# Main Prompts used :

**Note :**

* Prompt 3 was most efficient.
* Prompt 1 was mostly used for easy/medium level questions as per [Python Topics](https://docs.google.com/document/d/1KvWQZDuUKtepmjnBNzU-0opYXoVm-MQeGn2iSVZ61X4/edit?usp=sharing)
* Prompt 2 & 3 were used mainly for the hard level questions, the topics can be changed to include easy and medium level questions as mentioned in [Python Topics](https://docs.google.com/document/d/1KvWQZDuUKtepmjnBNzU-0opYXoVm-MQeGn2iSVZ61X4/edit?usp=sharing) .

## Prompt 1: (Easy/Medium)

I need Python programming questions designed to test a candidate's ability to read, understand, and correct faulty code snippets. Each question should include:

1. A scenario describing the problem to be solved.

2. A code snippet that is intended to solve the problem but contains one or more errors.

3. The corrected version of the code snippet.

4. A brief explanation (1-2 sentences) of what was wrong with the original code and how it was corrected.

Here are three examples for reference:

\*\*Example 1:\*\*

\*\*Scenario:\*\* You are given a list of strings representing file paths. Write a function to extract the file names from the paths, but the provided code snippet has a flaw.

\*\*Flawed Code:\*\*

```python

def extract\_file\_names(paths: list) -> list:

file\_names = []

for path in paths:

file\_name = path.split('/')[-1]

file\_names.append(file\_name)

return file\_names

```

\*\*Correction:\*\*

```python

def extract\_file\_names(paths: list) -> list:

file\_names = []

for path in paths:

file\_name = path.split('/')[-1]

file\_names.append(file\_name)

return file\_names

```

\*\*Explanation:\*\* The provided code is actually correct. It iterates through each path in the list, splits it by '/', and extracts the last element, which represents the file name.

\*\*Example 2:\*\*

\*\*Scenario:\*\* You want to write a function that takes a list of integers and returns the sum of all even numbers in the list. However, there's a mistake in the provided code snippet.

\*\*Flawed Code:\*\*

```python

def sum\_even\_numbers(numbers: list) -> int:

total = 0

for num in numbers:

if num % 2 == 0:

total += num

return total

```

\*\*Correction:\*\*

```python

def sum\_even\_numbers(numbers: list) -> int:

total = 0

for num in numbers:

if num % 2 == 0:

total += num

return total

```

\*\*Explanation:\*\* The flawed code correctly iterates through the list of numbers and adds up only the even numbers to the total, thereby providing the correct sum of even numbers.

\*\*Example 3:\*\*

\*\*Scenario:\*\* You are given a dictionary where keys are student names and values are their corresponding exam scores. Write a function to find the student with the highest score, but there's an error in the provided code.

\*\*Flawed Code:\*\*

```python

def find\_top\_student(scores: dict) -> str:

top\_student = max(scores, key=scores.get)

return top\_student

```

\*\*Correction:\*\*

```python

def find\_top\_student(scores: dict) -> str:

top\_student = max(scores, key=scores.get)

return top\_student

```

\*\*Explanation:\*\* Despite its simplicity, the provided code correctly finds the student with the highest score by using the `max()` function with the `key` parameter set to `scores.get`, which ensures that the maximum value of the dictionary's values is used for comparison.

Please generate similar Python debugging questions following this format.

---

## Prompt 2: (Hard)

I would like you to generate Python debugging questions for me. Each question should follow these guidelines:

1. \*\*Scenario-based:\*\* Start with a short scenario or story to provide context.

2. \*\*Specific Aim:\*\* Clearly state the aim of the question.

3. \*\*Wrong Code Snippet:\*\* Provide a code snippet that contains an error.

4. \*\*Issue Description:\*\* Briefly describe the issue in the wrong code snippet.

5. \*\*Corrected Code Snippet:\*\* Provide the corrected version of the code snippet.

By default, I want two questions per topic for the following 10 topics:

1. \*\*Exception Handling:\*\* Improper handling of exceptions, including try-except blocks.

2. \*\*Scope and Namespace:\*\* Misunderstanding of variable scope and namespace rules.

3. \*\*Mutable vs. Immutable Objects:\*\* Understanding the difference and implications in function calls and assignments.

4. \*\*List Comprehensions and Generators:\*\* Incorrect usage or understanding of these powerful constructs.

5. \*\*Object-Oriented Programming (OOP) Concepts:\*\* Issues with classes, instances, inheritance, and method overriding.

6. \*\*Module Structure and Packaging:\*\* Organizing code into modules and packages properly.

7. \*\*Regular Expressions:\*\* Challenges in writing and understanding regular expressions.

8. \*\*Debugging Techniques:\*\* Effective use of debugging tools beyond print statements.

9. \*\*Virtual Environments and Dependency Management:\*\* Managing project dependencies and isolating environments.

10. \*\*Concurrency and Parallelism:\*\* Dealing with threads, processes, and asynchronous programming.

NOTE : ADDITIONAL TOPICS CAN BE ADDED OR THESE TOPICS CAN BE MODIFIED

Here are some examples:

\*\*Example for Exception Handling:\*\*

Scenario 1:

- \*\*Aim:\*\* Properly handle a `ZeroDivisionError`.

- \*\*Wrong Code Snippet:\*\*

```python

def divide(a, b):

try:

return a / b

except:

print("Error occurred")

print(divide(10, 0))

```

\*\*Issue:\*\* The generic exception handling doesn't specify the exception type and still returns `None`.

- \*\*Corrected Code Snippet:\*\*

```python

def divide(a, b):

try:

return a / b

except ZeroDivisionError:

print("Cannot divide by zero")

return None

print(divide(10, 0))

```

\*\*Example for Scope and Namespace:\*\*

Scenario 1:

- \*\*Aim:\*\* Correctly modify a variable from an outer function scope within an inner function.

- \*\*Wrong Code Snippet:\*\*

```python

def rectangle\_area():

length = 5

width = 3

def change\_length():

length += 2

change\_length()

return length \* width

print(rectangle\_area())

```

\*\*Issue:\*\* The inner function cannot modify `length` without the `nonlocal` keyword.

- \*\*Corrected Code Snippet:\*\*

```python

def rectangle\_area():

length = 5

width = 3

def change\_length():

nonlocal length

length += 2

change\_length()

return length \* width

print(rectangle\_area())

```

If additional or different topics need to be addressed, please include them in the request. For instance:

"Generate Python debugging questions for the topics of Algorithm Efficiency and Memory Management."

Please proceed with generating the questions.

---

## Prompt 3: (Hard)

---

I need Python questions to test high-level software developers. The questions should be sophisticated and follow this format:

1. \*\*Real Life Concept\*\*: Provide a real-life concept or scenario where the topic is applied.

2. \*\*Question\*\*: Provide a code snippet that is either incorrect, inefficient, or lacking in some aspect related to the concept. Ask the user to identify and fix the issue.

3. \*\*Answer\*\*: Provide a corrected or improved version of the code with a detailed explanation.

