ZyBooks Chapter 12: Files in Python

Jeremy Evert

October 20, 2025

Contents

Chapter 1

ZyBooks Chapter 12 Overview

This chapter explores Python file operations, including reading, writing, binary data, and file system interactions.

Chapter 2

12.1 Reading Files

Overview

In this section, students learn how to read from files using Python's built-in `open()` function. Files allow programs to save data permanently and retrieve it later—a sort of long-term memory for your code. Instead of typing data every time the program runs, we can read it directly from disk like civilized programmers.

Learning Goals:

- Understand how to open, read, and close text files safely.
- Differentiate between `read()`, `readline()`, and `readlines()`.
- Process and analyze file data (for example, computing averages).
- Use best practices that prevent data loss and mysterious crashes.

—

2.1 Reading from a File

The simplest way to read data from a file is to use `open()` together with `read()`. This reads the entire file at once into a single string.

```
# Example 1: Reading the entire contents of a file

from pathlib import Path

file_path = Path("journal.txt")

if file_path.exists():
    with open(file_path, "r", encoding="utf-8") as journal:
        contents = journal.read()
        print("[Info] Successfully opened journal.txt!\n")
        print(contents)
else:
    print("[Warning] File 'journal.txt' not found.")
```

Listing 2.1: Reading an entire file safely.

Key points:

- Always use `with open(...)` so Python closes the file automatically.
- Always specify `encoding="utf-8"` to avoid surprises on Windows.
- Check that the file exists before trying to read it—it's less embarrassing that way.

Sample File: `journal.txt`

```
Dear Journal,

Today I wrestled with a wild bug named SyntaxError.
After three print statements and one sigh, I prevailed.

Moral: Always close your parentheses before closing your laptop.
```

—

2.2 A More Complete Example

Let's take it up a notch: we'll add clear print messages, read the contents, and even perform a quick word analysis—because data deserves compliments too.

```python
# Example 2: Read a file and analyze it

from pathlib import Path

path = Path("myfile.txt")

if path.exists():
    with open(path, "r", encoding="utf-8") as f:
        print("[Info] Opening myfile.txt...")
        contents = f.read()
        print("[Success] File read successfully!\n")

    print("--- File Contents ---")
    print(contents)
    print("----------------------\n")

    words = contents.split()
    print(f"[Stats] Word count: {len(words)}")
    print(f"[Stats] Longest word: {max(words, key=len)}")
else:
    print("[Warning] File 'myfile.txt' not found.")
```

Listing 2.2: Reading and analyzing file contents.

Sample File: `myfile.txt`

```
Python is elegant.
It reads like poetry.
Sometimes, the bugs are haikus.
```

—

2.3 Reading Line by Line

The `readlines()` method reads a file into a list, where each element is one line of text. This makes it easy to loop through lines individually.

```
# Example 3: Read lines from a file

from pathlib import Path

file_path = Path("readme.txt")

if file_path.exists():
    with open(file_path, "r", encoding="utf-8") as f:
        lines = f.readlines()

    print(f"[Info] Found {len(lines)} lines in readme.txt.\n")

    for i, line in enumerate(lines, start=1):
        print(f"Line {i:>2}: {line.strip()}")

    all_words = " ".join(lines).split()
    print(f"\n[Result] Longest word: {max(all_words, key=len)}")
else:
    print("[Warning] File 'readme.txt' not found.")
```

Listing 2.3: Reading all lines into a list.

Sample File: `readme.txt`

```
Welcome to the File Reading Zone.
Line 1: Preparation.
Line 2: Curiosity.
Line 3: Revelation.
Line 4: Triumph.
```

—

2.4 Processing Data from a File

Files often hold numbers, and it's common to process them to compute things like sums or averages. Here's a calm, methodical way to do that without blowing up your CPU.

```
# Example 4: Calculate the average of numbers stored in a file

from pathlib import Path

data_file = Path("mydata.txt")

if data_file.exists():
    with open(data_file, "r", encoding="utf-8") as f:
        numbers = [int(line.strip()) for line in f if line.strip().isdigit()]

    average = sum(numbers) / len(numbers)
    print(f"[Result] Average value: {average:.2f}")
```

```
else:
    print("[Warning] File 'mydata.txt' not found.")
```

Listing 2.4: Computing an average from file data.

