PW Assignment
Files, exceptional handling, logging and memory management

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

Ans>

Interpreted Language	**Compiled Language**
Code is executed line by line by an interpreter.	Entire code is translated to machine code before execution by a compiler.
Slower (because interpretation happens at runtime).	Faster (machine code is precompiled and directly executed).
Errors are detected at runtime (when the line is executed).	Errors are detected at compile time, before execution.
No separate executable file is generated.	Generates an executable file (.exe, .out, etc.).
More portable – can run on any machine with the interpreter.	Less portable – compiled code is platform-dependent.
Examples: Python, JavaScript, Ruby,	Examples: C, C++, Rust, Go

2.What is exception handling in Python?

Ans>An exception is an event that disrupts the normal flow of a program.
In Python, we can handle exceptions using try–except blocks.

When an exception occurs, Python stops the normal execution of the program unless it is properly handled.
Therefore, any code that might cause an error should be placed inside a try block for safe execution.

Syntax:

```
try:

   # Code that might raise an exception

except:

   # Executes when an exception occurs
```

```
else:
   # Executes if no exception occurs in the try block
finally:
   # Always executes, regardless of whether an exception occurred or not
```

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

Ans> The finally block is used to **execute code no matter what happens —** whether an exception occurs or not.

It is typically used for **cleanup activities** such as:

Closing files

Releasing resources (like database connections)

Resetting variables or states

The code inside finally always runs:

Even if there’s an exception

Even if there’s a return, break, or continue statement in the try or except block

```
try:
   file = open("data.txt", "r")
   print(file.read())
except FileNotFoundError:
   print("File not found.")
finally:
   file.close()
   print("File closed (executed no matter what).")
```

4. What is logging in Python?

Ans> **Logging** in Python is the process of **recording events or messages** that happen during program execution.

It helps developers **track the flow**, **find errors**, and **debug issues** without interrupting the program.

The messages can be saved to a **file**, **console**, or other outputs.

Why Use Logging?

- To record **errors**, **warnings**, and **information** about program behaviour.
- To help in **debugging and troubleshooting**.
- To maintain a **history of events** for auditing or monitoring.

```
import logging

logging.basicConfig(filename='app.log', level=logging.INFO)

logging.info("Program started")

logging.warning("Low disk space")

logging.error("File not found")
```

Level	**Description**
DEBUG	Detailed information for debugging
INFO	General information about program execution
WARNING	Something unexpected, but program still runs
ERROR	Serious problem; program may not continue correctly
CRITICAL	Severe error; program may stop running

5. What is the significance of the __del__ method in Python?

Ans>

The __del__() method is a **destructor** in Python.

It is **automatically called** when an object is about to be **destroyed** (i.e., when it goes out of scope or the program ends).

Its main purpose is to **release resources** or **perform cleanup tasks** before the object is deleted from memory.

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

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

```
obj = Demo("Test")

del obj   # Manually deleting the object
```

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

Ans>

import statement	from ... import statement
Imports the entire module.	Imports specific functions, classes, or variables from a module.
Syntax: import module_name	Syntax: from module_name import function_name
You must use the module name as a prefix to access items.	You can access imported items directly without the module name.
Example:import mathprint(math.sqrt(16))	Example:from math import sqrtprint(sqrt(16))
Used when you need many functions or classes from a module.	Used when you need only a few specific items and want shorter code.

7. How can you handle multiple exceptions in Python?

Ans> In Python, you can handle **multiple exceptions** using:

Multiple except blocks – one block for each exception type.

A single except block with a tuple of exceptions – handles multiple types together.

```
try:

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

   result = 10 / x

except ValueError:

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

except ZeroDivisionError:

   print(" Division by zero is not allowed.")
```

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except (ValueError, ZeroDivisionError) as e:
    print(f"❌ An error occurred: {e}")
```

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

Ans>

- The with statement is used to open files safely and automatically manage resources.
- It ensures that the file is properly closed after its suite finishes, even if an exception occurs.
- Using with reduces the need to explicitly call file.close(), making code cleaner and safer.

