files & exceptional handling assignment

QUESTION1: What is the difference between interpreted and compiled languages?

ANSWER1: Difference Between Interpreted and Compiled Languages

Interpreted and compiled languages refer to two different methods of translating high-level programming code into machine code that a computer can understand and execute.

1. Compiled Languages:
   A compiled language uses a compiler to translate the entire program's source code into machine code before execution. The result is a standalone executable file that can be run independently of the source code or compiler. Compilation typically occurs once, and the program can then be executed multiple times without recompiling.

- Translation: Done before execution.
- Speed: Faster execution due to precompiled code.
- Error Detection: Errors are detected during the compilation process.
- Examples: C, C++, Rust, Go.

2. Interpreted Languages:
   An interpreted language uses an interpreter to read and execute the source code line-by-line at runtime. There is no separate executable file; instead, the interpreter must be present every time the code runs. This approach provides flexibility and easier debugging but can lead to slower performance.

- Translation: Done at runtime, line by line.
- Speed: Slower due to continuous interpretation.
- Error Detection: Errors appear during execution.
- Examples: Python, JavaScript, Ruby, PHP.

QUESTION2: What is exception handling in Python? ANSWER2: Exception handling in Python is a process used to manage errors that occur during the execution of a program. These errors, called exceptions, can cause the program to stop unexpectedly. Python provides a way to handle these errors using special blocks of code, allowing the program to continue running smoothly.

Purpose of Exception Handling:

- To prevent program crashes
- To handle errors gracefully
- To ensure proper cleanup of resources (like files or network connections)
- To inform users about the nature of the error in a readable way

Main Keywords:

- try: Contains code that may cause an error.
- except: Defines how to respond to specific exceptions.
- else: (Optional) Runs if no exception occurs.
- finally: (Optional) Runs no matter what, often used for cleanup.
- Example:

[ ]

```python
try:

    number = int(input("Enter a number: "))

    result = 10 / number

except ZeroDivisionError:

    print("You cannot divide by zero.")

except ValueError:

    print("Invalid input. Please enter a number.")

else:

    print("Result is:", result)

finally:

    print("This block always runs.")
```

```
Enter a number: 4587126
Result is: 2.1800142398530147e-06
This block always runs.
```

QUESTION3: What is the purpose of the finally block in exception handling?

ANSWER3: The finally block in exception handling is used to define a section of code that always executes, regardless of whether an exception was raised or handled. Its primary purpose is to ensure that important cleanup actions—such as closing files, releasing resources, or disconnecting from a network or database—are performed no matter what happens in the try or except blocks.

Key Features:

- Executes after the try block and any except blocks.
- Runs regardless of whether an exception was raised, handled, or unhandled.
- Executes even if the try or except block contains a return, break, or continue statement.

Why It's Important:

The finally block ensures that critical resources are properly released and that the program remains stable and predictable, even in the face of errors.

Example:

```
[ ]
```

```
try:

    file = None

    file = open("data.txt", "r")

    content = file.read()

except FileNotFoundError:

    print("The file was not found.")

finally:

    if file is not None:

        file.close()
```

```
The file was not found.
```

QUESTION4: What is logging in Python?

ANSWER4: Logging in Python is the process of recording messages that describe events occurring during the execution of a program. It is commonly used for debugging, monitoring, and troubleshooting applications.

Python provides a built-in module called logging that allows developers to:

- Track events at different severity levels.
- Output logs to various destinations (console, files, network, etc.).
- Format log messages for clarity and structure.
- Control what messages are captured based on their importance.

Common Logging Levels:

From lowest to highest severity:

- DEBUG: Detailed information, typically useful for diagnosing problems.
- INFO: General messages that confirm the program is working as expected.
- WARNING: Indicates a potential issue or something unexpected.
- ERROR: A more serious problem; the program may still run.
- CRITICAL: A severe error indicating the program may not continue.
- Example:

[ ]

```
import logging



logging.basicConfig(level=logging.INFO, format='%(levelname)s: %(message)s')



logging.debug("This is a debug message")

logging.info("This is an info message")
logging.warning("This is a warning")

logging.error("An error has occurred")

logging.critical("Critical issue!")
```

```
WARNING:root:This is a warning
ERROR:root:An error has occurred
CRITICAL:root:Critical issue!
```

QUESTION5: What is the significance of the **del** method in Python? ANSWER5: The **del** method in Python is a special method known as a destructor. It is called automatically when an object is about to be destroyed, typically when its reference count drops to zero (i.e., no more references to the object exist).

