The with Statement, Appending & Checking File Existence

AiCE Software Engineering Foundations — Python Programming (Intermediate)

Manually opening and closing files works, but it's easy to forget a .close() call, especially once error handling enters the picture. This lesson introduces the professional alternative, the with statement, then covers appending to files without destroying what's already there, and safely checking whether a file exists before you try to open it.

1. The with Statement — The Professional Way

The with statement is Python's recommended way to handle files. It automatically closes the file when the indented block finishes — even if an error occurs inside the block. You never have to call .close() manually.

```
# The with statement handles closing automatically
with open("students.txt", "r") as file:
    content = file.read()
# File is already closed here — safely and automatically
print(content)
```

The syntax is with open(filename, mode) as variable_name:. Everything indented inside the with block has access to the file. The moment the block ends — whether normally or because of an error — Python closes the file.

> **The Standard From Here On**
> *From this point forward, every file operation in this program uses the with statement. It is not just more convenient — it is safer and more professional, and it's what you'll see in virtually all real Python code.*

Reading With with — Line by Line

```
# Reading a file line by line with the with statement
with open("students.txt", "r") as file:
```

```
    for line in file:    # Iterate directly over the file
object
        name = line.strip()
        if name:    # Skip empty lines
            print(f"Student: {name}")
```

Iterating directly over the file object (for line in file:) is the most memory-efficient way to read a large file — it reads one line at a time rather than loading the entire file into memory at once. This is the correct approach for files that could be very large.

Writing With with

```
new_students = ["Zainab Musa", "Diallo Bah", "Aisha Kamara"]

with open("new_students.txt", "w") as file:
    for student in new_students:
        file.write(student + "\n")

# Confirm the content
with open("new_students.txt", "r") as file:
    print(file.read())
```

2. Appending to a File

Append mode ("a") adds content to the end of an existing file without disturbing what's already there. If the file doesn't exist, it's created. This is the correct mode when you want to accumulate records over time — like a log file or a growing student registry.

```
# First run: write the initial records
with open("registry.txt", "w") as file:
    file.write("Amara Diallo\n")
    file.write("Kofi Mensah\n")

# Second addition: append without overwriting
with open("registry.txt", "a") as file:
    file.write("Ngozi Adeyemi\n")

# Read and display the complete registry
```

```
with open("registry.txt", "r") as file:
    for line in file:
        print(f"  {line.strip()}")
```

All three names appear despite being written across two separate open() calls. Append mode preserved the existing content each time. This is how log files, audit trails, and growing registries are built — you open in append mode, add the new record, and close. The existing data is never at risk.

> **Write vs. Append — The Critical Difference**
> *open('file.txt', 'w') destroys existing content before writing. open('file.txt', 'a') preserves existing content and adds to the end. Mixing these up is one of the most painful file handling mistakes — you can accidentally erase an entire dataset. Before writing to any file, ask yourself: do I want to replace everything, or add to what's there?*

3. Checking If a File Exists

Attempting to open a non-existent file in read mode raises a FileNotFoundError and crashes your program. Before reading any file, check that it exists using the os module from the previous unit:

```
import os

def read_file_safely(filename):
    """Read a file only if it exists. Return content or an
error message."""
    if not os.path.exists(filename):
        return None, f"File not found: {filename}"
    with open(filename, "r") as file:
        content = file.read()
    return content, None

# Test with existing and non-existing files
content, error = read_file_safely("students.txt")
if error:
    print(f"Error: {error}")
```

```
else:
    print("File content:")
    print(content)

content, error = read_file_safely("missing_file.txt")
if error:
    print(f"Error: {error}")
```

The first call succeeds because students.txt exists. The second call returns an error message gracefully instead of crashing. This pattern — check before opening — is the safe, professional approach to file reading. The next unit introduces exception handling, which provides an even more robust solution for the same problem.

Key Takeaways

The with statement closes a file automatically once its block ends, even if an error occurs inside it — and from here on, it's the only way files should be opened in this program. Append mode ("a") adds to a file without destroying what's there, while write mode ("w") always overwrites — confusing the two is one of the most damaging file handling mistakes you can make. Checking os.path.exists() before opening a file in read mode prevents a FileNotFoundError from crashing your program outright. The next lesson covers working with structured data in files — CSV format, both parsed manually and with Python's built-in csv module — and the read-process-write pattern that powers real data pipelines.