These are the variables that are shared across all instances of a class. It is defined at the class level, outside any methods. All objects of the class share the same value for a class variable unless explicitly overridden in an object.

Instance Variables

Variables that are unique to each instance (object) of a class. These are defined within the __init__ method or other instance methods. Each object maintains its own copy of instance variables, independent of other objects.

```
class Dog:
    # Class variable
    species = "Canine"
    def __init__(self, name, age):
        # Instance variables
```

```
        self.name = name
        self.age = age

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)

# Access class and instance variables
print(dog1.species)  # (Class variable)
print(dog1.name)     # (Instance variable)
print(dog2.name)     # (Instance variable)

# Modify instance variables
dog1.name = "Max"
print(dog1.name)     # (Updated instance variable)

# Modify class variable
Dog.species = "Feline"
print(dog1.species)  # (Updated class variable)
print(dog2.species)
```

✅ SQL Scalar Functions

Definition: Return a single value based on input values.
Types: Can be either built-in or user-defined.

Examples:
UPPER('hello') → 'HELLO'
LEN('hello') → 5
ROUND(3.1415, 2) → 3.14

✅ Character Functions (Subset of Scalar Functions)

These are used specifically to manipulate strings/characters.

Examples:
UPPER(), LOWER()
SUBSTRING()
TRIM(), LTRIM(), RTRIM()
CHARINDEX(), REPLACE()

✅ Main Types of SQL Functions

✅ 1. Scalar Functions

A scalar function returns a single value for each row.
It operates on individual data elements.

📌 Examples (Assume a table employee):

id	name	salary
1	Alice	50000
2	Bob	60000
3	Carol	55000

🧪 **Query using Scalar Functions:**

*SELECT name, UPPER(name) AS upper_name, salary, ROUND(salary * 0.1, 2) AS bonus*
FROM employee;

🧾 **Output:**

name	upper_name	salary	bonus
Alice	ALICE	50000	5000.0
Bob	BOB	60000	6000.0
Carol	CAROL	55000	5500.0

Explanation:

UPPER(name) converts names to uppercase.
*ROUND(salary * 0.1, 2) calculates a 10% bonus and rounds to 2 decimals.*

✅ 2. Aggregate Functions

Aggregate functions operate on multiple rows and return a single value (e.g., total, average).

🧪 **Query using Aggregate Functions:**

SELECT COUNT(*) AS total_employees,
AVG(salary) AS average_salary,
MAX(salary) AS highest_salary
FROM employee;

🧾 **Output:**

total_employees	average_salary	highest_salary
3	55000	60000

Explanation:

COUNT(*) counts the number of employees.
AVG(salary) gives the average salary.
MAX(salary) returns the highest salary.

🔑 **Summary Table:**

Type	Function Examples	Returns	Works On
Scalar	`UPPER()`, `ROUND()`, `LEN()`	One value per row	Each row individually
Aggregate	`SUM()`, `AVG()`, `COUNT()`, `MAX()`	One value per group/table	Set of rows

✅ What is a set in Python?

A set is an unordered, mutable collection of unique elements in Python.
Duplicate elements are automatically removed.
It is implemented using a hash table, which makes it very efficient for membership testing (i.e., checking if an element exists).

🔍 Example:

```
my_set = {1, 2, 3, 4, 5}
# Add an element
my_set.add(6)

# Check if an element exists
print(3 in my_set)  # Output: True

# Try adding duplicate
my_set.add(3)  # No effect, set still has unique values only

print(my_set)  # Output: {1, 2, 3, 4, 5, 6}
```

⏱️ Time Complexity in Python set

Operation	Average Time Complexity	Worst Case Time Complexity
`x in set` (search)	**O(1)** (constant time)	O(n) (rare, due to hash collisions)
`add(x)`	O(1)	O(n)
`remove(x)`	O(1)	O(n)

In practice, search in a set is very fast and considered O(1) most of the time due to hashing.

🧠 Why is it fast?

Because the Python set uses a has- table internally. When you search for an element, Python computes its hash, then looks it up directly — like finding a book by its index in a catalog.

Difference between SET and DICT?

Definition:

Feature	`set`	`dict`
Purpose	Collection of **unique values**	Collection of **key-value pairs**
Syntax	`{1, 2, 3}` or `set()`	`{'a': 1, 'b': 2}` or `dict()`

Time Complexity:

Operation	`set`	`dict`
Search	O(1) average case	O(1) average case
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)

Both use **hash tables internally**, which gives them similar performance.

Note:
Both use ***hash tables*** internally, which gives them similar performance.

