TOPIC

Introduction to File Handling, File Operations, and Directories in Python.

What is File Handling?

File Handling :

The ability to interact with files stored on your computer's storage system, including reading, writing, and modifying their contents.

Introduction to File Handling

- Python provides easy-to-use modules to work with files and directories.
- File handling involves creating, reading, writing, and deleting files.
- Commonly used for:
  - Data storage
  - Configuration files
  - Logging

Importance

File handling is essential for managing data, storing information, and creating programs that can access and manipulate files.

File Modes:

- `'r'`: Read mode
- `'w'`: Write mode
- `'a'`: Append mode
- `'r+'`: Read and write mode

Basic File Operations

1 **Open**

Establishes a connection between your program and the file, allowing access to its contents.

2 **Read**

Retrieves data from an existing file and makes it available to your program.

3 **Write**

Sends data to a file, either creating a new file or modifying an existing one.

4 **Close**

Terminates the connection between your program and the file, ensuring proper data integrity.

File Modes

Read Mode ('r')

Opens a file for reading only. The file must exist.

Write Mode ('w')

Opens a file for writing only. If the file exists, it is overwritten. If it doesn't exist, it is created.

Append Mode ('a')

Opens a file for appending. If the file exists, data is added to the end. If it doesn't exist, it is created.

Binary Mode ('b')

Opens a file in binary mode, suitable for working with images, audio, or other non-text data.

Opening and Closing Files

- **Syntax :**

```
file = open("filename.txt", "mode") file.close()
```

- **Using "`with`" Statement:**

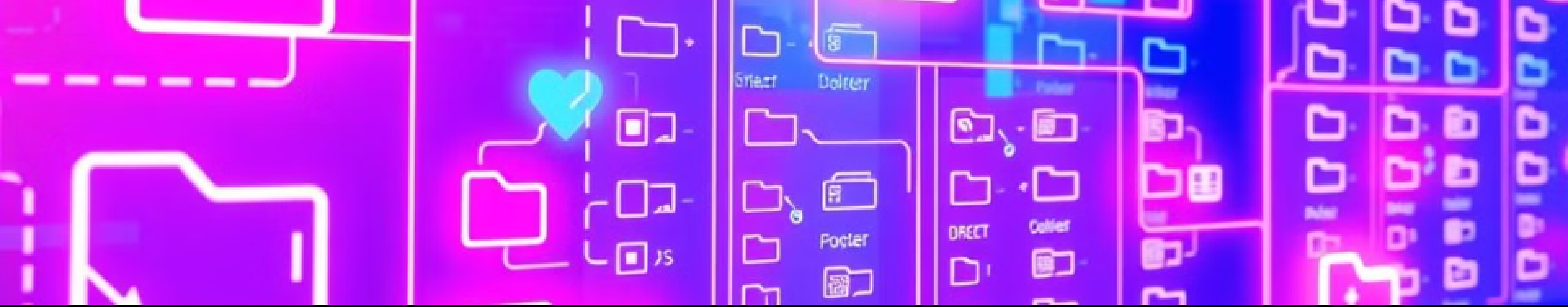
```
with open("filename.txt", "r") as file: data = file.read()
```

Reading Files

- **Methods:**
  - `read()`: Reads the entire file.
  - `readline()`: Reads one line at a time.
  - `readlines()`: Returns a list of lines.

Example:

```
with open("example.txt", "r") as file: print(file.read())
```



Handling Directories

Creating Directories

Create new directories using the 'os. mkdir()' function.

Deleting Directories

Remove directories using the 'os.rmdir()' function.

Navigating Directories

Move between directories using the 'os. chdir()' function.

Working with File Paths

1 **Absolute Paths**

Provide the full path from the root directory to the desired file or directory.

2 **Relative Paths**

Specify the path based on the current working directory, using '.' for the current directory and '..' for the parent directory.

3 **Path Manipulation**

Python provides modules like 'os.path' for manipulating file paths, including joining, splitting, and getting directory names.

Writing to Files

- **Methods:**
  - `write()`: Writes a string to the file.
  - `writelines()`: Writes a list of strings.
- **Example:**

```
with open("example.txt", "w") as file: file.write("Hello, Python!")
```

File Operations

- **Common Functions:**
  - `os.rename("old.txt", "new.txt")`
  - `os.remove("file.txt")`
  - `os.path.exists("file.txt"`

File and Directory Metadata

1

Size

The size of a file in bytes using 'os.path.getsize()'.

2

Creation Date

The time of creation of a file using 'os.path.getctime()'.

3

Permissions

Access control details for files and directories using 'os.stat()'.

Directories in Python

- **What are Directories?**
  - Containers to organize files

- **Key Functions (from `os` module):**
  - `os.mkdir("dirname")`: Create a directory.
  - `os.rmdir("dirname")`: Remove an empty directory.
  - `os.listdir("path")`: List contents of a directory.

Directories In Python

What are Directories?

- Containers to organize files.

Key Functions (from `os` module):

- `os.mkdir("dirname")`: Create a directory.
- `os.rmdir("dirname")`: Remove an empty directory.
- `os.listdir("path")`: List contents of a directory.

Working with Directories

- **Changing Directory:**

```
os.chdir("new_folder") print("Current Directory:", os.getcwd())
```

- **Check Current Working Directory:**

```
import os print("Current Directory:", os.getcwd())
```

Error Handling

1

Try-Except

Enclose file operations within a 'try' block to catch any potential exceptions.

2

Exceptions

Common file handling exceptions include 'FileNotFoundError', 'PermissionError', and 'IOError'.

3

Handling Exceptions

Use 'except' blocks to handle specific exceptions gracefully and provide appropriate feedback or alternative actions.

Best Practices

1

Close Files

Always close files after you're done using them to prevent resource leaks and data corruption.

2

Context Managers

Use 'with open()' for automatic file closing, ensuring that the file is closed even if an exception occurs.

3

Avoid Hardcoded Paths

Use relative paths or environment variables for flexibility and portability.

Exception Handling in File Operations :

- **Why Use It?**
  - Prevent runtime errors.
- **Example:**

```
try: with open("nonexistent.txt", "r") as file:
print(file.read()) except FileNotFoundError:
print("File not found!")
```

Summary

Key Takeaways:

- Python simplifies file and directory management with built-in modules like `os` and file handling methods.
- You can perform operations such as:
  - Reading (`read()`, `readlines()`), writing (`write()`, `writelines()`), and appending to files.
  - Renaming, deleting, and checking for file existence using the `os` module.
  - Creating, removing, and navigating directories.

Best Practices:

Use the `with` statement for efficient file handling.

- Implement exception handling to ensure error-free operations.

Why Learn This?

- File handling is essential for real-world applications like data storage, logging, and automation.