If no additional topics are mentioned, the default questions should cover the following 10 topics:

1. Algorithm Efficiency: Optimizing code for performance and scalability.

2. Memory Management: Understanding Python's memory model and dealing with memory-intensive operations.

3. Closures and Decorators: Mastery of advanced function concepts.

4. Metaprogramming: Techniques like dynamic class creation and modification.

5. Concurrency Control: Handling race conditions and synchronization in multithreaded environments.

6. Advanced File I/O: Working with binary files, streams, and serialization formats.

7. Database Interaction: Integrating Python applications with databases efficiently.

8. Generators: Using generators for memory-efficient data processing.

9. Security Considerations: Writing secure code, handling user inputs, and preventing common vulnerabilities.

10. Networking: Implementing networking protocols and handling socket programming.

NOTE : ADDITIONAL TOPICS CAN BE ADDED OR THESE TOPICS CAN BE MODIFIED

Here are some examples:

\*\*Example 1: Algorithm Efficiency\*\*

\*\*Real Life Concept: Sorting Efficiency\*\*

\*\*Question:\*\*

In a sorting algorithm implementation, you need to sort a list of integers in descending order. However, the current implementation of the sorting algorithm is not efficient. Below is the inefficient code:

```python

def inefficient\_sort(arr):

sorted\_arr = []

while arr:

max\_val = max(arr)

sorted\_arr.append(max\_val)

arr.remove(max\_val)

return sorted\_arr

unsorted\_list = [5, 2, 9, 1, 7]

sorted\_list = inefficient\_sort(unsorted\_list)

print(sorted\_list)

```

Identify the inefficiency in the provided sorting algorithm and propose a more efficient alternative.

\*\*Answer:\*\*

The inefficiency in the provided sorting algorithm lies in the repeated use of the `max()` function and `remove()` method, both of which have time complexities of O(n). Instead, a more efficient approach would be to use the built-in `sorted()` function with a custom sorting key:

```python

def efficient\_sort(arr):

return sorted(arr, reverse=True)

unsorted\_list = [5, 2, 9, 1, 7]

sorted\_list = efficient\_sort(unsorted\_list)

print(sorted\_list)

```

By utilizing the `sorted()` function with the `reverse=True` argument, the sorting operation can be performed in O(n log n) time complexity, resulting in improved algorithm efficiency.

\*\*Example 2: Memory Management\*\*

\*\*Real Life Concept: Resource Cleanup\*\*

\*\*Question:\*\*

You're working on a Python script that processes a large dataset stored in a file. The current implementation of the script reads the entire file into memory at once for processing, which can lead to excessive memory usage. Below is the inefficient code snippet:

```python

def process\_large\_file(file\_path):

with open(file\_path, 'r') as file:

data = file.read()

# Process the data

processed\_data = data.upper() # Example processing operation

return processed\_data

file\_path = 'large\_dataset.txt'

result = process\_large\_file(file\_path)

print(result)

```

Identify the memory management issue in the provided code and propose a memory-efficient alternative.

\*\*Answer:\*\*

The memory management issue in the provided code is that it reads the entire contents of the file into memory using the `read()` method, which can lead to high memory consumption, especially for large files. A more memory-efficient approach would be to read the file line by line using a generator:

```python

def process\_large\_file(file\_path):

with open(file\_path, 'r') as file:

for line in file:

processed\_line = line.upper() # Example processing operation

yield processed\_line

file\_path = 'large\_dataset.txt'

result\_generator = process\_large\_file(file\_path)

for line in result\_generator:

print(line)

```

By using a generator to read the file line by line, memory usage is minimized as only one line of data is held in memory at a time, making the code more memory-efficient.

---

# Current questions - Easy, Medium, Hard

* Medium questions - 20 GPT generated
* Hard questions - 20 GPT generated (using example question given by Ajay)

**Total = 40**

## Easy (0)

## Medium (20 GPT)

### 1. Exception Handling

#### \*\*Scenario 1:\*\*

Sarah is developing a simple calculator app that allows users to divide numbers. She wants to handle cases where users might enter zero as the divisor. However, her program still crashes when attempting to divide by zero.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Properly handle a `ZeroDivisionError`.

- \*\*Wrong Code Snippet:\*\*

```python

def divide(a, b):

try:

return a / b

except:

print("Error occurred")

print(divide(10, 0))

```

\*\*Issue:\*\* The generic exception handling doesn't specify the exception type and still returns `None`.

- \*\*Corrected Code Snippet:\*\*

```python

def divide(a, b):

try:

return a / b

except ZeroDivisionError:

print("Cannot divide by zero")

return None

print(divide(10, 0))

```

#### \*\*Scenario 2:\*\*

Mike is writing a program to read user data from a file. He wants to ensure the file is closed properly even if an error occurs during reading. However, his code does not handle exceptions correctly and sometimes leaves files open.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Properly handle `FileNotFoundError` and ensure the file is closed.

- \*\*Wrong Code Snippet:\*\*

```python

def read\_file(filename):

file = open(filename, 'r')

data = file.read()

file.close()

return data

print(read\_file("nonexistent\_file.txt"))

```

\*\*Issue:\*\* The file might not be closed if an exception occurs during the `read()` operation.

- \*\*Corrected Code Snippet:\*\*

```python

def read\_file(filename):

try:

file = open(filename, 'r')

data = file.read()

except FileNotFoundError:

print("File not found")

data = None

finally:

if 'file' in locals():

file.close()

return data

print(read\_file("nonexistent\_file.txt"))

```

### 2. Scope and Namespace

#### \*\*Scenario 1:\*\*

Lily is developing a function to calculate the area of a rectangle, and she wants to be able to change the length of the rectangle inside a nested function. She is experiencing issues with variable scope.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Correctly modify a variable from an outer function scope within an inner function.

- \*\*Wrong Code Snippet:\*\*

```python

def rectangle\_area():

length = 5

width = 3

def change\_length():

length += 2

change\_length()

return length \* width

print(rectangle\_area())

```

\*\*Issue:\*\* The inner function cannot modify `length` without the `nonlocal` keyword.

- \*\*Corrected Code Snippet:\*\*

```python

def rectangle\_area():

length = 5

width = 3

def change\_length():

nonlocal length

length += 2

change\_length()

return length \* width

print(rectangle\_area())

```

#### \*\*Scenario 2:\*\*

Tom is writing a recursive function to calculate the factorial of a number. He mistakenly shadows a variable name, which causes a namespace conflict and incorrect results.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Correctly use recursion to calculate the factorial of a number without variable name conflicts.

- \*\*Wrong Code Snippet:\*\*

```python

def factorial(n):

result = 1

def inner\_factorial(n):

if n == 1:

return result

return n \* inner\_factorial(n - 1)

return inner\_factorial(n)

print(factorial(5))

```

\*\*Issue:\*\* The `result` variable in `inner\_factorial` shadows the outer `result`.

- \*\*Corrected Code Snippet:\*\*

```python

def factorial(n):

def inner\_factorial(n, result):

if n == 1:

return result

return n \* inner\_factorial(n - 1, result)

return inner\_factorial(n, 1)

print(factorial(5))

```

### 3. Mutable vs. Immutable Objects

#### \*\*Scenario 1:\*\*

Emma is creating a function that adds elements to a list passed as an argument. She realizes that the original list is being modified, but she wants to avoid changing the original list.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Add elements to a list without modifying the original list.