Sample File: `mydata.txt`

```
10
15
25
20
30
```

—

2.5 Iterating Directly Over a File Object

For very large files, it's better to read one line at a time instead of loading the whole file. This approach uses minimal memory and maximum patience.

```
# Example 5: The efficient way to read a file

with open("myfile.txt", "r", encoding="utf-8") as f:
    for line_number, line in enumerate(f, start=1):
        print(f"Line {line_number}: {line.strip()}")
```

Listing 2.5: Memory-efficient iteration through a file.

This method works beautifully on files of any size, even the ones so large you can hear your hard drive whisper, "Are you sure about this?"

—

2.6 Practice Exercise

Challenge: Write a program that asks the user for a filename, reads its contents, and prints them in uppercase. Bonus points if you make it sound enthusiastic.

```
# Example 6: Read and transform a file

filename = input("Enter filename to shoutify: ")

try:
    with open(filename, "r", encoding="utf-8") as f:
        contents = f.read()
    print("\n[Result] SHOUTING MODE ACTIVATED:\n")
    print(contents.upper())
except FileNotFoundError:
    print("[Warning] File not found. Please try again.")
```

Listing 2.6: Challenge Activity: Transform file contents.

—

2.7 Explore More

For more detailed tutorials and examples:

- Python Docs: Reading and Writing Files
- Real Python: Working with Files
- W3Schools: File Handling in Python

—

Summary

- Use `with open()` for clean and automatic file handling.
- `read()`, `readline()`, and `readlines()` each serve different use cases.
- Always specify `encoding="utf-8"`.
- Check file existence before opening it.
- Never fear the file system—treat it like a friend that occasionally misplaces your stuff.

Chapter 3

12.2 Writing Files

Overview

Reading files is like visiting the library—you take in information. Writing files, on the other hand, is like *becoming* the author. In this section, students learn how to create, modify, and save text files safely.

Python's built-in `open()` function provides multiple "modes" for writing, appending, and creating files. You'll also learn why some write operations fail, how to avoid data loss, and how to make your programs polite authors who close their notebooks when finished.

Learning goals:

- Understand how file modes (`w`, `a`, `x`, etc.) affect writing behavior.
- Learn to handle common write errors gracefully.
- Explore buffering and flushing to ensure data is saved properly.
- Appreciate the importance of file safety and reproducibility.

—

3.1 Basic File Writing

The `write()` method records text into a file. Opening a file in mode `"w"` will create it if missing or overwrite it if it already exists. Think of it as starting a new diary page—sometimes that's what you want, sometimes it's heartbreak.

```
def write_basic_example():
    """Write two lines to a new file."""
    with open("myfile.txt", "w", encoding="utf-8") as f:
        f.write("This is a brand-new file.\n")
        f.write("Second line: this one overwrites any previous content.\n")
    print("[Success] File written successfully!")

write_basic_example()
```

Listing 3.1: Example 1: Writing text to a file.

Sample File Output: `myfile.txt`

```
This is a brand-new file.
Second line: this one overwrites any previous content.
```

—

3.2 Why Some Writes Fail

You can only write strings to text files. Attempting to write a number directly will cause Python to raise a dramatic `TypeError`.

```python
def demonstrate_wrong_write():
    """Show why writing numbers directly causes a TypeError."""
    try:
        with open("wrong_write.txt", "w", encoding="utf-8") as f:
            # This will fail: write() only accepts strings.
            f.write(3.14159)
    except TypeError as e:
        print("[Error]", e)
        print("[Hint] Convert numbers to strings using str() or f-strings.")

    # Correct version
    with open("right_write.txt", "w", encoding="utf-8") as f:
        num1, num2 = 5, 7.5
        f.write(f"{num1} + {num2} = {num1 + num2}\n")
    print("[Fixed] Math written successfully!")

demonstrate_wrong_write()
```

Listing 3.2: Example 2: Handling write errors with style.

—

3.3 File Modes

File modes are like the different moods of a writer—each one changes the tone of what happens next.

