9️⃣

Chapter 9

📁 Chapter 9: File I/O in Python

> Learn how to read, write, append, and manage files in Python

---

🧠 Main Code

📄 `file.py` – Reading a File

```
# 📂 FILE I/O in Python

f = open("file.txt", "r")  # 📖 Opens file.txt in read mode
data = f.read()            # 🧠 Reads the entire file content into a string
print(data)                # 🖨️ Prints the content on screen
f.close()                  # 🔐 Closes the file to free resources
```

📝 **Note:** Always remember to close the file unless you're using a `with` block (we'll cover that soon!).

✍️ `file_write.py` – Writing to a File

```
st = "Hey Prathamesh You are amazing and cool"

f = open("Myfile.txt", "w")  # ✏️ Opens Myfile.txt in write mode (creates file i
f it doesn't exist)
f.write(st)              # 💾 Writes the string to the file (⚠️ Overwrites existing
content!)
f.close()                # 🔐 Always close the file
```

💡 **Use Case:** Writing logs, saving user data, etc.

📜 more_file_types.py – Reading Line by Line

```
f = open("file.txt")        # Default mode is "r" (read)

line = f.readline()         # 📥 Reads first line

while line != "":           # 🔁 Loop until end of file
    print(line)             # 🖨️ Print current line
    line = f.readline()     # Move to next line
```

📝 **Tip:** This method is useful when reading large files line-by-line to save memory.

➕ file_append.py – Appending to a File

```
st = "Hey Prathamesh You are amazing and cool"

f = open("Myfile.txt", "a")  # ➕ Opens file in append mode (won't delete exi
sting content)
f.write(st)              # Adds the string to the end of file
f.close()                # 🔐 Close after writing
```

📌 **Perfect for:** Adding logs, updating data, or journaling programs.

🔒 with.py – Using with (Context Manager)

```
f = open("file.txt")
print(f.read())
```

```
f.close()
```

⬇️ **Can be simplified to this:**

```
# ✅ Efficient way to open and read a file
with open("file.txt") as f:
    print(f.read())  # File is automatically closed after block
```

✨ **Benefits of `with`:**

- No need to call `f.close()`
- Cleaner syntax
- Fewer chances of forgetting to close the file

🔬 Deep Dive

🧠 File Modes Explained

Mode	Description
`"r"`	Read (default)
`"w"`	Write (overwrites existing file)
`"a"`	Append (adds to file without deleting)
`"r+"`	Read + Write
`"w+"`	Write + Read (overwrites file)
`"a+"`	Append + Read

📦 Real World Examples:

- Logging errors to a file → `"a"`
- Reading config files → `"r"`
- Saving high scores in a game → `"w"` or `"a"`

🚧 Common Mistakes

- ❌ Forgetting to close a file (especially outside `with`)
- ❌ Opening a non-existent file in `"r"` mode causes an error
- ❌ Using `"w"` when you meant `"a"` — you'll **lose** existing content!

🧪 Summary

✅ You can:

- Read files with `.read()`, `.readline()`, `.readlines()`
- Write using `"w"`, append using `"a"`
- Use `with` to handle files more safely

⚡ **Remember this mini flow:**

```
with open("filename.txt", "mode") as f:
    data = f.read()
    # do something
```

🎯 Handy rule:

> "If you open it, you must close it. Unless you're polite and use 'with'." 😉

🔬 Deep Dive – Advanced File I/O Concepts in Python

🔧 1. Different File Modes (Beyond r, w, a)

👷 **Use case:**

```
with open("image.jpg", "rb") as f:
    data = f.read()  # Reads binary data of the image
```

🪵 2. Buffering and File Object Internals

When you open a file, Python buffers it (holds a chunk in memory). This affects how data is read/written.

```
open("file.txt", "r", buffering=0)  # No buffering (rarely used)
open("file.txt", "r", buffering=1)  # Line buffered (default in text mode)
open("file.txt", "rb", buffering=4096)  # Custom buffer size in bytes
```

💡 Use this when working with **very large files** to optimize performance.

