Structured Data, Read-Process-Write & the Student Registry

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on file handling covers structured data — CSV, the most common format for real-world text data — the read-process-write pattern behind every data pipeline, and closes with a complete, persistent student registry that brings this entire unit together.

1. Working With Structured Data in Files

Most real-world file data is structured — not just names, but records with multiple fields. The most common format for structured text data is CSV (Comma-Separated Values). Each line is a record; each field within the line is separated by a comma.

Writing CSV Data Manually

```
students = [
    {"name": "Amara Diallo",  "country": "Senegal", "score":
88, "grade": "A"},
    {"name": "Kofi Mensah",   "country": "Ghana",   "score":
45, "grade": "F"},
]

with open("students.csv", "w") as file:
    # Write the header row
    file.write("name,country,score,grade\n")
    # Write each student record
    for student in students:
        line =
f'{student["name"]},{student["country"]},{student["score"]},{s
tudent["grade"]}\n'
        file.write(line)

print("CSV file written.")
```

The output is a header row followed by one record per line. This file can be opened in Microsoft Excel or Google Sheets and will display as a proper table.

Reading CSV Data Manually

```
students = []

with open("students.csv", "r") as file:
    lines = file.readlines()

# First line is the header — skip it
header = lines[0].strip().split(",")

# Process each data line
for line in lines[1:]:
    parts = line.strip().split(",")
    student = {
        "name"   : parts[0],
        "country": parts[1],
        "score"  : int(parts[2]),
        "grade"  : parts[3],
    }
    students.append(student)

print(f"Loaded {len(students)} students.")
```

The program reads the CSV file, skips the header, splits each line by commas, converts the score to an integer, and builds a list of student dictionaries — exactly as if the data had been entered by the user, but loaded from a persistent file instead.

Using Python's csv Module

Python's built-in csv module handles CSV files more robustly than manual string splitting — correctly handling commas inside quoted fields, different delimiters, and encoding issues:

```
import csv

students = [
    ["Amara Diallo", "Senegal", 88, "A"],
    ["Kofi Mensah",  "Ghana",   45, "F"],
]

with open("students_v2.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Country", "Score", "Grade"])
# Header
    writer.writerows(students)
# All rows

# Reading with csv.DictReader
with open("students_v2.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        # Each row is a dictionary — keys are the header names
        print(f"{row['Name']:<22} | {row['Country']:<10} |
{row['Score']:>3} | {row['Grade']}")
```

csv.DictReader is particularly powerful — it uses the header row as keys and gives you each row as a dictionary automatically. This is the preferred way to read CSV data in professional Python programs.

> **A Program That Can Learn**
> *A program that forgets everything when it stops is a program that cannot learn. File handling is how your code develops memory — the ability to carry information from one session to the next, to accumulate, to grow.*

2. Reading and Writing in the Same Session

Sometimes you need to read existing data, process it, and write the results back. The pattern is: read first, process in memory, then write the output:

```
# Step 1: Read the raw names from the source file
with open("students.txt", "r") as file:
    raw_names = file.readlines()

# Step 2: Clean and process in memory
processed = []
for i, name in enumerate(raw_names, 1):
    clean = name.strip()
    if clean:
        processed.append(f"{i:02d}. {clean}")

# Step 3: Write the formatted output to a new file
with open("student_report.txt", "w") as file:
    file.write("AiCE STUDENT REGISTRY\n")
    file.write("=" * 30 + "\n")
    for entry in processed:
        file.write(entry + "\n")

print(f"Report written: {len(processed)} students processed.")
```

This read-process-write pattern is the foundation of data transformation pipelines — the kind used in ETL (Extract, Transform, Load) systems that move and reshape data between systems across the continent and the world.

3. File Handling — Complete Reference

Operation	Code	Notes
Open for reading	open("file.txt", "r")	FileNotFoundError if file missing
Open for writing	open("file.txt", "w")	Creates file; overwrites if exists
Open for appending	open("file.txt", "a")	Creates file; preserves existing content
Read all content	file.read()	Returns single string
Read one line	file.readline()	Returns one line including \n

Operation	Code	Notes
Read all lines	file.readlines()	Returns list of lines including \n
Iterate over lines	for line in file:	Most memory-efficient for large files
Write a string	file.write("text\n")	Does not add newline automatically
Close file	file.close()	Always close — or use with
Auto-close with with	with open(...) as f:	Recommended — always closes safely
Check file exists	os.path.exists("file.txt")	Import os first
CSV reader	csv.DictReader(file)	Import csv; rows as dicts

4. Bringing It All Together — The AiCE Persistent Student Registry

This unit's capstone is a persistent student registry — data survives between program runs because it's stored in a file:

```
import os
import csv
import datetime

REGISTRY_FILE = "aice_registry.csv"

def load_students():
    """Load all students from the registry file."""
    if not os.path.exists(REGISTRY_FILE):
        return []
    students = []
    with open(REGISTRY_FILE, "r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            students.append(row)
```

```
    return students

def save_students(students):
    """Save all students to the registry file."""
    with open(REGISTRY_FILE, "w", newline="") as file:
        fieldnames = ["name", "country", "course",
"enrolled_date"]
        writer = csv.DictWriter(file, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(students)

def add_student(students):
    """Collect student details and add to the registry."""
    name    = input("Name   : ").strip().title()
    country = input("Country: ").strip().title()
    course  = input("Course : ").strip().title()
    date    = datetime.date.today().strftime("%d/%m/%Y")

    student = {"name": name, "country": country, "course":
course, "enrolled_date": date}
    students.append(student)
    save_students(students)
    print(f"{name} enrolled successfully.")
    return students

def run():
    """Main application loop."""
    students = load_students()
    print(f"{len(students)} student(s) loaded from file.")

    while True:
        choice = input("\n1. View  2. Enroll  3. Exit:
").strip()
        if choice == "1":
            for s in students:
                print(f"{s['name']} | {s['country']} |
{s['course']}")
        elif choice == "2":
            students = add_student(students)
        elif choice == "3":
```

```
            print(f"Registry saved. {len(students)} student(s)
on file. Goodbye!")
            break

if __name__ == "__main__":
    run()
```

Enroll two or three students, then exit. Run the program again — the students you enrolled are still there, loaded from the CSV file. The program now has persistent memory. This is a complete, functional application, and every concept from this program so far — functions, scope, return values, modules, file handling — is working together here.

Key Takeaways

CSV is the standard format for structured text data, readable in a program or in a spreadsheet application, and Python's csv module — especially csv.DictReader and csv.DictWriter — handles it more robustly than manual comma splitting. The read-process-write pattern (read the source, transform it in memory, write the result) is the foundation of real data pipelines. With file handling complete, your programs can now genuinely persist information between runs. The next unit introduces exception handling — the technique that makes your programs resilient to the inevitable errors that arise from bad input, missing files, and the unpredictable real world.