# API - PROMPTING

# QUESTION 1

# 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()

# ```

## QUESTION 2

# # #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))

# ```

## QUESTION 3

# 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 4

# 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 5

# prompt 9, result 2

# 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 6

# prompt 9, result 2

# 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.

## QUESTION 7

# 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 8

# 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 9

# 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 10

# 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 11

# 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()

*```*

## QUESTION 12

(needs another file also - some test "hello.txt" file)

# prompt 13 response

# Question 4

# 0 - 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 13

# 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 14

# 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 15

# 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]])))

*```*

## QUESTION 16

# 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 17

# 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'

*```*

# GPT Link

<https://chatgpt.com/share/fff88f8f-4710-40ac-b6a7-b25b735cc9fb>

## QUESTION 18

\*\*0 - Topics\*\*: Deep Learning, Data Handling, and Object-Oriented Programming

\*\*1 - Scenario\*\*: You are developing a neural network for image classification using TensorFlow and Keras. There is an issue in the implementation of the data loading and model training process.

\*\*2 - Code\*\*:

```python

import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

class ImageDataLoader:

def \_\_init\_\_(self, image\_paths, labels):

self.image\_paths = image\_paths

self.labels = labels

def load\_images(self):

images = []

for path in self.image\_paths:

image = tf.io.read\_file(path)

image = tf.image.decode\_jpeg(image, channels=3)

images.append(image)

return np.array(images), np.array(self.labels)

image\_paths = ["image1.jpg", "image2.jpg", "image3.jpg"]

labels = [0, 1, 0]

data\_loader = ImageDataLoader(image\_paths, labels)

images, labels = data\_loader.load\_images()

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(128, 128, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

history = model.fit(images, labels, epochs=10, batch\_size=2)

```

\*\*3 - Question\*\*: The code above is intended to load images, build a convolutional neural network, and train the model. There are multiple issues in the code. Identify and explain the issues.

\*\*4 - Explanation\*\*:

1. The images are loaded without resizing, which will cause dimension mismatches during model training.

2. Images are not normalized, which can affect the training performance.

3. Labels are not one-hot encoded, which can lead to issues during training.

4. The images are converted to NumPy arrays before preprocessing, leading to incorrect data types.

\*\*5 - Answer\*\*:

```python

import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

class ImageDataLoader:

def \_\_init\_\_(self, image\_paths, labels):

self.image\_paths = image\_paths

self.labels = labels

def load\_images(self):

images = []

for path in self.image\_paths:

image = tf.io.read\_file(path)

image = tf.image.decode\_jpeg(image, channels=3)

image = tf.image.resize(image, [128, 128])

image = image / 255.0 # Normalize the image

images.append(image)

return np.array(images), np.array(self.labels)

image\_paths = ["image1.jpg", "image2.jpg", "image3.jpg"]

labels = [0, 1, 0]

data\_loader = ImageDataLoader(image\_paths, labels)

images, labels = data\_loader.load\_images()

model = models.Sequential()

model.add(layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(128, 128, 3)))

model.add(layers.MaxPooling2D((2, 2)))

model.add(layers.Flatten())

model.add(layers.Dense(64, activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

history = model.fit(images, labels, epochs=10, batch\_size=2)

```

## QUESTION 19

\*\*0 - Topics\*\*: Multithreading, Data Synchronization, and Concurrency

\*\*1 - Scenario\*\*: You are developing a multithreaded application that processes data from multiple sources and combines the results. There is an issue with data synchronization.

\*\*2 - Code\*\*:

```python

import threading

import time

class DataProcessor:

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

self.data = []

def process\_data(self, data\_source):

for data in data\_source:

self.data.append(data)

time.sleep(0.1) # Simulate data processing time

data\_processor = DataProcessor()

data\_source1 = [1, 2, 3, 4, 5]

data\_source2 = [6, 7, 8, 9, 10]

thread1 = threading.Thread(target=data\_processor.process\_data, args=(data\_source1,))

thread2 = threading.Thread(target=data\_processor.process\_data, args=(data\_source2,))

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print("Processed data:", data\_processor.data)

```

\*\*3 - Question\*\*: The code above is intended to process data from two sources in parallel using multithreading. There is an issue with the data processing. Identify and explain the issues.