📄 3. Tell() and Seek(): Navigating Inside a File

🧭 `.tell()` – shows where you are (cursor position)

```
f = open("file.txt", "r")
print(f.tell())  # 👀 Returns byte position (e.g., 0 at start)
```

🕹️ `.seek(offset, whence)` – move around in the file

```
f.seek(10)     # ⏩ Move to the 10th byte
f.seek(0)      # ⏮️ Go back to start
f.seek(-2, 2)  # 🔚 Move 2 bytes before EOF (whence=2: end of file)
```

Why care? Super useful for:

- File parsers
- Rewinding/fast-forwarding
- Skipping headers or metadata

📁 4. Working with Directories and Files using `os` & `pathlib`

Sometimes, it's not just about reading a file—it's about managing many files.

```
import os

print(os.getcwd())  # 📍 Get current working directory
os.chdir("folder/")  # 📂 Change directory
os.listdir()        # 📑 List files in folder
```

`pathlib` (cleaner alternative in Python 3.6+)

```
from pathlib import Path

p = Path("folder/")
for file in p.iterdir():
    print(file.name)
```

🧪 5. Exception Handling While Opening Files

You **must** handle errors in real-world apps.

```
try:
    with open("data.txt", "r") as f:
        content = f.read()
except FileNotFoundError:
    print("❌ File not found!")
except IOError:
    print("⚠️ IO Error occurred")
```

This prevents your app from crashing and helps you debug.

📚 6. Reading Large Files Without Killing RAM

Instead of `.read()` (which loads entire file), use generators:

```
def read_large_file(filename):
    with open(filename) as f:
        for line in f:
            yield line  # ✅ Generator: reads one line at a time
```

⚡ Use this for log processing, CSVs, etc.

📂 7. Saving Structured Data (JSON, CSV)

Instead of writing raw text, we often store structured data.

📊 CSV

```
import csv

with open("data.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "Score"])
    writer.writerow(["Prathamesh", 100])
```

🧠 JSON

```
import json

data = {"name": "Prathamesh", "coolness": "MAX"}
```

```
with open("data.json", "w") as f:
    json.dump(data, f)  # 🔒 Writes JSON to file
```

And to read:

```
with open("data.json", "r") as f:
    info = json.load(f)
```

🧽 8. File Cleanup and Temporary Files

You don't always want to save files forever.

🧽 Delete files

```
import os
os.remove("useless_file.txt")  # 💣 Deletes file
```

🧪 Temp files

```
import tempfile

with tempfile.TemporaryFile() as temp:
    temp.write(b"Hello!")  # Use binary mode
    temp.seek(0)
    print(temp.read())  # ⚡ Temporary, auto-deletes!
```

Great for sensitive data, scratchpads, or testing.

🧾 TL;DR Cheat Sheet

Concept	Description	Use Case
`seek()` / `tell()`	Navigate inside files	Parsers, scanners
`with open(...)`	Auto-close files	Best practice
`a+`, `r+`	Read + write combined	Logs, editable files
`json`, `csv`	Structured file formats	APIs, datasets
`tempfile`	Auto-delete scratch files	Testing, cache

`os`, `pathlib`	File/directory handling	Automations, scripts

🧠 Practice Problems – Chapter 9: File I/O

📘 Problem 1: Detecting a Word in a File

```
# ✅ Goal: Check if the word "twinkle" is present in the file 'poems.txt'

with open("CH9_PS/poems.txt") as f:
    data = f.read()

# 🔍 Convert to lowercase to handle any case ("Twinkle", "twinkle", etc.)
if "twinkle" in data.lower():
    print("This file contains 'twinkle'")
else:
    print("It doesn't contain 'twinkle'")
```

🧠 **Why?** This is basic keyword scanning — useful for detecting forbidden words, tags, or keywords.

