Theory Questions :- **Files, exceptional handling, logging and memory management.**

(by Ritesh)

Q1. What is the difference between compiled and interpreted languages?

Concept Breakdown:

Programming languages can be broadly divided into **compiled** and **interpreted**, based on how the source code is converted into machine code.

→ Compiled Languages

- **Process**: Source code → Compiler → Machine code (binary) → Executed directly by CPU.
- **Execution speed**: Very fast (since the program is already in machine language).
- **Examples**: C, C++, Rust, Go.
- **Analogy**: Think of it like translating an entire book into Hindi before reading.
  Once translated, you can read it anytime without needing the translator again.

→ Interpreted Languages

- **Process**: Source code → Interpreter → Line-by-line execution.
- **Execution speed**: Slower (since translation happens during execution).
- **Examples**: Python, JavaScript, Ruby.
- **Analogy**: Like having a translator read each sentence of an English book aloud in Hindi while you listen.
  The translator is always needed during reading.

→ Mixed Languages

Some languages (like Java, Python) use a **hybrid approach**:

- Java → Compiled into bytecode → Run by the JVM (Java Virtual Machine).
- Python → Internally compiled into bytecode (.pyc files) → Run by the Python interpreter.

→Tabular Comparison

Feature	Compiled Languages	Interpreted Languages
Translation method	Entire code compiled first	Line by line during runtime
Execution speed	Faster	Slower
Dependency	No interpreter at runtime	Interpreter needed
Example languages	C, C++, Rust, Go	Python, JavaScript, Ruby
Error detection	At compile time	At runtime

Q2. What is Exception Handling in Python?

Concept Breakdown:

- **Exception** = An error that occurs during program execution (e.g., dividing by zero, accessing a file that doesn't exist).
- If not handled, the program **crashes** immediately.
- **Exception Handling** in Python allows us to detect errors and deal with them gracefully using keywords:
  - `try` → Code block to test for errors.
  - `except` → Code block that runs if an error occurs.
  - `else` → Runs if no error occurs.
  - `finally` → Runs no matter what (used for cleanup like closing files).

Analogy:

Imagine you're driving. If a tire bursts (error), instead of letting the car crash, you pull over safely (handle the exception) and fix it.

Example Code :

```
try:
    x = 10 / 0    # risky code
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
finally:
    print("Program finished.")


# Output:
Error: Cannot divide by zero.
Program finished.
```

note :->

This way, program does not crash.
Used in file handling, user input, databases, etc.

Q3. What is the purpose of the `finally` block in exception handling?

The `finally` block is used to write code that **always runs**,
whether an exception occurs or not.

Commonly used for cleanup tasks like:

- Closing files
- Releasing resources
- Disconnecting from databases

Example Code :

```
try:
    x = 10 / 0     # risky code
except ZeroDivisionError:
    print("Error: Division by zero.")
finally:
    print("This will always run.")


# Output:
 Error: Division by zero.
 This will always run.
```

Purpose → To ensure important cleanup code is executed every time.

Q4. What is Logging in Python?

- **Logging** is a way to record messages about what a program is doing.
- It helps in **debugging, monitoring, and tracking errors** without stopping the program.
- Unlike `print()`, logging is more powerful:
    - Different levels of messages (INFO, WARNING, ERROR, CRITICAL).

- Can save logs to a **file** for future analysis.
- Used in **real-world applications** like servers, banking systems, and AI models to trace issues.

Analogy:

Think of it like a black box in an airplane: it records everything for later investigation.

Example Code :

```
import logging

# Configure logging (level and format)
logging.basicConfig(level=logging.INFO, format="%(levelname)s - %(message)s")

logging.info("Program started")      # Normal info message
logging.warning("Low disk space")    # Warning message
logging.error("File not found")      # Error message

# Output:
INFO - Program started
WARNING - Low disk space
ERROR - File not found
```

→ Key Point:

- Logging is better than `print()` for real projects.
- It helps developers **track issues, debug faster, and maintain programs** safely.

Q5. What is the significance of the `__del__` method in Python?

- In Python, `__del__` is called a **destructor method**.
- It is executed **automatically when an object is deleted** or goes out of scope.
- Purpose:
  - Release resources (close files, free memory, disconnect database).
  - Perform final cleanup before the object is destroyed.
- It is the opposite of the constructor (`__init__`).

Think of `__init__` as "birth of an object" and `__del__` as "last rites of an object".

