



MODULE 1: CORE PYTHON & DATA

WEEK: 3 LECTURE: 11

DATE: 01/09/2025

INSTRUCTOR: HAMZA SAJID

BEYOND VOLATILE MEMORY: INTRODUCTION TO FILE I/O

Welcome back! Until now, our programs have had amnesia. Every time we run them, they start from a blank slate, and any data we create—like a to-do list or a contact book—vanishes the moment the program ends. Today, we solve that. We'll learn how to read from and write to files, allowing our applications to save their state, load data, and interact with the outside world.

TODAY'S AGENDA

- **The Fundamentals of Text Files (.txt)**

- What is File I/O? Making Programs Remember
- The `with open(...)` Statement: The Safe and Modern Way
- File Modes: Reading ('r'), Writing ('w'), and Appending ('a')
- Reading from Files: `.read()`, `.readlines()`, and Looping
- Writing to Files: `.write()`
- *Interactive Exercise: Saving and Reading a To-Do List (Plain Text)*

- **Working with Structured Data (.csv & JSON)**

- The Problem with Plain Text for Complex Data
- Introduction to CSV (Comma-Separated Values)
- Reading CSV Files Manually and with the `csv` Module
- **Callback: A Better Way to Save Our Contact Book**
- Introduction to JSON: The Language of Web APIs
- Using the `json` Module: `json.dump()` and `json.load()`
- *Interactive Exercise: Saving and Loading a Dictionary as JSON*

- **Best Practices & The Hands-On Lab**

- Handling File Paths with `pathlib`
- Error Handling for Files: `try...except` `FileNotFoundError`
- **Hands-On Lab: Log File Analyzer**
- Q&A and Wrap-up

THE FUNDAMENTALS OF TEXT FILES (.TXT)

WHAT IS FILE I/O?

- File I/O (Input/Output) is the process of a program reading data *from* a file (input) or writing data *to* a file (output). This is how we achieve **persistence**—the ability for data to survive after the program has stopped running.

THE WITH OPEN(...) STATEMENT: THE SAFE WAY

- To work with a file, you must first open it. The modern, recommended way to do this is with a with statement. This creates a context where the file is open, and it **automatically and safely closes the file for you** when you are done, even if errors occur.

```
# The 'with' block handles opening
and closing the file
with open("my_file.txt", "r") as f:
    # 'f' is the variable we use to
    interact with the file object
    content = f.read()
    print(content)
```

FILE MODES: 'R', 'W', AND 'A'

The second argument to `open()` is the **mode**. It tells Python what you intend to do with the file.

- 'r' (Read): **Default mode**. Opens a file for reading. Throws an error if the file does not exist.
- 'w' (Write): Opens a file for writing. **It will create the file if it doesn't exist. If it does exist, it will completely overwrite the contents.**
- 'a' (Append): Opens a file for appending. It will create the file if it doesn't exist. If it exists, it will add new content to the **end** of the file.



```
# --- WRITING ---
```

```
lines_to_write = ["First line.\n", "Second line.\n"]
```

```
with open("story.txt", "w") as f:
```

```
    f.write("This is a story.\n")
```

```
    f.writelines(lines_to_write) # writelines takes a list of strings
```

```
# --- READING ---
```

```
with open("story.txt", "r") as f:
```

```
    # Option 1: Read the whole file into one string
```

```
    # full_content = f.read()
```

```
    # Option 2 (Better for line-by-line processing): Loop!
```

```
    for line in f:
```

```
        print(line.strip()) # .strip() removes the newline character
```



IN-CLASS EXERCISE: SAVING A TO-DO LIST

- Create a list of strings: `tasks = ["Buy groceries", "Pay bills", "Walk the dog"]`.
- Write this list to a file named `todo.txt`. Each task should be on a new line.
(Hint: You'll need a for loop and to add the `\n` newline character yourself).
- Write a second piece of code that opens `todo.txt` for reading and prints each task.

WORKING WITH STRUCTURED DATA (.CSV & JSON)

THE PROBLEM WITH PLAIN TEXT

- Our todo.txt is simple. But what if we wanted to save our contact book from a few lectures ago?
- `contact_book = {"Alice": "555-1234", "Bob": "555-5678"}`
- How would we save this in a .txt file and be able to read it back into a dictionary? It's complicated. We need formats designed for structured data.