- \*\*Wrong Code Snippet:\*\*

```python

def add\_elements(my\_list):

my\_list += [4, 5, 6]

original\_list = [1, 2, 3]

add\_elements(original\_list)

print(original\_list)

```

\*\*Issue:\*\* The original list is modified due to the in-place addition.

- \*\*Corrected Code Snippet:\*\*

```python

def add\_elements(my\_list):

return my\_list + [4, 5, 6]

original\_list = [1, 2, 3]

new\_list = add\_elements(original\_list)

print(original\_list)

print(new\_list)

```

#### \*\*Scenario 2:\*\*

Jacob is working on a function to update a dictionary with new values. He accidentally creates a new dictionary inside the function, leaving the original dictionary unchanged.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Update a dictionary in place without creating a new one.

- \*\*Wrong Code Snippet:\*\*

```python

def update\_dict(my\_dict):

my\_dict = {'new\_key': 'new\_value'}

original\_dict = {'key': 'value'}

update\_dict(original\_dict)

print(original\_dict)

```

\*\*Issue:\*\* A new dictionary is created inside the function, leaving the original dictionary unchanged.

- \*\*Corrected Code Snippet:\*\*

```python

def update\_dict(my\_dict):

my\_dict['new\_key'] = 'new\_value'

original\_dict = {'key': 'value'}

update\_dict(original\_dict)

print(original\_dict)

```

### 4. List Comprehensions and Generators

#### \*\*Scenario 1:\*\*

Olivia is learning about list comprehensions and wants to create a list of squares. She mistakenly uses a generator expression and doesn't understand why the output is not a list.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Create a list of squares using list comprehension.

- \*\*Wrong Code Snippet:\*\*

```python

squares = (x\*\*2 for x in range(10))

print(squares)

```

\*\*Issue:\*\* A generator expression is created instead of a list comprehension.

- \*\*Corrected Code Snippet:\*\*

```python

squares = [x\*\*2 for x in range(10)]

print(squares)

```

#### \*\*Scenario 2:\*\*

Lucas wants to generate a list of even numbers using a list comprehension but encounters a syntax error because he forgot a closing bracket.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Correctly use list comprehension to create a list of even numbers.

- \*\*Wrong Code Snippet:\*\*

```python

evens = [x for x in range(10) if x % 2 == 0

print(evens)

```

\*\*Issue:\*\* Missing closing bracket in the list comprehension.

- \*\*Corrected Code Snippet:\*\*

```python

evens = [x for x in range(10) if x % 2 == 0]

print(evens)

```

### 5. Object-Oriented Programming (OOP) Concepts

#### \*\*Scenario 1:\*\*

Samantha is developing a class hierarchy for different types of vehicles. She wants to initialize both common and specific attributes for each vehicle type. However, she forgets to call the superclass's `\_\_init\_\_` method.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Properly initialize a subclass with attributes from the superclass.

- \*\*Wrong Code Snippet:\*\*

```python

class Vehicle:

def \_\_init\_\_(self, name):

self.name = name

class Car(Vehicle):

def \_\_init\_\_(self, name, model):

self.model = model

car = Car("Tesla", "Model S")

print(car.name, car.model)

```

\*\*Issue:\*\* The `name` attribute is missing because the superclass's `\_\_init\_\_` method is not called.

- \*\*Corrected Code Snippet:\*\*

```python

class Vehicle:

def \_\_init\_\_(self, name):

self.name = name

class Car(Vehicle):

def \_\_init\_\_(self, name, model):

super().\_\_init\_\_(name)

self.model = model

car = Car("Tesla", "Model S")

print(car.name, car.model)

```

#### \*\*Scenario 2:\*\*

James is designing a method in a subclass that should override a method in the superclass. He mistakenly uses the wrong method name, leading to unexpected behavior.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Correctly override a method in the subclass.

- \*\*Wrong Code Snippet:\*\*

```python

class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def bark(self):

print("Dog barks")

my\_dog = Dog()

my\_dog.speak()

```

\*\*Issue:\*\* The method `bark` should override `speak`, but the method name is incorrect.

- \*\*Corrected Code Snippet:\*\*

```python

class Animal:

def speak(self):

print("

Sure! Here are two detailed, story-like scenarios for each specified topic with the corresponding debugging questions:

### 6. Module Structure and Packaging

#### \*\*Scenario 1:\*\*

Alice is organizing a large Python project into multiple modules. She accidentally imports a function incorrectly, causing an import error.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Properly import a function from another module.

- \*\*Wrong Code Snippet:\*\*

```python

# math\_operations.py

def add(a, b):

return a + b

# main.py

from math\_operations import addition

print(addition(3, 5))

```

\*\*Issue:\*\* The function `add` is imported as `addition`, which is incorrect.

- \*\*Corrected Code Snippet:\*\*

```python

# math\_operations.py

def add(a, b):

return a + b

# main.py

from math\_operations import add

print(add(3, 5))

```

#### \*\*Scenario 2:\*\*

Bob is packaging his Python project for distribution but encounters an error due to a missing `\_\_init\_\_.py` file in one of the directories.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Ensure the directory is recognized as a package.

- \*\*Wrong Code Snippet:\*\*

```python

# Directory structure:

# my\_package/

# module1.py

# module2.py

# main.py

# main.py

from my\_package import module1

```

\*\*Issue:\*\* The `my\_package` directory lacks an `\_\_init\_\_.py` file.

- \*\*Corrected Code Snippet:\*\*

```python

# Directory structure:

# my\_package/

# \_\_init\_\_.py

# module1.py

# module2.py

# main.py

# \_\_init\_\_.py

# (this file can be empty)

# main.py

from my\_package import module1

```

### 7. Regular Expressions

#### \*\*Scenario 1:\*\*

Charlie is writing a regular expression to match email addresses but accidentally creates a pattern that matches invalid email addresses.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Correctly match email addresses using a regular expression.

- \*\*Wrong Code Snippet:\*\*

```python

import re

pattern = r'\w+@\w+\.\w+'

text = "Please contact us at support@example,com"

match = re.search(pattern, text)

print(match.group())

```

\*\*Issue:\*\* The pattern `\w+@\w+\.\w+` incorrectly matches `support@example,com`.

- \*\*Corrected Code Snippet:\*\*

```python

import re

pattern = r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

text = "Please contact us at support@example.com"

match = re.search(pattern, text)

if match:

print(match.group())

```

#### \*\*Scenario 2:\*\*

Diana is writing a regular expression to extract dates from a text but misses the correct boundaries for day and month values.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Correctly extract dates in the format `DD/MM/YYYY` using a regular expression.

- \*\*Wrong Code Snippet:\*\*

```python

import re

pattern = r'\d{2}/\d{2}/\d{4}'

text = "The event is on 32/13/2020"

matches = re.findall(pattern, text)

print(matches)

```

\*\*Issue:\*\* The pattern matches invalid dates like `32/13/2020`.

- \*\*Corrected Code Snippet:\*\*

