# PROMPTS: USER

## prompt1 ="""

generate a Python debugging coding question. \

generate debugging questions in python that test common but important mistakes made in python while on the job. \

The questions you generate have to have the following format :

{ 1 - Scenario : Real world scenario along with an explanation of the code being used

2 - Code : This should have the wrong code or the code that must be debugged using important and senior level topics.

The code must be around 20-50 lines

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both, and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2), what concepts were being

tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code

} \

You generate questions that test important and relevant questions (for a job) for senior python developers.\

You ask mainly in terms of efficiency and debugging.\

Your code snippets have 20 to 50 lines of code. \

In the case that user gives invalid python topics, or invalid debugging question requests, say "invalid" and give one line of explanation for why it is invalid.\

"""

## prompt2 = """

generate a Python debugging coding question that tests important mistakes made in python while on the job. There is a special

emphasis on important, relevant, and high level python topics \

The questions you generate have to have the following format :

{ 0 - Concept : what is the hard concept or combination of concepts being used, list them.

1 - Scenario : Real world scenario along with an explanation of the code being used

2 - Code : This should have the wrong code (code that must be debugged) using important and senior level topics.

The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both, and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2), what concepts were being

tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code

} \

questions must test important and relevant questions (for a job) for senior python developers.\

You ask mainly in terms of debugging as in fixing actual bugs in the code. These questions must be solvable if and only if

the hard topics being tested are actually conceptually understood by the solvers.\

"""

## prompt3 = """

generate a Python debugging coding question with the main criteria marked using the triple backticks

```tests important mistakes made in advanced python topics while on the job```.

There is a special emphasis on important, relevant, and advanced level python topics

The questions you generate must have the following format :

``` 0 - Topics : what is the hard concept or combination of concepts being used, list them if there are multiple.

1 - Scenario : Real world scenario along with an explanation of the code being used. The scenario

should not always have a senior developer in the scenario description.

2 - Code : This should have the wrong code (code that must be debugged) using important and advanced

level topics. The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both,

and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2),

what concepts were being tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code

```

The questions must be solvable if and only if the solver is someone who has had experience and conceptually

understands the advanced topics being tested. No college going student should be able to solve it easily.

You ask mainly in terms of debugging as in fixing actual bugs in the code.

"""

## prompt4 = """

generate a Python debugging coding question with the main criteria marked using the triple backticks

```tests important mistakes made in advanced python topics while on the job that are beyond syntax mistakes```.

There is a special emphasis on important, relevant, and advanced level python topics

The questions you generate must have the following format :

``` 0 - Topics : what is the hard concept or combination of concepts being used, list them if there are multiple.

1 - Scenario : Real world scenario along with an explanation of the code being used. The scenario

should not always have a senior developer in the scenario description.

2 - Code : This should have the wrong code (code that must be debugged) using important and advanced

level topics. The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both,

and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2),

what concepts were being tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code

```

The questions must be solvable if and only if the solver is someone who has had experience and conceptually

understands the advanced topics being tested. An important note is marked in triple backticks

``` do not make the bugs all very syntactical```. Additionally the questions must be at a difficulty level

where no junior developer should be able to solve it easily, but can be solved by senior developers.

You ask mainly in terms of debugging as in fixing actual bugs in the code (conceptual).

"""

## prompt5 = """

generate 2 Python debugging coding question with the main criteria marked using the triple backticks

```tests important mistakes made in different advanced python topics while on the job that are beyond syntax mistakes```.

There is a special emphasis on important, relevant, and advanced level python topics

The questions you generate must have the following format :

``` 0 - Topics : what is the hard concept or combination of concepts being used, list them if there are multiple.

1 - Scenario : Real world scenario along with an explanation of the code being used. The scenario

should not always have a senior developer in the scenario description.

2 - Code : This should have the wrong code (code that must be debugged) using important and advanced

level topics. The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both,

and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2),

what concepts were being tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code

```

The questions must be slovable if and only if the solver is someone who has had experience and conceptually

understands the advanced topics being tested. An important note is marked in triple backticks

``` do not make the bugs all very syntactical```. Additionally the questions must be at adifficulty level

where no junior developer should be able to solve it easily, but can be solved by senior developers.

You ask mainly in terms of debugging as in fixing actual bugs in the code (conceptual).

"""

## Prompt6 (good so far) = """

generate 2 Python debugging coding question with the main criteria marked using the triple backticks

```tests important mistakes made in different advanced python topics while on the job that are beyond syntax mistakes```.

There is a special emphasis on important, relevant, and advanced level python topics

The questions you generate must have the following format :

``` 0 - Topics : what is the hard concept or combination of concepts being used, list them if there are multiple.

1 - Scenario : Real world scenario along with an explanation of the code being used. The scenario

should not always have a senior developer in the scenario description.

2 - Code : This should have the wrong code (code that must be debugged) using important and advanced

level topics. The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both,

and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2),

what concepts were being tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code, with no bugs or compiling issues.

```

The questions must be solvable if and only if the solver is someone who has had experience and conceptually

understands the advanced topics being tested. Also the answer version of the code must properly run on a python compiler

with no issues. An important note is marked in triple backticks

``` do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code. This

difference has to be conceptual and not something silly```. Additionally the questions must be at a difficulty level

where no junior developer should be able to solve it easily, but can be solved by senior developers.

You ask mainly in terms of debugging as in fixing actual bugs in the code (conceptual).

"""

## prompt7 = """

generate 5 Python debugging coding question with the main criteria marked using the triple backticks

```tests important mistakes made in different advanced python topics while on the job that are beyond syntax mistakes```.

There is a special emphasis on important, relevant, and advanced level python topics

The questions you generate must have the following format :

``` 0 - Topics : what is the hard concept or combination of concepts being used, list them if there are multiple.

1 - Scenario : Real world scenario along with an explanation of the code being used. The scenario

should not always have a senior developer in the scenario description.

2 - Code : This should have the wrong code (code that must be debugged) using important and advanced

level topics. The code must be around 30-50 lines of meaningful code that adds to the context.

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both,

and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2),

what concepts were being tested, and idea of the changes that must be made

5 - Answer : Implementing the explanation, this part must have the corrected code, with no bugs or compiling issues.

```

The questions must be solvable if and only if the solver is someone who has had experience and conceptually

understands the advanced topics being tested. Also the answer version of the code must properly run on a python compiler

with no issues. An important note is marked in triple backticks

``` do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code. This

difference has to be conceptual and not something silly```. Additionally the questions must be at a difficulty level

where no junior developer should be able to solve it easily, but can be solved by senior developers.

You ask mainly in terms of debugging as in fixing actual bugs in the code (conceptual).

"""

## prompt9 = """

generate 3 Python debugging coding questions such that they ```test important mistakes made in different advanced python topics

while on the job that are beyond syntax mistakes```.

The questions you generate must have the following format :

``` 0 - Topics

1 - Scenario : Real world application of code

2 - Code : 50 line snippet of wrong code (code that must be debugged) using important and advanced level topics.

3 - Question : explain that there is some issue in the code

4 - Explanation : explanation regarding what was wrong in the code (as mentioned in part 2),

5 - Answer : this part must have the corrected code(in relation to part 2,4), with no bugs or compiling issues.

```

Do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code that tests relevant

concepts for high level developers. The questions must be at a difficulty level such that it can be solved only by senior

developers, not by any juniors.

"""

## prompt12 = f"""

generate 5 Python debugging coding questions such that they ```test important mistakes made in different advanced python topics

while on the job that are beyond syntax mistakes```.

The questions you generate must have the following format :

``` 0 - Topics

1 - Scenario : Real world application of code

2 - Code : 50 line snippet of wrong code (code that must be debugged) using important and advanced level topics.

3 - Question : explain that there is some issue in the code

4 - Explanation : explanation regarding what was wrong in the code (as mentioned in part 2),

5 - Answer : this part must have the corrected code(in relation to part 2,4), with no bugs or compiling issues.

```

Do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code that tests relevant

concepts for high level developers. The questions must be at a difficulty level such that it can be solved only by senior

developers, not by any juniors.

some good examples are ```{sample2c}``` and ```{sample3c}``. One example that is bad is ```{sample1w}``` this is because

the question has only 3 lines of code, and tests syntax based square and closed brackets. Its too simple and does not test for an

advanced conceptual understanding.

Ensure that you generate 5 distinct questions following the format provided, take your time and give me good quality questions.

Please start the response by enumerating each question from 1 to 5.

Here are 5 distinct Python debugging coding questions:

"""

## prompt13 = f"""(sillyyyyy)

generate 5 Python debugging coding questions such that they ```test important mistakes made in different advanced python topics

while on the job that are beyond syntax mistakes```.

The questions you generate must have the following format :

``` 0 - Topics

1 - Scenario : Real world application of code

2 - Code : 50 line snippet of wrong code (code that must be debugged) using important and advanced level topics.

