

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

Ans:- ♦ Compiled Languages

How it works: The entire code is translated into machine code by a compiler before execution.

Examples: C, C++, Rust, Go

Pros:

- Faster execution after compilation
- Better performance for large programs

Cons:

- Slower development cycle (needs recompilation after changes)
- Platform-dependent (needs different binaries for different systems)

♦ Interpreted Languages

How it works: The code is executed line-by-line by an interpreter, without converting it into machine code ahead of time.

Examples: Python, JavaScript, PHP, Ruby

Pros:

- Easier to debug and test
- Cross-platform and more flexible

Cons:

- Slower execution compared to compiled languages
- Can be less efficient for heavy computation

### Q2.What is exception handling in Python ?

- Ans:- Exception handling in Python is a way to manage errors or unexpected situations that occur during program execution, so your code doesn't crash.

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

- Ans:- ♦ Purpose of the finally Block in Python Exception Handling: The finally block is used to define code that must run no matter what – whether an exception was raised or not.

### Q4.What is logging in Python ?

-🌀 Logging is a means of tracking events that happen when some software runs. Logging is important for software developing, debugging, and running.

- There are five built-in levels of the log message.
- ☒ Debug
- ☒ Info
- ☒ Warning
- ☒ Error
- ☒ Critical

### Q5 What is the significance of the del method in Python ?

- 🗑️ The **del** method is a special method in Python that is called when an object is about to be destroyed. It allows you to define specific cleanup actions that should be taken when an object is garbage collected.

☒ Purpose:

To perform cleanup operations, like:

Closing files or network connections

Releasing external resources

Logging the deletion of an object.

**Q6. What is the difference between import and from ... import in Python ?** Ans:- ♦ Difference between import and from ... import in Python  
Both are used to include external modules or specific components into your Python code, but they differ in how they bring them in.

✅ 1. import module

- Imports the entire module.
- You access everything using the module name as a prefix.

🔧 Example:

```
import math
```

```
print(math.sqrt(25)) # Access via math.sqrt
```

✅ 2. from module import something

- Imports specific functions, classes, or variables directly.
- You can use them without the module name prefix.

🔧 Example:

```
from math import sqrt
```

```
print(sqrt(25)) # No need for math.sqrt
```

**Q7. How can you handle multiple exceptions in Python ?**

Ans:- ♦ How to Handle Multiple Exceptions in Python

Python allows you to handle different types of exceptions separately or together using the try-except block.

✅ Method 1: Multiple except Blocks (Best for Specific Handling)

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

✅ Method 2: Single except Block with a Tuple

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except (ValueError, ZeroDivisionError):
    print("An error occurred: either invalid input or division by zero.")
```

✅ Method 3: Catching All Exceptions (Not Recommended Unless Logging)

```
try:
    # risky code
except Exception as e:
    print(f"Unexpected error: {e}")
```

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

-Ans:- ♦ Purpose of the with Statement in File Handling (Python) The with statement in Python is used to simplify file handling by ensuring that resources like files are automatically closed after use — even if an error occurs.

✅ Why Use with?

- Automatically closes the file

- Makes code cleaner and more readable
- Reduces the risk of forgetting to close the file (which can cause memory leaks or file locks)

#### Q9. What is the difference between multithreading and multiprocessing ?

- Ans:- ♦ Difference Between Multithreading and Multiprocessing in Python Both are techniques for achieving concurrent execution, but they differ in how they use system resources.

##### ✓ 1. Multithreading

- Uses multiple threads within a single process
- Threads share the same memory space
- Best for I/O-bound tasks (e.g., file handling, network operations)

🧠 Example Use:

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
```

##### ✓ 2. Multiprocessing

- Uses multiple processes
- Each process has its own memory space
- Best for CPU-bound tasks (e.g., data crunching, machine learning)

🧠 Example Use:

```
import multiprocessing

def print_numbers():
    for i in range(5):
        print(i)

process = multiprocessing.Process(target=print_numbers)
process.start()
```

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

- Ans:- ♦ Advantages of Using Logging in a Python Program Logging is much more powerful and flexible than using print() statements. Here's why it's a best practice in real-world programming:

##### ✓ 1. Tracks Program Execution

- Logs help you monitor the flow of your program.
- You can trace what happened and when — useful for debugging and auditing.

##### ✓ 2. Captures Errors and Warnings

- Automatically record exceptions, warnings, or unexpected behavior.
- Helps in identifying and fixing bugs quickly.