🏆 Problem 2: Update High Score

```
# 🚀 This would be used in a game to track and save the highest score

score = int(input("Enter your score: "))

with open("CH9_PS/high_score.txt", "r") as f:
    content = f.readline().strip()

# 🛠️ Handle empty file (no score saved yet)
high_score = int(content) if content else 0

# 🆕 Update only if current score is higher
if score > high_score:
    with open("CH9_PS/high_score.txt", "w") as f:
        f.write(str(score))
    print("New high score saved!")
```

```
else:
    print("No new high score.")
```

💡 Could be modified later to use a `game()` function returning random scores.

👩 Problem 3: Table Generator (Age 13 Friendly)

```
# 🎯 Generate tables from 2 to 20 and save each in a file

def tableGenerate(n):
    table = ""
    for i in range(1, 11):
        table += f"{n} X {i} = {n*i}\n"

    # 💾 Save to individual files per number
    with open(f"CH9_PS/tables/table_{n}.txt", "w") as f:
        f.write(table)

# 🔁 Generate for numbers 2 to 20
for i in range(2, 21):
    tableGenerate(i)
```

🧠 You could use this idea to dynamically create educational content or personalized homework sheets.

🚫 Problem 4: Word Censoring (Single Word)

```
# 🛑 Replace "donkey" with #####

with open("CH9_PS/donkey.txt", "r") as f:
    content = f.read()

# 👮 Replace the word with #####
censored = content.replace("donkey", "#####")

with open("CH9_PS/donkey.txt", "w") as f:
    f.write(censored)
```

💡 Real-world use: Filtering chat messages or censoring inappropriate words.

🧹 Problem 5: Censor a List of Words

```
# 🚫 Replace multiple bad words with #####

words = ["world", "donkey", "prestigious", "their"]

with open("CH9_PS/para.txt", "r") as f:
    content = f.read().lower()

# 🔄 Replace each bad word with the same number of #
for word in words:
    content = content.replace(word, "#" * len(word))

with open("CH9_PS/para.txt", "w") as f:
    f.write(content)
```

🧠 Dynamically adjusts the number of `#` based on word length — smart censorship technique.

📜 Problem 6: Log File Keyword Search

```
# 🔍 Check if "python" exists in a log file

with open("CH9_PS/log.txt", "r") as f:
    content = f.read().lower()

if "python" in content:
    print("It contains 'python'")
else:
    print("No mention of 'python'")
```

✅ Useful in log analysis tools, error scanning systems, etc.

🕵️ Problem 7: Find Line Number of a Word

```
# 📍 Find the first line that contains "python"

with open("CH9_PS/log.txt", "r") as f:
```

```
    lines = f.readlines()

for lineno, line in enumerate(lines, start=1):
    if "python" in line.lower():
        print(f"'python' found at line {lineno}")
        break
else:
    print("'python' not found")
```

📌 Use `enumerate()` for clean line tracking — powerful for code scanning and logs.

📁 Problem 8: Copy File Content

```
# 📋 Make a duplicate of a file

with open("CH9_PS/this.txt", "r") as f:
    content = f.read()

with open("CH9_PS/this2.txt", "w") as f:
    f.write(content)
```

🧠 Use this in backup systems or cloning templates.

🧬 Problem 9: Compare Two Files

```
# 🔍 Check if two files have identical content

with open("CH9_PS/this.txt", "r") as f:
    content1 = f.read()

with open("CH9_PS/this1.txt", "r") as f:
    content2 = f.read()

if content1 == content2:
    print("The files are identical ✅")
else:
    print("The files are different ❌")
```

🔐 Useful for file integrity checks, version control.

🧨 Problem 10: Wipe Out File Content

```
# 🧼 Clear everything from a file

with open("CH9_PS/this2.txt", "w") as f:
    f.write("")  # Writing empty string wipes the file
```

🧹 Used for log resets, cache clearing, etc.

