Opening, Writing & Reading Files

AiCE Software Engineering Foundations — Python Programming (Intermediate)

Every program you've written so far forgets everything the moment it stops running — variables, lists, dictionaries, all of it vanishes when Python exits. File handling changes that. It gives your programs the ability to read data that existed before they ran, and to write data that persists after they finish. This lesson opens that capability: opening, writing to, and reading from files.

1. Why File Handling Matters

Consider an agricultural data platform tracking rainfall records for farmers in northern Nigeria. Every morning, new readings arrive. Every evening, agronomists need to review the full historical dataset. The platform cannot store everything in memory — it would be lost each time the program restarts. It must read from files and write to files.

Or consider an AiCE student management system. When a new student registers, their record must be saved somewhere. When an administrator logs in the next day, those records must be retrievable. Files — and later, databases — are how programs persist information across sessions.

> **The Bridge Between Temporary and Permanent**
> *File handling is the bridge between the temporary world of running programs and the permanent world of stored data. Every real application uses it.*

2. Opening a File — The open() Function

Python's built-in open() function is the gateway to file handling. It opens a file and returns a file object — a connection between your program and the file on disk. Through this connection, you can read, write, or append.

```
file_object = open("filename.txt", "mode")
```

The first argument is the file path; the second is the mode — a short string telling Python what you intend to do with the file:

Mode	Name	What It Does	File Must Exist?
"r"	Read	Opens for reading. Default mode.	Yes — FileNotFoundError if missing
"w"	Write	Opens for writing. Creates file if missing. Overwrites if it exists.	No — creates if missing
"a"	Append	Opens for appending. Adds to the end without overwriting.	No — creates if missing
"x"	Exclusive Create	Creates a new file. Fails if it already exists.	No — FileExistsError if it does

Always Close What You Open

When you open a file with open(), Python establishes a connection between your program and the file on disk. If you don't close that connection, data may not be written properly, and system resources are wasted. Always close files with file_object.close() — or better yet, use the with statement, covered later in this unit, which closes the file automatically.

3. Writing to a File

The most natural starting point is writing — creating a file and putting content into it:

```
# Writing to a file
file = open("students.txt", "w")
file.write("Amara Diallo\n")
file.write("Kofi Mensah\n")
file.write("Ngozi Adeyemi\n")
file.close()
print("File written successfully.")
```

Checking the file from the terminal with cat students.txt shows three names, each on its own line.

The \n at the end of each write() call inserts a newline character — without it, everything would appear on a single line. open("students.txt", "w") created the file because it didn't exist. If you run the program again, it will overwrite the file completely — all previous content is gone. This is the critical behavior of write mode.

Writing Multiple Lines with writelines()

Instead of calling write() once per line, writelines() writes all lines from a list in one call:

```
lines = [
    "Adesina Olagunju\n",
    "Founder of AiCE\n",
]

file = open("about_aice.txt", "w")
file.writelines(lines)
file.close()
```

writelines() writes each item in the list sequentially. It does not add newlines automatically — you must include \n in each string yourself.

4. Reading from a File

Once a file exists, you can read its content back. Python provides three methods for reading, each suited to different situations.

.read() — Read the Entire File as One String

```
# Read the entire file content at once
file = open("students.txt", "r")
content = file.read()
file.close()
print(content)
print(f"Total characters: {len(content)}")
```

.read() returns the entire file as a single string — newlines and all. This is fine for small files but impractical for large ones that might be hundreds of megabytes.

.readline() — Read One Line at a Time

```
# Read one line at a time
file = open("students.txt", "r")
line1 = file.readline()
line2 = file.readline()
file.close()
print(f"Line 1: {line1.strip()}")
print(f"Line 2: {line2.strip()}")
```

.readline() reads and returns one line each time it's called, advancing the file position. The .strip() removes the trailing newline character — you almost always want to strip when reading lines.

.readlines() — Read All Lines as a List

```
# Read all lines into a list
file = open("students.txt", "r")
lines = file.readlines()
file.close()

print(f"Total students: {len(lines)}")
for i, line in enumerate(lines, 1):
    name = line.strip()
    print(f"  {i}. {name}")
```

.readlines() returns a list where each item is one line from the file, including the \n character at the end. Always .strip() each line when you use the content. This method is ideal when you need to process all lines as a collection.

Key Takeaways

open(filename, mode) is the gateway to file handling — "r" to read, "w" to write (overwriting existing content), "a" to append, and "x" to create exclusively. write() and writelines() need explicit \n characters, while read(), readline(), and readlines() each read content back in different shapes, suited to different situations. Every file you open must eventually be closed. The next lesson introduces the with statement — the professional,

automatic way to handle that closing — along with appending to files and safely checking whether a file exists before you try to read it.