 # Set
my_set = {1, 2, 3}
my_set.add(4)
print(2 in my_set) # True

 # Dict
my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3
print('b' in my_dict) # True
print(my_dict['b'])   # 2

Beginner-Friendly Expanded Q&A with Examples

Python Programming

Q: What are Python decorators?
A: A decorator is a special function that modifies the behavior of another function **without changing its code**.

Think of it like wrapping a gift – you don't change the gift, but you can add extra features like ribbons.
In Python, decorators are written using the @ syntax.

Example:

```
def greet_decorator(func):
    def wrapper():
        print("Hello!")
        func()
    return wrapper

@greet_decorator
def say_name():
    print("I am Vaibhav")

say_name()
# Output:
# Hello!
# I am Vaibhav
```

Q: Difference between @staticmethod and @classmethod?
A:

- **@staticmethod** → behaves like a normal function inside a class. It doesn't know about the instance (self) or the class (cls).
- **@classmethod** → gets the class (cls) as the first argument, so it can modify class variables.

Example:

```
class Example:
    count = 0

    @staticmethod
    def greet():
        print("Hello!")
```

```
    @classmethod
    def increment_count(cls):
        cls.count += 1

Example.greet()             # Output: Hello!
Example.increment_count()
print(Example.count)        # Output: 1
```

Q: Explain list comprehension and generator expressions.
A:

- **List comprehension**: Creates a list directly in memory.
- **Generator expression**: Creates an iterator that yields values one by one (saves memory).

Example:

```
# List comprehension
nums = [x * x for x in range(5)]
print(nums)  # [0, 1, 4, 9, 16]

# Generator expression
nums_gen = (x * x for x in range(5))
print(list(nums_gen))  # [0, 1, 4, 9, 16]
```

If the list is huge, generators are more memory efficient.

Q: How does Python memory management work?
A: Python keeps track of how many variables reference an object (reference count).
When no variable references it, the object is deleted automatically.
It also has a **garbage collector** to clean up circular references.

Example:

```
import sys
a = [1, 2, 3]
print(sys.getrefcount(a))  # Shows reference count
```

Q: How would you implement a custom iterator and generator class?
A:

- **Iterator class**: Must have __iter__() and __next__() methods.
- **Generator**: Use yield to produce values.

Example (iterator):

```
class Counter:

    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.current < self.limit:
            self.current += 1
            return self.current
        else:
            raise StopIteration

for num in Counter(3):
    print(num)
```

Example (generator):

```
def counter(limit):
    for i in range(1, limit + 1):
        yield i

for num in counter(3):
    print(num)
```

Flask / FastAPI

Q: How do you create an API route?
A:

- **Flask**: Use @app.route(“path/”, methods = [“GET”,”POST”)
- **FastAPI**: Use @app.get(“path/”) or @app.post(“path/”)

Example (Flask):

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

@app.route("/hello")
def hello():
    return "Hello World"
```

Example (FastAPI):

```
from fastapi import FastAPI
app = FastAPI()

@app.get("/hello")
async def hello():
    return {"message": "Hello World"}
```

Q: What is Pydantic in FastAPI?
A: Pydantic is a library for **validating and parsing data** based on Python type hints.
It ensures that incoming request data is in the correct format before your function uses it.

Example:

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int

@app.post("/user")
```

```
async def create_user(user: User):
    return user
```

Q: How to handle request validation & response models in FastAPI?
A:

- **Request validation**: Use Pydantic models in function parameters.
- **Response models**: Use response_model=YourModel in the route decorator to ensure consistent output.

Example:

```
@app.post("/items", response_model=Item)
async def create_item(item: Item):
    return item
```

Q: How to implement JWT authentication?
A:

- Use pyjwt to create tokens with user info.
- Store token in client, verify in every request using middleware/dependency.

Example:

```
import jwt
token = jwt.encode({"user_id": 1}, "secret", algorithm="HS256")
data = jwt.decode(token, "secret", algorithms=["HS256"])
```

Authentication & Authorization

Q: How to implement basic authentication in Flask?
A: Use request.authorization to get the username/password and check them.

Example:

```
from flask import request, Response

@app.route("/secure")
def secure():
    auth = request.authorization
    if auth and auth.username == "admin" and auth.password == "pass":
        return "Welcome"
    return Response("Unauthorized", 401)
```

Q: What is OAuth2?
A: OAuth2 is a system that lets users grant apps permission to access resources without sharing passwords.
Example: "Login with Google".

Q: How to implement RBAC (Role-Based Access Control)?
A: Store user roles in the database, check their role before performing actions.