Purpose of **del**:

- The **del** method is used to define cleanup behavior for objects, such as:
- Releasing external resources (e.g., files, network connections).
- Logging destruction events for debugging.
- Performing other finalization tasks before an object is garbage-collected.
- Example:

[ ]

```
class FileHandler:

  def __init__(self, filename):

    try:

      with open(filename, 'r') as f:

        pass

    except FileNotFoundError:

      with open(filename, 'w') as f:
            f.write("This is a dummy file.\n")

        print(f"Created dummy file: {filename}")



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

    print("File opened.")



def __del__(self):

    if self.file is not None and not self.file.closed:

        self.file.close()

        print("File closed.")



handler = FileHandler("data.txt")
```

```
Created dummy file: data.txt
File opened.
```

QUESTION6: What is the difference between import and from ... import in Python?

ANSWER6:

1. import Statement

[ ]

```
import math
```

- What it does: Imports the entire module. You can access all the functions, classes, and variables defined in the module, but you need to refer to the module name each time.

- How to access: You need to use the module name as a prefix to access any item from the module.

Example:

```
[ ]
```

```python
import math

print(math.sqrt(16))
```

```
4.0
```

- Advantages:
    2. Keeps the namespace clear because you need to use the module name when accessing its functions or variables, reducing the risk of name conflicts.
    3. Useful when you need multiple functions or classes from a module.
- Disadvantages:
    2. It can be more verbose since you always need to type the module name to access its functions or variables.
    3. from ... import Statement
- What it does: Imports specific items (functions, classes, variables) from the module. You can directly access the imported items without needing the module name as a prefix.
- How to access: You can use the imported items directly without the module name.

Example:

```
[ ]
```

```python
from math import sqrt

print(sqrt(16))
```

```
4.0
```

- Advantages:
    - More concise: You can access specific items directly without needing the module name.
    - Ideal when you only need a few functions or classes from a module.
- Disadvantages:
    - Can cause namespace pollution if you import many items or use *, potentially causing naming conflicts if different modules have functions or variables with the same name.
    - Makes it harder to know where a function or variable came from, especially in large projects.

QUESTION7: How can you handle multiple exceptions in Python?

ANSWER7:

1. Using Multiple except Blocks

You can catch different types of exceptions using multiple except blocks. Each block handles a specific exception.

Example:

[ ]

```
try:

  x = 1 / 0

except ZeroDivisionError:

  print("Cannot divide by zero!")



try:

  num = int("abc")

except ValueError:
    print("Invalid input! Cannot convert to integer.")
```

```
Cannot divide by zero!
Invalid input! Cannot convert to integer.
```

---

2. Catching Multiple Exceptions in a Single except Block

You can specify multiple exceptions in a tuple within a single except block to handle different exceptions in the same way.

Example:

```
[ ]
```

```
try:
    x = 1 / 0
    num = int("abc")
except (ZeroDivisionError, ValueError) as e:
    print(f"Error occurred: {e}")
```

```
Error occurred: division by zero
```

---

3. Handling Different Exceptions with Different Responses

You can handle each exception type with separate except blocks, allowing you to respond to each type of error differently.

Example:

```
[ ]
```

```
try:

    x = 1 / 0

    num = int("abc")

except ZeroDivisionError:

    print("Cannot divide by zero!")

except ValueError:

    print("Invalid value for conversion!")
```

```
Cannot divide by zero!
```

---

4. Using a Generic except Block

You can use a generic except block to catch any exception, though it's generally recommended to catch specific exceptions for better clarity and debugging.

Example:

```
[ ]
```

```
try:

    num = int("xyz")

except Exception as e:

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

```
An error occurred: invalid literal for int() with base 10: 'xyz'
```

---

5. Using else with try/except

The else block is executed if no exception is raised in the try block. It's often used when you want to perform an action only if the try block succeeds.

Example:

```
[ ]
```

```
try:

    result = 10 / 2

except ZeroDivisionError:

    print("Cannot divide by zero!")

else:

    print(f"Result is: {result}")
```

```
Result is: 5.0
```

---

6. Using finally for Cleanup

A finally block will execute regardless of whether an exception is raised or not. This is useful for cleanup actions like closing files or releasing resources.

Example:

```
[ ]
```

```
try:

    file = open("example.txt", "r")

    content = file.read()

except FileNotFoundError:
    print("File not found!")
finally:
    if file:
        file.close()
```

```
File not found!
```

QUESTION8: What is the purpose of the with statement when handling files in Python? ANSWER8: The with statement in Python is used to simplify file handling by automatically managing resources, such as opening and closing files.

Purpose of the with statement:

- It ensures that the file is properly closed after its suite finishes, even if an error occurs during file operations.
- It makes the code cleaner, safer, and more readable.

Without with statement:

You have to manually close the file:

```
[ ]
```

```
try:
    file = open("example.txt", "r")
    try:
        content = file.read()
        print("File content:", content)
    finally:
        if file:
```

```
        file.close()

        print("File closed.")

except FileNotFoundError:

    print("Error: The file 'example.txt' was not found.")

except Exception as e:

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

```
Error: The file 'example.txt' was not found.
```

- With with statement:

Python handles the closing automatically:

[ ]

```
with open("example.txt", "r") as file:

    content = file.read()
```

[ ]

```
import os



file_path = "example.txt"

if not os.path.exists(file_path):

    try:

        with open(file_path, "w") as f:

            f.write("This is some content in example.txt\n")

        print(f"'{file_path}' created.")
    except IOError as e:

        print(f"Error creating file '{file_path}': {e}")

try:

    with open(file_path, "r") as file:

        content = file.read()

        print("File content:", content)

except FileNotFoundError:

    print(f"Error: The file '{file_path}' was not found.")

except Exception as e:

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

```
'example.txt' created.
File content: This is some content in example.txt
```

QUESTION9: What is the difference between multithreading and multiprocessing?
ANSWER9: Difference Between Multithreading and Multiprocessing in Python. Both multithreading and multiprocessing are techniques used to achieve concurrent execution in Python, but they work differently and are suited for different types of task.

Multithreading

- Definition: Uses multiple threads (lightweight subprocesses) within a single process.
- Shared memory: All threads share the same memory space.
- Best for: I/O-bound tasks (like file operations, web scraping, network calls).
- Limitation: Affected by Python's Global Interpreter Lock (GIL), which prevents multiple threads from executing Python bytecode simultaneously in multi-core CPUs.

Example Use Case:

[ ]

```
import threading



def print_numbers():

    for i in range(5):

        print(i)



thread = threading.Thread(target=print_numbers)

thread.start()
```

```
0
1
2
3
4
```

Multiprocessing

- Definition: Uses multiple processes, each with its own Python interpreter and memory space.
- Separate memory: Processes do not share memory, making them more isolated and safer for CPU-bound tasks.
- Best for: CPU-bound tasks (like complex calculations, data processing).
- Bypasses GIL: Since each process has its own interpreter, Python's GIL doesn't restrict it.

Example Use Case:

[ ]

```
import multiprocessing
```

```
def square_numbers():

    for i in range(5):

        print(i * i)



process = multiprocessing.Process(target=square_numbers)

process.start()
```

QUESTION10: What are the advantages of using logging in a program?

ANSWER10: Logging is a way to track events that happen when software runs. It provides a better alternative to using print() statements for debugging and monitoring.

Main Advantages of Logging:

1. Helps with Debugging and Troubleshooting
- Logs provide detailed insights into what the program was doing when something went wrong.
- Easier to trace errors and fix bugs based on historical data.
2. Records Runtime Information
- You can log inputs, outputs, function calls, and exceptions.
- Useful for understanding how your program behaves in production.
3. Flexible Log Levels
- Python's logging module supports different log levels:
- DEBUG: Detailed info for diagnosing problems
- INFO: General information (e.g., successful operations)
- WARNING: Something unexpected happened
- ERROR: A serious problem occurred
- CRITICAL: Very severe error
4. Saves Logs to Files
- Logs can be written to files for long-term storage.
- You can analyze past logs to identify patterns or recurring issues.
5. Non-intrusive

- Unlike print(), logging can be toggled on/off or filtered without changing the program logic.
- Helps in separating diagnostic output from normal output.

6. Customizable and Configurable

- Supports formatting, timestamps, module names, etc.
- Can direct logs to different outputs: console, file, email, or remote server.

7. Useful in Multi-User or Multi-Threaded Programs

- Logs can help track actions of different users or threads.
- Helps monitor complex applications (like web servers or APIs).

QUESTION11: What is memory management in Python?

ANSWER11: Memory management in Python refers to the process of allocating, tracking, and freeing memory used by variables, objects, and data structures during program execution.

How Python Manages Memory:

1. Automatic Memory Management

- Python handles most memory operations automatically, so developers don't need to allocate or free memory manually.
- It uses a private heap space to store all objects and data structures.

2. Reference Counting

- Python uses reference counting as its primary memory management technique.
- Every object has a reference count, which increases when a reference is made to the object and decreases when references are deleted.
- When the count reaches zero, the memory is released.

Example:

[ ]

```
a = "hello"

b = a

del a
```

---

3. Garbage Collection
- To handle circular references (e.g., when two objects reference each other), Python includes a garbage collector.
- It detects and deletes objects that are no longer accessible.
4. Memory Pools (Pymalloc)
- Python uses a custom allocator called pymalloc for performance.
- It creates memory pools for frequently used object sizes to reduce the overhead of system-level memory allocation.
5. Dynamic Typing
- Python variables are dynamically typed, meaning memory is allocated at runtime based on the type and size of the object.

QUESTION12: What are the basic steps involved in exception handling in Python?

ANSWER12: Exception handling in Python is used to manage errors gracefully without crashing the program. It allows you to write safer and more robust code by dealing with unexpected situations.

Basic Steps in Exception Handling:

1. try Block - Detect the Error
- Wrap the code that might raise an exception in a try block.
- If an error occurs, Python stops executing the try block and jumps to the except block.
2. except Block – Handle the Error
- Catch and handle the exception.
- You can handle specific exceptions or use a general one.
3. (Optional) else Block – Run if No Exception Occurs
- The else block runs only if no exception occurs in the try block.
4. (Optional) finally Block – Always Runs
- This block executes no matter what – whether an exception occurs or not.
- Useful for cleanup actions like closing files or releasing resources.
  Example

