**PYTHON PROGRAMMING**

**UNIT - 5**

**FILES, MODULES, PACKAGES**

**File Handling in Python**

## 1. Introduction to File Handling in Python

File handling in Python allows users to create, read, write, and manipulate files. Python provides built-in functions to interact with files, making it easy to handle file operations efficiently.

### Why File Handling?

* Persistent data storage
* Easy data exchange between programs
* Logging and debugging purposes
* Large-scale data processing

## 2. Opening a File in Python

In Python, the open() function is used to open a file. It returns a file object which allows interaction with the file.

### Syntax:

file\_object = open("filename", "mode")

### Example:

file = open("example.txt", "r") # Opens file in read mode

print(file.read()) # Reads content from file

file.close() # Closes the file



## 3. File Open Modes

Python provides various modes to open a file:

| **Mode** | **Description** |
| --- | --- |
| r | Read mode (default) |
| w | Write mode (overwrites existing content) |
| a | Append mode (adds data to the end of file) |
| x | Exclusive creation (fails if file exists) |
| t | Text mode (default mode) |
| b | Binary mode |

### Example of Different Modes:

# Writing to a file

file = open("test.txt", "w")

file.write("Hello, Python File Handling!")

file.close()

# Reading the file

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

print(file.read())

file.close()

# Appending to the file

file = open("test.txt", "a")

file.write("\nAppending new data!")

file.close()



**Output:**

Hello, Python File Handling!

Appending new data!



## 4. Writing to a File

The write() function is used to write text to a file.

### Example:

file = open("sample.txt", "w")

file.write("Welcome to Python File Handling.")

file.close()



## 5. Handling Exceptions in File Handling

File operations may result in errors, such as trying to open a non-existent file. Python provides exception handling using try-except blocks.

### Example:

try:

file = open("non\_existent\_file.txt", "r")

print(file.read())

except FileNotFoundError:

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

finally:

file.close()



**Output:**

Error: The file does not exist!



**Exception Handling in Programming**

## What are Exceptions?

Exceptions are unexpected or unwanted events that occur during the execution of a program. These events disrupt the normal flow of the program and can lead to crashes if not handled properly. Exception handling provides a way to manage these errors gracefully, ensuring the program can continue executing or terminate with an appropriate message.

### Example of an Exception:

# Division by zero error

num1 = 10

num2 = 0

result = num1 / num2 # This will raise a ZeroDivisionError

print("Result:", result)



**Output:**

ZeroDivisionError: division by zero

## Types of Exceptions

Exceptions can be broadly classified into two categories:

1. **Built-in Exceptions:** Predefined exceptions in programming languages that handle common runtime errors.
2. **User-defined Exceptions:** Custom exceptions created by programmers to handle specific scenarios.

### Common Built-in Exceptions:

* **ZeroDivisionError**: Raised when a number is divided by zero.
* **IndexError**: Raised when accessing an index that is out of range.
* **KeyError**: Raised when accessing a non-existent key in a dictionary.
* **TypeError**: Raised when an operation is performed on an incompatible data type.
* **ValueError**: Raised when an incorrect value is provided to a function.

### Example of Built-in Exceptions:

try:

my\_list = [1, 2, 3]

print(my\_list[5]) # Trying to access an out-of-range index

except IndexError as e:

print("Exception caught:", e)



**Output:**

Exception caught: list index out of range

### User-Defined Exception:

class CustomException(Exception):

pass

try:

raise CustomException("This is a user-defined exception!")

except CustomException as e:

print("Caught a custom exception:", e)



**Output:**

Caught a custom exception: This is a user-defined exception!

## Difference between Syntax Errors and Exceptions

| **Feature** | **Syntax Error** | **Exception** |
| --- | --- | --- |
| Occurrence | At compile-time | At runtime |
| Reason | Incorrect syntax | Unforeseen issues during execution |
| Handling | Must be fixed in code | Can be handled using exception handling |

### Example of Syntax Error:

print("Hello World" # Missing closing parenthesis



**Output:**

SyntaxError: unexpected EOF while parsing

### Example of an Exception:

num = int("abc") # Invalid conversion



**Output:**

ValueError: invalid literal for int() with base 10: 'abc'

## Exception Handling Syntax

Most programming languages provide a structured way to handle exceptions using try-except blocks.