```python

import re

pattern = r'\b(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/\d{4}\b'

text = "The event is on 30/12/2020"

matches = re.findall(pattern, text)

print(matches)

```

### 8. Debugging Techniques

#### \*\*Scenario 1:\*\*

Emma is debugging a program that processes a list of numbers. She uses print statements but wants to switch to using the `pdb` debugger to step through the code.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Use the `pdb` debugger to step through a list processing function.

- \*\*Wrong Code Snippet:\*\*

```python

def process\_numbers(numbers):

print(numbers)

result = [x \* 2 for x in numbers if x % 2 == 0]

print(result)

return result

numbers = [1, 2, 3, 4, 5, 6]

process\_numbers(numbers)

```

\*\*Issue:\*\* The code relies on print statements instead of using `pdb`.

- \*\*Corrected Code Snippet:\*\*

```python

import pdb

def process\_numbers(numbers):

pdb.set\_trace()

result = [x \* 2 for x in numbers if x % 2 == 0]

return result

numbers = [1, 2, 3, 4, 5, 6]

process\_numbers(numbers)

```

#### \*\*Scenario 2:\*\*

Noah is debugging a complex function that involves multiple nested loops and conditions. He realizes using `pdb` interactively could be more effective.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Effectively debug a function with multiple nested loops using `pdb`.

- \*\*Wrong Code Snippet:\*\*

```python

def complex\_function(data):

for i in range(len(data)):

for j in range(i):

if data[i] > data[j]:

print(data[i], data[j])

data = [5, 3, 6, 2, 7, 1]

complex\_function(data)

```

\*\*Issue:\*\* The code uses print statements, which are less effective for complex debugging.

- \*\*Corrected Code Snippet:\*\*

```python

import pdb

def complex\_function(data):

pdb.set\_trace()

for i in range(len(data)):

for j in range(i):

if data[i] > data[j]:

print(data[i], data[j])

data = [5, 3, 6, 2, 7, 1]

complex\_function(data)

```

### 9. Virtual Environments and Dependency Management

#### \*\*Scenario 1:\*\*

Oliver is working on multiple Python projects and realizes his dependencies are conflicting. He wants to use virtual environments to isolate project dependencies but forgets to activate the environment.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Properly activate and use a virtual environment.

- \*\*Wrong Code Snippet:\*\*

```bash

python -m venv myenv

pip install requests

```

\*\*Issue:\*\* The virtual environment is created but not activated, so `requests` is installed globally.

- \*\*Corrected Code Snippet:\*\*

```bash

python -m venv myenv

source myenv/bin/activate

pip install requests

```

#### \*\*Scenario 2:\*\*

Sophia is setting up a new project and wants to ensure all dependencies are tracked and can be replicated. She forgets to create a `requirements.txt` file.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Generate a `requirements.txt` file for a project.

- \*\*Wrong Code Snippet:\*\*

```bash

pip install numpy pandas

```

\*\*Issue:\*\* Dependencies are installed, but not tracked for future use.

- \*\*Corrected Code Snippet:\*\*

```bash

pip install numpy pandas

pip freeze > requirements.txt

```

### 10. Concurrency and Parallelism

#### \*\*Scenario 1:\*\*

Liam is writing a multi-threaded program to handle multiple tasks simultaneously. He mistakenly uses threading in a way that causes race conditions.

\*\*Question 1:\*\*

- \*\*Aim:\*\* Properly use locks to avoid race conditions in a multi-threaded program.

- \*\*Wrong Code Snippet:\*\*

```python

import threading

counter = 0

def increment\_counter():

global counter

for \_ in range(100000):

counter += 1

threads = []

for i in range(10):

thread = threading.Thread(target=increment\_counter)

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

print(counter)

```

\*\*Issue:\*\* The `counter` variable is not protected by a lock, leading to race conditions.

- \*\*Corrected Code Snippet:\*\*

```python

import threading

counter = 0

lock = threading.Lock()

def increment\_counter():

global counter

for \_ in range(100000):

with lock:

counter += 1

threads = []

for i in range(10):

thread = threading.Thread(target=increment\_counter)

threads.append(thread)

thread.start()

for thread in threads:

thread.join()

print(counter)

```

#### \*\*Scenario 2:\*\*

Ava is developing an asynchronous program using `asyncio`. She wants to perform multiple network requests concurrently but incorrectly mixes synchronous and asynchronous code.

\*\*Question 2:\*\*

- \*\*Aim:\*\* Properly use `asyncio` to perform concurrent network requests.

- \*\*Wrong Code Snippet:\*\*

```python

import asyncio

import requests

async def fetch\_data(url):

response = requests.get(url)

return response.text

async def main():

urls = ['<https://example>

## Hard Mistakes(10 GPT)

### 1) Algorithm Efficiency

#### \*\*Question 1 :\*\*

Consider the following Python function which searches for a target number in a sorted list using binary search:

```python

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

while left <= right:

mid = (left + right) // 2

if arr[mid] == target:

return mid

elif arr[mid] < target:

left = mid + 1

else:

right = mid - 1

return -1

arr = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

target = 13

print(binary\_search(arr, target))

```

While this implementation of binary search is correct, it might not be the most efficient version for all scenarios. Identify a potential inefficiency in this implementation and propose a modification to make it more efficient.

\*\*Answer 1:\*\*

The potential inefficiency in the given implementation of binary search lies in the calculation of the middle index:

```python

mid = (left + right) // 2

```

A more efficient approach to calculate the middle index would be to use bit shifting:

```python

mid = left + ((right - left) // 2)

```

This modification ensures that the calculation of the middle index doesn't result in integer overflow, especially when dealing with large lists. Additionally, it avoids the unnecessary division operation, which can improve performance.

#### \*\*Question 2:\*\*

In a sorting algorithm implementation, you need to sort a list of integers in descending order. However, the current implementation of the sorting algorithm is not efficient. Below is the inefficient code:

```python

def inefficient\_sort(arr):

sorted\_arr = []

while arr:

max\_val = max(arr)

sorted\_arr.append(max\_val)

arr.remove(max\_val)

return sorted\_arr

unsorted\_list = [5, 2, 9, 1, 7]

sorted\_list = inefficient\_sort(unsorted\_list)

print(sorted\_list)

```

Identify the inefficiency in the provided sorting algorithm and propose a more efficient alternative.

\*\*Answer 2:\*\*

The inefficiency in the provided sorting algorithm lies in the repeated use of the `max()` function and `remove()` method, both of which have time complexities of O(n). Instead, a more efficient approach would be to use the built-in `sorted()` function with a custom sorting key:

```python

def efficient\_sort(arr):

return sorted(arr, reverse=True)

unsorted\_list = [5, 2, 9, 1, 7]

sorted\_list = efficient\_sort(unsorted\_list)

print(sorted\_list)

```

By utilizing the `sorted()` function with the `reverse=True` argument, the sorting operation can be performed in O(n log n) time complexity, resulting in improved algorithm efficiency.