```
[ ]
```

```
try:
    num = int(input("Enter a number: "))

    result = 10 / num

except ValueError:

    print("Invalid input! Please enter a number.")

except ZeroDivisionError:

    print("Cannot divide by zero.")

else:

    print("Result is:", result)

finally:

    print("Program finished.")
```

```
Enter a number: 548748
Result is: 1.822330104164389e-05
Program finished.
```

QUESTION13: Why is memory management important in Python?

ANSWER13: Memory management is a critical part of any programming language, including Python, because it ensures your program runs efficiently, reliably, and scalably.

Reasons Why Memory Management is Important:

1. Prevents Memory Leaks

- Proper memory management ensures that unused memory is freed automatically.
- In Python, this is done using reference counting and garbage collection.
- Without it, your program could consume more and more memory over time, leading to crashes.

2. Improves Performance

- Efficient memory usage helps your program run faster and respond quicker.

- Python's built-in memory manager (e.g., pymalloc) optimizes the use of memory by reusing objects and minimizing overhead.

3. Automatic Cleanup

- Python automatically deallocates memory that is no longer needed, reducing the need for manual memory management (unlike C/C++).
- This allows developers to focus more on writing logic and less on managing memory manually.

4. Supports Large Applications

- Good memory management makes it easier to build and maintain large-scale applications that use lots of data or run for a long time.

5. Prevents Program Crashes

- Poor memory handling can lead to memory overflow, causing the program to crash or behave unpredictably.
- Python's memory management reduces this risk significantly.

6. Optimizes Resource Usage

- Especially important in resource-limited environments like mobile apps, IoT devices, or cloud services.
- Proper memory usage ensures the application doesn’t overuse CPU or RAM.

QUESTION14: What is the role of try and except in exception handling?

ANSWER14: In Python, try and except are used to catch and handle exceptions (errors) that may occur during program execution. They help prevent the program from crashing and allow you to define what should happen when an error occurs.

1. try Block – Detect Errors

- The try block is used to wrap the code that might raise an exception.
- If an error occurs inside the try block, Python stops execution at that point and looks for an except block to handle the error.

Example:

[ ]

```
try:

  number = int(input("Enter a number: "))

  result = 10 / number
```

```
except:

  pass
```

```
Enter a number: 46132164244
```

---

2. except Block – Handle Errors

- The except block is used to catch and respond to exceptions raised in the try block.
- You can catch specific types of exceptions (like ZeroDivisionError, ValueError) or use a generic except to catch all.

QUESTION15: How does Python's garbage collection system work?

ANSWER15: Python's garbage collection (GC) system is responsible for automatically managing memory — specifically, reclaiming memory from objects that are no longer needed, so your program runs efficiently without manual memory cleanup.

Main Components of Python's Garbage Collection System:

1. Reference Counting (Primary Method)

- Every object in Python has an internal reference count — a counter of how many references point to it.
- When the reference count drops to zero, the object is immediately deleted from memory.

Example:

```
[ ]
```

```
a = "hello"

b = a

del a

del b
```

2. Garbage Collector (Handles Circular References)

- Reference counting can't handle circular references (e.g., two objects referencing each other but not used elsewhere).
- Python's garbage collector, part of the gc module, detects and cleans up unreachable objects in cycles.

Example of Circular Reference:

[ ]

```
import gc



class Node:

    def __init__(self):

        self.ref = self



node = Node()

del node



gc.collect()
```

```
Exception ignored in: <function FileHandler.__del__ at 0x7c9c1ac8e020>
Traceback (most recent call last):
  File "<ipython-input-7-f6843028fe66>", line 7, in __del__
AttributeError: 'FileHandler' object has no attribute 'file'


257
```

QUESTION16: What is the purpose of the else block in exception handling?

ANSWER16: In Python, the else block in exception handling is used to define code that should run only if no exceptions are raised in the try block.

Purpose of the else Block:

- It separates the normal execution path from the error handling path.
- Code in the else block runs only if the try block doesn't raise any exceptions.
- It helps keep the try block clean by placing non-error-related code in the else.

Example

```
[ ]
```

```python
try:

    number = int(input("Enter a number: "))

except ValueError:

    print("That's not a valid number.")

else:

    print(f"You entered: {number}")
```

```
Enter a number: 58614684132165
You entered: 58614684132165
```

QUESTION17: What are the common logging levels in Python?

ANSWER17: In Python, the logging module provides several predefined logging levels to categorize messages based on their severity. These levels help developers track events in a program, ranging from detailed debugging information to critical error messages.

Common Logging Levels:

1. DEBUG (10)
- Used for detailed diagnostic information.
- Typically used only during development.
2. INFO (20)
- Used to confirm that things are working as expected.
- Suitable for general application progress messages.
3. WARNING (30)
- Indicates that something unexpected happened, or there may be a problem soon.
- The program continues to run.
4. ERROR (40)
- Indicates a more serious problem that prevented a part of the program from functioning.
5. CRITICAL (50)
- A very serious error that may prevent the program from continuing to run.

Example:

```
[ ]
```

```
import logging



logging.basicConfig(level=logging.DEBUG)



logging.debug("This is a debug message.")

logging.info("This is an info message.")

logging.warning("This is a warning.")

logging.error("This is an error.")

logging.critical("This is critical.")
```

```
WARNING:root:This is a warning.
ERROR:root:This is an error.
CRITICAL:root:This is critical.
```

QUESTION18: What is the difference between os.fork() and multiprocessing in Python? ANSWER18:Both os.fork() and the multiprocessing module are used to create new processes in Python, but they differ significantly in terms of usage, portability, and abstraction.

1. os.fork()

- Definition: os.fork() is a low-level system call that creates a child process by duplicating the current process.
- Platform: Available only on Unix/Linux systems (not available on Windows).
- Complexity: Lower-level and requires manual handling of inter-process communication.
- Usage: Suitable when you need direct control over process creation and behavior.

Example:

```
[ ]
```

```python
import os



pid = os.fork()



if pid == 0:

    print("This is the child process")

else:

    print("This is the parent process")
```

```
This is the parent process
This is the child process
```

2. multiprocessing Module

- Definition: A high-level Python module that provides a simple API to create and manage multiple processes.
- Platform: Cross-platform (works on Windows, macOS, and Linux).
- Ease of Use: Easier to use and manage than os.fork(). Supports process pools, queues, pipes, and shared memory.
- Safe: Avoids many common bugs and pitfalls of low-level process management.

Example:

[ ]

```
from multiprocessing import Process



def worker():

    print("Worker function running")



p = Process(target=worker)

p.start()

p.join()
```

QUESTION19: What is the importance of closing a file in Python?

ANSWER19: Closing a file in Python is an essential step after performing file operations like reading or writing. It ensures that system resources are released properly and that all data is safely written to the file.

1. Resource Management

- Open files consume system resources like memory and file descriptors.
- Closing files frees these resources, preventing potential memory leaks or file handle exhaustion.

2. Data Integrity

- When writing to a file, data is often stored in a temporary buffer before being written to disk.
- Closing the file flushes the buffer, ensuring all data is properly saved.

3. Avoiding Errors

- Keeping files open longer than needed can lead to file corruption, data loss, or errors in programs with many file operations.

4. File Locks

- On some systems, open files may be locked and inaccessible to other programs. Closing the file releases the lock.

Example:

[ ]

```
with open("example.txt", "w") as file:

    file.write("Hello, world!")
```

QUESTION20: What is the difference between file.read() and file.readline() in Python? ANSWER20: In Python, both file.read() and file.readline() are used to read content from a file, but they behave differently in how much data they read at a time.

1. file.read()

- Function: Reads the entire file (or a specified number of characters).
- Usage: Useful when you want to process the whole content at once.

Example:

[ ]

```
with open("example.txt", "r") as file:

    content = file.read()
```

```
print(content)
```

---

2. file.readline()
- Function: Reads the file one line at a time.
- Usage: Useful when processing large files or when reading line-by-line is needed.

Example:

```
[ ]
```

```
with open("example.txt", "r") as file:
    line = file.readline()
    print(line)
```

QUESTION21: What is the logging module in Python used for? The logging module in Python is used to record messages about a program’s execution. These messages can help developers track events, debug issues, and monitor application behavior during development or after deployment.

Purpose of the Logging Module:

1. Debugging
- Helps trace and fix bugs by recording detailed messages about what the program is doing.
2. Monitoring
- Allows you to monitor the program's behavior in real time or by reviewing log files later.
3. Error Tracking
- Captures errors and exceptions to help identify the cause of failures.
4. Flexible Output

- Logs can be displayed in the console, written to a file, or sent to remote servers.

5. Level Control

- You can set the severity level of messages (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).
  Example Usage:

[ ]

```
import logging



logging.basicConfig(level=logging.INFO)



logging.debug("This is a debug message")

logging.info("This is an info message")

logging.warning("This is a warning")

logging.error("This is an error")

logging.critical("This is critical")
```

QUESTION22: What is the os module in Python used for in file handling?

ANSWER22: The os module in Python provides a way to interact with the operating system. In file handling, it is especially useful for performing operations like creating, deleting, renaming, or navigating files and directories.

Common Uses of the os Module in File Handling:

1.Checking if a File or Directory Exists

[ ]

```
import os

os.path.exists("example.txt")
```

---

2. Creating Directories

[ ]

```
os.mkdir("new_folder")
```

---

3. Removing Files or Directories

[ ]