3 - Question : explain that there is some issue in the code

4 - Explanation : explanation regarding what was wrong in the code (as mentioned in part 2),

5 - Answer : this part must have the corrected code(in relation to part 2,4), with no bugs or compiling issues.

```

Do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code that tests relevant

concepts for high level developers. The questions must be at a difficulty level such that it can be solved only by senior

developers, not by any juniors.

some good examples are ```{sample2c}``` and ```{sample3c}``. One example that is bad is ```{sample1w}``` this is because

the question has only 3 lines of code, and tests syntax based square and closed brackets. Its too simple and does not test for an

advanced conceptual understanding. Another bad example is ```{sample2w}``` this is because the answer version also does not compile

and has a runtime error. The answer should not have such errors.

Ensure that you generate 5 distinct questions following the format provided, take your time and give me good quality questions.

Please start the response by enumerating each question from 1 to 5.

Here are 5 distinct Python debugging coding questions:

"""

## prompt14 = f"""

generate 5 Python debugging coding questions such that they ```test important mistakes made in different advanced python topics

while on the job that are beyond syntax mistakes```.

The questions you generate must have the following format :

``` 0 - Topics

1 - Scenario : Real world application of code

2 - Code : 50 line snippet of wrong code (code that must be debugged) using important and advanced level topics.

3 - Question : explain that there is some issue in the code

4 - Explanation : explanation regarding what was wrong in the code (as mentioned in part 2),

5 - Answer : this part must have the corrected code(in relation to part 2,4), with no bugs or compiling issues on interpreter.

```

Do not make the bugs all very syntactical, and there must be a difference in the wrong code and corrected code that tests relevant

concepts for high level developers. The questions must be at a difficulty level such that it can be solved only by senior

developers, not by any juniors.

some good examples are ```{sample2c}``` and ```{sample3c}``. One bad example is ```{sample1w}``` because

the question has only 3 lines of code, and tests syntax based square and closed brackets. Its too simple and does not test for an

advanced conceptual understanding.

Ensure that you generate 5 distinct questions following the format provided, take your time and give me good quality questions.

Please start the response by enumerating each question from 1 to 5.

Here are 5 distinct Python debugging coding questions:

"""

# RESULTS:USER

## Result: prompt1\_result 3.5gpt

: {

# 1 - Scenario:

# A senior Python developer is working on a project that involves processing a large amount of data from a CSV file.

# The code they have written is supposed to read the data from the CSV file, perform some calculations,

# and then output the results. However, they are experiencing performance issues and the code is taking a long time

# to run.

# 2 - Code:

*```python*

import csv

data = []

with open('data.csv', 'r') as file:

reader = csv.reader(file)

for row in reader:

data.append(row)

results = []

for row in data:

result = row[0] \* row[1]

results.append(result)

for result in results:

print(result)

*```*

# 3 - Question:

# What is the issue with the code provided? How can the performance be improved?

# 4 - Explanation:

# The issue with the code is that it is reading the data from the CSV file row by row,

# which can be inefficient for large datasets. Additionally, the calculation being performed on each row

# is incorrect as it is trying to multiply two strings instead of converting them to integers first.

# To improve performance, the code should read the entire CSV file into memory at once and convert the data

# to integers before performing the calculations.

# 5 - Answer:

*```python*

import csv

data = []

with open('data.csv', 'r') as file:

reader = csv.reader(file)

data = [list(map(int, row)) for row in reader]

results = []

for row in data:

result = row[0] \* row[1]

results.append(result)

for result in results:

print(result)

*```*

# }

## Result: prompt2\_response 3.5gpt:

# 0 - Concept: Object-Oriented Programming, Inheritance, Method Overriding

# 1 - Scenario: In a software development project, a senior developer is working on a Python program that simulates a zoo management system. The program has a base class called Animal, with subclasses for different types of animals such as Lion, Elephant, and Giraffe. Each subclass has a method called make\_sound() that is supposed to return the sound the animal makes.

# 2 - Code:

*```python*

class Animal:

def make\_sound(*self*):

return "Some generic sound"

class Lion(Animal):

def make\_sound(*self*):

return "Roar"

class Elephant(Animal):

def make\_sound(*self*):

return "Trumpet"

class Giraffe(Animal):

def make\_sound(*self*):

return "Giraffe sound"

zoo = [Lion(), Elephant(), Giraffe()]

for animal in zoo:

print(animal.make\_sound())

*```*

# 3 - Question: There is a bug in the code that is causing all animals to make the same generic sound instead of their

# specific sounds. How would you fix this issue?

# 4 - Explanation: The issue in the code is due to method overriding. The make\_sound() method in the Animal class

# is being called for all instances of the subclasses, instead of the overridden methods in the Lion, Elephant,

# and Giraffe classes. To fix this, we need to make sure that the overridden methods in the subclass are called

# instead of the method in the base class.

# 5 - Answer:

*```python*

class Animal:

def make\_sound(*self*):

return "Some generic sound"

class Lion(Animal):

def make\_sound(*self*):

return "Roar"

class Elephant(Animal):

def make\_sound(*self*):

return "Trumpet"

class Giraffe(Animal):

def make\_sound(*self*):

return "Giraffe sound"

zoo = [Lion(), Elephant(), Giraffe()]

for animal in zoo:

print(animal.make\_sound())

*```*

## Result: prompt3\_response 4gpt, temp 0.6

# 0 - Topics : Multithreading and Exception Handling in Python

# 1 - Scenario : You are a software engineer working on a real-time application that requires a high level of

# concurrency. You decided to use Python's threading module to achieve this. You have a function that performs

# a time-consuming task and you want to run multiple instances of this function concurrently. However, your code

# is not running as expected.

# 2 - Code :

*```python*

import threading

import time

def time\_consuming\_task():

try:

for i in range(10):

print(f"Task running on thread {threading.current\_thread().name}")

time.sleep(1)

except Exception as e:

print(f"An error occurred: {e}")

threads = []

for i in range(5):

t = threading.Thread(*target*=time\_consuming\_task)

t.start()

threads.append(t)

for thread in threads:

thread.join()

*```*

# 3 - Question : There is an issue with the above code. The threads are not running concurrently as expected,

# instead, they are running sequentially. Find the bug and fix it.

# 4 - Explanation : The problem here is that we are not correctly creating and starting threads.

# When creating threads, we need to pass the function name without parentheses to the target parameter.

# If we include parentheses, the function will be executed immediately instead of being passed

# to the thread for execution. This is a common mistake when dealing with multithreading in Python.

# 5 - Answer :

*```python*

import threading

import time

def time\_consuming\_task():

try:

for i in range(10):

print(f"Task running on thread {threading.current\_thread().name}")

time.sleep(1)

except Exception as e:

print(f"An error occurred: {e}")

threads = []

for i in range(5):

t = threading.Thread(*target*=time\_consuming\_task)

t.start()

threads.append(t)

for thread in threads:

thread.join()

*```*

# In the corrected code, we remove the parentheses when passing the function to the target parameter.

# This allows the function to be passed to the thread for execution, allowing for concurrent execution of the threads.

## Result: prompt4\_response\_4gpt, temp=0.6

(when regenerated it kind of tests similar concepts)

# 0 - Topics : Multithreading and Global Interpreter Lock (GIL)

# 1 - Scenario : You are a software engineer working on a high-performance application that requires

# heavy computation. You decide to use multithreading to speed up your program.

# You use the threading module in Python and create multiple threads to perform the computation concurrently.

# However, you notice that the program is not running as fast as expected.

# 2 - Code :

*```python*

import threading

import time

numbers = list(range(1, 1000000))

def compute\_square(*numbers*):

for n in *numbers*:

n \* n

def compute\_cube(*numbers*):

for n in *numbers*:

n \* n \* n

start = time.time()

t1 = threading.Thread(*target*=compute\_square, *args*=(numbers,))

t2 = threading.Thread(*target*=compute\_cube, *args*=(numbers,))

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print("Time taken in seconds -", end - start)

*```*

# 3 - Question : The code is supposed to compute the square and cube of numbers in the range 1 to 1,000,000

# concurrently using multithreading. However, the performance is not as expected. Identify the issue and suggest

# a way to improve the efficiency of the code.

# 4 - Explanation : The issue here is related to the Global Interpreter Lock (GIL) in Python. The GIL is a

# mechanism used in CPython to synchronize the execution of threads so that only one native thread executes

# Python bytecode at a time. This means that even though Python provides a threading module,

# it does not allow for true concurrent execution due to the GIL. In this case, the usage of threads does not

# improve the performance as expected due to the GIL. A better approach would be to use multiprocessing instead

# of multithreading, as it allows for true concurrent execution by creating separate Python interpreter processes.

# 5 - Answer :

*```python*

import multiprocessing

import time

numbers = list(range(1, 1000000))

def compute\_square(*numbers*):

for n in *numbers*:

n \* n

def compute\_cube(*numbers*):

for n in *numbers*:

n \* n \* n

start = time.time()

p1 = multiprocessing.Process(*target*=compute\_square, *args*=(numbers,))

p2 = multiprocessing.Process(*target*=compute\_cube, *args*=(numbers,))

p1.start()

p2.start()

p1.join()

p2.join()

end = time.time()

print("Time taken in seconds -", end - start)

## Result: prompt4\_response\_4gpt, temp=0.9

# Here's a challenging debugging python question.

# ```

# 0 - Topics: Multithreading, file manipulation

# 1 - Scenario: As a senior developer at a data analysis company, you are required to write a script that takes

# a log file as input, processes it and writes the output into another file. The script is supposed to work fine,

# however, it stalls occasionally and no output is produced.

# 2 - Code:

*```python*

import threading

import time

class WorkerThread(*threading*.Thread):

def \_\_init\_\_(*self*, *file*):

threading.Thread.\_\_init\_\_(*self*)

*self*.file = *file*

def run(*self*):

with open(*self*.file, 'r') as f:

lines = f.readlines()

output\_lines = []

for line in lines:

output\_lines.append(process\_line(line))

with open('output.txt', 'w') as f:

for line in output\_lines:

f.write(line)

def process\_line(*line*):

# Some complex line processing goes here

time.sleep(0.01) # Simulating processing delay

return *line*

worker = WorkerThread('logfile.txt')

worker.start()

worker.join()

*```*

# 3 - Question: The code above sometimes hangs with no output produced, which means there's a bug in it.

# Your task is to find and fix the bug.

# 4 - Explanation: The code above has a deadlock problem. The worker thread sometimes doesn't finish because

# the main thread is waiting for it to complete using join() and the worker thread is waiting for the main thread

# to close the file it opened. This is a typical deadlock situation which is common in multithreading applications.

# The concepts being tested include threading, file manipulation and deadlock situations.

# 5 - Answer: The solution for this problem is to make sure the main thread doesn't wait for the worker thread

# indefinitely. This can be done by using a timeout with join(). Below is the corrected code:

*```python*

import threading

import time

class WorkerThread(*threading*.Thread):

def \_\_init\_\_(*self*, *file*):

threading.Thread.\_\_init\_\_(*self*)

*self*.file = *file*

def run(*self*):

with open(*self*.file, 'r') as f:

lines = f.readlines()

output\_lines = []

for line in lines:

output\_lines.append(process\_line(line))

with open('output.txt', 'w') as f:

for line in output\_lines:

f.write(line)

def process\_line(*line*):

# Some complex line processing goes here

time.sleep(0.01) # Simulating processing delay

return *line*

worker = WorkerThread('logfile.txt')

worker.start()

worker.join(1) # Only wait for 1 second for the worker thread to finish

if worker.is\_alive():

print("Processing took longer than expected. Try again.")

else:

print("Processing finished.")

*```*

*```*

## Result: Prompt5\_response,4o-gpt,temp=0.9

### Question 1