\*\*4 - Explanation\*\*:

1. The code is not thread-safe; multiple threads can modify `self.data` simultaneously, leading to data corruption.

2. The `self.data` list should be protected using a lock to ensure only one thread modifies it at a time.

\*\*5 - Answer\*\*:

```python

import threading

import time

class DataProcessor:

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

self.data = []

self.lock = threading.Lock()

def process\_data(self, data\_source):

for data in data\_source:

with self.lock:

self.data.append(data)

time.sleep(0.1) # Simulate data processing time

data\_processor = DataProcessor()

data\_source1 = [1, 2, 3, 4, 5]

data\_source2 = [6, 7, 8, 9, 10]

thread1 = threading.Thread(target=data\_processor.process\_data, args=(data\_source1,))

thread2 = threading.Thread(target=data\_processor.process\_data, args=(data\_source2,))

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print("Processed data:", data\_processor.data)

```

## QUESTION 20

\*\*0 - Topics\*\*: Data Handling, Pandas, and Error Handling

\*\*1 - Scenario\*\*: You are working on a data analysis project that involves loading and processing a large CSV file using Pandas. There is an issue in the data loading and processing steps.

\*\*2 - Code\*\*:

```python

import pandas as pd

class DataAnalyzer:

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

self.file\_path = file\_path

self.data = None

def load\_data(self):

self.data = pd.read\_csv(self.file\_path)

def process\_data(self):

# Drop rows with missing values

self.data.dropna(inplace=True)

# Convert column 'age' to integer

self.data['age'] = self.data['age'].astype(int)

def analyze\_data(self):

# Calculate mean age

mean\_age = self.data['age'].mean()

return mean\_age

analyzer = DataAnalyzer('data.csv')

analyzer.load\_data()

analyzer.process\_data()

mean\_age = analyzer.analyze\_data()

print("Mean Age:", mean\_age)

```

\*\*3 - Question\*\*: The code above is intended to load a CSV file, process the data by removing rows with missing values, and then calculate the mean age. There is an issue in the data processing step. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The code does not handle the case where the 'age' column may contain non-numeric values, which will cause a failure when converting to integers.

\*\*5 - Answer\*\*:

```python

import pandas as pd

class DataAnalyzer:

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

self.file\_path = file\_path

self.data = None

def load\_data(self):

self.data = pd.read\_csv(self.file\_path)

def process\_data(self):

# Drop rows with missing values

self.data.dropna(inplace=True)

# Convert column 'age' to integer, handling non-numeric values

self.data = self.data[pd.to\_numeric(self.data['age'], errors='coerce').notnull()]

self.data['age'] = self.data['age'].astype(int)

def analyze\_data(self):

# Calculate mean age

mean\_age = self.data['age'].mean()

return mean\_age

analyzer = DataAnalyzer('data.csv')

analyzer.load\_data()

analyzer.process\_data()

mean\_age = analyzer.analyze\_data()

print("Mean Age:", mean\_age)

```

—

## QUESTION 21

\*\*0 - Topics\*\*: Object-Oriented Programming, Inheritance, and Polymorphism

\*\*1 - Scenario\*\*: You are developing a library management system using OOP principles. There is an issue with the class inheritance and method overriding.

\*\*2 - Code\*\*:

```python

class LibraryItem:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

def get\_info(self):

return f"Title: {self.title}, Author: {self.author}"

class Book(LibraryItem):

def \_\_init\_\_(self, title, author, num\_pages):

super().\_\_init\_\_(title, author)

self.num\_pages = num\_pages

def get\_info(self):

return f"Title: {self.title}, Author: {self.author}, Pages: {self.num\_pages}"

class Magazine(LibraryItem):

def \_\_init\_\_(self, title, author, issue\_number):

super().\_\_init\_\_(title, author)

self.issue\_number = issue\_number

def get\_info(self):

return f"Title: {self.title}, Author: {self.author}, Issue: {self.issue\_number}"

items = [

Book("Book One", "Author A", 300),

Magazine("Magazine One", "Author B", 5)

]

for item in items:

print(item.get\_info())

```

\*\*3 - Question\*\*: The code above is intended to create a list of library items (books and magazines) and print their information. There is an issue with the class inheritance and method overriding. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `Magazine` class should not have an `author` attribute since magazines typically do not have a single author.

2. The `Magazine` class's `get\_info` method does not correctly override the `get\_info` method from the `LibraryItem` class.