```
os.remove("example.txt")

os.rmdir("old_folder")
```

QUESTION23: What are the challenges associated with memory management in Python?

ANSWER23: Memory management in Python is mostly handled automatically through reference counting and garbage collection. However, certain challenges can arise, especially in large or long-running programs.

Common Challenges in Python Memory Management:

1. Reference Cycles

- Objects that reference each other (circular references) may not be freed immediately.

- Python’s garbage collector handles this, but it may not always do so efficiently.

2. Memory Leaks

- Long-lived references (e.g., global variables, caches) can unintentionally keep objects in memory.
- Not freeing up unused memory can lead to excessive memory use over time.

3. High Memory Usage in Large Data Structures

- Structures like large lists, dictionaries, or NumPy arrays can consume a lot of memory.
- Poor data structure choices may lead to inefficient memory usage.

4. Inefficient Object Reuse

- Creating many short-lived objects (like strings in a loop) can create memory pressure.
- Using generators instead of lists can help reduce memory use.

5. Manual Memory Control is Limited

- Unlike languages like C/C++, Python does not allow manual memory allocation/deallocation.
- Developers must rely on Python's garbage collector, which might delay cleanup.

6. Third-Party Library Leaks

- Some third-party libraries may not manage memory well, leading to hidden leaks.
- These issues are harder to trace and require profiling tools to detect.

Question 24. How do you raise an exception manually in Python

Answer - 1. The raise statement can be used with an exception type (like ValueError, TypeError, or a custom exception class).

You can provide a message that will be shown when the exception is raised.

You can use raise to re-raise an exception in an except block if you want to propagate it after handling it.

Question 25. Why is it important to use multithreading in certain applications?

Answer - Multithreading is important in certain applications to improve performance, efficiency, and responsiveness. For I/O-bound tasks, it allows concurrent execution while waiting for external resources, reducing idle time. In CPU-bound tasks, it enables parallel processing across multiple cores, speeding up computations. Multithreading also enhances user interface responsiveness by allowing background tasks without freezing the GUI. Additionally, it supports real-time systems and task parallelism, ensuring timely execution of critical operations. While multithreading optimizes resource use, it requires careful management to avoid issues like race conditions and deadlocks, making it essential for complex, concurrent applications.

Practical Question

```
[ ]
```

```
#1. How can you open a file for writing in Python and write a string to it?

with open('example.txt', 'w') as file:

    file.write("Hello, world!")
```

```
[ ]
```

```
#2. Write a Python program to read the contents of a file and print each line.

with open('example.txt', 'r') as file:

   for line in file:

      print(line.strip())
```

```
Hello, world!
```

```
[ ]
```

```
#3. How would you handle a case where the file doesn't exist while trying to open it for
reading?
```

```
try:

  with open("non_existent_file.txt", "r") as file:

    content = file.read()

    print(content)

except FileNotFoundError:

  print("Error: The file does not exist.")
```

```
Error: The file does not exist.
```

```
[ ]
```

```
#4. Write a Python script that reads from one file and writes its content to another file

def copy_file(source_file, destination_file):

  """Copies the content of one file to another.



  Args:

    source_file: The path to the source file.

    destination_file: The path to the destination file.

  """

  try:

    with open(source_file, 'r') as source, open(destination_file, 'w') as destination:

      for line in source:

        destination.write(line)

    print(f"File '{source_file}' copied to '{destination_file}' successfully.")

  except FileNotFoundError:
        print(f"Error: Source file '{source_file}' not found.")

    except Exception as e:

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



# Example usage

source_file = 'input.txt'  # Replace with your source file path

destination_file = 'output.txt'  # Replace with your destination file path

copy_file(source_file, destination_file)
```

```
Error: Source file 'input.txt' not found.
```

```
[ ]
```

```
#5. How would you catch and handle division by zero error in Python

try:

    numerator = 10

    denominator = 0

    result = numerator / denominator

    print("Result:", result)

except ZeroDivisionError:

    print("Error: Cannot divide by zero.")
```

```
Error: Cannot divide by zero.
```

```
[ ]
```

```
#6.  Write a Python program that logs an error message to a log file when a division by zero exception occurs

import logging
```

```python
# Configure logging
logging.basicConfig(filename='error.log', level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s')


# Division operation with error handling
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        logging.error("Attempted to divide by zero: %s", e)
        print("Error: Cannot divide by zero.")