Mode	Description	Read?	Write?	Overwrite?
`r`	Read only	Yes	No	No
`w`	Write (overwrite)	No	Yes	Yes
`a`	Append to end	No	Yes	No
`r+`	Read and write (must exist)	Yes	Yes	No
`w+`	Read and write (truncates)	Yes	Yes	Yes
`a+`	Read and append	Yes	Yes	No
`x`	Create new file (error if exists)	No	Yes	N/A

```python
from pathlib import Path
import io

def show_file(path):
    p = Path(path)
    print(f"\n[{p.name}] contents:")
    print(p.read_text(encoding="utf-8") if p.exists() else "<missing>")
```

```
def demonstrate_modes():
    path = "modes_demo.txt"
    Path(path).write_text("START\n", encoding="utf-8")

    # Wrong: open in read mode, try to write
    try:
        with open(path, "r", encoding="utf-8") as f:
            f.write("APPEND\n")
    except io.UnsupportedOperation as e:
        print("[Error]", e)
        print("[Hint] 'r' is read-only; use 'a' for append.")

    # Correct: append mode
    with open(path, "a", encoding="utf-8") as f:
        f.write("APPEND\n")

    show_file(path)

demonstrate_modes()
```

Listing 3.3: Example 3: Trying wrong modes, then fixing them.

—

3.4 Append Mode: The Diary Approach

The a mode appends to an existing file—great for logs, journals, and confessions you don't want erased.

```
from datetime import datetime

def append_to_log(entry, filename="daily_log.txt"):
    """Append timestamped entries to a log file."""
    with open(filename, "a", encoding="utf-8") as f:
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        f.write(f"[{timestamp}] {entry}\n")
    print("[Info] Log entry added.")

# Let's test a few entries
append_to_log("Started Chapter 12 examples.")
append_to_log("Tried append mode. It worked!")
append_to_log("Feeling confident about file handling.")
```

Listing 3.4: Example 4: Appending to a file repeatedly.

—

3.5 Buffered Output and Flushing

When you write to a file, Python first stores the data in memory before writing it to disk. This is called *buffering*—it makes writing faster, but also means data might not appear immediately.

```
import os, time
from pathlib import Path
```

```
def buffering_demo(path="buffer_demo.txt"):
    p = Path(path)
    if p.exists():
        p.unlink()

    f = open(path, "w", encoding="utf-8")
    f.write("Write me (still in memory)...")
    print("[Info] File opened and data written (but not yet saved).")
    time.sleep(1)
    f.flush()          # Push data from Python to OS
    os.fsync(f.fileno())  # Ensure the OS writes it to disk
    print("[Success] Data flushed to disk.")
    f.close()

buffering_demo()
```

Listing 3.5: Example 5: Forcing a buffer flush.

—

3.6 Safe File Creation

The "x" mode is the “no overwrite allowed” option. It’s perfect for protecting students’ lab reports, thesis drafts, or personal manifestos.

```
def create_once(filename="create_once.txt"):
    try:
        with open(filename, "x", encoding="utf-8") as f:
            f.write("This file will never be overwritten.\n")
        print(f"[Created] New file: {filename}")
    except FileExistsError:
        print(f"[Skipped] '{filename}' already exists; not overwritten.")

create_once()
create_once()
```

Listing 3.6: Example 6: Using ’x’ mode to prevent overwrite.

—

3.7 Fun Activity: The Compliment Machine

Let’s make something a bit sillier—a program that takes user input and writes each compliment to a file so you can re-read your greatness later.

```
def compliment_machine():
    filename = "compliments.txt"
    print("Welcome to the Compliment Machine!")
    print("Type compliments to save them; press Enter on an empty line to quit.\n")

    with open(filename, "a", encoding="utf-8") as f:
        while True:
            compliment = input("Say something nice: ")
            if not compliment.strip():
                break
```

```
            f.write(compliment + "\n")
            print("[Saved] Compliment added!\n")

    print(f"All compliments saved to {filename}.")

compliment_machine()
```

Listing 3.7: Example 7: The Compliment Machine.