```
with open("example.txt", "r") as file:
    data = file.read()
    print(data)
# File is automatically closed here
```

9. What is the difference between multithreading and multiprocessing?

Ans>

Multithreading	Multiprocessing
Runs multiple threads within a single process.	Runs multiple processes, each with its own memory space.
Shares the same memory space, which makes data sharing easier but can cause race conditions.	Each process has separate memory, so data sharing requires IPC (inter-process communication).
Suitable for I/O-bound tasks (e.g., file operations, network calls).	Suitable for CPU-bound tasks (e.g., heavy computations).
Threads are lighter and start faster than processes.	Processes are heavier and take more time to start.

Multithreading	Multiprocessing
Limited by Python’s GIL (Global Interpreter Lock) — only one thread executes Python bytecode at a time.	Not limited by GIL — multiple processes can run truly in parallel.
Examples: threading module in Python.	Examples: multiprocessing module in Python.

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

Ans>

Debugging Made Easier

Logs provide detailed information about program execution, helping developers **identify and fix errors** faster.

Record of Program Execution

Maintains a **history of events**, which is useful for auditing or tracking program behavior over time.

Better Error Monitoring

Errors and warnings can be recorded without stopping the program, enabling **proactive issue detection**.

Separation from User Output

Unlike print(), logging messages can be **directed to files, console, or other destinations** without cluttering user-facing output.

Multiple Logging Levels

Allows developers to **categorize messages** (DEBUG, INFO, WARNING, ERROR, CRITICAL) for better organization and filtering.

Persistent Record

Logs can be saved to **files or databases**, which helps in **post-mortem analysis** of issues after the program has run.

Configurable and Flexible

Logging in Python can be easily configured to **change format, level, and destination** without changing program logic.

11. What is memory management in Python?

Ans> **Memory management** in Python refers to the process of **allocating, using, and releasing memory** for objects during program execution. Python automatically handles memory to ensure efficient use of resources.

Key Features

Automatic Allocation

When you create objects (like integers, lists, dictionaries), Python automatically allocates memory for them.

Garbage Collection

Python automatically **frees memory** for objects that are no longer in use using its **garbage collector**.

Helps prevent memory leaks.

Reference Counting

Python keeps track of how many references point to an object.

When the reference count drops to zero, the memory is released.

Dynamic Memory Allocation

Python can dynamically allocate memory at runtime, making it flexible for growing data structures.

Memory Pooling

Python internally uses **pymalloc** to manage small memory blocks efficiently, reducing overhead.

```
a = [1, 2, 3]  # Memory allocated for the list

b = a

del a

del b
```

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

Ans>

Identify Risky Code

Determine the parts of your code that might raise an exception (e.g., file operations, user input, division).

Use try Block

Place the risky code inside a try block.

Python will **monitor this block for exceptions**.

Handle Exceptions with except Block

Define one or more except blocks to **catch specific exceptions** and handle them gracefully.

Example: except ValueError: or except ZeroDivisionError:

Optional: else Block

Use else to **execute code only if no exception occurs** in the try block.

Optional: finally Block

Use finally to **execute code regardless of whether an exception occurred**.

Commonly used for cleanup, like closing files or releasing resources.

```
try:
   num = int(input("Enter a number: "))
   result = 10 / num
except ValueError:
   print("Invalid input! Enter a number.")
except ZeroDivisionError:
   print(" Division by zero is not allowed.")
else:
   print(f"Result is {result}")
finally:
   print("Execution completed.")
```

13. Why is memory management important in Python?

Ans>

Efficient Resource Utilization

Proper memory management ensures that **memory is allocated and used efficiently**, preventing wastage.

Prevents Memory Leaks

Automatically freeing memory of unused objects helps **avoid memory leaks**, which can slow down or crash programs.

Improves Program Performance

Efficient allocation and deallocation of memory **speeds up program execution**.

Supports Large and Dynamic Data

Python programs often deal with **dynamic data structures** (lists, dictionaries). Good memory management ensures these structures can grow without issues.

Simplifies Developer Work

Python's automatic memory management (reference counting + garbage collection) **reduces the need for manual memory handling**, allowing developers to focus on logic instead of resources.

Ensures Stability

Prevents **unexpected crashes or errors** due to running out of memory or using invalid memory locations.

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

Ans>

1. try Block

The try block contains code that might raise an exception.

Python monitors this block during execution.

If an exception occurs in the try block, normal program execution stops, and control moves to the corresponding except block.

2. except Block

The except block catches and handles the exception raised in the try block.

You can handle specific exceptions (like ValueError, ZeroDivisionError) or all exceptions using a generic except.

This prevents the program from crashing and allows it to continue running gracefully.

15. How does Python's garbage collection system work?

Ans> **Garbage collection** in Python is the process of **automatically freeing memory** occupied by objects that are no longer in use, ensuring efficient memory management.

- Python keeps track of the **number of references** to each object.
- When an object's reference count drops to **zero**, it is no longer accessible and its memory can be **reclaimed automatically**.