Example:

```
if current_user.role != "admin":
    raise HTTPException(status_code=403, detail="Access Denied")
```

Q: How to manage refresh tokens securely?
A: Store them in **HTTP-only cookies** or encrypted DB, rotate them after use, and delete them on logout.

API Integration

Q: How to make REST API calls in Python?
A: Use the requests library.

```
import requests
res = requests.get("https://api.example.com/data")
print(res.json())
```

Q: Typical status codes?
A:

- **200 OK** – success
- **201 Created** – resource created
- **400 Bad Request** – invalid data
- **401 Unauthorized** – login required
- **403 Forbidden** – no permission
- **404 Not Found** – resource missing
- **500 Internal Server Error** – server crashed

Q: How to build middleware to log API traffic?
A: Wrap requests and log info like path, method, request body, and response.

Q: How to handle rate-limiting programmatically?
A: Use **Redis** to track request counts per IP, and block if the limit is exceeded.

LLMs & Vector DB

Q: What is prompt & temperature?
A:

- **Prompt**: The input you give the AI.
- **Temperature**: Controls randomness (0 = strict, 1 = creative).

Q: How to call GPT API?
A:

```
import openai
openai.api_key = "your-key"
res = openai.ChatCompletion.create(
    model="gpt-4",
```

```
    messages=[{"role": "user", "content": "Hello"}]
)
```

Q: Function calling/tool use?
A: You define a function schema, give it to the model, and the model returns structured JSON with arguments.

Q: What is a sentence embedding?
A: A list of numbers representing meaning of a sentence, used for similarity search.

Q: What is FAISS?
A: A library for **fast similarity search** among embeddings.
Q: Compare ChatCompletion vs Completion?
A:

- **ChatCompletion**: Multiple messages, roles like "user" and "assistant".
- **Completion**: Single prompt and response.

Q: How to do hybrid search?
A: Combine vector similarity search with metadata filtering.

Q: Cosine vs dot product?
A:

- **Cosine similarity**: Measures angle between vectors (ignores length).
- **Dot product**: Measures projection, sensitive to magnitude.

Question: Rate limiting using middleware or third-party libraries like **slowapi**, **fastapi-limiter**, or custom logic with Redis.

Saved memory full ⓘ

✅ 2. Using `fastapi-limiter` (Redis-based, production-ready)

Install:

bash Copy Edit

```
pip install fastapi-limiter redis
```

Setup:

python Copy Edit

```python
import aioredis
from fastapi import FastAPI, Depends, Request
from fastapi_limiter import FastAPILimiter
from fastapi_limiter.depends import RateLimiter

app = FastAPI()

@app.on_event("startup")
async def startup():
    redis = await aioredis.from_url("redis://localhost", encoding="utf8", decode_responses=True)
    await FastAPILimiter.init(redis)

@app.get("/items/", dependencies=[Depends(RateLimiter(times=5, seconds=60))])
async def get_items():
    return {"message": "You are allowed!"}
```

Q: What is Composite indexing?

Composite indexing (also called multi-column index) means creating an index on two or more columns together in a database table, instead of just one column.

- Composite indexing = indexing multiple columns together to optimize queries that filter or sort on those columns.
- It's best used when you frequently query with conditions on multiple fields.

To create Indexing for more than 1 column

NOTE: ⚠️ Don't over-index — too many indexes slow down INSERT/UPDATE/DELETE.

Ex:

CREATE INDEX idx_lastname_department
ON employees (last_name, department);

Q: How to optimize query operations?

🔹 **Query Optimization Best Practices**

- ✅ **Use Indexes**
  - Single-column index for frequently searched columns.
  - Composite index for queries involving multiple columns.

    - Always check with `EXPLAIN ANALYZE` if indexes are being used.
- ✅ **Select Only Needed Columns**
    - Avoid `SELECT *`, fetch only required fields.
- ✅ **Filter Early**
    - Apply `WHERE` conditions in SQL instead of filtering in application code.
- ✅ **Optimize Joins**
    - Ensure foreign keys are indexed.
    - Prefer `JOIN` over subqueries for performance.
- ✅ **Cache Expensive Queries**
    - Use Redis/Memcached for repeated heavy queries.
    - Especially useful for reports/aggregates.

- ✅ **Paginate Result**
    - Use `LIMIT` + `OFFSET` (or keyset pagination) instead of returning all rows.
- ✅ **Normalize / Denormalize Appropriately**
    - Normalize for fast writes and less redundancy.
    - Denormalize when you need fewer joins for read-heavy queries.
- ✅ **Profile Queries**
    - Use `EXPLAIN` / `EXPLAIN ANALYZE` to find slow queries.
    - Look out for **Seq Scan** (bad) → aim for **Index Scan**.
- ✅ **Batch Inserts/Updates**
    - Insert/update multiple rows at once instead of looping row by row.
    - Use PostgreSQL `COPY` for bulk loads.
- ✅ **Use Connection Pooling**
    - Avoid opening/closing new DB connections per request.
    - Configure pool size in SQLAlchemy / psycopg2.
- ✅ **Avoid N+1 Queries (ORM)**
    - Use `selectinload`, `joinedload` (SQLAlchemy) or joins instead of querying inside loops.