##### ✓ 3. Customizable Output Levels

- Supports different severity levels: DEBUG, INFO, WARNING, ERROR, CRITICAL.
- You can filter logs based on importance.

##### ✓ 4. Saves to Files

- Logs can be stored in files, making it easy to review later — especially helpful in production environments.

### Q11. What is memory management in Python ?

- Ans:- Memory management in Python refers to the process of allocating and releasing memory to ensure efficient use of resources while the program runs. Python handles memory management automatically through its garbage collection mechanism, but it also provides developers with tools to manage memory more effectively.

#### 1. Automatic Memory Management (Garbage Collection)

- Python uses automatic garbage collection to free up memory that is no longer in use (i.e., when objects are no longer referenced).
- The garbage collector identifies and removes objects that are not reachable, preventing memory leaks.

#### 2. Memory Allocation

- When a variable or object is created, memory is allocated for it.
- Python manages this allocation using the heap (for dynamically created objects) and the stack (for function calls and local variables).

#### 3. Reference Counting

- Each object in Python has a reference count that tracks the number of references to the object.
- When the reference count drops to zero (i.e., no more references), the object is automatically deallocated.

#### 4. Garbage Collection (GC)

- Python uses cyclic garbage collection to handle circular references (objects referencing each other).
- The gc module provides developers the ability to interact with the garbage collector if needed.

### Q12. What are the basic steps involved in exception handling in Python ?

- Ans:- ♦ Basic Steps Involved in Exception Handling in Python Exception handling in Python helps you manage errors gracefully and prevent your program from crashing unexpectedly.

#### ✓ 1. Use try Block

- Wrap the code that might raise an exception inside a try block.

try:

```
# Risky code here
x = 10 / 0
```

#### ✓ 2. Add except Block(s)

- Catch specific or general exceptions and define what to do when they occur.

```
except ZeroDivisionError:
    print("You can't divide by zero!")
```

#### ✓ 3. (Optional) Use else Block

- Executes only if no exceptions occur in the try block.

```
else:
    print("No errors occurred.")
```

#### ✓ 4. (Optional) Use finally Block

- Executes no matter what, whether an exception occurs or not (often used for cleanup).

```
finally:
    print("Execution completed.")
```

### Q13. Why is memory management important in Python ?

- Ans:- Memory management is crucial in Python to ensure that your program runs efficiently, smoothly, and without crashing due to running out of memory. Even though Python handles memory automatically, understanding and managing it helps write better, more

scalable code.

#### ✓ 1. Prevents Memory Leaks

- Proper memory handling ensures that unused objects don't stay in memory forever.
- Without it, memory usage would grow unnecessarily, eventually slowing or crashing the program.

#### ✓ 2. Improves Performance

- Efficient memory use results in faster execution.
- Python's garbage collector and memory manager reduce the time and effort spent on manual allocation and cleanup.

#### ✓ 3. Supports Large Applications

- Big projects (e.g., web apps, ML models) use a lot of memory.
- Good memory management helps them stay stable and responsive over time.

#### ✓ 4. Avoids Crashes

- Uncontrolled memory usage can lead to MemoryError or system crashes.
- Python's memory manager helps avoid such fatal issues by freeing unused memory.

#### ✓ 5. Enhances Resource Efficiency

- Makes sure memory is used only when needed, and released when it's not — keeping the system resources free for other tasks.

#### ✓ 6. Safer Programs

- Helps prevent bugs caused by uninitialized or corrupted memory.

### Q14.What is the role of try and except in exception handling ?

- Ans:- ♦ Role of try and except in Exception Handling (Python)
- The try and except blocks are core components of Python's exception handling mechanism. They help your program handle errors gracefully without crashing.

#### ✓ 1. try Block

- Used to wrap risky code — the part of your program where an error might occur.

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
```

- Python tries to run the code.
- If no error: it proceeds.
- If an error occurs: it jumps to the corresponding except block.

#### ✓ 2. except Block

- Used to catch and handle the error that was raised in the try block.