### Python Exception Handling Syntax:

try:

# Code that may raise an exception

result = 10 / 0

except ZeroDivisionError:

# Handling the exception

print("Cannot divide by zero!")

finally:

print("Execution completed!")



**Output:**

Cannot divide by zero!

Execution completed!

### Java Exception Handling Syntax:

public class ExceptionExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // This will throw an exception

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

} finally {

System.out.println("Execution completed!");

}

}

}



**Output:**

Cannot divide by zero!

Execution completed!



Exception handling is crucial for writing robust and error-free programs. Using proper exception handling techniques ensures that programs run smoothly even in unexpected conditions.

**Advanced Exception Handling in Python**

Exception handling is a crucial part of programming that ensures an application can gracefully handle runtime errors. Python provides an advanced mechanism for handling exceptions using else, finally blocks, raising exceptions, and custom exception handling.

## **1. else and finally Blocks**

### **1.1 else Block**

The else block in Python is used with a try block to execute code when no exceptions are raised. This block allows the program to differentiate between successful execution and an exception case.

**Syntax:**

try:

# Code that may raise an exception

except ExceptionType:

# Code to handle the exception

else:

# Code to execute if no exception occurs



**Example:**

try:

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

result = 10 / num

except ZeroDivisionError:

print("Cannot divide by zero!")

else:

print("Division successful! Result:", result)



**Output:**

Enter a number: 5

Division successful! Result: 2.0

### **1.2 finally Block**

The finally block executes code regardless of whether an exception occurs. It is typically used to release resources like file handles or database connections.

**Syntax:**

try:

# Code that may raise an exception

except ExceptionType:

# Code to handle the exception

finally:

# Code that runs no matter what



**Example:**

try:

file = open("sample.txt", "w")

file.write("Hello, World!")

except IOError:

print("Error in writing to file")

finally:

file.close()

print("File closed successfully")



**Output:**

File closed successfully



## **2. Raising Exceptions**

Python allows manual triggering of exceptions using the raise statement. This is useful when certain conditions in a program need to be explicitly flagged as errors.

**Syntax:**

raise ExceptionType("Error Message")



**Example:**

def check\_age(age):

if age < 18:

raise ValueError("Age must be 18 or above!")

else:

print("Access granted!")

try:

check\_age(16)

except ValueError as e:

print("Exception occurred:", e)



**Output:**

Exception occurred: Age must be 18 or above!



## **3. Custom Exception Handling**

In Python, custom exceptions can be defined by inheriting from the built-in Exception class. This is useful for creating meaningful error messages for specific application needs.

**Syntax:**

class CustomError(Exception):

def \_\_init\_\_(self, message):

self.message = message

super().\_\_init\_\_(self.message)



**Example:**

class NegativeNumberError(Exception):

def \_\_init\_\_(self, message="Negative numbers are not allowed!"):

super().\_\_init\_\_(message)

def square\_root(num):

if num < 0:

raise NegativeNumberError()

return num \*\* 0.5

try:

print(square\_root(-9))

except NegativeNumberError as e:

print("Custom Exception:", e)



**Output:**

Custom Exception: Negative numbers are not allowed!



**Introduction to Modules in Python**

## What is a Module?

A module in Python is a file containing Python code that defines functions, classes, and variables, which can be reused in other programs. It helps in organizing code into separate files, making it more readable and maintainable.

### Example:

# mymodule.py

def greet(name):

return f"Hello, {name}!"



## Importance and Uses of Modules in Python

1. **Code Reusability**: Avoids code repetition by using functions and variables from existing modules.
2. **Modularity**: Helps organize code into separate files, improving maintainability.
3. **Namespace Management**: Prevents variable name conflicts.
4. **Built-in Functionality**: Accesses a vast range of pre-built Python modules.
5. **Improved Readability**: Makes large programs easier to manage.