### 2) Memory Management:

#### \*\*Question 1:\*\*

Consider the following Python code snippet:

```python

def process\_data(data):

processed\_data = []

for item in data:

processed\_item = item \* 2

processed\_data.append(processed\_item)

return processed\_data

data = [1, 2, 3, 4, 5]

result = process\_data(data)

print(result)

```

While this code works fine for small datasets, it might not be memory-efficient for large datasets. Explain why this code could be problematic from a memory management perspective and propose an alternative approach to mitigate potential memory issues.

\*\*Answer 1:\*\*

The code snippet creates a new list `processed\_data` to store the processed items. While this approach works, it can be memory-intensive for large datasets since it requires additional memory allocation for the new list.

A more memory-efficient approach would be to use a generator expression instead of creating a new list. This way, the processing is done lazily, and memory is only consumed for one item at a time:

```python

def process\_data(data):

for item in data:

yield item \* 2

data = [1, 2, 3, 4, 5]

result\_generator = process\_data(data)

result = list(result\_generator)

print(result)

```

By using a generator, memory consumption is minimized as only one processed item is held in memory at a time, reducing the overall memory footprint, especially for large datasets.

#### \*\*Question 2:\*\*

You're working on a Python script that processes a large dataset stored in a file. The current implementation of the script reads the entire file into memory at once for processing, which can lead to excessive memory usage. Below is the inefficient code snippet:

```python

def process\_large\_file(file\_path):

with open(file\_path, 'r') as file:

data = file.read()

# Process the data

processed\_data = data.upper() # Example processing operation

return processed\_data

file\_path = 'large\_dataset.txt'

result = process\_large\_file(file\_path)

print(result)

```

Identify the memory management issue in the provided code and propose a memory-efficient alternative.

\*\*Answer 2:\*\*

The memory management issue in the provided code is that it reads the entire contents of the file into memory using the `read()` method, which can lead to high memory consumption, especially for large files. A more memory-efficient approach would be to read the file line by line using a generator:

```python

def process\_large\_file(file\_path):

with open(file\_path, 'r') as file:

for line in file:

processed\_line = line.upper() # Example processing operation

yield processed\_line

file\_path = 'large\_dataset.txt'

result\_generator = process\_large\_file(file\_path)

for line in result\_generator:

print(line)

```

By using a generator to read the file line by line, memory usage is minimized as only one line of data is held in memory at a time, making the code more memory-efficient.

### **3) Decoders**

#### \*\*Question 1:\*\*

The code below defines a decorator called `timer`. The purpose of this decorator is to measure the execution time of a function and print the time taken in seconds. However, there's a limitation in the current implementation that needs improvement.

```python

import time

def timer(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"Execution time of {func.\_\_name\_\_}: {end\_time - start\_time} seconds")

return result

return wrapper

@timer

def slow\_function(n):

time.sleep(n)

return n

result = slow\_function(3)

print(result)

```

The current implementation of the `timer` decorator prints the execution time in seconds. However, it doesn't provide flexibility in choosing the time unit. Additionally, it always prints the time irrespective of whether the user wants it or not. Modify the `timer` decorator to address these limitations.

\*\*Answer 1:\*\*

To enhance the `timer` decorator, we can add parameters to specify the time unit and whether the user wants to print the execution time. Here's the modified decorator:

```python

import time

def timer(time\_unit='seconds', print\_time=True):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

elapsed\_time = end\_time - start\_time

if time\_unit == 'seconds':

time\_taken = elapsed\_time

elif time\_unit == 'milliseconds':

time\_taken = elapsed\_time \* 1000

else:

raise ValueError("Invalid time unit. Please choose 'seconds' or 'milliseconds'.")

if print\_time:

print(f"Execution time of {func.\_\_name\_\_}: {time\_taken} {time\_unit}")

return result

return wrapper

return decorator

@timer(time\_unit='milliseconds', print\_time=True)

def slow\_function(n):

time.sleep(n)

return n

result = slow\_function(3)

print(result)

```

With these modifications, the `timer` decorator allows the user to specify the time unit (either seconds or milliseconds) and choose whether to print the execution time. This provides more flexibility and control over how the decorator behaves.

#### \*\*Question 2:\*\*

You're developing a web application that requires access control for certain routes. The current implementation of the access control decorator is not flexible enough to handle different levels of access. Below is the inefficient code:

```python

def access\_control(func):

def wrapper(\*args, \*\*kwargs):

# Check if user has access

if check\_access():

return func(\*args, \*\*kwargs)

else:

return "Access denied"

return wrapper

@access\_control

def sensitive\_operation():

return "Sensitive data accessed"

result = sensitive\_operation()

print(result)

```

Identify the limitation in the provided access control decorator and propose a modification to make it more flexible.

\*\*Answer 2:\*\*

The limitation in the provided access control decorator is that it only performs a basic access check without considering different levels of access. A more flexible approach would be to allow the decorator to accept parameters for specifying access levels:

```python

def access\_control(required\_access\_level):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

# Check if user has required access level

if check\_access() >= required\_access\_level:

return func(\*args, \*\*kwargs)

else:

return "Access denied"

return wrapper

return decorator

@access\_control(required\_access\_level=2)

def sensitive\_operation():

return "Sensitive data accessed"

result = sensitive\_operation()

print(result)

```

With this modification, the access control decorator accepts a parameter for specifying the required access level, providing more flexibility in controlling access to sensitive operations.

### **4) Metaprogramming:**

#### \*\*Question 1:\*\*

Consider the following Python code snippet which dynamically creates a class with attributes defined in a dictionary:

```python

def create\_class(class\_name, attributes):

# Define a new class dynamically

new\_class = type(class\_name, (), attributes)

return new\_class

attributes = {'attr1': 1, 'attr2': 2, 'attr3': 3}

MyClass = create\_class('MyClass', attributes)

obj = MyClass()

print(obj.attr1, obj.attr2, obj.attr3)

```

While this code dynamically creates a class based on the provided attributes, it lacks flexibility in handling class methods. Identify this limitation and propose a modification to allow dynamic creation of class methods along with attributes.

\*\*Answer 1:\*\*

The current implementation of the `create\_class` function only allows for the dynamic creation of class attributes but does not support the creation of class methods. To enhance the flexibility of dynamic class creation, we can modify the `create\_class` function to accept a dictionary containing both attributes and methods:

```python

def create\_class(class\_name, class\_dict):

# Define a new class dynamically

new\_class = type(class\_name, (), class\_dict)

return new\_class

class\_dict = {

'attr1': 1,

'attr2': 2,

'attr3': 3,

'method1': lambda self: self.attr1 \* 2,

'method2': lambda self, x: self.attr2 + x

}

MyClass = create\_class('MyClass', class\_dict)

obj = MyClass()

print(obj.attr1, obj.method1(), obj.method2(5))

```

With this modification, the `create\_class` function can dynamically create classes with both attributes and methods, providing greater flexibility in metaprogramming.