```
except ZeroDivisionError:
    print("You cannot divide by zero!")
```

- It prevents the program from crashing.
- You can have multiple except blocks for different error types.


### Q15.How does Python's garbage collection system work ?

- Ans- ♦ How Python's Garbage Collection System Works
- Python's garbage collection (GC) system automatically reclaims memory by removing objects that are no longer needed, helping your program run efficiently without memory leaks.

#### ✓ 1. Reference Counting (Primary Mechanism)

- Every object in Python has a reference count — the number of variables that point to it.

- When an object's reference count drops to zero, Python automatically deletes it.

 Example:

```
a = [1, 2, 3] # Reference count = 1
b = a        # Reference count = 2
del a        # Reference count = 1
del b        # Reference count = 0 → Collected
```

## ✓ 2. Garbage Collector for Cyclic References

- Sometimes, objects reference each other in a cycle, so their reference count never reaches 0 — like:

```
a = {}
b = {}
a["b"] = b
b["a"] = a
```

## ✓ 3. Generational Garbage Collection

- Python divides objects into 3 generations:
- Gen 0: New objects
- Gen 1: Survived one GC cycle
- Gen 2: Long-lived objects

## ✓ 4. Manual Control (Optional)

- You can interact with the GC using the gc module:


```
import gc
gc.collect() # Manually trigger garbage collection
gc.disable() # Turn off automatic GC (not recommended)
```

## Q16. What is the purpose of the else block in exception handling ?

- Ans: ♦ Purpose of the else Block in Exception Handling (Python) The else block in Python's exception handling is used to run code only if no exceptions were raised in the try block.

## ✓ Why Use else?

- Keeps your error-handling (except) separate from your normal logic
- Makes code cleaner and easier to understand

 Syntax:

```
try:
    # Risky code
except SomeException:
    # Handle the exception
else:
    # Run only if no exception occurred
```

## Example

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print(f"Result: {result}")
```

### Q17.What are the common logging levels in Python ?

- Ans:- ♦ Common Logging Levels in Python Python's logging module provides 5 standard logging levels to categorize the importance and severity of messages in your application.

#### ✓ 1. DEBUG (10)

- Lowest level.
- Used for detailed information, mostly useful during development.

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

#### ✓ 2. INFO (20)

- General events that confirm the program is working as expected.

```
logging.info("Process started successfully.")
```

#### ✓ 3. WARNING (30)

- Indicates something unexpected happened, but the program is still running.

```
logging.warning("Low disk space.")
```

#### ✓ 4. ERROR (40)

- A serious problem that prevents a function from performing its task.

```
logging.error("Failed to connect to database.")
```

#### ✓ 5. CRITICAL (50)

- A very serious error, possibly shutting down the application.

```
logging.critical("System crash! Immediate attention needed.")
```

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

- Ans:- ♦ Difference Between os.fork() and multiprocessing in Python
- Both os.fork() and the multiprocessing module are used to create new processes, but they differ significantly in usage, portability, and abstraction level.

#### ✓ 1. os.fork()

- 🧠 Low-level system call (UNIX/Linux only).
- Creates a child process by duplicating the parent.
- Returns:
  - 0 to the child process,
  - PID of the child to the parent.

```
import os

pid = os.fork()
if pid == 0:
    print("Child process")
else:
    print("Parent process")
```

#### ⚠ Limitation:

- Not available on Windows.
- Requires manual handling of shared data, communication, and synchronization.

#### ✓ 2. multiprocessing Module

- 🧠 High-level API for cross-platform process-based parallelism.
- Spawns independent Python interpreter processes.
- Safer and easier to use.

- from multiprocessing import Process

def worker():

```
    print("Worker process")
```

```
p = Process(target=worker) p.start() p.join()
```

#### ✓ Advantages:

- Portable (works on Windows, macOS, Linux)
- Built-in support for:
  - Queues
  - Pipes
  - Shared memory
  - Synchronization tools (like Lock, Event)

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

- Ans:- ♦ Importance of Closing a File in Python When working with files in Python, it's essential to close them properly after you're done. This ensures your program runs safely and efficiently.

#### ✓ 1. Frees System Resources

- Files consume system-level resources (like file handles).
- Closing the file releases those resources back to the OS.

```
file = open("data.txt", "r")
# ... use the file
file.close() # Frees memory and file handle
```

#### ✓ 2. Saves Data Properly

- In write (w) or append (a) mode, changes may be buffered and not written immediately.
- close() ensures all buffered data is flushed to disk.

```
file = open("log.txt", "w")
file.write("Important log")
file.close() # Ensures the data is actually saved
```

#### ✓ 3. Avoids Data Corruption or Loss

- Keeping files open for too long, especially during writing, increases the risk of corruption if the program crashes.
- Closing them reduces that risk.