Example Code :

```
class Demo:
    def __init__(self, name):
        self.name = name
        print(f"Object {self.name} created.")

    def __del__(self):
        print(f"Object {self.name} destroyed. Resources cleaned.")

# Create and delete object
obj = Demo("Test")
del obj   # explicitly deleting object


# Output:
Object Test created.
Object Test destroyed. Resources cleaned.
```

Constructor vs Destructor in Python

Feature	Constructor (`__init__`)	Destructor (`__del__`)
Purpose	Initialize object (setup)	Cleanup before object is destroyed
Called when	Object is created	Object is deleted / goes out of scope
Usage examples	Setting initial values, open files	Closing files, free memory

---

→ Key Point:

- Use `__init__` to **prepare** an object.
- Use `__del__` to **clean up** after an object.
  Together, they ensure smooth lifecycle management.

Q6. What is the difference between `import` and `from ... import` in Python?

Python allows us to reuse code by importing modules.

- **`import module`**
  - Brings in the *whole module*.
  - You must use the module name as a prefix to access functions/variables.

- **`from module import name`**
  - Brings in *specific functions or variables* directly.
  - You can use them **without prefixing** the module name.

Analogy:

- `import` is like bringing the **entire toolbox** (you must open it to pick a tool).
- `from ... import` is like taking **only the specific tool you need** from the box.

Example Code :

```
import math
print("Using import:", math.sqrt(16))   # need module name prefix

from math import sqrt
print("Using from ... import:", sqrt(25))   # directly accessible

# Output:
Using import: 4.0
Using from ... import: 5.0
```

Comparison Table

Feature	`import module`	`from module import name`
Import style	Whole module	Specific part of module
Access	`module.function()`	`function()` directly
Namespace clarity	Clear (avoids name conflicts)	Can cause conflicts if names overlap
Example	`import math` → `math.sqrt(16)`	`from math import sqrt` → `sqrt(16)`

→ Key Point:

- Use `import` when you need **many functions** from a module.
- Use `from ... import` when you need **only a few specific functions** for cleaner code.

Q7. How can you handle multiple exceptions in Python?

- In Python, sometimes more than one type of error can occur.
- You can handle **multiple exceptions** using:
  1. **Multiple `except` blocks** → One block for each error type.
  2. **Single `except` with tuple** → Catch multiple errors in one block.

This makes programs more **robust** by covering different possible problems.

Example 1: Multiple except blocks

```
try:
   num = int("abc")   # ValueError
   result = 10 / 0    # ZeroDivisionError
except ValueError:
   print(" Invalid number conversion.")
except ZeroDivisionError:
   print(" Division by zero error.")

# Output:
Invalid number conversion.
```

Example 2: Single except with tuple

```
try:
    num = int("abc")   # ValueError
except (ValueError, ZeroDivisionError) as e:
    print(" Error occurred:", e)


# Output:
Error occurred: invalid literal for int() with base 10: 'abc'
```

→ Key Points:

- Use **separate `except` blocks** for different handling.
- Use a **tuple in one block** if same handling is fine for multiple errors.
- Always try to catch **specific exceptions** instead of a generic one (`except:`).

This makes your code safer and easier to debug.

Q8. What is the purpose of the `with` statement when handling files in Python?

- The `with` statement is used for **resource management** in Python.
- When working with files, it ensures:
    1. The file is **opened safely**.

2. The file is **automatically closed** after use, even if an error occurs.

- Without `with`, you must close files manually using `file.close()`.
- With `with`, Python handles cleanup automatically.

Analogy:

Think of it like borrowing a library book.
Even if you forget to return it, the librarian (`with`) makes sure it gets returned.

Example Code: File handling with and without `with`

```
# Without 'with'
file = open("sample.txt", "w")
file.write("Hello, World!")
file.close()   # must close manually

# With 'with'
with open("sample.txt", "w") as f:
    f.write("Hello, Python!")   # file auto-closes after block ends
```

→ Key Points:

- Prevents resource leaks (files left open).
- Makes code shorter and cleaner.
- Used for **files, databases, sockets, and more**.

Best practice: Always use `with` when handling files in Python.

Q9. What is the difference between Multithreading and Multiprocessing?

- **Multithreading**
  - Uses multiple **threads** within a single process.
  - Threads share the same memory space.
  - Best for tasks that wait (I/O-bound tasks like downloading files, reading data).
- **Multiprocessing**
  - Uses multiple **processes**, each with its own memory space.
  - True parallelism (runs on multiple CPU cores).

- Best for heavy computations (CPU-bound tasks like data processing, ML training).

Analogy:

- Multithreading = Many workers in **one kitchen** sharing ingredients.
- Multiprocessing = Many workers in **different kitchens**, each with their own ingredients.