## Types of Modules in Python

1. **Built-in Modules**: Predefined modules that come with Python (e.g., math, os, sys).
2. **User-defined Modules**: Custom modules created by users.
3. **External Modules**: Modules that need to be installed separately (e.g., numpy, pandas).

### Example:

import math

print(math.sqrt(16)) # Output: 4.0



## How to Create a Module

A module is simply a Python file (.py) containing functions, classes, or variables.

### Example:

# mymodule.py

def add(a, b):

return a + b



## How to Import a Module

Modules can be imported using the import statement.

### Example:

import mymodule

print(mymodule.add(3, 5)) # Output: 8



You can also use aliasing:

import mymodule as mm

print(mm.add(3, 5)) # Output: 8



## Import All Names from a Module

Use from module import \* to import everything from a module.

### Example:

from mymodule import \*

print(add(3, 5)) # Output: 8



## Locating Python Modules

Python searches for modules in directories listed in sys.path.

### Example:

import sys

print(sys.path)



To add a custom path:

sys.path.append("/custom/path")



**Built-in Modules in Python**

## Introduction to Built-in Modules

Python provides several built-in modules that help in performing various tasks without requiring external libraries. These modules come pre-installed with Python and offer functionalities ranging from mathematical operations to file handling and system interaction.

Some of the commonly used built-in modules include:

* **math**: Provides mathematical functions.
* **random**: Deals with generating random numbers.
* **datetime**: Handles date and time functionalities.
* **os**: Interacts with the operating system.

## 1. Math Module

The math module in Python provides various mathematical functions such as trigonometric, logarithmic, and exponential calculations.

### Example 1: Basic Mathematical Operations

import math

print(math.sqrt(25)) # Square root

print(math.pow(2, 3)) # Exponentiation

print(math.pi) # Pi constant

print(math.factorial(5)) # Factorial

### Output

5.0

8.0

3.141592653589793

120



## 2. Random Module

The random module is used to generate random numbers, shuffle sequences, and select random elements.

### Example 1: Generating Random Numbers

import random

print(random.randint(1, 10)) # Random integer between 1 and 10

print(random.random()) # Random float between 0 and 1

print(random.choice(['apple', 'banana', 'cherry'])) # Random choice from a list

### Output (may vary)

7

0.567894

banana



## 3. Datetime Module

The datetime module allows manipulation of dates and times.

### Example 1: Getting Current Date and Time

import datetime

now = datetime.datetime.now()

print("Current Date and Time:", now)

print("Year:", now.year)

print("Month:", now.month)

print("Day:", now.day)

### Output (may vary)

Current Date and Time: 2025-03-05 10:30:45.123456

Year: 2025

Month: 3

Day: 5



## 4. OS Module

The os module is used for interacting with the operating system.

### Example 1: Listing Files in a Directory

import os

print("Current Working Directory:", os.getcwd())

print("List of Files:", os.listdir())

### Output (may vary)

Current Working Directory: /Users/Desktop

List of Files: ['file1.py', 'file2.txt', 'folder1']



**Packages in Python**

## What is a Package?

A **package** in Python is a collection of modules organized in directories, allowing for better code organization and reuse. Packages help avoid module name conflicts and enable hierarchical structuring of the code.

### Key Features of a Package:

* It is a directory containing a special \_\_init\_\_.py file.
* Contains multiple modules that can be imported and used in Python programs.
* Facilitates modular programming and code reuse.

### Example:

A simple package structure:

my\_package/

|-- \_\_init\_\_.py

|-- module1.py

|-- module2.py

## Structure of Packages

A package follows a specific structure:

package\_name/

|-- \_\_init\_\_.py # Initialization file

|-- module1.py # Module 1

|-- module2.py # Module 2

|-- sub\_package/ # Subpackage

|-- \_\_init\_\_.py

|-- sub\_module.py

### Explanation:

* \_\_init\_\_.py: This file makes Python treat the directory as a package. It can be empty or contain initialization code.
* module1.py, module2.py: Individual modules containing functions and classes.
* sub\_package/: A subpackage containing its own modules.

## How to Create a Package?

### Steps to Create a Package:

1. **Create a Directory**: Create a folder for the package.
2. **Add an \_\_init\_\_.py file**: This makes it a package.
3. **Create Modules**: Add Python files containing functions or classes.