#### ✓ 4. Prevents File Lock Issues

- On some systems, an open file may be locked (other programs or processes can't access it).
- Closing the file removes that lock.

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

- Ans:- ♦ Difference Between file.read() and file.readline() in Python Both file.read() and file.readline() are used to read content from a file, but they work differently in how much data they return.

#### ✓ 1. file.read()

- Reads the entire file content at once (or a specified number of bytes).
- Returns a single string.

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



```
content = file.read()
- print(content)
```

## ✔ 2. file.readline()

- Reads only one line at a time from the file.
- Returns a string including the newline \n.

```
with open("sample.txt", "r") as file:
    line1 = file.readline()
    line2 = file.readline()
    print(line1, line2)
```

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

- Ans:- The logging module in Python is used to track events, errors, and status updates during a program's execution — making it easier to debug, monitor, and maintain your code.

### ✔ Key Uses of the logging Module:

#### 1.Debugging

- Logs detailed info about the code's behavior.

```
logging.debug("Variable x has value %d", x)
```

#### 2.Error Tracking

- Captures errors and exceptions without stopping the program.

```
logging.error("Something went wrong!", exc_info=True)
```

#### 3.Monitoring

- Helps track program status, execution flow, or performance over time.

#### 4.Reporting

- Saves logs to files for later analysis — useful for production environments.

#### 5.Alerts

- Use higher-level logs (WARNING, ERROR, CRITICAL) to highlight serious issues.

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

- Ans:- The os module in Python provides a way to interact with the operating system, especially for file and directory management — making it essential for file handling tasks.

### ✔ Key Uses of os in File Handling:

#### 1.Working with Files and Directories

#### 2.Checking File/Folder Existence

#### 3.Creating and Deleting Folders

#### 4.Listing Directory Contents

#### 5.Changing the Current Directory

#### 6.Getting Current Working Directory

## Q.23 What are the challenges associated with memory management in Python ?

- Ans:- ♦ Challenges Associated with Memory Management in Python Although Python automates memory management using its garbage collection and reference counting systems, there are still several challenges developers might face.

### ⚠ 1. Memory Leaks

- Caused when objects are unintentionally kept alive due to lingering references.
- Often occurs with circular references or global/static variables.

### ⚠ 2. Inefficient Memory Usage

- Python's high-level abstractions (lists, dictionaries, objects) may use more memory than low-level languages like C.
- Data structures can become memory-heavy if not optimized.

### ⚠ 3. Circular References

- Reference counting alone can't handle them.
- Python's garbage collector detects and collects these, but it's not always perfect or fast.

### ⚠ 4. Unreleased Resources

- Objects like file handles, DB connections, or sockets consume memory if not explicitly closed.

### ⚠ 5. Global Variables

- Keeping large objects in the global scope may delay garbage collection or increase memory footprint unnecessarily.

### ⚠ 6. Misuse of C Extensions or Third-party Libraries

- Some C-based libraries may not integrate well with Python's garbage collector, leading to leaks.

### ⚠ 7. Long-Lived Objects in Loops

- Creating large objects repeatedly without clearing them or letting them expire can cause memory bloat over time

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

- Ans ♦ How to Raise an Exception Manually in Python In Python, you can manually raise an exception using the raise statement. This is helpful when you want to flag custom errors or enforce rules in your code.

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

- Ans:- Multithreading allows a program to run multiple tasks concurrently within the same process — which can significantly improve performance and responsiveness in the right scenarios.

### ✅ Key Reasons to Use Multithreading:

#### 1.Improved Responsiveness

- In GUI applications, threads prevent the interface from freezing during long-running tasks.

#### 2.Better Resource Utilization

- Threads can share memory and data easily within the same process, making communication between tasks more efficient than multiprocessing.

#### 3.Efficient I/O-bound Task Handling

- For tasks like reading files, downloading data, or database access, multithreading allows the CPU to keep working while waiting for I/O.