``` 0 - Topics: Multi-threading, Concurrency, Race Conditions

# 1 - Scenario: You are a software engineer tasked with developing a new feature for an e-commerce application.

# This feature involves processing user orders and updating an inventory database in real-time. Your team has

# created a multi-threaded implementation to handle concurrent orders to improve efficiency.

# The code is supposed to ensure that the inventory count is updated accurately without any race conditions.

# 2 - Code:

*```python*

import threading

class Inventory:

def \_\_init\_\_(*self*,*stock*):

*self*.stock = *stock*

def update\_stock(*self*, *item*, *quantity*):

if *self*.stock[*item*] >= *quantity*:

*self*.stock[*item*] -= *quantity*

print(f"Order processed: {*item*} - {*quantity*}")

else:

print(f"Insufficient stock for item: {*item*}")

inventory = Inventory({'item1': 10, 'item2': 20})

def process\_order(*item*, *quantity*):

inventory.update\_stock(*item*, *quantity*)

threads = []

orders = [('item1', 2), ('item2', 5), ('item1', 7), ('item2', 15)]

for order in orders:

t = threading.Thread(*target*=process\_order, *args*=order)

threads.append(t)

t.start()

for t in threads:

t.join()

print(f"Final stock: {inventory.stock}")

# 3 - Question: The given code is supposed to handle concurrent orders and update the inventory without

# any issues. However, there are potential bugs related to concurrency and race conditions. Identify and

# fix the issues to ensure the inventory is updated correctly.

# 4 - Explanation: The code fails to handle concurrency issues properly. The update\_stock method modifies

# the inventory without any synchronization mechanism, leading to race conditions where multiple threads might

# modify the stock simultaneously. This can result in incorrect inventory counts.

# 5 - Answer:

*```python*

import threading

class Inventory:

def \_\_init\_\_(*self*, *stock*):

*self*.stock = *stock*

*self*.lock = threading.Lock()

def update\_stock(*self*, *item*, *quantity*):

with *self*.lock:

if *self*.stock[*item*] >= *quantity*:

*self*.stock[*item*] -= *quantity*

print(f"Order processed: {*item*} - {*quantity*}")

else:

print(f"Insufficient stock for item: {*item*}")

inventory = Inventory({'item1': 10, 'item2': 20})

def process\_order(*item*, *quantity*):

inventory.update\_stock(*item*, *quantity*)

threads = []

orders = [('item1', 2), ('item2', 5), ('item1', 7), ('item2', 15)]

for order in orders:

t = threading.Thread(*target*=process\_order, *args*=order)

threads.append(t)

t.start()

for t in threads:

t.join()

print(f"Final stock: {inventory.stock}")

*```*

### Question 2

#``` 0 - Topics: Asynchronous programming, asyncio, Deadlocks

# 1 - Scenario: You are developing a microservice that retrieves data from two different APIs concurrently

# to combine and process the data. This microservice uses Python's asyncio library to perform asynchronous tasks

# and improve the response time. However, your teammate reported that the service sometimes hangs indefinitely.

# 2 - Code:

*```python*

import asyncio

import aiohttp

async def fetch\_data(*session*, *url*):

async with *session*.get(*url*) as response:

return await response.json()

async def gather\_data():

async with aiohttp.ClientSession() as session:

task1 = asyncio.create\_task(fetch\_data(session, 'https://api.example.com/data1'))

task2 = asyncio.create\_task(fetch\_data(session, 'https://api.example.com/data2'))

data1 = await task1

data2 = await task2

return data1, data2

async def main():

data1, data2 = await gather\_data()

combined\_data = {\*\*data1, \*\*data2}

print("Combined Data:", combined\_data)

asyncio.run(main())

# 3 - Question: The code is intended to fetch data from two APIs concurrently using asyncio. Occasionally, the

# microservice hangs indefinitely without completing the task. Identify and fix the issue to ensure the

# microservice fetches and processes the data correctly.

# 4 - Explanation: The issue lies in the structure of the asynchronous calls. If either of the tasks take

# longer or get stuck, the entire system might hang because there are no timeouts or proper exception handling

# to deal with such scenarios. Proper handling of these cases is crucial for robust asynchronous programming.

# 5 - Answer:

*```python*

import asyncio

import aiohttp

async def fetch\_data(*session*, *url*):

try:

async with *session*.get(*url*, *timeout*=10) as response:

return await response.json()

except asyncio.TimeoutError:

print(f"Timeout while fetching data from {*url*}")

return {}

async def gather\_data():

async with aiohttp.ClientSession() as session:

task1 = asyncio.create\_task(fetch\_data(session, 'https://api.example.com/data1'))

task2 = asyncio.create\_task(fetch\_data(session, 'https://api.example.com/data2'))

data1, data2 = await asyncio.gather(task1, task2, *return\_exceptions*=True)

if isinstance(data1, Exception) or isinstance(data2, Exception):

print("Error occurred during fetching data")

return {}, {}

return data1, data2

async def main():

data1, data2 = await gather\_data()

combined\_data = {\*\*data1, \*\*data2}

print("Combined Data:", combined\_data)

asyncio.run(main())

## Result:prompt6\_response,4gpt,temp = 0.7

### Question 1

# 0 - Topics: Multithreading, Concurrency, Global Interpreter Lock (GIL)

# 1 - Scenario: A software team is working on a web scraping project where they need to fetch data from multiple URLs

# simultaneously. They decided to use multithreading for concurrent execution. The junior developer in the team wrote the

# following python code for the same but the code doesn't seem to be working as expected.

# 2 - Code:

# ```python

import threading

import urllib.request

class FetchUrls(*threading*.Thread):

def \_\_init\_\_(*self*, *urls*, *output*):

threading.Thread.\_\_init\_\_(*self*)

*self*.urls = *urls*

*self*.output = *output*

def run(*self*):

while *self*.urls:

url = *self*.urls.pop()

result = urllib.request.urlopen(url)

*self*.output.write(result.read())

return

def main(*urls*, *output\_path*):

f = open(*output\_path*, 'w')

fetcher = FetchUrls(*urls*, f)

fetcher.start()