Example: Simple demo

```
import threading, multiprocessing, time

def task(name):
    print(f"Task {name} started")
    time.sleep(1)
    print(f"Task {name} finished")

# Multithreading
print("\n--- Multithreading ---")
t1 = threading.Thread(target=task, args=("A",))
t2 = threading.Thread(target=task, args=("B",))
t1.start(); t2.start()
t1.join(); t2.join()

# Multiprocessing
print("\n--- Multiprocessing ---")
p1 = multiprocessing.Process(target=task, args=("X",))
p2 = multiprocessing.Process(target=task, args=("Y",))
p1.start(); p2.start()
p1.join(); p2.join()

# Output:

--- Multithreading ---
Task A started
Task B started
Task A finished
Task B finished

--- Multiprocessing ---
Task X started
Task Y started
Task X finished
Task Y finished
```

Comparison Table

Feature	Multithreading	Multiprocessing
Definition	Multiple threads in one process	Multiple independent processes
Memory	Shared memory	Separate memory
Best for	I/O-bound tasks	CPU-bound tasks
Performance	Limited by GIL (in CPython)	True parallel execution
Example use case	Web servers, file I/O	Data science, image processing

---

⟶ Key Point:

- Use **multithreading** for tasks that wait on I/O.
- Use **multiprocessing** for heavy CPU work that benefits from parallel cores.

Q10. What are the advantages of using Logging in a program?

Logging is a way to record events, errors, and information about a program's execution.
It is **better than using print statements** because it is more flexible and professional.

Advantages of Logging:

1. **Error Tracking** ⟶ Helps find where and why errors occurred.
2. **Debugging** ⟶ Easier to debug large applications.
3. **Different Levels** ⟶ Can record info as INFO, WARNING, ERROR, CRITICAL.
4. **Persistent Records** ⟶ Logs can be saved to a file for future analysis.
5. **Non-intrusive** ⟶ Doesn't interrupt program flow like print().
6. **Scalability** ⟶ Used in large projects, servers, and production systems.

Analogy:

Logging is like keeping a **diary** of everything your program does,
so you can look back and understand what went wrong.

Example Code:

```
import logging

logging.basicConfig(level=logging.INFO, format="%(levelname)s - %(message)s")
```

```
logging.info("Program started")      # Info message
logging.warning("Low memory")        # Warning
logging.error("File not found")      # Error


# Output:
INFO - Program started
WARNING - Low memory
ERROR - File not found
```

⟶ Key Point:

- Logging makes programs **more reliable and easier to maintain**.
- In real-world projects, logging is essential for **debugging, monitoring, and audits**.

Q11. What is Memory Management in Python?

- **Memory management** means how Python allocates and frees memory for objects.
- Python does this **automatically** using:
    1. **Private Heap Space** ⟶ All Python objects and data are stored in a special memory area.
    2. **Memory Manager** ⟶ Allocates memory for objects.
    3. **Garbage Collector** ⟶ Frees memory by removing objects that are no longer used.

You don't need to manually manage memory like in C/C++;
Python handles it, making programming easier and safer.

Example Code: Garbage Collection Demo

```
import gc

class Demo:
    def __del__(self):
        print("Object destroyed, memory freed.")

obj = Demo()
del obj        # explicitly delete object
gc.collect()   # force garbage collection
```

```
# Output:
Object destroyed, memory freed.
```

Key Points of Python Memory Management

Feature	Explanation
Heap Space	All objects are stored here
Automatic Handling	No need for manual memory allocation (like malloc)
Garbage Collector	Removes unused objects to free memory
Reference Counting	Tracks how many variables point to an object

---

Real-World Insight:

- Helps Python programs run smoothly without memory leaks.
- Important in **large projects, data analysis, and AI/ML** where memory usage is heavy.

Q12. What are the basic steps involved in Exception Handling in Python?

Exception handling allows programs to **handle errors gracefully** instead of crashing.
The basic steps are:

1. **try** → Place risky code inside this block.
2. **except** → Handles the error if it occurs.
3. **else** → Runs if no error happens (optional).
4. **finally** → Runs always (for cleanup, optional).

Think of it like a safety net: If something goes wrong, your program falls safely into `except`.

Example Code

```
try:
    num = int("abc")   # risky code (will cause ValueError)
except ValueError:
    print("Error: Invalid conversion to integer.")
else:
    print(" No error occurred.")
finally:
    print(" Execution finished.")

# Output:
```

```
Error: Invalid conversion to integer.
Execution finished
```

Key Points:

- `try` → Write risky code.
- `except` → Handle the error.
- `else` → Runs if no exception.
- `finally` → Always runs, good for cleanup.

These steps make Python programs **robust, reliable, and professional**.

Q13. Why is Memory Management important in Python?

Memory management ensures that Python programs use memory **efficiently and safely**. It prevents waste, avoids crashes, and keeps performance smooth.