#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, this is a sample text.")
```

#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, this is a sample text.

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

```
try:
    with open("data.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found. Creating a new one...")
    with open("data.txt", "w") as file:
        file.write("This is a newly created file.")
```

🔄 File not found. Creating a new one...

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

```
try:
    with open("source.txt", "r") as src:
        content = src.read()

    with open("destination.txt", "w") as dest:
        dest.write(content)

    print("File copied successfully!")

except FileNotFoundError:
    print("Source file not found.")
except Exception as e:
    print("An error occurred:", e)
```

➦ Source file not found.

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

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
    print("Result:", result)

except ZeroDivisionError:
    print("Error: Denominator cannot be zero.")
except ValueError:
    print("Error: Please enter valid numbers.")
```

➦ Enter numerator: 20  
Enter denominator: 10  
Result: 2.0

#6 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",      # Log file name
    level=logging.ERROR,          # Log only errors and above
    format="%(asctime)s - %(levelname)s - %(message)s"
)

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

# Test the function
result = divide(10, 0)
```

➦ ERROR:root:Division by zero attempted: division by zero  
Error: Cannot divide by zero.

#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,          # Set the lowest log level to capture everything
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log messages at different levels
logging.debug("This is a DEBUG message (for diagnosing problems).")
logging.info("This is an INFO message (for general events).")
```

```
logging.warning("This is a WARNING message (something unexpected happened).")
logging.error("This is an ERROR message (a serious problem occurred).")
logging.critical("This is a CRITICAL message (very serious error).")
```

```
➦ WARNING:root:This is a WARNING message (something unexpected happened).
  ERROR:root:This is an ERROR message (a serious problem occurred).
  CRITICAL:root:This is a CRITICAL message (very serious error).
```

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

```
try:
    with open("myfile.txt", "r") as file:
        content = file.read()
        print("File content:")
        print(content)

except FileNotFoundError:
    print("Error: The file 'myfile.txt' was not found.")

except IOError:
    print("Error: An I/O error occurred while trying to read the file.")
```

```
➦ Error: The file 'myfile.txt' was not found.
```

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

```
lines = []

try:
    with open("sample.txt", "r") as file:
        lines = file.readlines() # Reads all lines into a list

    print("File content as a list:")
    print(lines)

except FileNotFoundError:
    print("Error: File not found.")
```

```
➦ Error: File not found.
```

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

```
with open("example.txt", "a+") as file:
    file.write("\nAnother appended line.")
    file.seek(0) # Move to the beginning of the file
    print(file.read())
```

```
➦ Hello, this is a sample text.
  Another appended line.
  Another appended 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
student = {
    "name": "Alice",
    "age": 20
}

# Attempt to access a key that may not exist
try:
    grade = student["grade"] # This key doesn't exist
    print("Grade:", grade)

except KeyError:
    print("Error: 'grade' key not found in the dictionary.")
```

```
➦ Error: 'grade' key 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)

my_list = [1, 2, 3]
print("Fourth item:", my_list[3]) # This will raise IndexError

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

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

except IndexError:
    print("Error: List index out of range.")

except Exception as e:
    print("An unexpected error occurred:", e)

```

```

➦ Enter a number: 10
Result: 1.0
Error: List index out of range.

```

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

```

```

➦ File content:
Hello, this is a sample text.
Another appended line.
Another appended line.

```

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

```

import logging

# Configure logging to write messages to a file
logging.basicConfig(
    filename="app.log",          # Log file name
    level=logging.DEBUG,         # Capture DEBUG and higher level logs
    format="%(asctime)s - %(levelname)s - %(message)s"
)

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

# Simulating an error scenario
try:
    result = 10 / 0 # This will raise a ZeroDivisionError
except ZeroDivisionError as e:
    logging.error(f"Error: {e}")

```

```

➦ ERROR:root:Error: division by zero

```

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

```

# Function to read and print file content
def read_file(file_path):
    try:
        with open(file_path, "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(f"Error: The file '{file_path}' was not found.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

# Specify the file path
file_path = "example.txt"
read_file(file_path)

```

🔄 File Content:  
Hello, this is a sample text.  
Another appended line.  
Another appended line.

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

- Ans:- Step 1: Install the memory\_profiler package

If you don't have the memory\_profiler package installed, you can install it via pip:

```
pip install memory-profiler
```

Step 2: Write a simple program to profile

Let's write a small Python program that calculates the sum of squares of numbers in a list:

```

from memory_profiler import profile

@profile
def sum_of_squares():
    result = []
    for i in range(1000000):
        result.append(i ** 2)
    return sum(result)

if __name__ == "__main__":
    sum_of_squares()

```

Step 3: Run the program with memory profiling

To see memory usage, run the script with the following command in the terminal:

```
python -m memory_profiler your_script.py
```

Step 4: Output analysis

You will see output that looks like this:

Line #	Mem usage	Increment	Line Contents
3	12.3 MiB	12.3 MiB	@profile
4	12.4 MiB	0.1 MiB	def sum_of_squares():
5	12.4 MiB	0.0 MiB	result = []
6	12.4 MiB	0.0 MiB	for i in range(1000000):
7	38.0 MiB	25.6 MiB	result.append(i ** 2)
8	38.0 MiB	0.0 MiB	return sum(result)

#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 ('w')
with open('numbers.txt', 'w') as file:
    # Iterate through the list and write each number to the file

```

```

    for number in numbers:
        file.write(f"{number}\n")

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

```

➦ Numbers have been written to 'numbers.txt' successfully.

