**Chapter 7: File I/O Handling and Exception Handling**

**Lecture Notes**

**Introduction:**

In this chapter, we will cover two important topics: File I/O Handling and Exception Handling. File I/O operations are essential for working with external files, such as reading data from files and writing data to files. Exception handling allows us to gracefully handle errors during program execution. By the end of this module, you will have a solid understanding of file manipulation, directory operations, different types of errors, and how to handle exceptions in Python.

**1. File I/O Handling:**

**1.1 Opening and Closing Files:**

Files are opened using the open() function and closed using the close() method. The open() function takes two parameters: the file path and the opening mode. The opening mode determines whether the file will be opened for reading, writing, or appending.

Syntax:

file\_object = open(file\_path, mode)

Examples:

# Open a file for reading

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

# Open a file for writing

file = open('output.txt', 'w')

# Open a file for appending

file = open('log.txt', 'a')

**1.2 Reading Data from Files:**

Python provides various methods for reading data from files:

Reading files line by line:

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

for line in file:

print(line)

Reading the entire file at once:

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

contents = file.read()

print(contents)

Reading specific characters or bytes:

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

file.seek(5) # Move the file pointer to the 6th character

data = file.read(10) # Read the next 10 characters

print(data)

**1.3 Writing Data to Files:**

Writing data to files is done using the write() method. By default, the write() method overwrites the existing file contents. To append data to an existing file, use the 'a' mode when opening the file.

Syntax:

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

file.write('Hello, World!')

Appending data to a file:

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

file.write('Error: File not found')

**1.4 File Opening Modes:**

File opening modes determine the purpose of opening a file. The most commonly used modes are:

'r': Read mode (default). Opens the file for reading.

'w': Write mode. Opens the file for writing. If the file exists, it is truncated. If it doesn't exist, a new file is created.

'a': Append mode. Opens the file for appending. If the file exists, the data is appended to it. If it doesn't exist, a new file is created.

Additional modes include 'x' for exclusive creation, 'b' for binary mode, and more.

**1.5 Manipulating Files and Directories:**

Python provides several functions and modules for manipulating files and directories. The commonly used ones are:

Creating new files:

with open('new\_file.txt', 'w') as file:

# File created

Renaming files:

import os

os.rename('old\_file.txt', 'new\_file.txt')

Deleting files:

import os

os.remove('file\_to\_delete.txt')

Checking file and directory existence:

import os

file\_exists = os.path.exists('file.txt')

directory\_exists = os.path.isdir('directory')

**1.6 The OS and SYS Modules:**

The os module provides functions for interacting with the operating system, while the sys module provides access to system-specific parameters and functions.

**The OS module:**

Creating and removing directories: os.mkdir() and os.rmdir()

Listing directory contents: os.listdir()

Checking file size and permissions: os.path.getsize() and os.access()

**The sys module:**

Accessing command-line arguments: sys.argv

Terminating the program: sys.exit()

**2. Exception Handling:**

**2.1 Types of Errors:**

In Python, there are three main types of errors:

Syntax errors: Errors that occur due to incorrect syntax. The code fails to compile.

Runtime errors (exceptions): Errors that occur during program execution. These can be handled using exception handling.

Logical errors: Errors resulting from flawed program logic. These are more difficult to identify and fix.

**2.2 Error Processing:**

Error processing involves handling exceptions using the try-except block. The try block contains the code that may raise an exception, while the except block specifies how to handle the exception if it occurs. The else and finally clauses provide additional control flow.

Syntax:

try:

# Code that may raise an exception

except ExceptionType:

# Code to handle the exception

else:

# Code to execute if no exception occurred

finally:

# Code that is always executed, whether an exception occurred or not

**2.3 Exception Raising:**

Python allows you to manually raise exceptions using the raise statement. You can also create custom exception classes to handle specific scenarios.

Syntax:

raise ExceptionType("Error message")

**2.4 Handling Exceptions:**

In addition to handling specific exceptions, you can handle multiple exceptions using separate except blocks. Exceptions in Python form a hierarchy, and you can handle multiple exceptions using the base exception class. Understanding the exception hierarchy and inheritance is crucial for effective exception handling.

**Example-**

def divide\_numbers(num1, num2):

try:

result = num1 / num2

return result

except ZeroDivisionError:

return "Division by zero is not allowed!"

except TypeError:

return "Please enter numbers only!"

finally:

print("Function executed")

# Testing the function

print(divide\_numbers(10, 2)) # Expected: 5.0

print(divide\_numbers(10, 0)) # Expected: Division by zero is not allowed!

print(divide\_numbers(10, 'a')) # Expected: Please enter numbers only!