Importance:

1. **Efficient Resource Use** → Prevents memory wastage by reusing space.
2. **Program Stability** → Avoids crashes due to "out of memory" errors.
3. **Automatic Garbage Collection** → Frees unused objects automatically.
4. **Scalability** → Important for big applications (AI, ML, Data Science) where large datasets are used.
5. **Security** → Proper management prevents memory leaks or corruption.

Without memory management, programs would slow down, consume too much RAM, and sometimes even **crash the system**.

Example Code

```
import gc

x = [i for i in range(1000000)]  # large memory usage
print("Big list created.")

del x        # free memory
gc.collect() # force garbage collection
print("Memory cleaned.")


# Output:
```

```
Big list created.
Memory cleaned.
```

Key Point:

- Memory management is important because it keeps Python programs **fast, safe, and efficient**,
  especially in large-scale applications.

Q14. What is the role of `try` and `except` in Exception Handling?

- **try block** → Contains code that might raise an error (risky code).
- **except block** → Contains code to handle the error if it occurs.

Together, they prevent the program from **crashing** and allow it to handle errors gracefully.

Analogy:

- `try` is like testing a new gadget carefully.
- `except` is like the safety net that catches it if it breaks.

Example Code:

```
try:
    num = int("abc")   # risky operation (ValueError)
except ValueError:
    print(" Error: Invalid number conversion.")


# Output:
Error: Invalid number conversion.
```

Key Point:

- **try** → place risky code.
- **except** → handle error if it happens.

This ensures the program continues running safely instead of crashing.

Q15. How does Python's Garbage Collection system work?

- Garbage Collection (GC) = Automatic process of **freeing memory** that is no longer in use.
- Python mainly uses:
    1. **Reference Counting** → Every object has a counter of references.
       When the counter = 0 → object is deleted.
    2. **Generational Garbage Collector** → Handles cases where objects refer to each other (circular references).
        - Divides objects into generations (0, 1, 2).
        - Frequently checks young objects, rarely checks old ones (optimization).

→ Why it matters:

- Prevents **memory leaks**.
- Makes Python programs efficient without manual memory management.

Example: Reference counting & garbage collection

```
import gc

class Demo:
    def __del__(self):
        print("Object destroyed, memory freed.")

# Create object
obj = Demo()

# Delete reference
del obj

# Force garbage collection manually
gc.collect()


# Output:
Object destroyed, memory freed.
```

Key Point:

Python's Garbage Collection system works by:

- Counting references,
- Cleaning up unused objects,
- Using a **generational GC** for efficiency.

This ensures memory is always managed automatically and safely.

Q16. What is the purpose of the `else` block in Exception Handling?

- The `else` block is used **after the try block**.
- It runs **only if no exception occurs** inside `try`.
- Helps separate **error-handling code** (`except`) from **normal code** (`else`).

→ Think of it as:

- `try` = risky work
- `except` = handle errors
- `else` = run extra code if everything was safe
- `finally` = cleanup (always runs)

This makes the code **clearer and more structured**.

Example Code :

```
try:
    num = int("10")   # safe code
except ValueError:
    print("Error: Invalid number.")
else:
    print(" Success! No error, so else block runs.")
finally:
    print(" Finally always runs.")

# Output :
Success! No error, so else block runs.
Finally always runs.
```

→ Key Point:

- `else` = runs only when **no exception occurs**.
- Makes the program flow more **organized** and readable.

Q17. What are the common logging levels in Python?

Python's `logging` module provides **different levels** to show the importance/severity of messages.
Each level helps developers **track, debug, or monitor** programs.

Common Logging Levels:

1. **DEBUG (10)** → Detailed information (for developers).
2. **INFO (20)** → General events, confirmations program is working.
3. **WARNING (30)** → Something unexpected, but program still works.
4. **ERROR (40)** → Serious issue, program may not continue a part.
5. **CRITICAL (50)** → Very serious error, program may crash.

→ Levels are ordered: `DEBUG < INFO < WARNING < ERROR < CRITICAL`

Example Code : Using logging levels

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG)

logging.debug("This is DEBUG (detailed info).")
logging.info("This is INFO (normal flow).")
logging.warning("This is WARNING (something unusual).")
logging.error("This is ERROR (something failed).")
logging.critical("This is CRITICAL (program may crash).")


