File Handling

- File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations.
- Python file handling refers to the process of working with files on the filesystem. It involves operations such as reading from files, writing to files, appending data, and managing file pointers.
- Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

- The following operations can be performed with files.
  - Opening
  - Reading
  - Writing
  - Adding (Appending)
  - Closing

Types of File

- **Text File**: Text file usually we use to store character data. For example, test.txt
- **Binary File**: The binary files are used to store binary data such as images, video files, audio files, etc.

File Path

A file path defines the location of a file or folder in the computer system. There are two ways to specify a file path.

1. **Absolute path**: which always begins with the root folder
2. **Relative path**: which is relative to the program's current working directory

Different Modes to Open a File in Python

Mode	Description
`r`	It opens an existing file to read-only mode. The file pointer exists at the beginning.
`rb`	It opens the file to read-only in binary format. The file pointer exists at the beginning.
`r+`	It opens the file to read and write both. The file pointer exists at the beginning.
`rb+`	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
`w`	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name.

`wb`	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists.
`w+`	It opens the file to write and read data. It will override existing data.
`wb+`	It opens the file to write and read both in binary format
`a`	It opens the file in the append mode. It will not override existing data. It creates a new file if no file exists with the same name.
`ab`	It opens the file in the append mode in binary format.
`a+`	It opens a file to append and read both.
`ab+`	It opens a file to append and read both in binary format.

Opening Files

- In Python, we need to open a file first to perform any operations on it—we use the `open()` function to do so.
- Let's look at an example:
  - Suppose we have a file named `file1.txt`.
  - To open this file, we can use the `open()` function.
    ```
    file1 = open("file1.txt")
    ```
  - Here, we have created a file object named file1. Now, we can use this object to work with files.

Example: Opening file with relative path

```
# Opening the file in the same folder
fp = open('FH.txt', 'r')
# read file
for each in fp:
    print(each)
# Closing the file after reading
```

```
Welcome to MLW01

This is a sample.txt

Line 3

Line 4

Line 5
```

Example: Opening file with absolute path

```
1  # Opening the file with absolute path
2  fp = open(r"C:\Users\gyana\OneDrive\CM_FDP\AIML\MLW Course 2024\Programs\FH.txt", 'r')
3
4  # read file
5  print(fp.read())
6  # Closing the file after reading
7  fp.close()
8
9  # Reading a part of the file
10 fp = open('FH.txt', "r")
11 print (fp.read(7))
12 fp.close()
```

```
Welcome to MLW01
This is a sample.txt
Line 3
Line 4
Line 5
Welcome
```

- We can also split lines while reading files in Python. The `split()` function splits the variable when space is encountered. You can also split using any characters as you wish.

Example

```
# Python code to illustrate split() function
with open("FH.txt", "r") as file:
    data = file.readlines()
    for line in data:
        word = line.split()
        print (word)
```

```
['Welcome', 'to', 'MLW01']
['This', 'is', 'a', 'sample.txt']
['Line', '3']
['Line', '4']
['Line', '5']
```

Writing to a File

- To write content into a file, Use the access mode `w` to open a file in a write mode.
- **Note:**
  - If a file already exists, it truncates the existing content and places the file handle at the beginning of the file. A new file is created if the mentioned file doesn't exist.
  - If you want to add content at the end of the file, use the access mode a to open a file in append mode

Example: writing to a file

```
text = "This is new content"

# writing new content to the file
fp = open('FH.txt', 'w')
fp.write(text)
print('Done Writing')
fp.close()

# reading the file contents
fp = open('FH.txt', 'r')
for each in fp:
    print(each)

# Closing the file after reading
fp.close()
```

```
Done Writing
This is new content
```

- We can also use the written statement along with the `with()` function.

Example: writing to a file using with()

```
# Python code to illustrate with() alongwith write()
with open("FH.txt", "w") as fp:
    fp.write("Writing line 1\n")
    fp.write("Writing line 2")

fp = open('FH.txt', 'r')
for each in fp:
    print(each)

# Closing the file after reading
fp.close()

```

```
Writing line 1

Writing line 2
```

Append Mode

- Let us see how the append mode works.

Example

```
# Python code to illustrate append() mode
file = open('FH.txt', 'a')
file.write("\nThis will add this line")
file.close()

fp = open("FH.txt", 'r')
# read file
print(fp.read())
# Closing the file after reading
fp.close()
```

```
Writing line 1
Writing line 2
This will add this line
```

Move File Pointer

- The `seek()` method is used to change or move the file's handle position to the specified location.
- The cursor defines where the data must be read or written in the file.
- The position (index) of the first character in files is zero, just like the string index.

