3. Opening and Closing Files

❖Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

Modes :

- **'r'** – open for **reading only**. The file must exist. The pointer starts at the beginning.
- **'w'** – open for **writing only**. If the file exists, it's **truncated** (emptied); if not, it's created. Pointer at start.
- **'a'** – open for **writing only**, but **always appending**. Creates file if missing. Pointer at end.

- **'r+'** – read *and* write, no truncation; file must exist; pointer at the beginning.
- **'w+'** – truncate/create, then read/write from the start.
- **'a+'** – read/write, but writing always goes to the file's **current end**, regardless of seek(); file created if needed.

❖Using the open() function to create and access files.

open() Function:

Python’s open() is the core file-handling function. It returns a file object used for reading, writing, appending, or creating files. You use it to connect with files stored on disk, and then operate on them via methods like .read(), .write(), .readline(), .truncate(), and .seek() .

Syntax :

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
    newline=None, closefd=True, opener=None)
```

Creating a new file :

```
f = open('new.txt', 'x')  # fails if exists
```

Writing (overwrite or create):

```
with open('file.txt', 'w', encoding='utf-8') as f:
   f.write("Hello\n")
```

Appending (adding without erase) :

```
with open('file.txt', 'a', encoding='utf-8') as f:
    f.write("New line\n")
```

Read and Write (+ modes) :

```
with open('file.txt', 'r+', encoding='utf-8') as f:
    data = f.read()
    f.seek(0)
    f.write("Updated start\n")
    f.truncate()
```

Handling binary files :

```
with open('img.png', 'rb') as src, open('copy.png', 'wb') as dst:
    dst.write(src.read())
```

❖Closing files using close().

Why Close Files?

1. **Flush Buffers & Save Data :**
   The close() method flushes any buffered data—ensuring it's written from memory to disk. Without it, you risk losing data if your script crashes or exits unexpectedly.

2. **Release Operating System Resources**
   Every opened file consumes a file descriptor (or handle). These are limited by the OS—persistently opening files without closing can lead to resource exhaustion.

3. **Avoid File Locking Issues**
   On systems like Windows, files remain locked while open, preventing other processes from accessing them. Closing releases the lock.

4. **Better Portability & Predictability**
   Relying on garbage collection to close files isn’t reliable across Python implementations .

Syntax:

```
f = open('example.txt', 'w')

f.write('Hello')

f.close()
```