# ```

# 3 - Question: Can you find out what's wrong with this code? The developer seems to miss out on some important concepts

# of threading and concurrency.

# 4 - Explanation: The FetchUrls thread is not thread-safe. The urls are shared among all threads but python list

# is not thread-safe. Also, file write operation is also not thread-safe. This code will lead to race conditions.

# 5 - Answer:

# ```python

import threading

import urllib.request

import queue

class FetchUrls(*threading*.Thread):

def \_\_init\_\_(*self*, *queue*, *output*):

threading.Thread.\_\_init\_\_(*self*)

*self*.queue = *queue*

*self*.output = *output*

def run(*self*):

while True:

url = *self*.queue.get()

result = urllib.request.urlopen(url)

*self*.output.write(result.read())

*self*.queue.task\_done()

def main(*urls*, *output\_path*):

url\_queue = queue.Queue()

output = open(*output\_path*, 'w')

for url in *urls*:

url\_queue.put(url)

for \_ in range(5):

fetcher = FetchUrls(url\_queue, output)

fetcher.start()

url\_queue.join()

output.close()

# ```

# # #prompt6, gpt4, temp = 0.7

### Question 2

# 0 - Topics: Decorators, Closures, Performance Optimization

# 1 - Scenario: A team is working on a performance-critical application where function execution time is very important.

# The team lead wants to have a mechanism to measure the execution time of all the functions in the application. A junior

# developer tried to use decorators for the same but facing some issues.

# 2 - Code:

# ```python

import time

def timer\_decorator(*func*):

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

start\_time = time.perf\_counter()

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

end\_time = time.perf\_counter()

print(f"Function {*func*.\_\_name\_\_} took {end\_time - start\_time} seconds to execute")

return result

return wrapper

@timer\_decorator

def sum\_numbers(*n*):

return sum(range(*n*))

print(sum\_numbers(1000000))

# ```

# 3 - Question: The above code is not working as expected. Can you find out the mistake and correct it?

# 4 - Explanation: The problem is with the function call inside the wrapper function. The arguments are not passed

# correctly to the function. Instead of passing `args` and `kwargs` as two separate arguments, it should be unpacked

# while calling the function.

# 5 - Answer:

# ```python

import time

def timer\_decorator(*func*):

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

start\_time = time.perf\_counter()

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

end\_time = time.perf\_counter()

print(f"Function {*func*.\_\_name\_\_} took {end\_time - start\_time} seconds to execute")

return result

return wrapper

@timer\_decorator

def sum\_numbers(*n*):

return sum(range(*n*))

print(sum\_numbers(1000000))

# ```

## Result: prompt7\_response , gpt4 , temp = 0.7

### Question 1:

# - \*\*Topics\*\*: Multithreading, Concurrency, Global Interpreter Lock (GIL)

# - \*\*Scenario\*\*: A python programmer in a large tech company is working on a script that needs to download several files

# concurrently to optimize time. He decided to use Python threading module.

# - \*\*Code\*\*:

# ```python

import threading

import urllib.request

def download\_file(*url*, *file\_name*):

response = urllib.request.urlopen(*url*)

data = response.read()

with open(*file\_name*, 'wb') as file:

file.write(data)

urls = [

'http://example.com/file1.txt',

'http://example.com/file2.txt',

'http://example.com/file3.txt'

]

for i, url in enumerate(urls):

threading.Thread(*target*=download\_file, *args*=(url, f'file{i}.txt')).start()

*```*

# - \*\*Question\*\*: The code has an efficiency issue due to Python's Global Interpreter Lock (GIL).

# The code is not as fast as expected. What can be done to improve this?

# - \*\*Explanation\*\*: The GIL is a mechanism in CPython that only allows one thread to execute Python bytecodes

# at a time in a single process. Threading in Python does not take advantage of multiple cores because of the GIL.

# A better alternative for I/O bound tasks like downloading files from the internet would be to use asynchronous I/O

# or multiprocessing.

# - \*\*Answer\*\*:

# ```python

from concurrent.futures import ThreadPoolExecutor

import urllib.request

def download\_file(*url*, *file\_name*):

response = urllib.request.urlopen(*url*)

data = response.read()

with open(*file\_name*, 'wb') as file:

file.write(data)

urls = [

'http://example.com/file1.txt',

'http://example.com/file2.txt',

'http://example.com/file3.txt'

]

with ThreadPoolExecutor() as executor:

for i, url in enumerate(urls):

executor.submit(download\_file, url, f'file{i}.txt')

*```*

### Question 2:

# - \*\*Topics\*\*: Decorators, Function Wrapping, Debugging

# - \*\*Scenario\*\*: A Python developer is trying to create a decorator that will log the execution time of a function.

# However, the decorator is not working as expected.

# - \*\*Code\*\*:

# ```python

import time

def timer(*func*):

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

start = time.time()

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

end = time.time()

print(f'{*func*.\_\_name\_\_} ran in: {end - start} sec')

return *func*

@timer

def long\_running\_function():

time.sleep(5)

long\_running\_function()

*```*

# - \*\*Question\*\*: The timer decorator is not working as expected. What is wrong with this code and how can it be fixed?

# - \*\*Explanation\*\*: The timer function is not returning the wrapper function, it is returning the original function instead.

# This means the timing code will never be run.

# - \*\*Answer\*\*:

# ```python

import time

def timer(*func*):

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

start = time.time()

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

end = time.time()

print(f'{*func*.\_\_name\_\_} ran in: {end - start} sec')

return wrapper

@timer

def long\_running\_function():

time.sleep(5)

long\_running\_function()

*```*

### Question 3:

# - \*\*Topics\*\*: Generators, Memory Efficiency

# - \*\*Scenario\*\*: A data engineer is trying to process a large file line by line using a generator but the code is consuming

# too much memory.

# - \*\*Code\*\*:

# ```python

def read\_file(*file\_name*):

with open(*file\_name*, 'r') as file:

data = file.readlines()

for line in data:

yield line.strip()

for line in read\_file('large\_file.txt'):

print(line)

*```*

# - \*\*Question\*\*: The code is not as memory efficient as expected. What is wrong with this code and how can it be fixed?

# - \*\*Explanation\*\*: The readlines() method loads the entire file into memory. To make it more memory efficient, the lines

# should be read one at a time.

# - \*\*Answer\*\*:

# ```python

def read\_file(*file\_name*):

with open(*file\_name*, 'r') as file:

for line in file:

yield line.strip()

for line in read\_file('large\_file.txt'):

print(line)

*```*

### Question 4:

# - \*\*Topics\*\*: Error Handling, Exception Propagation

# - \*\*Scenario\*\*: A Python developer is trying to handle exceptions in a nested function call. However, the exceptions

# are not being caught as expected.

# - \*\*Code\*\*:

# ```python

def inner\_function():

raise ValueError('An error occurred')

def outer\_function():

try:

inner\_function()

except Exception:

print('Caught exception')

outer\_function()

*```*

# - \*\*Question\*\*: The exception is not being caught as expected. What is wrong with this code and how can it be fixed?

# - \*\*Explanation\*\*: The exception is being raised in the inner\_function but it is not being caught because the inner\_function

# is not called inside the try block.

# - \*\*Answer\*\*:

# ```python

def inner\_function():

raise ValueError('An error occurred')

def outer\_function():

try:

inner\_function()

except ValueError as e:

print('Caught exception:', e)

outer\_function()

*```*

### Question 5:

# - \*\*Topics\*\*: Python's Object Model, Mutable Default Arguments

# - \*\*Scenario\*\*: A software engineer is trying to create a class to model a tree data structure. However, the behavior

# of the code is not as expected.

# - \*\*Code\*\*:

# ```python

class TreeNode:

def \_\_init\_\_(*self*, *value*, *children*=[]):

*self*.value = *value*

*self*.children = *children*

a = TreeNode(1)

b = TreeNode(2)

a.children.append(b)

c = TreeNode(3)

print(c.children)

*```*

# - \*\*Question\*\*: The code is not behaving as expected. What is wrong with this code and how can it be fixed?

# - \*\*Explanation\*\*: The problem is that in Python, default arguments are evaluated once when the function is defined,

# not each time the function is called. This means that if the default value is a mutable object and you change it, the change

# will persist in subsequent function calls.

# - \*\*Answer\*\*:

# ```python

class TreeNode:

def \_\_init\_\_(*self*, *value*, *children*=None):

*self*.value = *value*

*self*.children = *children* if *children* is not None else []

a = TreeNode(1)

b = TreeNode(2)

a.children.append(b)

c = TreeNode(3)

print(c.children)

*```*

## prompt9\_response part2, temp = 0.7, gpt 4(also good I think)

### Question 1

# 0 - Topics : Multithreading and Exception Handling in Python

# 1 - Scenario : An application that downloads multiple files concurrently using multithreading.

# 2 - Code :

# ```python

import threading

import urllib.request

class Downloader(*threading*.Thread):

def \_\_init\_\_(*self*, *url*):

threading.Thread.\_\_init\_\_(*self*)