\*\*5 - Answer\*\*:

```python

class LibraryItem:

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

self.title = title

def get\_info(self):

return f"Title: {self.title}"

class Book(LibraryItem):

def \_\_init\_\_(self, title, author, num\_pages):

super().\_\_init\_\_(title)

self.author = author

self.num\_pages = num\_pages

def get\_info(self):

return f"Title: {self.title}, Author: {self.author}, Pages: {self.num\_pages}"

class Magazine(LibraryItem):

def \_\_init\_\_(self, title, issue\_number):

super().\_\_init\_\_(title)

self.issue\_number = issue\_number

def get\_info(self):

return f"Title: {self.title}, Issue: {self.issue\_number}"

items = [

Book("Book One", "Author A", 300),

Magazine("Magazine One", 5)

]

for item in items:

print(item.get\_info())

```

---

## QUESTION 22

\*\*0 - Topics\*\*: Multithreading and Resource Management

\*\*1 - Scenario\*\*: You are developing a multithreaded application that performs I/O operations on files. There is an issue with resource management in the multithreaded environment.

\*\*2 - Code\*\*:

```python

import threading

class FileHandler:

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

self.file\_path = file\_path

def read\_file(self):

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

data = file.read()

print(data)

file\_paths = ['file1.txt', 'file2.txt', 'file3.txt']

handlers = [FileHandler(path) for path in file\_paths]

threads = [threading.Thread(target=handler.read\_file) for handler in handlers]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

```

\*\*3 - Question\*\*: The code above is intended to read multiple files concurrently using multithreading. There is an issue with resource management. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The code does not ensure that each thread completes its I/O operation safely, potentially leading to race conditions if the files are being written to by another process.

\*\*5 - Answer\*\*:

```python

import threading

class FileHandler:

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

self.file\_path = file\_path

self.lock = threading.Lock()

def read\_file(self):

with self.lock:

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

data = file.read()

print(data)

file\_paths = ['file1.txt', 'file2.txt', 'file3.txt']

handlers = [FileHandler(path) for path in file\_paths]

threads = [threading.Thread(target=handler.read\_file) for handler in handlers]

for thread in threads:

thread.start()

for thread in threads:

thread.join()

```

## QUESTION 23

\*\*0 - Topics\*\*: Generators, Iterators, and Memory Management

\*\*1 - Scenario\*\*: You are developing a large-scale data processing pipeline and using generators to handle the data efficiently. There is an issue with the generator function that processes the data.

\*\*2 - Code\*\*:

```python

class DataProcessor:

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

self.data = data

def data\_generator(self):

for item in self.data:

yield item \* 2

def process\_data(self):

result = []

for item in self.data\_generator():

result.append(item)

return result

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

processor = DataProcessor(data)

processed\_data = processor.process\_data()

print("Processed Data:", processed\_data)

```

\*\*3 - Question\*\*: The code above is intended to process data by doubling each item using a generator function. There is an issue with the data processing pipeline. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The generator function `data\_generator` is not leveraging the full potential of generators for memory efficiency. The current implementation collects all results in a list, which defeats the purpose of using a generator.

\*\*5 - Answer\*\*:

```python

class DataProcessor:

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

self.data = data

def data\_generator(self):

for item in self.data:

yield item \* 2

def process\_data(self):

return list(self.data\_generator()) # Directly converting generator to a list

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

processor = DataProcessor(data)

processed\_data = processor.process\_data()

print("Processed Data:", processed\_data)

```

---

## QUESTION 24

\*\*0 - Topics\*\*: Metaprogramming and Decorators

\*\*1 - Scenario\*\*: You are developing a logging mechanism using decorators to automatically log function calls and their arguments. There is an issue with the decorator implementation.

\*\*2 - Code\*\*:

```python

import functools

def log\_decorator(func):

@functools.wraps(func)

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

print(f"Calling {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")

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

return wrapper

class Calculator:

@log\_decorator

def add(self, a, b):

return a + b

@log\_decorator

def multiply(self, a, b):

return a \* b

calc = Calculator()

result\_add = calc.add(3, 5)

result\_multiply = calc.multiply(4, 6)

print("Add Result:", result\_add)

print("Multiply Result:", result\_multiply)

```

\*\*3 - Question\*\*: The code above is intended to log function calls and their arguments for the methods in the `Calculator` class. There is an issue with the decorator implementation. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The decorator does not correctly log the method calls because it does not handle instance method binding properly. The `self` argument should be passed correctly.