### Example:

1. Create a directory named math\_operations.
2. Inside, create \_\_init\_\_.py (can be empty).
3. Add modules addition.py and subtraction.py.

#### **addition.py**

# addition.py

def add(a, b):

return a + b

#### **subtraction.py**

# subtraction.py

def subtract(a, b):

return a - b

### Using the Package:

from math\_operations import addition, subtraction

print(addition.add(5, 3)) # Output: 8

print(subtraction.subtract(10, 4)) # Output: 6

## How to Install Packages?

Packages can be installed using **pip**, Python’s package manager.

### Installing a Package:

Use the following command in the terminal or command prompt:

pip install package\_name

### Example:

Installing the numpy package:

pip install numpy

### Installing a Specific Version:

pip install numpy==1.21.0

### Listing Installed Packages:

pip list

### Uninstalling a Package:

pip uninstall package\_name

### Example:

pip uninstall numpy

**Summary and Practice**

**File Handling in Python**

**Summary:** File handling in Python allows users to read, write, and manipulate files. The open() function is used to open files, and different file modes determine how a file is accessed. Python also provides mechanisms to handle exceptions that may occur during file operations.

**Key Topics:**

* File handling in Python
* File Open (open() function)
* File Open Modes (r, w, a, b, x)
* File Write (write(), writelines())
* Handling Exceptions (try-except)

**Practice:**

1. Open a file in write mode and add content.
2. Read content from a file and display it.
3. Append new data to an existing file.
4. Handle file exceptions using try-except.

**Exception Handling**

**Summary:** Exceptions in Python occur when the interpreter encounters an error during execution. They can be handled using the try-except block to ensure the program runs smoothly.

**Key Topics:**

* What are exceptions?
* Types of Exceptions
* Difference between syntax errors and exceptions
* Exception handling syntax (try-except-finally)

**Practice:**

1. Write a program that handles ZeroDivisionError.
2. Implement exception handling for file reading.
3. Use multiple except blocks to handle different exceptions.

**Advanced Exception Handling**

**Summary:** Advanced exception handling techniques in Python involve else and finally blocks, raising custom exceptions, and defining user-defined exception classes.

**Key Topics:**

* else and finally blocks
* Raising exceptions using raise
* Creating custom exceptions

**Practice:**

1. Implement try-except-else-finally for file operations.
2. Raise a ValueError when an invalid input is given.
3. Create a custom exception class and use it in a program.

**Introduction to Modules**

**Summary:** A module in Python is a file containing Python code, which can include functions, classes, and variables. Modules help in code reusability and better organization.

**Key Topics:**

* What is a module?
* Importance and uses of modules
* Types of Modules (Built-in, User-defined, External)
* Creating a module
* Importing modules (import, from ... import, import \*)
* Locating Python modules

**Practice:**

1. Create a simple module with functions and import it.
2. Use import \* to access all functions in a module.
3. Locate and check installed Python modules.

**Built-in Modules**

**Summary:** Python comes with built-in modules that provide useful functionalities like mathematical operations, random number generation, date-time handling, and OS interactions.

**Key Topics:**

* Introduction to built-in modules
* math module (e.g., sqrt(), pow(), ceil(), floor())
* random module (e.g., randint(), choice(), shuffle())
* datetime module (e.g., datetime.now(), strftime(), timedelta)
* os module (e.g., os.getcwd(), os.listdir(), os.remove())

**Practice:**

1. Use the math module to perform basic calculations.
2. Generate random numbers using the random module.
3. Work with date and time using the datetime module.
4. Perform file operations using the os module.

**Packages in Python**

**Summary:** A package in Python is a collection of modules organized in directories, making it easier to manage large projects.

**Key Topics:**

* What is a package?
* Structure of packages
* Creating a package (\_\_init\_\_.py)
* Installing external packages using pip

**Practice:**

1. Create a package with multiple modules.
2. Install and use an external package (e.g., numpy).
3. Explore pip commands (install, uninstall, list).

**END OF UNIT - 5**