*self*.url = *url*

def run(*self*):

urllib.request.urlretrieve(*self*.url)

urls = ["http://example.com/file1.zip", "http://example.com/file2.zip", "http://example.com/file3.zip"]

for url in urls:

t = Downloader(url)

t.start()

*```*

# 3 - Question : The above code is supposed to download multiple files concurrently, however, the code crashes when it encounters

# a non-existing URL. What is wrong?

# 4 - Explanation : The code does not handle exceptions that might occur during the download process. If the URL is not valid

# or the file does not exist, a URLError will be raised, which is not caught, causing the thread and subsequently the whole

# program to crash.

# 5 - Answer :

# ```python

import threading

import urllib.request

class Downloader(*threading*.Thread):

def \_\_init\_\_(*self*, *url*):

threading.Thread.\_\_init\_\_(*self*)

*self*.url = *url*

def run(*self*):

try:

urllib.request.urlretrieve(*self*.url)

except urllib.error.URLError:

print(f"Failed to download {*self*.url}")

urls = ["http://example.com/file1.zip", "http://example.com/file2.zip", "http://example.com/file3.zip"]

for url in urls:

t = Downloader(url)

t.start()

*```*

# In the corrected code, we handle the URLError exception that might occur while trying to download the file.

### Question 2

# 0 - Topics : Python Decorators and Generator Functions

# 1 - Scenario : Caching the results of a function that yields multiple values using a decorator

# 2 - Code :

# ```python

def cache(*func*):

results = {}

def wrapper(\**args*):

if *args* not in results:

results[*args*] = *func*(\**args*)

return results[*args*]

return wrapper

@cache

def fib(*n*):

a, b = 0, 1

for \_ in range(*n*):

yield a

a, b = b, a + b

for i in fib(10):

print(i)

*```*

# 3 - Question : The above code is supposed to cache the results of the Fibonacci generator function, but it doesn't work

# as expected. What is the issue?

# 4 - Explanation : The issue is that the `cache` decorator is trying to cache the generator object itself, not the

# it produces. When the generator function is called again with the same arguments, it returns the cached generator object,

# which is already exhausted, thus producing no values.

# 5 - Answer :

# ```python

def cache(*func*):

results = {}

def wrapper(\**args*):

if *args* not in results:

results[*args*] = list(*func*(\**args*))

return results[*args*]

return wrapper

@cache

def fib(*n*):

a, b = 0, 1

for \_ in range(*n*):

yield a

a, b = b, a + b

for i in fib(10):

print(i)

*```*

# In the corrected code, we convert the generator to a list before caching it, thus storing the actual values it produces.

# ---

### Question 3

0 - Topics : Python metaclasses

# 1 - Scenario : Creating a Singleton class using a metaclass

# 2 - Code :

# ```python

class SingletonMeta(type):

\_instances = {}

def \_\_call\_\_(*cls*, \**args*, \*\**kwargs*):

if *cls* not in *cls*.\_instances:

*cls*.\_instances[*cls*] = super().\_\_call\_\_(\**args*, \*\**kwargs*)

return *cls*.\_instances[*cls*]

class Singleton(*metaclass*=SingletonMeta):

def \_\_init\_\_(*self*, *value*):

*self*.value = *value*

s1 = Singleton(1)

s2 = Singleton(2)

print(s1.value)

print(s2.value)

*```*

# 3 - Question : The above code is supposed to create a Singleton class, but the value of `s1` changes after creating `s2`.

# What is wrong?

# 4 - Explanation : The problem is that the `\_\_init\_\_` method is still called every time the `Singleton` class is instantiated,

# even though the metaclass prevents multiple instances from being created. This leads to the `value` attribute being overwritten

# every time a new `Singleton` is created.

# 5 - Answer :

# ```python

class SingletonMeta(type):

\_instances = {}

def \_\_call\_\_(*cls*, \**args*, \*\**kwargs*):

if *cls* not in *cls*.\_instances:

instance = super().\_\_call\_\_(\**args*, \*\**kwargs*)

*cls*.\_instances[*cls*] = instance

return *cls*.\_instances[*cls*]

class Singleton(*metaclass*=SingletonMeta):

def \_\_init\_\_(*self*, *value*):

if not hasattr(*self*, "value"):

*self*.value = *value*

s1 = Singleton(1)

s2 = Singleton(2)

print(s1.value)

print(s2.value)

*```*

# In the corrected code, we only initialize the `value` attribute if it has not been initialized before, preserving

# state of the Singleton instance.

## (show this,last one not compiling) prompt12\_response, gpt4, temperature 0.7

### Question 1

# 0 - Topics: Memory Management, Garbage Collection

# 1 - Scenario: A large application is experiencing memory leaks and overall slowdowns. The issue seems to be related

# to a specific class that is heavily used throughout the app.

# 2 - Code:

# ```python

class ExpensiveObject:

def \_\_init\_\_(*self*, *data*):

*self*.data = *data*

*self*.other\_objects = []

def add\_other\_object(*self*, *obj*):

*self*.other\_objects.append(*obj*)

# Simulate heavy use of ExpensiveObject

objects = []

for i in range(100000):

obj = ExpensiveObject(i)

if objects:

obj.add\_other\_object(objects[0])

objects.append(obj)

del objects

*```*

# 3 - Question: Despite the deletion of the `objects` list, the memory used by the program does not decrease as expected.

# Identify and fix the issue.

# 4 - Explanation: The problem here is a circular reference. Each `ExpensiveObject` references the first object in the `objects`

# list, creating a reference cycle that the Python garbage collector can't handle automatically.

# 5 - Answer:

# ```python

import weakref

class ExpensiveObject:

def \_\_init\_\_(*self*, *data*):

*self*.data = *data*

*self*.other\_objects = []

def add\_other\_object(*self*, *obj*):

*self*.other\_objects.append(weakref.ref(*obj*))

# Simulate heavy use of ExpensiveObject

objects = []

for i in range(100000):

obj = ExpensiveObject(i)

if objects:

obj.add\_other\_object(objects[0])

objects.append(obj)

del objects

*```*

### Question 2

# 0 - Topics: Decorators, Function Wrapping

# 1 - Scenario: You are using a decorator to measure the execution time of various functions in your codebase. However, the

# decorated functions seem to lose their original `\_\_name\_\_` and `\_\_doc\_\_` attributes.

# 2 - Code:

# ```python

import time

def timer\_decorator(*func*):

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

start = time.time()

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

end = time.time()

print(f'Function {*func*.\_\_name\_\_} took {end - start} seconds')

return result

return wrapper

@timer\_decorator

def expensive\_function():

"""This is a very expensive function"""

time.sleep(2)

expensive\_function()

print(expensive\_function.\_\_name\_\_)

print(expensive\_function.\_\_doc\_\_)

*```*

# 3 - Question: The `\_\_name\_\_` and `\_\_doc\_\_` attributes of `expensive\_function` are not what they're expected to be. Identify

# and fix the issue.

# 4 - Explanation: The issue is that the `timer\_decorator` function returns `wrapper`, which overwrites the `expensive\_function`.

# The `\_\_name\_\_` and `\_\_doc\_\_` attributes of the original function are lost in the process.

# 5 - Answer:

# ```python

import time

import functools

def timer\_decorator(*func*):

@functools.wraps(*func*)

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

start = time.time()

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

end = time.time()

print(f'Function {*func*.\_\_name\_\_} took {end - start} seconds')

return result

return wrapper

@timer\_decorator

def expensive\_function():

"""This is a very expensive function"""

time.sleep(2)

expensive\_function()

print(expensive\_function.\_\_name\_\_)

print(expensive\_function.\_\_doc\_\_)

*```*

### Question 3

# 0 - Topics: Exception Handling, Context Managers

# 1 - Scenario: You are developing a library for working with files. You have a `FileReader` class that is intended to be used

# as a context manager. However, exceptions raised within the `with` block are not being handled as expected.

# 2 - Code:

# ```python

class FileReader:

def \_\_init\_\_(*self*, *filename*):

*self*.filename = *filename*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, 'r')

return *self*.file

def \_\_exit\_\_(*self*, *exc\_type*, *exc\_val*, *exc\_tb*):

*self*.file.close()

try:

with FileReader('non\_existent\_file.txt') as f:

print(f.read())

except:

print("An error occurred")

*```*

# 3 - Question: The `FileReader` class is not properly handling exceptions raised within the `with` block. Identify

# and fix the issue.

# 4 - Explanation: The `\_\_exit\_\_` method in a context manager is responsible for handling exceptions. However, in its

# current state, the `FileReader` class does not do anything with the exception information it receives.

# 5 - Answer:

# ```python

class FileReader:

def \_\_init\_\_(*self*, *filename*):

*self*.filename = *filename*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, 'r')

return *self*.file

def \_\_exit\_\_(*self*, *exc\_type*, *exc\_val*, *exc\_tb*):

*self*.file.close()

if *exc\_type* is not None:

print("An error occurred")

return True

try:

with FileReader('non\_existent\_file.txt') as f:

print(f.read())

except:

pass

*```*

### question 4

# 0 - Topics: Monkey Patching, Dynamic Attributes

# 1 - Scenario: You are trying to add a method to an existing class at runtime, but the method doesn't seem to work as expected.