\*\*5 - Answer\*\*:

```python

import functools

def log\_decorator(func):

@functools.wraps(func)

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

print(f"Calling {func.\_\_name\_\_} with args: {args}, kwargs: {kwargs}")

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

return wrapper

class Calculator:

@log\_decorator

def add(self, a, b):

return a + b

@log\_decorator

def multiply(self, a, b):

return a \* b

calc = Calculator()

result\_add = calc.add(3, 5)

result\_multiply = calc.multiply(4, 6)

print("Add Result:", result\_add)

print("Multiply Result:", result\_multiply)

```

---

## QUESTION 25

\*\*0 - Topics\*\*: Context Managers and Resource Management

\*\*1 - Scenario\*\*: You are developing a file processing system that uses context managers to handle file operations. There is an issue with the custom context manager implementation.

\*\*2 - Code\*\*:

```python

class FileManager:

def \_\_init\_\_(self, file\_name, mode):

self.file\_name = file\_name

self.mode = mode

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

self.file = open(self.file\_name, self.mode)

return self.file

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

self.file.close()

file\_name = 'example.txt'

with FileManager(file\_name, 'w') as f:

f.write("Hello, world!")

with FileManager(file\_name, 'r') as f:

content = f.read()

print("File Content:", content)

```

\*\*3 - Question\*\*: The code above is intended to create a custom context manager for file operations. There is an issue with the context manager implementation. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `\_\_exit\_\_` method does not handle exceptions, which can lead to resource leaks if an exception occurs during file operations.

\*\*5 - Answer\*\*:

```python

class FileManager:

def \_\_init\_\_(self, file\_name, mode):

self.file\_name = file\_name

self.mode = mode

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

self.file = open(self.file\_name, self.mode)

return self.file

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

self.file.close()

if exc\_type is not None:

print(f"Exception occurred: {exc\_val}")

return True # Suppress the exception

file\_name = 'example.txt'

with FileManager(file\_name, 'w') as f:

f.write("Hello, world!")

with FileManager(file\_name, 'r') as f:

content = f.read()

print("File Content:", content)

```

## QUESTION 26

Certainly! Here are more questions that test advanced Python topics:

---

\*\*0 - Topics\*\*: Concurrency with `asyncio` and Error Handling

\*\*1 - Scenario\*\*: You are developing an application that fetches data concurrently from multiple sources using `asyncio`. There is an issue with error handling in the coroutine.

\*\*2 - Code\*\*:

```python

import asyncio

async def fetch\_data(url):

print(f"Fetching data from {url}")

await asyncio.sleep(2) # Simulate network delay

if "error" in url:

raise ValueError(f"Failed to fetch data from {url}")

return f"Data from {url}"

async def fetch\_all\_data(urls):

tasks = [fetch\_data(url) for url in urls]

results = await asyncio.gather(\*tasks)

return results

urls = [

'https://example.com/data1',

'https://example.com/error',

'https://example.com/data3'

]

async def main():

results = await fetch\_all\_data(urls)

for result in results:

print(result)

asyncio.run(main())

```

\*\*3 - Question\*\*: The code above is intended to fetch data from multiple URLs concurrently and handle errors. However, there is an issue with the error handling in the coroutine. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `asyncio.gather` function does not handle exceptions correctly when one of the coroutines raises an exception. The entire operation will fail, and no results will be returned.

\*\*5 - Answer\*\*:

```python

import asyncio

async def fetch\_data(url):

print(f"Fetching data from {url}")

await asyncio.sleep(2) # Simulate network delay

if "error" in url:

raise ValueError(f"Failed to fetch data from {url}")

return f"Data from {url}"

async def fetch\_all\_data(urls):

tasks = [fetch\_data(url) for url in urls]

results = await asyncio.gather(\*tasks, return\_exceptions=True)

return results

urls = [

'https://example.com/data1',

'https://example.com/error',

'https://example.com/data3'

]

async def main():

results = await fetch\_all\_data(urls)

for result in results:

if isinstance(result, Exception):

print(f"Error: {result}")

else:

print(result)

asyncio.run(main())

```

—

## QUESTION 27

\*\*0 - Topics\*\*: Advanced Decorators and Dynamic Function Modification

\*\*1 - Scenario\*\*: You are developing a logging system that dynamically adds logging functionality to existing functions. There is an issue with the decorator implementation that modifies functions dynamically.