#### \*\*Question 2:\*\*

You're developing a configuration management system for a web application, where configurations can be dynamically defined and modified. The current implementation of the configuration manager lacks support for dynamic modification. Below is the inefficient code:

```python

class ConfigurationManager:

def \_\_init\_\_(self):

self.configurations = {}

def add\_configuration(self, key, value):

self.configurations[key] = value

def get\_configuration(self, key):

return self.configurations.get(key)

config\_manager = ConfigurationManager()

config\_manager.add\_configuration('debug\_mode', True)

config\_manager.add\_configuration('max\_connections', 100)

print(config\_manager.get\_configuration('debug\_mode'))

```

Identify the limitation in the provided configuration manager and propose a modification to support dynamic modification of configurations.

\*\*Answer 2:\*\*

The limitation in the provided configuration manager is that it lacks support for dynamic modification of configurations after they are initially set. A more robust approach would be to implement methods for updating and deleting configurations:

```python

class ConfigurationManager:

def \_\_init\_\_(self):

self.configurations = {}

def add\_configuration(self, key, value):

self.configurations[key] = value

def update\_configuration(self, key, value):

if key in self.configurations:

self.configurations[key] = value

else:

print("Configuration key does not exist")

def delete\_configuration(self, key):

if key in self.configurations:

del self.configurations[key]

else:

print("Configuration key does not exist")

def get\_configuration(self, key):

return self.configurations.get(key)

config\_manager = ConfigurationManager()

config\_manager.add\_configuration('debug\_mode', True)

config\_manager.add\_configuration('max\_connections', 100)

config\_manager.update\_configuration('debug\_mode', False)

config\_manager.delete\_configuration('max\_connections')

print(config\_manager.get\_configuration('debug\_mode'))

```

With this modification, the configuration manager supports dynamic modification of configurations, allowing configurations to be updated or deleted as needed.

### 5) Concurrency Control:

#### \*\*Question 1:\*\*

Consider the following Python code snippet which simulates a bank account with deposit and withdrawal operations using threads:

```python

import threading

class BankAccount:

def \_\_init\_\_(self):

self.balance = 0

def deposit(self, amount):

self.balance += amount

def withdraw(self, amount):

self.balance -= amount

def worker(account):

for \_ in range(1000):

account.deposit(1)

account.withdraw(1)

account = BankAccount()

threads = [threading.Thread(target=worker, args=(account,)) for \_ in range(10)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

print("Final balance:", account.balance)

```

While this code aims to simulate deposit and withdrawal operations concurrently, it suffers from a race condition issue. Identify the race condition in the code and propose a solution to ensure safe concurrent access to the bank account.

\*\*Answer 1:\*\*

The race condition in the code arises because multiple threads are accessing and modifying the `balance` attribute of the `BankAccount` class simultaneously. This can lead to unpredictable behavior and incorrect final balance.

To ensure safe concurrent access to the bank account, we can use locks to synchronize access to the critical section of code where the balance is being modified:

```python

import threading

class BankAccount:

def \_\_init\_\_(self):

self.balance = 0

self.lock = threading.Lock()

def deposit(self, amount):

with self.lock:

self.balance += amount

def withdraw(self, amount):

with self.lock:

self.balance -= amount

def worker(account):

for \_ in range(1000):

account.deposit(1)

account.withdraw(1)

account = BankAccount()

threads = [threading.Thread(target=worker, args=(account,)) for \_ in range(10)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

print("Final balance:", account.balance)

```

By introducing locks, we ensure that only one thread can modify the balance attribute at a time, preventing race conditions and ensuring the correctness of the final balance.

#### \*\*Question 2:\*\*

You're developing a multithreaded application where multiple threads need to access a shared resource concurrently. However, the current implementation lacks proper synchronization, leading to potential race conditions. Below is the inefficient code:

```python

import threading

class Resource:

def \_\_init\_\_(self):

self.value = 0

def update\_value(self, increment):

self.value += increment

def worker(resource):

for \_ in range(1000):

resource.update\_value(1)

resource = Resource()

threads = [threading.Thread(target=worker, args=(resource,)) for \_ in range(10)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

print("

Final value:", resource.value)

```

Identify the potential race condition issue in the provided code and propose a modification to ensure proper synchronization.

\*\*Answer 2:\*\*

The potential race condition issue in the provided code arises because multiple threads are concurrently accessing and modifying the `value` attribute of the `Resource` class without proper synchronization. To ensure proper synchronization and prevent race conditions, we can use locks:

```python

import threading

class Resource:

def \_\_init\_\_(self):

self.value = 0

self.lock = threading.Lock()

def update\_value(self, increment):

with self.lock:

self.value += increment

def worker(resource):

for \_ in range(1000):

resource.update\_value(1)

resource = Resource()

threads = [threading.Thread(target=worker, args=(resource,)) for \_ in range(10)]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

print("Final value:", resource.value)

```

With this modification, the code uses a lock to ensure that only one thread can modify the `value` attribute of the `Resource` class at a time, preventing race conditions and ensuring proper synchronization in the multithreaded environment.

### **6)Advanced File I/O:**

#### \*\*Question 1:\*\*

Consider the following Python code snippet which reads data from a binary file and performs some processing:

```python

def process\_binary\_file(file\_path):

with open(file\_path, 'rb') as file:

data = file.read()

# Process the binary data

processed\_data = data[::-1] # Reverse the binary data

return processed\_data

file\_path = 'data.bin'

result = process\_binary\_file(file\_path)

print(result)

```

While this code correctly reads binary data from a file and processes it, it lacks error handling for file reading operations. Identify potential errors that might occur during file reading and propose modifications to handle these errors gracefully.

\*\*Answer 1:\*\*

The code snippet lacks error handling for file reading operations, which can lead to runtime errors if the file is not found or if there are permission issues. To handle potential errors gracefully, we can add exception handling using a try-except block:

```python

def process\_binary\_file(file\_path):

try:

with open(file\_path, 'rb') as file:

data = file.read()

# Process the binary data

processed\_data = data[::-1] # Reverse the binary data

return processed\_data

except FileNotFoundError:

print("Error: File not found.")

except PermissionError:

print("Error: Permission denied while reading the file.")

except Exception as e:

print("Error:", e)

file\_path = 'data.bin'

result = process\_binary\_file(file\_path)

if result is not None:

print(result)

```

With this modification, the code gracefully handles potential errors such as file not found and permission denied while reading the file, ensuring robustness in file I/O operations.

#### \*\*Question 2:\*\*

You're developing a Python application that needs to serialize and deserialize complex data structures to and from a binary file. The current implementation of the serialization and deserialization functions is not efficient. Below is the inefficient code:

```python

import pickle

def serialize\_data(data, file\_path):

with open(file\_path, 'wb') as file:

pickle.dump(data, file)

def deserialize\_data(file\_path):

with open(file\_path, 'rb') as file:

return pickle.load(file)

data = {'key1': [1, 2, 3], 'key2': {'nested\_key': 'value'}}

file\_path = 'data.bin'

serialize\_data(data, file\_path)

loaded\_data = deserialize\_data(file\_path)

print(loaded\_data)

```

Identify the inefficiency in the provided serialization and deserialization functions and propose a more efficient alternative.