# 2 - Code:

# ```python

class MyClass:

def \_\_init\_\_(*self*, *value*):

*self*.value = *value*

def get\_value(*self*):

return *self*.value

MyClass.get\_value = get\_value

obj = MyClass(42)

print(obj.get\_value())

*```*

# 3 - Question: The `get\_value` method is not working as expected when added to `MyClass` at runtime. Identify and fix the issue.

# 4 - Explanation: The problem is that the `get\_value` method is not bound to the `MyClass` instances. When a function is added

# to a class at runtime, it doesn't automatically become a bound method.

# 5 - Answer:

# ```python

class MyClass:

def \_\_init\_\_(*self*, *value*):

*self*.value = *value*

def get\_value(*self*):

return *self*.value

MyClass.get\_value = get\_value.\_\_get\_\_(None, MyClass)

obj = MyClass(42)

print(obj.get\_value())

*```*

### Question 5

# 0 - Topics: Asynchronous Programming, Coroutines

# 1 - Scenario: A piece of async code is not executing as expected, with certain parts of the code never executing at all.

# 2 - Code:

# ```python

import asyncio

async def my\_coroutine():

print("Coroutine started")

await asyncio.sleep(1)

print("Coroutine finished")

def main():

print("Main started")

my\_coroutine()

print("Main finished")

main()

*```*

# 3 - Question: The `my\_coroutine` function does not seem to execute at all. Identify and fix the issue.

# 4 - Explanation: The problem is that coroutines need to be scheduled to run on the event loop. Simply calling a

# coroutine function does not execute the coroutine.

# 5 - Answer:

# ```python

import asyncio

async def my\_coroutine():

print("Coroutine started")

await asyncio.sleep(1)

print("Coroutine finished")

def main():

print("Main started")

asyncio.run(my\_coroutine())

print("Main finished")

main()

*```*

## (silly) prompt13\_response , gpt4, temp = 0.7 (have to check compiling)

### Question 1

# - Topics: Metaclasses, Class Creation

# 1 - Scenario: You are trying to create a new class with a metaclass but you are getting an error.

# 2 - Code:

# ```python

class Meta(type):

def \_\_new\_\_(*cls*, *name*, *bases*, *attrs*):

*attrs*['added'] = True

return super().\_\_new\_\_(*cls*, *name*, *bases*, *attrs*)

class MyClass:

\_\_metaclass\_\_ = Meta

pass

print(MyClass.added)

*```*

# 3 - Question: The above code is supposed to add an `added` attribute to `MyClass` through a metaclass, but it's not working

# as expected. What's wrong?

# 4 - Explanation: The issue is that Python 3 has changed the way metaclasses are specified. Instead of adding a `\_\_metaclass\_\_`

# attribute, you should pass it as a keyword argument in the class definition.

# 5 - Answer:

# ```python

class Meta(type):

def \_\_new\_\_(*cls*, *name*, *bases*, *attrs*):

*attrs*['added'] = True

return super().\_\_new\_\_(*cls*, *name*, *bases*, *attrs*)

class MyClass(*metaclass*=Meta):

pass

print(MyClass.added)

*```*

### Question 2

# 2 - Topics: Decorators, Function Scope

# 1 - Scenario: You are trying to use a decorator to add a pre-processing step to a function, but the decorator is not working

# as expected.

# 2 - Code:

# ```python

def my\_decorator(*func*):

def wrapper():

print("Before function call")

*func*

print("After function call")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

*```*

# 3 - Question: The `say\_hello` function is not printing "Hello!" as expected. What's wrong with the code?

# 4 - Explanation: The issue is that the decorator's wrapper function is not calling the decorated function. It's just

# referencing it. To call a function, you need to include parentheses after its name.

# 5 - Answer:

# ```python

def my\_decorator(*func*):

def wrapper():

print("Before function call")

*func*()

print("After function call")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

say\_hello()

*```*

### Question 3

# 3 - Topics: Generators, Yield Statement

# 1 - Scenario: You are trying to create a generator function that yields a sequence of numbers, but it's not producing any output.

# 2 - Code:

# ```python

def count\_up\_to(*n*):

i = 1

while i <= *n*:

yield i

i += 1

for number in count\_up\_to(5):

print(number)

*```*

3 - Question: The *`count\_up\_to`* function is not printing the numbers 1 through 5 as expected. What's wrong with the code?

4 - Explanation: The issue is that the increment operation *`i += 1`* is outside the while loop. This means *`i`* is always 1 and the loop never ends.

5 - Answer:

*```python*

def count\_up\_to(*n*):

i = 1

while i <= *n*:

yield i

i += 1

for number in count\_up\_to(5):

print(number)

*```*

### Question 4

# 4 - Topics: Context Managers, Exception Handling

# 1 - Scenario: You are trying to use a context manager to handle file operations, but the file isn't being closed properly.

# 2 - Code:

# ```python

class ManagedFile:

def \_\_init\_\_(*self*, *filename*):

*self*.filename = *filename*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, 'r')

return *self*.file

def \_\_exit\_\_(*self*):

*self*.file.close()

with ManagedFile('hello.txt') as f:

contents = f.read()

print(f.closed)

*```*

# 3 - Question: The `ManagedFile` context manager is supposed to close the file after the `with` block, but the `f.closed`

# check is returning `False`. What's wrong with the code?

# 4 - Explanation: The issue is that the `\_\_exit\_\_` method doesn't handle exceptions. If an exception is raised within the

# `with` block, the `\_\_exit\_\_` method won't be called.

# 5 - Answer:

# ```python

class ManagedFile:

def \_\_init\_\_(*self*, *filename*):

*self*.filename = *filename*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, 'r')

return *self*.file

def \_\_exit\_\_(*self*, *exc\_type*, *exc\_val*, *exc\_tb*):

if *self*.file:

*self*.file.close()

with ManagedFile('hello.txt') as f:

contents = f.read()

print(f.closed)

*```*

### Question 5

# 5 - Topics: Python Descriptors, Object Attributes

# 1 - Scenario: You are trying to use a descriptor class to manage an attribute, but it's not working as expected.

# 2 - Code:

# ```python

class Descriptor:

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

*self*.value = 0

def \_\_get\_\_(*self*, *instance*, *owner*):

return *self*.value

def \_\_set\_\_(*self*, *instance*, *value*):

if *value* < 0:

raise ValueError("Cannot be negative.")

*self*.value = *value*

class MyClass:

attr = Descriptor()

obj = MyClass()

obj.attr = -10

*```*

# 3 - Question: The `Descriptor` class is supposed to prevent `attr` from being set to a negative value, but it's not raising an exception as expected. What's wrong with the code?

# 4 - Explanation: The issue is that the `\_\_set\_\_` method in the `Descriptor` class is not being called when setting `attr` to a negative value. This is because `attr` is not defined as a property in the `MyClass` class.

# 5 - Answer:

# ```python

class Descriptor:

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

*self*.value = 0

def \_\_get\_\_(*self*, *instance*, *owner*):

return *self*.value

def \_\_set\_\_(*self*, *instance*, *value*):

if *value* < 0:

raise ValueError("Cannot be negative.")

*self*.value = *value*

class MyClass:

attr = property(Descriptor())

obj = MyClass()

obj.attr = -10

*```*

## prompt14\_response , gpt4, temp = 0.7

### Question 1

# 0 - Topics: Decorators, Closures

# 1 - Scenario: You are implementing a simple caching mechanism for a function that calculates factorial of a number.

# You decide to use decorators for this task.

# 2 - Code:

# ```python

def memoize(*func*):

cache = {}

def wrapper(*n*):

if *n* not in cache:

cache[*n*] = *func*(*n*)

return cache[*n*]

return *func*

def factorial(*n*):

if *n* == 0:

return 1

else:

return *n* \* factorial(*n*-1)

factorial = memoize(factorial)

print(factorial(5))

*```*

# 3 - Question: The code is supposed to memoize the results of the factorial function, but it seems the memoization is not working

# as expected. Can you identify what's wrong?

# 4 - Explanation: The problem is in the 'memoize' decorator function. Instead of returning the 'wrapper' function, it's returning

# the original 'func'. Hence, the memoization is not working.

# 5 - Answer:

# ```python

def memoize(*func*):

cache = {}

def wrapper(*n*):

if *n* not in cache:

cache[*n*] = *func*(*n*)

return cache[*n*]

return wrapper

def factorial(*n*):

if *n* == 0:

return 1

else:

return *n* \* factorial(*n*-1)

factorial = memoize(factorial)

print(factorial(5))

*```*

### Question 2.

# 0 - Topics: AsyncIO, Concurrency

# 1 - Scenario: You are developing a program that fetches data from multiple URLs concurrently using AsyncIO.

# 2 - Code:

# ```python

import asyncio

import aiohttp

async def fetch(*session*, *url*):

async with *session*.get(*url*) as response:

return await response.text()

async def main():

urls = ['http://python.org', 'http://google.com']

async with aiohttp.ClientSession() as session:

for url in urls:

html = await fetch(session, url)

print(html)

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(main())

*```*

# 3 - Question: The program is supposed to fetch data from multiple URLs concurrently, but it's fetching them sequentially.