\*\*2 - Code\*\*:

```python

import functools

def add\_logging(func):

@functools.wraps(func)

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

print(f"Calling {func.\_\_name\_\_} with arguments {args} and {kwargs}")

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

print(f"{func.\_\_name\_\_} returned {result}")

return result

return wrapper

def log\_methods(cls):

for attr in dir(cls):

if callable(getattr(cls, attr)) and not attr.startswith("\_\_"):

original\_method = getattr(cls, attr)

decorated\_method = add\_logging(original\_method)

setattr(cls, attr, decorated\_method)

return cls

class MathOperations:

def add(self, a, b):

return a + b

def multiply(self, a, b):

return a \* b

MathOperations = log\_methods(MathOperations)

math\_ops = MathOperations()

result\_add = math\_ops.add(2, 3)

result\_multiply = math\_ops.multiply(4, 5)

```

\*\*3 - Question\*\*: The code above is intended to dynamically add logging to all methods of a class. There is an issue with the decorator implementation that modifies the methods. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The dynamic modification of class methods using the decorator does not correctly handle instance method bindings, which can cause the `self` argument to be missing or incorrect.

\*\*5 - Answer\*\*:

```python

import functools

def add\_logging(func):

@functools.wraps(func)

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

print(f"Calling {func.\_\_name\_\_} with arguments {args} and {kwargs}")

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

print(f"{func.\_\_name\_\_} returned {result}")

return result

return wrapper

def log\_methods(cls):

for attr in dir(cls):

if callable(getattr(cls, attr)) and not attr.startswith("\_\_"):

original\_method = getattr(cls, attr)

@functools.wraps(original\_method)

def wrapped\_method(self, \*args, \*\*kwargs):

return add\_logging(original\_method)(self, \*args, \*\*kwargs)

setattr(cls, attr, wrapped\_method)

return cls

@log\_methods

class MathOperations:

def add(self, a, b):

return a + b

def multiply(self, a, b):

return a \* b

math\_ops = MathOperations()

result\_add = math\_ops.add(2, 3)

result\_multiply = math\_ops.multiply(4, 5)

```

—

## QUESTION 28

\*\*0 - Topics\*\*: Serialization, Deserialization, and Error Handling with `pickle`

\*\*1 - Scenario\*\*: You are developing an application that saves and loads complex objects using `pickle` for serialization. There is an issue with the serialization and deserialization process.

\*\*2 - Code\*\*:

```python

import pickle

class ComplexObject:

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

self.name = name

self.data = data

def \_\_str\_\_(self):

return f"ComplexObject(name={self.name}, data={self.data})"

def save\_object(obj, filename):

with open(filename, 'wb') as file:

pickle.dump(obj, file)

def load\_object(filename):

with open(filename, 'rb') as file:

return pickle.load(file)

obj = ComplexObject("TestObject", [1, 2, 3, 4, 5])

filename = 'object.pkl'

save\_object(obj, filename)

loaded\_obj = load\_object(filename)

print("Loaded Object:", loaded\_obj)

```

\*\*3 - Question\*\*: The code above is intended to serialize and deserialize a complex object using `pickle`. There is an issue with the serialization and deserialization process. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `pickle` module does not handle custom error handling for the serialization and deserialization process. If an error occurs, it will not be caught.

\*\*5 - Answer\*\*:

```python

import pickle

class ComplexObject:

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

self.name = name

self.data = data

def \_\_str\_\_(self):

return f"ComplexObject(name={self.name}, data={self.data})"

def save\_object(obj, filename):

try:

with open(filename, 'wb') as file:

pickle.dump(obj, file)

except Exception as e:

print(f"Error saving object: {e}")

def load\_object(filename):

try:

with open(filename, 'rb') as file:

return pickle.load(file)

except Exception as e:

print(f"Error loading object: {e}")

return None

obj = ComplexObject("TestObject", [1, 2, 3, 4, 5])

filename = 'object.pkl'

save\_object(obj, filename)

loaded\_obj = load\_object(filename)

print("Loaded Object:", loaded\_obj)

```

—

## QUESTION 29

\*\*0 - Topics\*\*: Advanced Usage of `dataclass` and Custom Methods

\*\*1 - Scenario\*\*: You are developing a system that uses `dataclass` for managing data structures with custom methods. There is an issue with the implementation of the `dataclass`.