# Output:
DEBUG:root:This is DEBUG (detailed info).
INFO:root:This is INFO (normal flow).
WARNING:root:This is WARNING (something unusual).
ERROR:root:This is ERROR (something failed).
CRITICAL:root:This is CRITICAL (program may crash).
```

→ Key Point:

- Logging levels help **categorize messages**.
- Useful for **debugging, monitoring, and error tracking**.
- Default level = `WARNING` (messages below WARNING are ignored unless configured).

Q18. What is the difference between `os.fork()` and `multiprocessing` in Python?

Feature	os.fork()	multiprocessing module
Platform Support	Works only on **Unix/Linux** (not Windows)	Works on **all platforms** (cross-platform)
Usage	Creates a **child process** directly	Provides a **high-level API** for processes
Control	Low-level, manual process management	Easy to use, manages processes automatically
Communication	No built-in support for inter-process communication	Has **Queues, Pipes, Managers** for IPC
Safety	Can be error-prone if misused	Safer and recommended in Python

→ In short:

- `os.fork()` → **low-level** and platform-dependent.
- `multiprocessing` → **high-level, portable, and safer**.

Example: Using os.fork() (only works on Unix/Linux)

```
import os

def fork_example():
    pid = os.fork()
    if pid == 0:
        print(" Child process created")
    else:
        print(" Parent process continues")

# fork_example()   # Uncomment to run on Linux/Mac

# Output:
Child process created
Parent process continues
```

Example: Using multiprocessing (works everywhere)

```
from multiprocessing import Process

def worker():
    print("Child process is running (multiprocessing).")

if __name__ == "__main__":
    p = Process(target=worker)
    p.start()
    p.join()
```

```
    print("Parent process finished.")


# Output:
Child process is running (multiprocessing).
Parent process finished.
```

→ Key Point:

- Use **`multiprocessing`** for cross-platform, safe, and modern Python code.
- Use **`os.fork()`** only if you need **low-level control** on Unix/Linux.

Q19. What is the importance of closing a file in Python?

When we open a file in Python using `open()`, the program uses system resources (like memory and file handles).
Closing the file is **important** because:

1. **Frees Resources** → Releases memory and file handle back to the system.
2. **Saves Data** → Ensures all data is properly written from buffer to the file.
3. **Prevents Errors** → Avoids file corruption or accidental data loss.
4. **Best Practice** → Makes the program safe and efficient.

→ Use `file.close()` or (better) the `with` statement, which closes the file automatically.

Example Code: Without closing the file

```
f = open("sample.txt", "w")
f.write("Hello, World!")
# f.close()  # If not called, file may remain open and data may not be saved properly
```

Example Code: Using 'with' (auto closes the file)

```
with open("sample.txt", "w") as f:
    f.write("Hello safely with WITH statement!")
# File is automatically closed after the block
```

→ Key Point:

- Always close files to **save data and free resources**.
- Prefer using `with open(...)` because it is cleaner and safer.

Q20. What is the difference between `file.read()` and `file.readline()` in Python?

Method	Description
`file.read()`	Reads the **entire file** (or given number of characters) as a single string.
`file.readline()`	Reads the file **one line at a time** until a newline `\n` is found.

→ In short:

- `read()` → Whole content or chunk.
- `readline()` → Only the next line.

```
# Create a sample file
with open("sample.txt", "w") as f:
    f.write("Line 1\nLine 2\nLine 3")

# Using file.read()
with open("sample.txt", "r") as f:
    content = f.read()
    print("file.read() output:\n", content)

# Using file.readline()
with open("sample.txt", "r") as f:
    print("file.readline() output:")
    print(f.readline())  # Reads only first line
    print(f.readline())  # Reads only second line

# Output:
file.read() output:
Line 1
Line 2
Line 3

file.readline() output:
Line 1
Line 2
```

→ Key Point:

- Use **`read()`** when you need the full file content.
- Use **`readline()`** when processing files **line by line** (useful for big files).

Q21. What is the logging module in Python used for?

The `logging` module in Python is used to **track events** that happen while a program runs. It is mainly for **debugging, monitoring, and error tracking**.

⟶ Uses of logging module:

1. Records important information like errors, warnings, debug messages.
2. Helps developers **trace program flow** and diagnose problems.
3. Allows saving logs to **console, files, or external systems**.
4. Provides different **log levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL).

⟶ Unlike `print()`, `logging` is more flexible, professional, and suitable for real-world applications.

Example Code: Using logging module

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format="%(levelname)s - %(message)s")

logging.debug("This is a debug message")
logging.info("Program is running fine")
logging.warning("This is a warning")
logging.error("An error occurred")
logging.critical("Critical issue!")

# Output:
DEBUG - This is a debug message
INFO - Program is running fine
WARNING - This is a warning
ERROR - An error occurred
CRITICAL - Critical issue!
```

⟶ Key Point:

- The `logging` module is used to **record program events** for debugging and monitoring.
- It is **better than print()** for professional applications.