# Can you identify the problem?

# 4 - Explanation: The problem is in the 'main' function. Instead of waiting for each 'fetch' function to complete, we should

# gather all the tasks and then await them.

# 5 - Answer:

# ```python

import asyncio

import aiohttp

async def fetch(*session*, *url*):

async with *session*.get(*url*) as response:

return await response.text()

async def main():

urls = ['http://python.org', 'http://google.com']

async with aiohttp.ClientSession() as session:

tasks = [fetch(session, url) for url in urls]

htmls = await asyncio.gather(\*tasks)

for html in htmls:

print(html)

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(main())

*```*

### Question 3.

# 0 - Topics: Context Managers, Error Handling

# 1 - Scenario: You are implementing a context manager to handle file operations.

# 2 - Code:

# ```python

class File:

def \_\_init\_\_(*self*, *filename*, *mode*):

*self*.filename = *filename*

*self*.mode = *mode*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, *self*.mode)

return *self*.file

def \_\_exit\_\_(*self*, *type*, *value*, *traceback*):

*self*.file.close()

with File('test.txt', 'w') as f:

f.undefined\_function('Hello, world!')

*```*

# 3 - Question: The code is raising an unexpected exception. Can you identify and handle the exception properly?

# 4 - Explanation: The 'undefined\_function' is not a method of the file object, causing an AttributeError. The 'exit' method

# of the context manager should handle this exception.

# 5 - Answer:

# ```python

class File:

def \_\_init\_\_(*self*, *filename*, *mode*):

*self*.filename = *filename*

*self*.mode = *mode*

def \_\_enter\_\_(*self*):

*self*.file = open(*self*.filename, *self*.mode)

return *self*.file

def \_\_exit\_\_(*self*, *type*, *value*, *traceback*):

if *type* is AttributeError:

print(f"Error: {*value*}")

return True

*self*.file.close()

with File('test.txt', 'w') as f:

f.undefined\_function('Hello, world!')

*```*

### Question 4

# 0 - Topics: Metaclasses, Class Creation

# 1 - Scenario: You are creating a metaclass that ensures all classes using it have a 'greet' method.

# 2 - Code:

# ```python

class Meta(type):

def \_\_init\_\_(*cls*, *name*, *bases*, *attrs*):

if 'greet' not in *attrs*:

raise TypeError("Bad user class")

super().\_\_init\_\_(*name*, *bases*, *attrs*)

class MyClass(*metaclass*=Meta):

def hello(*self*):

pass

*```*

# 3 - Question: The code is supposed to raise a TypeError if a class using the metaclass doesn't implement the 'greet' method,

# but it's not working as expected. Can you identify the problem?

# 4 - Explanation: The problem is in the metaclass 'Meta'. The 'init' method of a metaclass is called after the class is created,

# so it's too late to raise a TypeError. Instead, we should override the 'new' method of the metaclass.

# 5 - Answer:

# ```python

class Meta(type):

def \_\_new\_\_(*mcls*, *name*, *bases*, *attrs*):

if 'greet' not in *attrs*:

raise TypeError("Bad user class")

return super().\_\_new\_\_(*mcls*, *name*, *bases*, *attrs*)

class MyClass(*metaclass*=Meta):

def greet(*self*):

pass

*```*

### Question 5

# 0 - Topics: Generators, Yield From

# 1 - Scenario: You are implementing a recursive function to flatten a nested list using generators.

# 2 - Code:

# ```python

def flatten(*lst*):

for x in *lst*:

if isinstance(x, list):

flatten(x)

else:

yield x

print(list(flatten([1, [2, [3, 4], 5]])))

*```*

# 3 - Question: The code is supposed to flatten a nested list, but it's not working as expected. Can you identify the problem?

# 4 - Explanation: The problem is in the 'flatten' function. When we recursively call 'flatten' on a nested list, we need to yield

# from it.

# 5 - Answer:

# ```python

def flatten(*lst*):

for x in *lst*:

if isinstance(x, list):

yield from flatten(x)

else:

yield x

print(list(flatten([1, [2, [3, 4], 5]])))

*```*

## # prompt15\_response, gpt4, temp=0.7

### Question 1

# 0 - Topic: Memory Management, Garbage Collection

# 1 - Scenario: You are developing a large-scale application that involves creating and deleting a large number of objects.

# You are noticing that the application's memory usage is continuously increasing.

# 2 - Code:

# ```python

import gc

class BigObject:

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

*self*.data = 'x' \* 1000000

objects = []

for \_ in range(10000):

objects.append(BigObject())

if len(objects) > 1000:

del objects[0]

*```*

# 3 - Question: Despite deleting unused objects, the program's memory usage is continuously increasing. Can you identify the issue

# and suggest a solution?

# 4 - Explanation: In Python, deleting an object or removing a reference to it does not immediately free up its memory.

# It only marks it as ready for garbage collection. The garbage collector only runs periodically to free up memory from

# unreferenced objects. If objects are being created and deleted at a high rate, this can lead to increased memory usage.

# 5 - Answer:

# ```python

import gc

class BigObject:

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

*self*.data = 'x' \* 1000000

objects = []

for \_ in range(10000):

objects.append(BigObject())

if len(objects) > 1000:

del objects[0]

gc.collect() # Force a garbage collection cycle

*```*

### 

### Question 2

# 0 - Topics: Exception Handling, Asynchronous Programming

# 1 - Scenario: You are developing an asynchronous application that makes HTTP requests and you want to handle HTTP errors.

# 2 - Code:

# ```python

import asyncio

import aiohttp

async def get(*url*):

async with aiohttp.ClientSession() as session:

async with session.get(*url*) as resp:

return await resp.text()

loop = asyncio.get\_event\_loop()

try:

loop.run\_until\_complete(get('http://example.com'))

except aiohttp.ClientResponseError:

print("HTTP error occurred")

*```*

# 3 - Question: The above code is supposed to catch HTTP errors, but it doesn't seem to be working. What is wrong?

# 4 - Explanation: The issue here is that exceptions raised inside an asyncio event loop are not propagated to the surrounding

# context. They need to be handled inside the event loop.

# 5 - Answer:

# ```python

import asyncio

import aiohttp

async def get(*url*):

try:

async with aiohttp.ClientSession() as session:

async with session.get(*url*) as resp:

return await resp.text()

except aiohttp.ClientResponseError:

print("HTTP error occurred")

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(get('http://example.com'))

*```*

### Question 3

# 0 - Topic: Metaclasses, Class Creation

# 1 - Scenario: You are creating a class dynamically using a metaclass, but the created class is not behaving as expected.

# 2 - Code:

# ```pythons

class Meta(type):

def \_\_new\_\_(*mcs*, *name*, *bases*, *attrs*):

*attrs*['greeting'] = 'Hello, world!'

return super().\_\_new\_\_(*mcs*, *name*, *bases*, *attrs*)

MyClass = Meta('MyClass', (), {})

print(MyClass.greeting) # Should print 'Hello, world!'

*```*

# 3 - Question: The MyClass object does not have the greeting attribute as expected. Can you identify the issue and suggest a

# solution?

# 4 - Explanation: The problem here is that metaclasses' `\_\_new\_\_` method doesn't add attributes to the class, it adds them to

# the instances of the class.

# 5 - Answer:

# ```python

class Meta(type):

def \_\_init\_\_(*cls*, *name*, *bases*, *attrs*):

*attrs*['greeting'] = 'Hello, world!'

super().\_\_init\_\_(*name*, *bases*, *attrs*)

MyClass = Meta('MyClass', (), {})

print(MyClass.greeting) # Should print 'Hello, world!'

*```*

### Question 4

# 0 - Topics: Decorators, Function Wrapping

# 1 - Scenario: You are using a decorator to add logging to a function, but the function's name and docstring are not preserved.

# 2 - Code:

# ```python

def log\_calls(*func*):

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

print(f"Calling {*func*.\_\_name\_\_}")

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

return wrapper

@log\_calls

def greet(*name*):

"""Greets the specified person"""

print(f"Hello, {*name*}!")

print(greet.\_\_name\_\_) # Should print 'greet'

print(greet.\_\_doc\_\_) # Should print 'Greets the specified person'

*```*

# 3 - Question: The name and docstring of the greet function are not preserved when it's decorated. Can you identify the issue

# and suggest a solution?

# 4 - Explanation: The issue here is that the wrapper function is replacing the original function, and the original function's

# name and docstring are not automatically copied over. You can use the `functools.wraps` decorator to preserve these attributes.

