Day 30: File Handling - Intermediate

File Modes Beyond Basics

- `'rb'` / `'wb'` – Read/Write in binary (e.g., images, PDFs)
  - Needed for working with non-text data

```
with open("image.jpg", "rb") as f:
    data = f.read(100)
```

- `'r+'` / `'a+'` / `'w+'` – Read and write in the same file

```
with open("example.txt", "r+") as f:
    print(f.read())       # Reads and prints content
    f.seek(0)             # Move pointer to beginning
    f.write("New Line")   # Overwrites from beginning
```

```
with open("example.txt", "a+") as f:
    f.write("\nAppended Line")  # Adds to the end
    f.seek(0)                   # Move pointer to beginning
    print(f.read())             # Reads full content
```

- `'x'` useful in sensitive file creation (e.g., logs, configs)

Using `with` and Custom Context Managers

- `with` handles file closing automatically, even on error
- Helps avoid resource leaks

```
with open("data.txt", "r") as f:
    print(f.readline())
```

Custom context manager:

- The `__enter__` method is called when you enter the with block, and its return value is assigned to a variable within that block.
- The `__exit__` method, on the other hand, is called when the `with` block exits, regardless of whether it finishes normally or with an exception.

```
class MyFile:
    def __enter__(self):
        self.f = open("log.txt", "w")
        return self.f
    def __exit__(self, exc_type, exc_val, exc_tb):
        self.f.close()

with MyFile() as f:
    f.write("Hello with custom context!")
```

Why this matters:

- Shows how context managers work internally
- Useful when writing classes that manage resources

File Pointer Manipulation

- `f.tell()` – returns current position
- `f.seek(offset)` – moves pointer

Use Case: Update a specific part of the file without rewriting everything.

```
with open("sample.txt", "r+") as f:
    f.seek(5)
```

```
    f.write("NEW")
```

Why this matters:

- Enables random access file editing
- Often used in database, logging, or caching systems

Memory-Efficient Large File Processing

- Why reading line-by-line is better than `.read()`
- Use generators or chunked reads

```
def read_large_file(filename):
    with open(filename, "r") as f:
        for line in f:
            yield line.strip()

for line in read_large_file("bigfile.txt"):
    print(line)
```

Why this matters:

- Saves memory
- Required when processing logs, analytics, etc.

📝 Test