```
a = [1, 2, 3]  # Reference count = 1

b = a        # Reference count = 2

del a        # Reference count = 1

del b        # Reference count = 0 → Object deleted
```

```
import gc

class Node:

   def __init__(self, name):

      self.name = name

      self.ref = None

a = Node("A")

b = Node("B")

a.ref = b

b.ref = a  # Creates a cycle

del a

del b

gc.collect()  # Explicitly triggers garbage collection
```

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

Ans> The **else block** is an **optional part** of exception handling.

It runs **only if no exception occurs** in the try block.

Useful for **code that should execute when everything in try succeeds**, keeping it separate from exception-handling logic.

```
try:

   # Code that might raise an exception

except SomeException:

   # Code to handle the exception

else:
   # Code that runs if no exception occurs
finally:
   # Code that runs always (optional)
```

17. What are the common logging levels in Python?

Ans> Python's logging module defines several standard levels to indicate the **severity of events**.

Level	**Description**
DEBUG	Detailed information, typically of interest only when diagnosing problems.
INFO	General information about program execution, e.g., "Process started."
WARNING	An indication that something unexpected happened, or may happen soon, but the program is still running.
ERROR	A serious problem that prevents some part of the program from functioning correctly.
CRITICAL	A very serious error indicating the program may be unable to continue running.

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.debug("Debugging information")
logging.info("Program started")
logging.warning("Low disk space")
logging.error("File not found")
logging.critical("System crash")
```

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

Ans>

os.fork()	**multiprocessing module**
Creates a **child process** by duplicating the current process.	Creates **new processes** using the Process class.
Available only on **Unix/Linux systems**.	Cross-platform: works on **Windows, Linux, macOS**.

os.fork()	**multiprocessing module**
Child process shares the **same code and data** initially, but changes are independent.	Each process runs in **separate memory space**, so data is isolated.
Low-level system call; programmer handles process management manually.	High-level API for process creation, communication (Queues, Pipes), and synchronization.
Less portable and less flexible for complex tasks.	Provides **better abstraction**, easier to write **parallel programs**.
Does not have built-in support for **task synchronization or communication**.	Supports **inter-process communication (IPC)** and synchronization mechanisms.

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

Ans> Closing a file in Python is important because it frees system resources, ensures all data is written to disk, prevents file corruption, avoids reaching the limit of open files, and allows safe access by other programs or parts of your code.

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

Ans>

file.read()	file.readline()
Reads the entire content of the file (or a specified number of characters).	Reads one line at a time from the file.
Returns a single string containing all the text.	Returns a string containing the current line, including the newline character.
After reading, the file pointer moves to the end of the file.	After reading, the file pointer moves to the next line.
Useful when you want to process all content at once.	Useful when you want to process the file line by line.
Example: content = file.read()	Example: line = file.readline()

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

Ans> The **logging module** in Python is used to **record events or messages** that occur during program execution. It helps developers **monitor, debug, and maintain** programs efficiently.

Key Uses

Debugging – Track the flow of execution and diagnose issues.

Error Monitoring – Record errors and warnings without stopping the program.

Information Tracking – Keep a history of program activities for auditing or analysis.

Flexible Output – Log messages can be sent to **console, files, or other destinations**.

Severity Levels – Categorize messages by importance (DEBUG, INFO, WARNING, ERROR, CRITICAL)

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

Ans> The os module in Python is used for interacting with the operating system, allowing programs to create, delete, rename, list, and inspect files and directories, making file and directory management easier.

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

Ans> Python's memory management is mostly automatic, but challenges like **memory leaks, garbage collection overhead, reference cycles, and large data handling** can affect performance and resource utilization.

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

Ans>In Python, you can **raise an exception manually** using the raise statement. This is useful when you want to **signal that an error condition has occurred** in your code.

```
raise ExceptionType("Error message")

def divide(a, b):

  if b == 0:

    raise ValueError("Cannot divide by zero")

  return a / b

divide(10, 0)


ValueError: Cannot divide by zero
```

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

Ans> Multithreading is important in certain applications because it allows **concurrent execution of tasks**, improves performance for I/O-bound operations, keeps programs responsive, and efficiently utilizes system resources.