Q22. What is the os module in Python used for in file handling?

The `os` (Operating System) module in Python provides functions to **interact with the operating system**.
In file handling, it is mainly used to:

1. **Create, remove, rename files and directories**
2. **Check if a file or folder exists**
3. **Work with file paths** (join, split, get current directory)
4. **Navigate directories** (change or list contents)

→ The `os` module gives Python programs **control over files and folders** like a file manager.

Example Code:

```
import os

# Create a new directory
os.mkdir("test_folder")

# Create a new file inside the folder
with open("test_folder/sample.txt", "w") as f:
    f.write("Hello, OS module!")

# List files inside the folder
print("Files in 'test_folder':", os.listdir("test_folder"))

# Rename the file
os.rename("test_folder/sample.txt", "test_folder/renamed_file.txt")

# Check if file exists
print("Does renamed_file.txt exist?", os.path.exists("test_folder/renamed_file.txt"))

# Remove the file
os.remove("test_folder/renamed_file.txt")

# Remove the folder
os.rmdir("test_folder")

# Output:
Files in 'test_folder': ['sample.txt']
Does renamed_file.txt exist? True
```

→ Key Point:

- The `os` module allows Python programs to **manage files and directories** directly from the operating system.
- It is essential for advanced **file handling automation**.

Q23. What are the challenges associated with memory management in Python?

Python manages memory automatically using a **private heap** and a **garbage collector**.
Still, some challenges exist:

1. **Reference Cycles**
    - Objects referring to each other may prevent proper garbage collection.
    - Example: Two objects holding references to each other but not used anywhere.
2. **Memory Leaks**
    - Unreleased memory caused by lingering references or poor coding practices.
    - Example: Storing large unused data in global variables.
3. **High Memory Usage**
    - Python objects (especially lists, dicts) take more memory compared to lower-level languages like C.
4. **Manual Control is Limited**
    - Developers have less direct control over memory (unlike C/C++ with malloc/free).
5. **Performance Overhead**
    - Automatic garbage collection and dynamic typing add overhead, which may slow down memory-intensive applications.

→ Key Point:

Memory management in Python is automatic but not perfect.
Developers must write efficient code, avoid unnecessary references, and use tools (like `gc` module, memory profilers) to optimize usage.

Example Code: Reference cycle causing memory issue

```
import gc
```

```
class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()

# Creating a cycle
a.ref = b
b.ref = a

# Deleting variables
del a
del b

# Forcing garbage collection
print("Garbage before collection:", gc.garbage)
gc.collect()
print("Garbage after collection:", gc.garbage)

# Output:
Garbage before collection: []
Garbage after collection: []
```

→ Summary:

- Python simplifies memory management but **reference cycles, memory leaks, and high memory usage** are common challenges.
- Careful coding and use of profiling tools help minimize these issues.

Q24. How do you raise an exception manually in Python?

- In Python, you can **manually raise exceptions** using the `raise` keyword.
- This is useful when you want to **stop the program** or **signal an error** if a condition is not met.

Syntax:

```
raise ExceptionType("Custom error message")
```

- ExceptionType → built-in or user-defined exception (e.g., ValueError, TypeError)
- "Custom error message" → explains the reason for raising the exception.

Example 1: Raising a ValueError

```
x = -5
if x < 0:
    raise ValueError("x cannot be negative")
```

Example 2: Raising a TypeError

```
def square(num):
    if not isinstance(num, (int, float)):
        raise TypeError(" Input must be a number")
    return num * num

print(square(4))     #  Works
print(square("hi"))  #  Raises TypeError

# Output:
16
Traceback (most recent call last):
...
TypeError:  Input must be a number
```

→ Summary:

- Use `raise` to **manually trigger exceptions**.
- Helps in **validating inputs** and enforcing correct usage of functions.

Q25. Why is it important to use multithreading in certain applications?

- **Multithreading** means running multiple threads (smaller units of a process) **concurrently**.
- It is important in applications where tasks can be performed **in parallel** without waiting for each other.

Importance of Multithreading:

1. **Improves performance** → multiple tasks run together.
2. **Better resource utilization** → CPU is not idle while waiting (e.g., waiting for I/O).
3. **Responsiveness** → useful in apps like GUIs, where the interface must stay active

Example Code:

```
import threading
import time

def task(name):
    print(f"Task {name} started")
    time.sleep(2)   # Simulate a delay (like downloading a file)
    print(f"Task {name} finished")

# Create multiple threads
t1 = threading.Thread(target=task, args=("A",))
t2 = threading.Thread(target=task, args=("B",))

# Start threads
t1.start()
t2.start()

# Wait for both threads to complete
t1.join()
t2.join()

print(" Both tasks completed.")


# Output:
Task A started
Task B started
Task A finished
Task B finished
Both tasks completed.
```

→ Summary:

- Multithreading makes programs **faster, responsive, and efficient**.
- Best for **I/O-heavy applications** like networking, file handling, or web scraping.