🏷️ Problem 11: Rename a File (Manual Copy)

```
# 📝 Copy content to a new file with a new name

with open("CH9_PS/good.txt", "r") as f:
    content = f.read()

with open("CH9_PS/renamed_by_python.txt", "w") as f:
    f.write(content)

# 🧽 Optional: Delete the original to complete renaming
# import os
# os.remove("CH9_PS/good.txt")
```

📦 You're simulating a file rename — though Python has a better way with `os.rename()` or `Path.rename()`.

📚 Chapter 9 Summary: File I/O in Python

🔢 No.	🧠 Concept	📝 Description	✅ Key Methods / Syntax	💡 Notes / Tips
1	File Handling Modes	Different ways to open a file	`'r'`, `'w'`, `'a'`, `'rb'`, `'wb'`, `'r+'`	`r`: read, `w`: write (overwrite), `a`: append
2	`open()` Function	Opens a file for a specific operation	`open("filename.txt", "mode")`	Returns a file object

3	`read()`	Reads the **entire content** as a single string	`f.read()`	Can be memory-heavy for large files
4	`readline()`	Reads one line at a time	`f.readline()`	Use in loops for line-by-line reading
5	`readlines()`	Reads all lines into a **list**	`f.readlines()`	Each list element is one line
6	`write()`	Writes content to a file	`f.write("some text")`	Overwrites if file already has content
7	`append()` mode	Adds content to the end of the file	`open("file.txt", "a")`	Does **not** overwrite
8	`with` Statement (Context Manager)	Automatically closes files after use	`with open("file.txt") as f:`	Safe & clean way to handle files
9	File Closing	Manually close the file	`f.close()`	Avoid forgetting it—better to use `with`
10	File Paths	Relative vs absolute paths	`"folder/file.txt"` or `"C:\\Users\\file.txt"`	Double backslashes `\\` in Windows
11	Reading in Loop	Read line-by-line using loop and `readline()`	`while line != "":`	Avoids memory overload with big files
12	Case Insensitive Matching	Helps find words regardless of case	`data.lower()`	Great for keyword detection
13	Replacing Content	Replace specific words with another	`data.replace("old", "new")`	Used for censorship, text cleanup
14	File Overwriting	Overwriting existing content	Use mode `"w"`	Always use with caution
15	Censoring Words	Replacing offensive or	Loop + `replace()`	Dynamically replace using

		specific words		`"#" * len(word)`
16	Creating Multiple Files	Write a loop that generates different files	`open(f"file_{n}.txt", "w")`	Used in generators like table, logs
17	File Duplication	Copy one file's content to another	Read from one → write to another	Used in backup systems
18	File Comparison	Check if contents of two files are identical	`if file1 == file2:`	Use `.read()` to get content for comparison
19	Wipe File Content	Clear all data from a file	`f.write("")`	Keep filename same, just empty it
20	Rename a File	Rename by copying + deleting OR using `os.rename()`	`os.rename("old.txt", "new.txt")`	Needs `import os`
21	`os` Module	For file operations (rename, delete, path ops)	`os.remove()`, `os.rename()`, `os.path.exists()`	Enables system-level file operations
22	Exception Handling in File Ops	Catch file-not-found or permission errors	`try-except` block around file ops	Prevents crash due to missing files
23	Binary File Handling	For images, videos, etc.	`open("img.png", "rb")`, `write()`	Use `"rb"`, `"wb"` for binary files
24	File Metadata	Get size, modified time, etc.	`os.stat("file.txt")`	Check file stats using `os` module

🧠 Good to Remember

- ✅ Always close files or use `with open(...) as f:` to auto-close
- 🛠️ `write()` only accepts strings, convert numbers using `str()`

- 🔐 File operations are **slow** — avoid unnecessary reads/writes
- 🧪 Use exception handling (`try-except`) in production code
- 🚀 `with` statement is the **best practice** for working with files