#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 logger
logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG) # Set the logging level (DEBUG, INFO, WARNING, etc.)

# Create a RotatingFileHandler
handler = RotatingFileHandler('my_log.log', maxBytes=1e6, backupCount=3) # 1 MB = 1e6 bytes
handler.setLevel(logging.DEBUG) # Set the level for the handler

# Create a formatter and set it for the handler
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
handler.setFormatter(formatter)

# Add the handler to the logger
logger.addHandler(handler)

# Example log entries
logger.debug('This is a debug message.')
logger.info('This is an info message.')
logger.warning('This is a warning message.')
logger.error('This is an error message.')
logger.critical('This is a critical message.')

```

➦ DEBUG:my\_logger:This is a debug message.  
 INFO:my\_logger:This is an info message.  
 WARNING:my\_logger:This is a warning message.  
 ERROR:my\_logger:This is an error message.  
 CRITICAL:my\_logger:This is a critical message.

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

```

def handle_errors():
    my_list = [10, 20, 30]
    my_dict = {'a': 1, 'b': 2, 'c': 3}

    try:
        # Attempting to access an invalid index in the list
        print(my_list[5])

        # Attempting to access a key that doesn't exist in the dictionary
        print(my_dict['d'])

    except IndexError as ie:
        print(f"IndexError occurred: {ie}")

    except KeyError as ke:
        print(f"KeyError occurred: {ke}")

# Call the function to test error handling
handle_errors()

```

➦ IndexError occurred: list index 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:
    # Read the entire contents of the file
    content = file.read()

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

➡ Hello, this is a sample text.  
Another appended line.  
Another appended 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, target_word):
    try:
        # Open the file in read mode using a context manager
        with open(file_name, 'r') as file:
            # Read the contents of the file
            content = file.read()

            # Count the occurrences of the target word (case-insensitive)
            word_count = content.lower().split().count(target_word.lower())

            # Print the result
            print(f"The word '{target_word}' occurred {word_count} times.")

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

# Example usage
file_name = 'example.txt' # Replace with the actual file path
target_word = 'python'    # Replace with the word you want to search for
count_word_occurrences(file_name, target_word)
```

➡ The word 'python' occurred 0 times.

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

1. Using `os.path.getsize()`:

You can check the size of the file. If the size is 0, the file is empty.

```
import os

file_path = "your_file.txt"

if os.path.getsize(file_path) > 0:
    with open(file_path, "r") as file:
        content = file.read()
        print(content)
else:
    print("The file is empty.")
```

2. Using `os.stat()`:

Another way to get the size of the file is using `os.stat()`, which also provides information about the file size.

```
import os

file_path = "your_file.txt"

if os.stat(file_path).st_size > 0:
    with open(file_path, "r") as file:
        content = file.read()
        print(content)
```



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

3. Checking if read() returns empty:

You can open the file and attempt to read its content directly. If read() returns an empty string, it means the file is empty.

```
with open("your_file.txt", "r") as file:
    content = file.read()
    if content:
        print(content)
    else:
        print("The file is empty.")
```

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

```
import logging

# Configure the logging
logging.basicConfig(
    filename='error_log.txt',
    level=logging.ERROR,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def divide(a, b):
    try:
        result = a / b
        print(f"Result: {result}")
    except Exception as e:
        logging.error("Error occurred while dividing", exc_info=True)
        print("An error occurred. Check error_log.txt for details.")

# Example usage
divide(10, 0) # This will cause a ZeroDivisionError
```

```
↩ ERROR:root:Error occurred while dividing
Traceback (most recent call last):
  File "<ipython-input-43-10cf20195654>", line 14, in divide
    result = a / b
    ~~~~
ZeroDivisionError: division by zero
An error occurred. Check error_log.txt for details.
```

[+ Code](#)[+ Text](#)

Start coding or [generate](#) with AI.