Practical Question :-

Q1. How can you open a file for writing in Python and write a string to it?

- Use Python's built-in `open()` function with mode `"w"` (write).
- `"w"` mode creates a new file or overwrites the existing file.
- Use `write()` method to write string data.
- Always close the file using `close()` OR use the `with` statement for automatic closing.

```
# Writing a string to a file
with open("example.txt", "w") as file:
    file.write("Hello, this is my first file write operation in Python!")

print("String written successfully to example.txt")
```

```
String written successfully to example.txt
```

Q2. Write a Python program to read the contents of a file and print each line

```
# Open the file in read mode
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())   # strip() removes extra newline characters
```

```
Hello, this is my first file write operation in Python!
```

Q3. How would you handle a case where the file doesn't exist while trying to open it for reading?

```
try:
    with open("non_existing_file.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("Error: The file does not exist.")
```

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

Q4. Write a Python script that reads from one file and writes its content to another file.

```
# Open the source file in read mode
with open("example.txt", "r") as source_file:
    content = source_file.read()

# Open the destination file in write mode
with open("copy_example.txt", "w") as dest_file:
    dest_file.write(content)

print("Content copied from example.txt to copy_example.txt")
```

```
Content copied from example.txt to copy_example.txt
```

Q5.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: Division by zero is not allowed.")
```

```
Error: Division by zero is not allowed.
```

Q6. Write a Python program that logs an error message to a log file when a division by zero exception occurs

```
import logging

# Configure logging to write to a file
logging.basicConfig(filename="error_log.txt", level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s')

try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
except ZeroDivisionError:
```

```
    logging.error("Division by zero occurred!")
    print("Error logged to error_log.txt")
```

```
ERROR:root:Division by zero occurred!
Error logged to error_log.txt
```

Q7. How do you log information at different levels (INFO, ERROR, WARNING) in Python using the logging module?

```
import logging

# Configure logging to write to a file
logging.basicConfig(filename="app_log.txt", level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# Logging messages at different levels
logging.info("This is an INFO message.")
logging.warning("This is a WARNING message.")
logging.error("This is an ERROR message.")

print(" Messages logged to app_log.txt")
```

```
WARNING:root:This is a WARNING message.
ERROR:root:This is an ERROR message.
 Messages logged to app_log.txt
```

Q8. Write a program to handle a file opening error using exception handling.

```
try:
    # Attempt to open a file that may not exist
    file = open("non_existing_file.txt", "r")
    content = file.read()
    print(content)
    file.close()
except FileNotFoundError:
    print("Error: The file could not be opened because it does not exist.")
```

```
Error: The file could not be opened because it does not exist.
```

Q9. How can you read a file line by line and store its content in a list in Python?

```
# Open the file in read mode
with open("example.txt", "r") as file:
    lines = file.readlines()  # reads all lines into a list

# Print the list
print(lines)
```

```
['Hello, this is my first file write operation in Python!']
```

Q10. 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("\nThis line is appended to the file.")

print("Data appended successfully to example.txt")
```

```
Data appended successfully to example.txt
```

Q11. 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.

```
my_dict = {"name": "Ritesh", "age": 21}

try:
    # Attempt to access a key that doesn't exist
    print(my_dict["city"])
except KeyError:
    print("Error: The key does not exist in the dictionary.")
```

```
Error: The key does not exist in the dictionary.
```

Q12. Write a program that demonstrates using multiple except blocks to handle different types of exceptions.

```
try:
    num1 = int(input("Enter numerator: "))
    num2 = int(input("Enter denominator: "))
    result = num1 / num2
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Please enter a valid integer.")
```

```
Enter numerator: 10
Enter denominator: 0
Error: Division by zero is not allowed.
```

Q13. How would you check if a file exists before attempting to read it in Python?

```
import os

file_name = "example.txt"

if os.path.exists(file_name):
    with open(file_name, "r") as file:
        content = file.read()
        print("File content:")
        print(content)
else:
    print("Error: The file does not exist.")
```

```
File content:
Hello, this is my first file write operation in Python!
This line is appended to the file.
```

Q14. Write a program that uses the logging module to log both informational and error messages.

```
import logging

# Configure logging to write to a file
```

```
logging.basicConfig(filename="app_log.txt", level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# Log informational message
logging.info("This is an informational message.")

# Log error message
logging.error("This is an error message.")

print("Informational and error messages have been logged to app_log.txt")
```

```
ERROR:root:This is an error message.
Informational and error messages have been logged to app_log.txt
```