\*\*2 - Code\*\*:

```python

from dataclasses import dataclass

@dataclass

class Product:

name: str

price: float

quantity: int

def total\_cost(self):

return self.price \* self.quantity

product = Product(name="Laptop", price=999.99, quantity=5)

print("Total Cost:", product.total\_cost())

product.price = "1000" # Incorrect data type

print("Updated Total Cost:", product.total\_cost())

```

\*\*3 - Question\*\*: The code above is intended to calculate the total cost of a product using a `dataclass` with custom methods. There is an issue with the implementation of the `dataclass`. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `dataclass` does not enforce type checking, which can lead to incorrect data types being assigned to fields, causing runtime errors in methods.

\*\*5 - Answer\*\*:

```python

from dataclasses import dataclass, field

from typing import Any

@dataclass

class Product:

name: str

price: float

quantity: int

def total\_cost(self):

return self.price \* self.quantity

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

if not isinstance(self.price, (int, float)):

raise TypeError(f"Expected price to be int or float, got {type(self.price).\_\_name\_\_}")

if not isinstance(self.quantity, int):

raise TypeError(f"Expected quantity to be int, got {type(self.quantity).\_\_name\_\_}")

product = Product(name="Laptop", price=999.99, quantity=5)

print("Total Cost:", product.total\_cost())

try:

product.price = "1000" # Incorrect data type

print("Updated Total Cost:", product.total\_cost())

except TypeError as e:

print(e)

```

## QUESTION 30

\*\*0 - Topics\*\*: Metaclasses and Dynamic Class Creation

\*\*1 - Scenario\*\*: You are working on a project that uses metaclasses to enforce specific constraints on class attributes dynamically. There is an issue with the metaclass implementation.

\*\*2 - Code\*\*:

```python

class AttributeValidator(type):

def \_\_new\_\_(cls, name, bases, dct):

for key, value in dct.items():

if key.startswith('\_'):

raise ValueError("Attributes cannot start with an underscore")

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

class Person(metaclass=AttributeValidator):

name = "John"

\_age = 30 # Invalid attribute

p = Person()

print(f"Person's name is {p.name}")

```

\*\*3 - Question\*\*: The code above is intended to use a metaclass to enforce that no class attribute starts with an underscore. There is an issue with the metaclass implementation. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The metaclass raises a `ValueError` for the invalid attribute `\_age`, but this exception is not correctly handled, and the class cannot be created.

\*\*5 - Answer\*\*:

```python

class AttributeValidator(type):

def \_\_new\_\_(cls, name, bases, dct):

for key, value in dct.items():

if key.startswith('\_'):

raise ValueError(f"Attributes cannot start with an underscore: {key}")

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

try:

class Person(metaclass=AttributeValidator):

name = "John"

\_age = 30 # Invalid attribute

except ValueError as e:

print(e)

class ValidPerson(metaclass=AttributeValidator):

name = "Jane"

age = 25 # Valid attribute

p = ValidPerson()

print(f"Person's name is {p.name}")

```

—

## QUESTION 31

\*\*0 - Topics\*\*: Advanced Use of `multiprocessing` and Shared Memory

\*\*1 - Scenario\*\*: You are developing an application that uses the `multiprocessing` module to parallelize computations and share data between processes. There is an issue with shared memory management.

\*\*2 - Code\*\*:

```python

from multiprocessing import Process, Array

import numpy as np

def compute\_square(numbers, results):

for idx, number in enumerate(numbers):

results[idx] = number \*\* 2

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

numbers = np.array([1, 2, 3, 4, 5])

results = Array('i', len(numbers))

p = Process(target=compute\_square, args=(numbers, results))

p.start()

p.join()

print("Squared Numbers:", results[:])

```

\*\*3 - Question\*\*: The code above is intended to compute the squares of a list of numbers using the `multiprocessing` module and store the results in shared memory. There is an issue with shared memory management. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `multiprocessing.Array` is correctly used for shared memory, but the data type specified ('i' for integer) may not handle the size of the squared values correctly if they exceed the range of a C `int`.