Q: How to apply pooling in fastapi?

Using Async SQLAlchemy (asyncpg)

If you're working async:

python Copy Edit

```python
from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "postgresql+asyncpg://user:password@localhost/mydb"

engine = create_async_engine(
    DATABASE_URL,
    pool_size=10,
    max_overflow=20,
    pool_timeout=30,
    pool_recycle=1800
)

AsyncSessionLocal = sessionmaker(
    bind=engine, class_=AsyncSession, expire_on_commit=False
)

async def get_db():
    async with AsyncSessionLocal() as session:
        yield session
```

Q: What is Containerisation ?
Containerisation is a way of packaging an application **and everything it needs to run** (code, dependencies, libraries, runtime, configs) into a **lightweight, portable container** so it can run consistently across different environments.

Port of frameworks:
flask default port 5000
Django default port 8000
fastapi default port 8000

can we deploy two services in same docker file then what is the advantages and disadvantages and when we can do this and when dont?

What is the cold start? related to serverless

principal behind rate limiter in api

kubernates work flow

difference between PostgreSQL and MySQL?

What is method overriding and method overloading in python?

Method Overloading:
Method Overloading is an example of Compile time polymorphism. In this, more than one method of the same class shares the same method name having different signatures. Method overloading is used to add more to the behavior of methods and there is no need of more than one class for method overloading.
Note: Python does not support method overloading. We may overload the methods but can only use the latest defined method.

Example:

```
def add(datatype, *args):
    # if datatype is int initialize answer as 0
    if datatype =='int':
        answer = 0

    # if datatype is str initialize answer as ''
    if datatype =='str':
        answer =''
    for x in args:
        answer = answer + x
    print(answer)

# Integer
add('int', 5, 6)
# String
add('str', 'Hi ', 'Geeks')
```

Output
11
Explanation: This code demonstrates method overloading in Python using *args. The add() function performs different operations based on the datatype argument. If the datatype is 'int', it sums the integers in args. If the datatype is 'str', it concatenates the strings in args. The function adapts to different input types, simulating overloading.

Method Overriding:
Method overriding is an example of run time polymorphism. In this, the specific implementation of the method that is already provided by the parent class is provided by the child class. It is used to change the behavior of existing methods and there is a need for at least two classes for method overriding. In method overriding, inheritance is always required as it is done between parent class(superclass) and child class(child class) methods. Example:

```
class A:
```

```
    def fun1(self):
        print('feature_1 of class A')
    def fun2(self):
        print('feature_2 of class A')

class B(A):
    # Modified function that already exist in class A
    def fun1(self):
        print('Modified feature_1 of class A by class B')
    def fun3(self):
        print('feature_3 of class B')

# Create instance
obj = B()
# Call the override function
obj.fun1()
```

Output

Modified feature_1 of class A by class B

Explanation: This code demonstrates method overriding in Python. Class B inherits from class A and overrides the fun1() method. When fun1() is called on an instance of B, the overridden method in class B is executed instead of the original method in class A.

Q: Can I use the same Docker volume in different Docker Compose projects?

✅ A:

Yes, you can use the same Docker volume across different Docker Compose projects by defining the volume as external in the second (or any additional) docker-compose.yml.
Make sure the volume already exists (either created by a previous compose or manually via docker volume create volume_name).

Q: How do I share a volume between different Docker Compose files?

✅ **A:**

Create or use an existing volume:

```
docker volume create shared_data
```

In the first compose file:

```
volumes:
  shared_data:
services:
  writer:
    volumes:
      - shared_data:/data
```

In the second compose file, mark the volume as external:

```
volumes:
  shared_data:
    external: true
services:
  reader:
    volumes:

      - shared_data:/data
```

Q: Why use external volumes in multiple Compose projects?

- To persist and share data between independent services.
- To decouple deployments but share the same data layer (logs, configs, datasets).

What are ACID or ACID properties in a database?

- **Atomicity :** All steps in a transaction succeed or none do.
- **Consistency :** Database remains in a valid state before and after the transaction.
- **Isolationc:** Transactions don’t interfere with each other.
- **Durability :** Once committed, data is saved permanently, even after a system crash.