Q15. Write a Python program that prints the content of a file and handles the case when the file is empty.

```
file_name = "example.txt"

try:
    with open(file_name, "r") as file:
        content = file.read()
        if content:  # Check if file is not empty
            print("File content:")
            print(content)
        else:
            print("The file is empty.")
except FileNotFoundError:
    print("Error: The file does not exist.")
```

```
File content:
Hello, this is my first file write operation in Python!
This line is appended to the file.
```

Q16. Demonstrate how to use memory profiling to check the memory usage of a small program.

```
# Q16. Demonstrate memory profiling in Python

# Step 1: Install memory-profiler (only need to run once)
!pip install memory-profiler

# Step 2: Import the module
from memory_profiler import memory_usage
```

```
# Step 3: Define a small function
def create_list():
    my_list = [i for i in range(100000)]  # creates a list of 100,000 numbers
    return my_list

# Step 4: Measure memory usage
mem_usage = memory_usage(create_list)
print("Memory usage (in MB) during execution:", mem_usage)
```

```
Requirement already satisfied: memory-profiler in /usr/local/lib/python3.12/dist-pack
Requirement already satisfied: psutil in /usr/local/lib/python3.12/dist-packages (fro
Memory usage (in MB) during execution: [128.265625, 128.265625, 128.265625, 128.26562
```

Q17. Write a Python program to create and write a list of numbers to a file, one number per line

```
numbers = [10, 20, 30, 40, 50]

# Open file in write mode
with open("numbers.txt", "w") as file:
    for num in numbers:
        file.write(str(num) + "\n")  # Convert number to string and add newline

print("Numbers written to numbers.txt successfully")
```

```
Numbers written to numbers.txt successfully
```

Q18. How would you implement a basic logging setup that logs to a file with rotation after 1MB?

```
import logging
from logging.handlers import RotatingFileHandler

# Create a rotating file handler
handler = RotatingFileHandler("rotating_log.txt", maxBytes=1_000_000, backupCount=3)
formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)

# Configure logger
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
logger.addHandler(handler)

# Log messages
```

```
logger.info("This is an INFO message.")
logger.warning("This is a WARNING message.")
logger.error("This is an ERROR message.")

print("Logging setup with rotation complete. Check rotating_log.txt")
```

```
INFO:root:This is an INFO message.
WARNING:root:This is a WARNING message.
ERROR:root:This is an ERROR message.
Logging setup with rotation complete. Check rotating_log.txt
```

Q19. Write a program that handles both IndexError and KeyError using a try-except block.

```
my_list = [10, 20, 30]
my_dict = {"name": "Ritesh", "age": 21}

try:
    # Accessing an index that may not exist
    print(my_list[5])

    # Accessing a key that may not exist
    print(my_dict["city"])

except IndexError:
    print("Error: The list index does not exist.")

except KeyError:
    print("Error: The dictionary key does not exist.")
```

```
Error: The list index does not exist.
```

Q20. How would you open a file and read its contents using a context manager in Python?

```
# Using 'with' ensures the file is automatically closed
with open("example.txt", "r") as file:
    content = file.read()

print("File content:")
print(content)
```

```
File content:
Hello, this is my first file write operation in Python!
```

```
This line is appended to the file.
```

Q21. Write a Python program that reads a file and prints the number of occurrences of a specific word.

```
word_to_count = "Python"  # Word to search for

with open("example.txt", "r") as file:
    content = file.read()

# Convert content to lowercase for case-insensitive counting
count = content.lower().count(word_to_count.lower())

print(f"The word '{word_to_count}' occurs {count} times in the file.")
```

```
The word 'Python' occurs 1 times in the file.
```

Q22. How can you check if a file is empty before attempting to read its contents?

```
file_name = "example.txt"

try:
    with open(file_name, "r") as file:
        content = file.read()
        if content:  # Check if file has content
            print("File content:")
            print(content)
        else:
            print("The file is empty.")
except FileNotFoundError:
    print("Error: The file does not exist.")
```

```
File content:
Hello, this is my first file write operation in Python!
This line is appended to the file.
```

Q23. Write a Python program that writes to a log file when an error occurs during file handling

```
import logging

# Configure logging to write to a file
logging.basicConfig(filename="file_error_log.txt", level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s')

file_name = "non_existing_file.txt"

try:
    with open(file_name, "r") as file:
        content = file.read()
except FileNotFoundError as e:
    logging.error(f"Error occurred: {e}")
    print("Error logged to file_error_log.txt")
```