# 5 - Answer:

# ```python

import functools

def log\_calls(*func*):

@functools.wraps(*func*)

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

print(f"Calling {*func*.\_\_name\_\_}")

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

return wrapper

@log\_calls

def greet(*name*):

"""Greets the specified person"""

print(f"Hello, {*name*}!")

print(greet.\_\_name\_\_) # Should print 'greet'

print(greet.\_\_doc\_\_) # Should print 'Greets the specified person'

*```*

### Question 5

# 0 - Topic: Generators, Iteration

# 1 - Scenario: You are writing a generator function to yield items from a nested data structure, but it's not yielding all

# the items.

# 2 - Code:

# ```python

def flatten(*lst*):

for item in *lst*:

if isinstance(item, list):

flatten(item)

else:

yield item

for item in flatten([1, [2, 3], 4]):

print(item) # Should print 1, 2, 3, 4

*```*

# 3 - Question: The flatten function is not yielding all the items from the nested list. Can you identify the issue and suggest

# a solution?

# 4 - Explanation: The issue here is that the recursive call to flatten is not connected to the outer generator function.

# The items yielded by the recursive call are not being yielded by the outer function.

# 5 - Answer:

# ```python

def flatten(*lst*):

for item in *lst*:

if isinstance(item, list):

yield from flatten(item)

else:

yield item

for item in flatten([1, [2, 3], 4]):

print(item) # Should print 1, 2, 3, 4

*```*

# PROMPTS: SYSTEM+USER

## Sys\_content\_1

= """

You are an automated system that generates 2 types of Python coding related questions. These are:

``` 1 - Debugging : These coding questions require a person solving it to, remove bugs in the code by modifying parts of the buggy code

2 - Efficiency Optimizing : These coding questions require a person solving it to, make changes in the code that make the code run faster,save memory, or other such things that make the code work in a more efficient manner

```

You generate questions with the following qualities:

``` 1 - conceptual : questions that actually test the solver's conceptual understanding. The question should be such that it can only be solved if and only if the solver actually knows and understands the concepts being tested.

2 - On Job Mistakes : questions are relevant and only test the solver in ways relevant to on the job mistakes. It tests the mistakes made in the job and not irrelevant ways or skills that are not really used on the job. They key word are 'relevant' and 'on the job'

3 High level - The questions must be used to weed out non-senior developers. So questions must be intended for the experienced and senior level developers. The questions must be of a high level, and not just basic concepts.

4 - In the case the user does not specify topics, you should generate questions by yourself choosing relevant topics to fit the difficult level as described by the user. If user does not describe difficulty level, from the context of the prompt predict the difficulty level being asked and proceed accordingly

```

The questions you generate have the following format :

``` 0 - Concept : what is the concept being used or combination of concepts being used. (if there are multiple - list them, else just state it). These concepts should be python concepts (example of concepts: syntax,generators,metaprogramming,etc)

1 - Scenario : This section must have the following:

```1 - Real world scenario

2 - An explanation of the code being used.

Note:

- The scenario does not need to always have a senior developer in the scenario description.

- It just needs to be a description so that the code and the purpose of solving it does not feel irrelevant```

2 - Code Snippet: This section must have the following:

```1 - The wrong (inefficient/buggy) code.

2 - The code must be around 30-50 lines

Note: - The question must test some concept/concepts that is relevant for on the job purposes, and not just some very basic concepts and topics that anyone who did the basics can also solve```

3 - Question : Must explain that there is some issue in the code - either efficiency or a bug or both, and the user must solve it.

4 - Explanation : This should have an explanation regarding what was wrong in the code (as mentioned in part 2), what concepts were being tested, and idea of the changes that must be made, and why it is important to do so

5 - Answer : Implementing the explanation, this part must have the corrected code.```

The list of topics(these are only examples, it is not exhaustive), organized by difficulty level is:

``` 1 - Easy:

1. Syntax Errors: Common errors such as missing colons, parentheses, or indentation issues.

2. Variable Naming: Misnaming variables or using reserved keywords.

3. Import Errors: Incorrect module names or paths.

4. Data Type Misuse: Attempting operations incompatible with data types.

5. Print Debugging: Misuse or omission of print statements for debugging purposes.

6. String Manipulation Errors: Mishandling of string operations like concatenation or slicing.

7. List Manipulation Errors: Incorrect usage of list methods like append, extend, or pop.

8. Basic Control Structures: Misunderstanding of loops (for, while) and conditional statements (if-else).

9. Function Definition and Invocation: Errors in defining or calling functions.

10. Basic File I/O: Mishandling file operations like opening, reading, or writing.```

2 - Medium:

```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.```

3 - Hard: ```

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

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

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

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

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

6. Generators

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

```

In the case that user gives invalid python topics, or invalid debugging question requests, say "invalid" and give one line of explanation for why it is invalid.

"""

## user\_prompt1

= """

Can you give me one hard debugging question for senior level programmers, it could be on any topic that is important at that level. but it has to be conceptually useful, and not just for loops or basic conversions

"""

## user\_prompt2

= """

Can you give me one hard debugging question that is not optimization/efficiency based.

It should be for senior level programmers, it could be on any topic that is important at that level. but it

has to be conceptually useful, and not just for loops or basic conversions

"""

# RESULT: SYSTEM+USER

## Result: prompt1, 3.5 turbo, temperature = 0,

# user\_response1 :

# 0 - Concept: Multithreading, Concurrency

# 1 - Scenario:

# 1 - Real world scenario:

# In a high-performance server application, multiple threads are used to handle incoming client requests

# concurrently to improve responsiveness.

# 2 - An explanation of the code being used:

# The code snippet provided below is a simplified version of a multithreaded server application

# that processes client requests. However, there is a bug in the code that is causing unexpected behavior and

# potential race conditions.

# 2 - Code Snippet:

*```python*

import threading

class Server:

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

*self*.lock = threading.Lock()

*self*.data = []

def process\_request(*self*, *request*):

with *self*.lock:

*self*.data.append(*request*)

print(f"Request {*request*} processed.")

def run\_server(*self*):

threads = []

for i in range(5):

t = threading.Thread(*target*=*self*.process\_request, *args*=(i,))

threads.append(t)

t.start()

for t in threads:

t.join()

if \_\_name\_\_ == "\_\_main\_\_":

server = Server()

server.run\_server()

*```*

# 3 - Question:

# Identify and fix the issue in the provided code that is causing unexpected behavior and potential race conditions

# in the multithreaded server application.

# 4 - Explanation:

# The issue in the code is related to the shared data structure `self.data` being accessed and modified concurrently

# by multiple threads without proper synchronization. This can lead to race conditions where threads interfere with

# each other's operations, causing unexpected behavior or data corruption. By using a lock (`self.lock`) to protect

# the critical section where `self.data` is modified, we can ensure thread safety and prevent race conditions.

# 5 - Answer:

*```python*

import threading

class Server:

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

*self*.lock = threading.Lock()

*self*.data = []

def process\_request(*self*, *request*):

with *self*.lock:

*self*.data.append(*request*)

print(f"Request {*request*} processed.")

def run\_server(*self*):

threads = []

for i in range(5):

t = threading.Thread(*target*=*self*.process\_request, *args*=(i,))

threads.append(t)

t.start()

for t in threads:

t.join()

if \_\_name\_\_ == "\_\_main\_\_":

server = Server()

server.run\_server()

*```*

# In the corrected code, a lock is used to synchronize access to the shared data structure `self.data`, ensuring thread safety and preventing race conditions.

## Result: prompt2\_response,4ogpt, temp = 0.9,

# \*\*Concept\*\* : Closures and Decorators

# \*\*Scenario\*\*:

# In a large-scale application development project, a senior developer is working on implementing

# a system that requires the extensive use of closures and decorators for handling authentication and authorization

# mechanisms. The developer is tasked with ensuring that only authorized users can access certain parts of the

# system based on their roles.

# \*\*Code Snippet\*\*:

*```python*

def check\_permission(*role*):

def wrapper(*func*):

if *role* == 'admin':

print("You have admin rights")

return *func*

else:

print("Access denied")

return wrapper

@check\_permission('user')

def view\_dashboard():

print("Viewing dashboard")

view\_dashboard()

*```*

# \*\*Question\*\*:

# There seems to be an issue in the given code related to permissions based on roles. Can you identify

# and rectify the problem?

# \*\*Explanation\*\*:

# The issue in the code lies in the decorator `check\_permission`. Currently, it directly prints whether the user

# has admin rights or access is denied based on the role provided. This approach is not handling the decorator's

# intended functionality properly. Decorators are commonly used to modify or enhance the behavior of functions.

# In this case, the decorator should either call the original function `func` or prevent its execution based

# on the role. The current implementation only prints messages but does not control the function's execution

# flow correctly.

# \*\*Answer\*\*:

*```python*

def check\_permission(*role*):

def real\_decorator(*func*):

def wrapper():

if *role* == 'admin':

print("You have admin rights")

return *func*()

else:

print("Access denied")

return wrapper

return real\_decorator

@check\_permission('user')

def view\_dashboard():

print("Viewing dashboard")

view\_dashboard()