Example

```
fp = open("FH.txt", "r")
# move to 11 character
fp.seek(11)
# read from 11th character
print(fp.read())
# Closing the file after reading
fp.close()
```

```
MLW01
This is a sample.txt
Line 3
Line 4
Line 5
```

- The `tell()` method to return the current position of the file pointer from the beginning of the file.

Example

```
f = open("FH.txt", "r")
# read first line
f.readline()
# get current position of file handle
print(f.tell())
```

```
18
```

Copy Files

- There are several ways to cop files in Python.
- The `shutil.copy()` method is used to copy the source file's content to the destination file.

Example

```
import shutil

src_path = "FH.txt"
dst_path = "FH_copy.txt"
shutil.copy(src_path, dst_path)
print('Copied')
```

```
Copied
```

File Object Methods

Method	Description
`read()`	Returns the file content.
`readline()`	Read single line
`readlines()`	Read file into a list
`truncate(size)`	Resizes the file to a specified size.
`write()`	Writes the specified string to the file.
`writelines()`	Writes a list of strings to the file.
`close()`	Closes the opened file.
`seek()`	Set file pointer position in a file
`tell()`	Returns the current file location.
`fileno()`	Returns a number that represents the stream, from the operating system's perspective.
`flush()`	Flushes the internal buffer.

Rename Files

- In Python, the `os` module provides the functions for file processing operations such as renaming, deleting the file, etc.
- The `os` module enables interaction with the operating system.
- The `os` module provides `rename()` method to rename the specified file name to the new name. The syntax of `rename()` method is shown below.

Example

```
# File Rename

import os

# Absolute path of a file
old_name = "FH.txt"
new_name = "FH_Rename.txt"

# Renaming the file
os.rename(old_name, new_name)
```

Functions to delete files and folders

Function	Description
`os.remove('file_path')`	Removes the specified file.
`os.unlink('file_path')`	Removes the specified file. Useful in UNIX environment.
`pathlib.Path("file_path").unlink()`	Delete the file or symbolic link in the mentioned path
`os.rmdir('empty_dir_path')`	Removes the empty folder.
`pathlib.Path(empty_dir_path).rmdir()`	Unlink and delete the empty folder.
`shutil.rmtree('dir_path')`	Delete a directory and the files contained in it.

- **Note:**
  - All above functions delete files and folders permanently.
  - The `pathlib` module was added in Python 3.4. It is appropriate when your application runs on different operating systems.

Delete Files

- In Python, the `os` module provides the `remove()` function to remove or delete file path.

Example

```
import os

# remove file with absolute path
os.remove("FH_copy.txt")
```

Check if File exist:

- To avoid getting an error, you might want to check if the file exists before you try to delete it.

Example

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

- To delete an entire folder, use the `os.rmdir()` method.
- You can only remove ***empty*** folders.

Example

```
import os
os.rmdir("myfolder")
```

Implementing all the functions in File Handling

- In this example, we will cover all the concepts that we have seen above.
- Other than those, we will also see how we can delete a file using the `remove()` function from Python `os` module .

Example

```
import os

def create_file(filename):
    try:
        with open(filename, 'w') as f:
            f.write('Hello, world!\n')
        print("File " + filename + " created successfully.")
    except IOError:
        print("Error: could not create file " + filename)

def read_file(filename):
    try:
        with open(filename, 'r') as f:
            contents = f.read()
            print(contents)
    except IOError:
        print("Error: could not read file " + filename)

def append_file(filename, text):
    try:
        with open(filename, 'a') as f:
            f.write(text)
        print("Text appended to file " + filename + " successfully.")
    except IOError:
```

```
        print("Error: could not append to file " + filename)

def rename_file(filename, new_filename):
    try:
        os.rename(filename, new_filename)
        print("File " + filename + " renamed to " + new_filename + "
              successfully.")
    except IOError:
        print("Error: could not rename file " + filename)

def delete_file(filename):
    try:
        os.remove(filename)
        print("File " + filename + " deleted successfully.")
    except IOError:
        print("Error: could not delete file " + filename)


if __name__ == '__main__':
    filename = "example.txt"
    new_filename = "new_example.txt"

    create_file(filename)
    read_file(filename)
    append_file(filename, "This is some additional text.\n")
    read_file(filename)
    rename_file(filename, new_filename)
    read_file(new_filename)
    delete_file(new_filename)
```

Result

```
File example.txt created successfully.
Hello, world!

Text appended to file example.txt successfully.
Hello, world!
This is some additional text.

File example.txt renamed to new_example.txt successfully.
Hello, world!
This is some additional text.

File new_example.txt deleted successfully.
```

Advantages of File Handling in Python

- **Versatility**: File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.
- **Flexibility**: File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files , etc.), and to perform different operations on files (e.g. read, write, append, etc.).
- **User – friendly**: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.
- **Cross-platform**: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling in Python

- **Error-prone:** File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks**: File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity**: File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.
- **Performance**: File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.