# Example usage
divide(10, 0)
```

```
ERROR:root:Attempted to divide by zero: division by zero
Error: Cannot divide by zero.
```

```
[ ]
```

```python
#7. How do you log information at different levels (INFO, ERROR, WARNING) in Python using the logging module?
import logging


# Configure the logging system
```

```
logging.basicConfig(

    filename='app.log',      # Log file name

    level=logging.DEBUG,      # Minimum level to capture

    format='%(asctime)s - %(levelname)s - %(message)s'

)



# Log messages at different levels

logging.debug("This is a debug message")      # For detailed diagnostic output

logging.info("This is an info message")       # For general information

logging.warning("This is a warning message")   # For something unexpected but not an error

logging.error("This is an error message")      # For errors that occur during execution

logging.critical("This is a critical message") # For very serious errors
```

```
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

```
[ ]
```

```
#8. Write a program to handle a file opening error using exception handling.

try:

    # Try to open a file that may not exist

    with open('nonexistent_file.txt', 'r') as file:

        content = file.read()

        print(content)
```

```
except FileNotFoundError as e:

    print("Error: File not found.")

    print(f"Details: {e}")
```

```
Error: File not found.
Details: [Errno 2] No such file or directory: 'nonexistent_file.txt'
```

```
[ ]
```

```
#9.  How can you read a file line by line and store its content in a list in Python

# Open the file for reading

with open('example.txt', 'r') as file:

    # Read lines into a list, stripping newline characters

    lines = [line.strip() for line in file]



# Print the list

print(lines)
```

```
['Hello, world!']
```

```
[ ]
```

```
#10. How can you append data to an existing file in Python

# Open the file in append mode

with open('example.txt', 'a+') as file:

    file.write("Another line.\n")

    file.seek(0)  # Move to the start to read the full content

    print(file.read())
```

```
Hello, world!Another line.
Another line.
```

```
[ ]
```

```
'''11. Write a Python program that uses a try-except block to handle an error when attempting to access a

dictionary key that doesn't exist'''

# Define a sample dictionary

person = {

    "name": "Alice",

    "age": 30

}



# Attempt to access a non-existent key with error handling

try:

    print("City:", person["city"])

except KeyError as e:

    print(f"Error: Key '{e}' not found in the dictionary.")
```

```
Error: Key ''city'' not found in the dictionary.
```

```
[ ]
```

```
#12. Write a program that demonstrates using multiple except blocks to handle different types of exceptions

try:

    num = int(input("Enter a number: "))

    result = 10 / num
    print("Result:", result)



    # Attempt to access a non-existent key

    data = {"name": "Alice"}

    print("Age:", data["age"])



except ZeroDivisionError:

    print("Error: Cannot divide by zero.")



except ValueError:

    print("Error: Invalid input. Please enter a number.")



except KeyError as e:

    print(f"Error: Key '{e}' not found in the dictionary.")
```

```
Enter a number: 21
Result: 0.47619047619047616
Error: Key ''age'' not found in the dictionary.
```

```
[ ]
```

```
#13. How would you check if a file exists before attempting to read it in Python

import os



file_path = 'example.txt'
```

```
if os.path.exists(file_path):

    with open(file_path, 'r') as file:

        content = file.read()

        print(content)

else:

    print("Error: File does not exist.")
```

```
Hello, world!Another line.
Another line.
```

```
[ ]
```

```
#14.  Write a program that uses the logging module to log both informational and error messages

import logging



# Configure the logging system

logging.basicConfig(

    filename='app.log',      # Log file name

    level=logging.DEBUG,     # Capture all levels of logs (DEBUG and above)

    format='%(asctime)s - %(levelname)s - %(message)s'

)



# Log an informational message

logging.info("This is an informational message.")
```

```
# Simulate an error and log it

try:

    result = 10 / 0  # Division by zero to trigger an error

except ZeroDivisionError as e:

    logging.error("An error occurred: %s", e)



# Log another informational message

logging.info("The program has completed successfully.")
```

```
INFO:root:This is an informational message.
ERROR:root:An error occurred: division by zero
INFO:root:The program has completed successfully.
```

[ ]

```
#15.  Write a Python program that prints the content of a file and handles the case when the file is empty

try:

    # Open the file for reading

    with open('example.txt', 'r') as file:

        content = file.read()



        # Check if the file is empty

        if content:

            print(content)

        else:
```

```
        print("The file is empty.")



except FileNotFoundError:

    print("Error: The file does not exist.")

except Exception as e:

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

```
Hello, world!Another line.
Another line.
```

keyboard_arrow_down

16. Demonstrate how to use memory profiling to check the memory usage of a small program

How it Works:

1. memory_profiler: This package provides tools for memory profiling in Python.
2. @profile decorator: This decorator is applied to the function you want to profile (process_data in this example).
3. %load_ext memory_profiler: This magic command loads the memory profiler extension in the IPython environment.
4. %mprun magic command: This command runs the specified function (process_data) and generates a memory usage report.

[ ]

```
!pip install memory-profiler
```

```
Requirement already satisfied: memory-profiler in
/usr/local/lib/python3.11/dist-packages (0.61.0)
Requirement already satisfied: psutil in
/usr/local/lib/python3.11/dist-packages (from memory-profiler) (5.9.5)
```