INTRODUCTION TO CSV

- CSV (Comma-Separated Values) is a simple format for tabular data. Each line is a row, and commas separate the values in that row.

- name,email,id
- Alice,alice@example.com,101
- Bob,bob@example.com,102

- Python has a built-in csv module to make working with these files easy.

```
import csv
```

```
with open("users.csv", "r") as f:
```

```
    csv_reader = csv.reader(f)
```

```
    next(csv_reader) # Skip the header row
```

```
    for row in csv_reader:
```

```
        # each 'row' is a list of strings
```

```
        print(f"Name: {row[0]}, Email: {row[1]}")
```

A BETTER WAY TO SAVE OUR DATA: JSON

- While CSV is good for tables, the best way to save native Python data structures like lists and dictionaries is **JSON (JavaScript Object Notation)**. It's human-readable and maps almost perfectly to Python's syntax.
- The `json` module is the tool for this.
 - `json.dump(python_object, file_object)`: **Dumps** a Python object into a file.
 - `json.load(file_object)`: **Loads** a JSON file back into a Python object.



```
import json
```

```
# Let's save our contact book
```


```
contact_book = {  
    "Alice": {"phone": "555-1234", "email": "alice@example.com"},  
    "Bob": {"phone": "555-5678", "email": "bob@example.com"}  
}
```

```
# --- SAVING (DUMPING) ---
```

```
with open("contacts.json", "w") as f:  
    json.dump(contact_book, f, indent=4) # indent makes it readable
```

```
# --- LOADING ---
```

```
with open("contacts.json", "r") as f:  
    loaded_contacts = json.load(f)  
  
print(loaded_contacts["Alice"]["email"]) # -> alice@example.com
```



IN-CLASS EXERCISE: SAVE AND LOAD A CONFIGURATION

- Create a Python dictionary representing a user's settings: `settings = {"theme": "dark", "notifications_enabled": True, "font_size": 14}`.
- Use the `json` module to dump this dictionary into a file named `settings.json`.
- Write a second piece of code that loads `settings.json` back into a new variable and prints the value of the `"theme"` key.

BEST PRACTICES & THE HANDS-ON LAB

ERROR HANDLING FOR FILES

- What happens if you try to read a file that doesn't exist? Your program crashes with a `FileNotFoundException`. We can handle this gracefully using `try...except`.

HANDS-ON LAB: LOG FILE ANALYZER

- **Goal:** You are given a log file from a server. Your task is to read the file, parse it, and count the number of "ERROR" and "WARNING" messages.
- **Sample log.txt**
- Create a new file log_analyzer.py.

PART 1: READING THE FILE

- Create a dictionary to store your counts: `log_counts = {"ERROR": 0, "WARNING": 0}`.
- Use a `with open("log.txt", "r") as f:` block to open the log file.
- Use a `for` loop to iterate through each line in the file.

PART 2: PARSING AND COUNTING

- Inside the loop, you need to check what kind of log message each line is.
- Use string methods. A good approach is to split the line by the colon (:) character. The first part will be the log level (e.g., 'INFO', 'ERROR').

```
parts = line.strip().split(":")  
log_level = parts[0]
```

- Use an if/elif statement to check if the log_level is "ERROR" or "WARNING".
- If it matches, increment the corresponding counter in your log_counts dictionary.

PART 3: REPORTING THE RESULTS

- After the loop has finished, print the final counts from your dictionary in a user-friendly format.

CHALLENGE / BONUS FEATURES:

- **Function Refactoring:** Encapsulate your logic into a function `analyze_log(filepath)` that takes the file path as an argument and returns the counts dictionary.
- **Command-Line Arguments:** Use `sys.argv` so the user can provide the log file path from the command line (e.g., `python log_analyzer.py log.txt`).
- **Error Handling:** Wrap your file opening logic in a `try...except FileNotFoundError` block to handle cases where the user provides a bad file path.
- **CSV Output:** After counting, write the results to a `report.csv` file with two columns: `LogLevel` and `Count`. This will require using the `csv` module.