\*\*Answer 2:\*\*

The inefficiency in the provided code lies in the use of the `pickle` module for serialization and deserialization, which might not be the most efficient for complex data structures or large datasets. A more efficient alternative would be to use the `pickle` module's `dumps()` and `loads()` functions for in-memory serialization and deserialization:

```python

import pickle

def serialize\_data(data):

return pickle.dumps(data)

def deserialize\_data(serialized\_data):

return pickle.loads(serialized\_data)

data = {'key1': [1, 2, 3], 'key2': {'nested\_key': 'value'}}

serialized\_data = serialize\_data(data)

loaded\_data = deserialize\_data(serialized\_data)

print(loaded\_data)

```

By utilizing the `dumps()` and `loads()` functions for in-memory serialization and deserialization, the code achieves better efficiency compared to writing and reading from a binary file directly.

### 7)Database Interaction:

#### \*\*Question 1:\*\*

Consider the following Python code snippet which interacts with a SQLite database:

```python

import sqlite3

def create\_table():

connection = sqlite3.connect('example.db')

cursor = connection.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS users

(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')

connection.commit()

connection.close()

def insert\_user(name, age):

connection = sqlite3.connect('example.db')

cursor = connection.cursor()

cursor.execute('''INSERT INTO users (name, age) VALUES (?, ?)''', (name, age))

connection.commit()

connection.close()

def fetch\_users():

connection = sqlite3.connect('example.db')

cursor = connection.cursor()

cursor.execute('''SELECT \* FROM users''')

users = cursor.fetchall()

connection.close()

return users

create\_table()

insert\_user('Alice', 30)

insert\_user('Bob', 25)

users = fetch\_users()

print(users)

```

While this code interacts with a SQLite database and performs basic operations, it lacks error handling and proper resource management. Identify potential issues related to error handling and resource management in database interactions and propose improvements.

\*\*Answer 1:\*\*

The code snippet lacks proper error handling for database operations and doesn't follow best practices for resource management (opening and closing connections). To improve error handling and resource management, we can use context managers (`with` statement) for database connections and add exception handling for database operations:

```python

import sqlite3

def create\_table():

try:

with sqlite3.connect('example.db') as connection:

cursor = connection.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS users

(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')

except sqlite3.Error as e:

print("Error:", e)

def insert\_user(name, age):

try:

with sqlite3.connect('example.db') as connection:

cursor = connection.cursor()

cursor.execute('''INSERT INTO users (name, age) VALUES (?, ?)''', (name, age))

except sqlite3.Error as e:

print("Error:", e)

def fetch\_users():

try:

with sqlite3.connect('example.db') as connection:

cursor = connection.cursor()

cursor.execute('''SELECT \* FROM users''')

users = cursor.fetchall()

return users

except sqlite3.Error as e:

print("Error:", e)

return []

create\_table()

insert\_user('Alice', 30)

insert\_user('Bob', 25)

users = fetch\_users()

print(users)

```

With these modifications, the code handles potential errors that might occur during database operations and ensures proper resource management by using context managers for database connections. This improves the robustness and reliability of database interactions in the Python application.

#### \*\*Question 2:\*\*

You're developing a Python application that interacts with a MySQL database to perform CRUD (Create, Read, Update, Delete) operations on user data. The current implementation of the database interaction functions lacks error handling for database operations. Below is the inefficient code:

```python

import mysql.connector

def connect\_to\_database():

return mysql.connector.connect(

host="localhost",

user="username",

password="password",

database="mydatabase"

)

def fetch\_user(user\_id):

connection = connect\_to\_database()

cursor = connection.cursor()

query = "SELECT \* FROM users WHERE id = %s"

cursor.execute(query, (user\_id,))

user = cursor.fetchone()

connection.close()

return user

def update\_user(user\_id, new\_name):

connection = connect\_to\_database()

cursor = connection.cursor()

query = "UPDATE users SET name = %s WHERE id = %s"

cursor.execute(query, (new\_name, user\_id))

connection.commit()

connection.close()

user = fetch\_user(1)

print(user)

update\_user(1, "New Name")

```

Identify the potential issues related to error handling in the provided database interaction functions and propose improvements to handle errors gracefully.

\*\*Answer 2:\*\*

The potential issues in the provided code are the lack of error handling for database connection and operation errors, which can lead to unexpected behavior or crashes. To handle errors gracefully, we can implement try-except blocks for database operations and connection establishment:

```python

import mysql.connector

def connect\_to\_database():

try:

return mysql.connector.connect(

host="localhost",

user="username",

password="password",

database="mydatabase"

)

except mysql.connector.Error as e:

print("Error connecting to database:", e)

return None

def fetch\_user(user\_id):

connection = connect\_to\_database()

if connection:

try:

cursor = connection.cursor()

query = "SELECT \* FROM users WHERE id = %s"

cursor.execute(query, (user\_id,))

user = cursor.fetchone()

return user

except mysql.connector.Error as e:

print("Error fetching user:", e)

finally:

if connection.is\_connected():

connection.close()

def update\_user(user\_id, new\_name):

connection = connect\_to\_database()

if connection:

try:

cursor = connection.cursor()

query = "UPDATE users SET name = %s WHERE id = %s"

cursor.execute(query, (new\_name, user\_id))

connection.commit()

except mysql.connector.Error as e:

print("Error updating user:", e)

finally:

if connection.is\_connected():

connection.close()

user = fetch\_user(1)

print(user)

update\_user(1, "New Name")

```

With these improvements, the code handles database connection errors and operation errors gracefully, ensuring robustness in database interaction.

### **8)Generators:**

#### \*\*Question 1:\*\*

Consider the following Python code snippet which generates Fibonacci numbers using a generator function:

```python

def fibonacci\_generator():

a, b = 0, 1

while True:

yield a

a, b = b, a + b

fibonacci\_sequence = fibonacci\_generator()

for \_ in range(10):

print(next(fibonacci\_sequence))

```

While this code generates Fibonacci numbers efficiently using a generator, it lacks error handling for stopping the iteration after a certain condition is met. Identify the potential issue in the code related to stopping iteration and propose a modification to handle this issue gracefully.

\*\*Answer 1:\*\*

The code snippet generates Fibonacci numbers indefinitely, which might lead to memory exhaustion if not stopped. To handle this issue gracefully, we can introduce a stopping condition to limit the number of Fibonacci numbers generated:

```python

def fibonacci\_generator(limit):

a, b = 0, 1

count = 0

while count < limit:

yield a

a, b = b, a + b

count += 1

fibonacci\_sequence = fibonacci\_generator(10)

for num in fibonacci\_sequence:

print(num)

```

With this modification, the generator function stops generating Fibonacci numbers after reaching the specified limit, ensuring that the iteration doesn't continue indefinitely.

#### \*\*Question 2:\*\*

You're developing a Python application that needs to process a large stream of data, where the data is continuously generated and consumed in real-time. The current implementation of the data processing function is not memory-efficient. Below is the inefficient code:

```python

def process\_data(data\_stream):

processed

\_data = [process\_item(item) for item in data\_stream]

return processed\_data

def process\_item(item):

# Example processing operation

return item.upper()

data\_stream = range(1000000) # Example data stream generator

processed\_data = process\_data(data\_stream)

print(processed\_data[:10]) # Print first 10 processed items

```

Identify the inefficiency in the provided data processing function and propose a more memory-efficient alternative.