```
[ ]
```

```python
from memory_profiler import profile



@profile

def process_data():

    a = [i for i in range(1000000)]

    b = [i * 2 for i in a]

    return b



if __name__ == "__main__":

    process_data()
```

```
ERROR: Could not find file <ipython-input-54-896f98296e17>
NOTE: %mprun can only be used on functions defined in physical files,
and not in the IPython environment.
```

```
[ ]
```

```python
%load_ext memory_profiler
```

```
The memory_profiler extension is already loaded. To reload it, use:
  %reload_ext memory_profiler
```

```
[ ]
```

```python
#17. Write a Python program to create and write a list of numbers to a file, one number per line

# List of numbers

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
# Open the file in write mode

with open('numbers.txt', 'w') as file:

    # Write each number to the file, one per line

    for number in numbers:

        file.write(f"{number}\n")



print("Numbers have been written to 'numbers.txt'.")
```

```
Numbers have been written to 'numbers.txt'.
```

```
[ ]
```

```
#18. How would you implement a basic logging setup that logs to a file
with rotation after 1MB?

import logging

from logging.handlers import RotatingFileHandler

# Set up a rotating file handler that rotates the log file after 1MB

handler = RotatingFileHandler('app.log', maxBytes=1e6, backupCount=3)
# 1MB = 1e6 bytes

handler.setLevel(logging.INFO)  # Log level to capture INFO and above

# Create a logging format

formatter = logging.Formatter('%(asctime)s - %(levelname)s -
%(message)s')

handler.setFormatter(formatter)

# Set up the root logger

logger = logging.getLogger()

logger.setLevel(logging.INFO)
logger.addHandler(handler)

# Example logging messages

logger.info("This is an informational message.")

logger.warning("This is a warning message.")

logger.error("This is an error message.")
```

```
INFO:root:This is an informational message.
WARNING:root:This is a warning message.
ERROR:root:This is an error message.
```

```
[ ]
```

```
#19. Write a program that handles both IndexError and KeyError using a try-except block

def handle_errors():

    # List and dictionary for demonstration

    my_list = [1, 2, 3]

    my_dict = {"name": "Alice", "age": 30}



    try:

        # Trying to access an invalid index in the list

        print(my_list[5])



        # Trying to access a non-existent key in the dictionary

        print(my_dict["address"])



    except IndexError as e:

        print(f"IndexError: {e} - The list index is out of range.")
    except KeyError as e:

        print(f"KeyError: {e} - The key does not exist in the dictionary.")



if __name__ == "__main__":

    handle_errors()
```

```
IndexError: list index out of range - The list index is out of range.
```

```
[ ]
```

```
#20. How would you open a file and read its contents using a context manager in Python.

# Using a context manager to open and read a file

with open('example.txt', 'r') as file:

    content = file.read()  # Read the entire file content

    print(content)  # Print the content of the file
```

```
Hello, world!Another line.
Another line.
```

```
[ ]
```

```
#21. Write a Python program that reads a file and prints the number of occurrences of a
specific word

def count_word_occurrences(file_name, word_to_find):

    try:

        # Open the file in read mode

        with open(file_name, 'r') as file:
            content = file.read()  # Read the entire content of the file

            # Count the occurrences of the specific word
            word_count = content.lower().split().count(word_to_find.lower())

            print(f"The word '{word_to_find}' appears {word_count} times in the file.")

    except FileNotFoundError:
        print(f"Error: The file '{file_name}' does not exist.")
    except Exception as e:
        print(f"An error occurred: {e}")

# Example usage
count_word_occurrences('example.txt', 'Python')
```

```
The word 'Python' appears 0 times in the file.
```

```
[ ]
```

```
#22.  How can you check if a file is empty before attempting to read its contents?
import os

file_path = 'example.txt'

# Check if the file exists and its size
```

```
if os.path.exists(file_path) and os.path.getsize(file_path) > 0:

    with open(file_path, 'r') as file:

        content = file.read()

        print(content)

else:

    print("The file is empty or does not exist.")
```

```
Hello, world!Another line.
Another line.
```

```
[ ]
```

```
#23. Write a Python program that writes to a log file when an error occurs during file handling.

import logging



# Configure the logging setup to write errors to a log file

logging.basicConfig(

    filename='file_handling_errors.log',  # Log file name

    level=logging.ERROR,  # Log only ERROR level messages and above

    format='%(asctime)s - %(levelname)s - %(message)s'

)



def read_file(file_path):

    try:

        # Try to open and read the file

        with open(file_path, 'r') as file:
            content = file.read()

            print(content)



    except FileNotFoundError:

        logging.error(f"File '{file_path}' not found.")

        print(f"Error: The file '{file_path}' does not exist.")



    except PermissionError:

        logging.error(f"Permission denied while trying to open '{file_path}'.")

        print(f"Error: Permission denied for file '{file_path}'.")



    except Exception as e:

        logging.error(f"An unexpected error occurred: {e}")

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

# Example usage

read_file('example.txt')
```

```
Hello, world!Another line.
Another line.
```