\*\*5 - Answer\*\*:

```python

from multiprocessing import Process, Array

import numpy as np

def compute\_square(numbers, results):

for idx, number in enumerate(numbers):

results[idx] = number \*\* 2

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

numbers = np.array([1, 2, 3, 4, 5])

results = Array('d', len(numbers)) # Use 'd' for double to handle larger values

p = Process(target=compute\_square, args=(numbers, results))

p.start()

p.join()

print("Squared Numbers:", results[:])

```

—

## QUESTION 32

\*\*0 - Topics\*\*: Coroutines and `asyncio` for I/O-bound Tasks

\*\*1 - Scenario\*\*: You are developing an application that uses coroutines with `asyncio` to handle multiple I/O-bound tasks concurrently. There is an issue with the coroutine implementation.

\*\*2 - Code\*\*:

```python

import asyncio

async def fetch\_data(url):

print(f"Fetching data from {url}")

await asyncio.sleep(1) # Simulate I/O-bound task

return f"Data from {url}"

async def main():

urls = ['https://example.com/data1', 'https://example.com/data2', 'https://example.com/data3']

results = []

for url in urls:

result = await fetch\_data(url)

results.append(result)

return results

loop = asyncio.get\_event\_loop()

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

for result in results:

print(result)

```

\*\*3 - Question\*\*: The code above is intended to fetch data from multiple URLs concurrently using coroutines and `asyncio`. There is an issue with the coroutine implementation. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The `main` function fetches data from URLs sequentially, not concurrently. This negates the benefits of using `asyncio` for concurrent tasks.

\*\*5 - Answer\*\*:

```python

import asyncio

async def fetch\_data(url):

print(f"Fetching data from {url}")

await asyncio.sleep(1) # Simulate I/O-bound task

return f"Data from {url}"

async def main():

urls = ['https://example.com/data1', 'https://example.com/data2', 'https://example.com/data3']

tasks = [fetch\_data(url) for url in urls]

results = await asyncio.gather(\*tasks)

return results

loop = asyncio.get\_event\_loop()

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

for result in results:

print(result)

```

---

## QUESTION 33

\*\*0 - Topics\*\*: Using `contextlib` for Custom Context Managers

\*\*1 - Scenario\*\*: You are developing an application that uses custom context managers to manage resources. There is an issue with the implementation of the context manager using `contextlib`.

\*\*2 - Code\*\*:

```python

from contextlib import contextmanager

@contextmanager

def open\_file(file\_name, mode):

file = open(file\_name, mode)

try:

yield file

finally:

file.close()

file\_name = 'example.txt'

with open\_file(file\_name, 'w') as f:

f.write("Hello, world!")

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

content = f.read()

print("File Content:", content)

```

\*\*3 - Question\*\*: The code above is intended to use a custom context manager to handle file operations. There is an issue with the implementation of the context manager using `contextlib`. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The context manager correctly handles the file operations, but it does not handle exceptions that might occur during the file operations, potentially leading to resource leaks or other issues.

\*\*5 - Answer\*\*:

```python

from contextlib import contextmanager

@contextmanager

def open\_file(file\_name, mode):

file = open(file\_name, mode)

try:

yield file

except Exception as e:

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

raise

finally:

file.close()

file\_name = 'example.txt'

try:

with open\_file(file\_name, 'w') as f:

f.write("Hello, world!")

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

content = f.read()

print("File Content:", content)

except Exception as e:

print(f"Failed to process file: {e}")

```

—

## QUESTION 34

\*\*0 - Topics\*\*: Advanced Regular Expressions

\*\*1 - Scenario\*\*: You are developing a text processing tool that uses advanced regular expressions to extract specific patterns from text. There is an issue with the regular expression implementation.

\*\*2 - Code\*\*:

```python

import re

text = "Contact us at contact@example.com or support@example.com"

# Regex to extract email addresses

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

emails = re.findall(email\_pattern, text)

print("Extracted Emails:", emails)

```

\*\*3 - Question\*\*: The code above is intended to extract email addresses from a given text using regular expressions. There is an issue with the regular expression implementation. Identify and explain the issue.

\*\*4 - Explanation\*\*:

1. The regular expression does not handle case insensitivity correctly, potentially missing email addresses with uppercase letters in the domain.

\*\*5 - Answer\*\*:

```python

import re

text = "Contact us at contact@example.com or SUPPORT@EXAMPLE.COM"

# Regex to extract email addresses with case insensitivity

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

emails = re.findall(email\_pattern, text, re.IGNORECASE)

print("Extracted Emails:", emails)

```