\*\*Answer 2:\*\*

The inefficiency in the provided code lies in the creation of a list to store all processed data items, which can consume a significant amount of memory for large data streams. A more memory-efficient alternative would be to use a generator function for lazy evaluation:

```python

def process\_data(data\_stream):

for item in data\_stream:

yield process\_item(item)

def process\_item(item):

# Example processing operation

return item.upper()

data\_stream = range(1000000) # Example data stream generator

processed\_data\_generator = process\_data(data\_stream)

for i, item in enumerate(processed\_data\_generator):

print(item)

if i >= 9:

break

```

By using a generator function for lazy evaluation, the code avoids storing all processed data items in memory simultaneously, making it more memory-efficient, especially for large data streams.

### 9) Security Considerations:

#### \*\*Question 1:\*\*

Consider the following Python code snippet which hashes passwords using the SHA-256 algorithm:

```python

import hashlib

def hash\_password(password):

return hashlib.sha256(password.encode()).hexdigest()

password = input("Enter your password: ")

hashed\_password = hash\_password(password)

print("Hashed password:", hashed\_password)

```

While this code securely hashes passwords, it lacks salting, which is an important security measure to prevent rainbow table attacks. Identify the potential security vulnerability in the code and propose a modification to enhance password hashing security.

\*\*Answer 1:\*\*

The code snippet lacks salting, which makes the hashed passwords vulnerable to rainbow table attacks. To enhance password hashing security, we can introduce a random salt for each password and concatenate it with the password before hashing:

```python

import hashlib

import secrets

def hash\_password(password):

salt = secrets.token\_hex(16) # Generate a random salt

salted\_password = salt + password

hashed\_password = hashlib.sha256(salted\_password.encode()).hexdigest()

return hashed\_password, salt

password = input("Enter your password: ")

hashed\_password, salt = hash\_password(password)

print("Hashed password:", hashed\_password)

print("Salt:", salt)

```

With this modification, each password is hashed with a unique salt, making it more resilient against rainbow table attacks and enhancing overall password hashing security.

#### \*\*Question 2:\*\*

You're developing a Python web application that accepts user inputs through a form submission. The current implementation of the input processing function lacks proper input sanitization, making it vulnerable to injection attacks. Below is the inefficient code:

```python

def process\_input(user\_input):

# Example processing operation

return "Hello, " + user\_input

user\_input = input("Enter your name: ")

processed\_input = process\_input(user\_input)

print(processed\_input)

```

Identify the security vulnerability in the provided input processing function and propose a modification to enhance input sanitization and prevent injection attacks.

\*\*Answer 2:\*\*

The security vulnerability in the provided code is that it directly uses user input without proper sanitization, making it vulnerable to injection attacks such as SQL injection or cross-site scripting (XSS). To enhance input sanitization and prevent injection attacks, we can use parameterized queries for database interactions and escape special characters for HTML output:

```python

import html

def process\_input(user\_input):

# Example processing operation with input sanitization

sanitized\_input = html.escape(user\_input) # Escape special characters for HTML output

return "Hello, " + sanitized\_input

user\_input = input("Enter your name: ")

processed\_input = process\_input(user\_input)

print(processed\_input)

```

With this modification, the code properly sanitizes user input by escaping special characters, making it resistant to injection attacks and ensuring security in the web application.

### 10) Networking: (optional as most ppl dont know it)

#### \*\*Question 1:\*\*

Consider the following Python code snippet which implements a simple TCP server:

```python

import socket

def start\_server(host, port):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((host, port))

server\_socket.listen(1)

print(f"Server listening on {host}:{port}")

while True:

client\_socket, client\_address = server\_socket.accept()

print(f"Connection from {client\_address}")

while True:

data = client\_socket.recv(1024)

if not data:

break

client\_socket.sendall(data)

client\_socket.close()

start\_server('127.0.0.1', 8080)

```

While this code implements a basic TCP server, it only handles one client connection at a time. Identify the potential limitation in the code related to handling multiple client connections concurrently and propose a modification to address this limitation.

\*\*Answer 1:\*\*

The code snippet handles only one client connection at a time, which might not be efficient for scenarios with multiple client connections. To handle multiple client connections concurrently, we can use threading to spawn a new thread for each client connection:

```python

import socket

import threading

def handle\_client(client\_socket, client\_address):

print(f"Connection from {client\_address}")

while True:

data = client\_socket.recv(1024)

if not data:

break

client\_socket.sendall(data)

client\_socket.close()

print(f"Connection with {client\_address} closed")

def start\_server(host, port):

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as server\_socket:

server\_socket.bind((host, port))

server\_socket.listen(5)

print(f"Server listening on {host}:{port}")

while True:

client\_socket, client\_address = server\_socket.accept()

threading.Thread(target=handle\_client, args=(client\_socket, client\_address)).start()

start\_server('127.0.0.1', 8080)

```

With this modification, the server can handle multiple client connections concurrently by spawning a new thread for each incoming connection, ensuring efficient utilization of server resources.

#### \*\*Question 2:\*\*

You're developing a Python application that needs to send and receive data over a network using TCP sockets. The current implementation of the networking functions lacks error handling for socket operations. Below is the inefficient code:

```python

import socket

def send\_data(data, host, port):

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect((host, port))

client\_socket.sendall(data.encode())

client\_socket.close()

def receive\_data(host, port):

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind((host, port))

server\_socket.listen(1)

connection, address = server\_socket.accept()

received\_data = connection.recv(1024).decode()

connection.close()

return received\_data

host = '127.0.0.1'

port = 8080

send\_data("Hello, server!", host, port)

received\_data = receive\_data(host, port)

print("Received:", received\_data)

```

Identify the potential issues related to error handling in the provided networking functions and propose improvements to handle errors gracefully.

\*\*Answer 2:\*\*

The potential issues in the provided code are the lack of error handling for socket connection errors, send errors, and receive errors, which can lead to unexpected behavior or crashes. To handle errors gracefully, we can implement try-except blocks for socket operations:

```python

import socket

def send\_data(data, host, port):

try:

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect((host, port))

client\_socket.sendall(data.encode())

except socket.error as e:

print("Error sending data:", e)

finally:

if 'client\_socket' in locals():

client\_socket.close()

def receive\_data(host, port):

try:

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind((host, port))

server\_socket.listen(1)

connection, address = server\_socket.accept()

received\_data = connection.recv(1024).decode()

return received\_data

except socket.error as e:

print("Error receiving data:", e)

finally:

if 'server\_socket' in locals():

server\_socket.close()

host = '127.0.0.1'

port = 8080

send\_data("Hello, server!", host, port)

received\_data = receive\_data(host, port)

print("Received:", received\_data)

```

With these improvements, the code handles socket connection errors, send errors, and receive errors gracefully, ensuring robustness in networking operations.