—

Summary

- Mode "w" overwrites, "a" appends, and "x" creates new.
- `write()` only accepts strings; use `str()` or f-strings for other data.
- Buffered writes may not appear immediately—use `flush()` or close the file.
- The safest way to write: `with open(...)` ensures automatic cleanup.
- Writing to files is how programs tell stories. Make yours a good one.

Chapter 4

12.3 Interacting with File Systems

Overview

Welcome to the neighborhood of directories, drives, and disk blocks. Your Python code is about to make friends with the operating system. This chapter teaches how to explore, navigate, and manipulate files and folders safely.

When your program touches the file system, several layers cooperate:

1. **Python layer:** you call functions like `open()`, `os.stat()`, or `pathlib.Path()`.
2. **C layer:** CPython translates those into system calls such as `open`, `read`, and `write`.
3. **Operating system kernel:** checks permissions, updates metadata, and interacts with file-system drivers.
4. **Hardware:** disks and controllers actually move bytes.

The key idea: Python provides a friendly façade, but the OS decides what is legal, safe, and durable.

Docs to Bookmark:

- `os` module
- `pathlib` module
- `shutil`: copy, move, remove
- Reading & Writing Files (Python Docs)

—

4.1 Portable Paths: Stop Fighting the Slashes

Hard-coding \ or `/` works until it doesn't. Use `os.path.join()` or `pathlib.Path()` so your code runs everywhere.

```python
import os
from pathlib import Path

def build_paths():
    # Wrong: brittle, Windows-only
    p_bad = "logs\\2025\\01\\log.txt"
    print("Brittle:", p_bad)

    # Better: OS-appropriate separator
    p_good = os.path.join("logs", "2025", "01", "log.txt")
    print("Portable:", p_good)

    # Best: Path objects are operator-friendly
    p = Path("logs") / "2025" / "01" / "log.txt"
    print("Pathlib:", p)

build_paths()
```

Listing 4.1: Building file paths the right way.

Tip: In Windows paths, a single backslash begins an escape. Use raw strings like `r"C:\Users\me\notes.txt"` to keep Python calm.

4.2 Checking Existence and Size

```python
import os
from pathlib import Path

def existence_and_type(path_str):
    print(f"\nTesting: {path_str}")
    print("-- os.path results --")
    print(" exists:", os.path.exists(path_str))
    print(" isfile:", os.path.isfile(path_str))
    print(" isdir:", os.path.isdir(path_str))

    print("-- pathlib results --")
    p = Path(path_str)
    print(" exists:", p.exists())
    print(" is_file:", p.is_file())
    print(" is_dir:", p.is_dir())
    if p.exists():
        print(" size:", p.stat().st_size, "bytes")

existence_and_type("myfile.txt")
existence_and_type("logs")
```

Listing 4.2: Check if something exists, and what kind of thing it is.

4.3 Reading File Metadata

```python
import datetime
from pathlib import Path

def show_stat(path):
    p = Path(path)
    if not p.exists():
        print(f"{path!r} does not exist.")
        return
    st = p.stat()
    print("\n-- File metadata --")
    print("size:", st.st_size, "bytes")
    print("modified:", datetime.datetime.fromtimestamp(st.st_mtime))
    print("created :", datetime.datetime.fromtimestamp(st.st_ctime))

show_stat("myfile.txt")
```

Listing 4.3: Inspect metadata and show friendly timestamps.

Note: On Windows, `st_ctime` means creation time. On Linux and macOS it means "metadata change time." Same name, different personality.

—

4.4 Walking a Directory Tree

```python
import os

def list_txt_files(root="."):
    print(f"\nSearching {root!r} for text files:")
    for dirpath, subdirs, files in os.walk(root):
        for name in files:
            if name.lower().endswith(".txt"):
                print(" ", os.path.join(dirpath, name))

list_txt_files("logs")
```

Listing 4.4: List .txt files using os.walk.

This is great for projects with nested folders or for grading dozens of student files.

—

4.5 Creating, Copying, and Deleting

```python
import os, shutil
from pathlib import Path

def safe_demo():
    d = Path("sandbox")
    d.mkdir(exist_ok=True)
    (d / "demo.txt").write_text("Hello, filesystem!\n")

    # Copy and rename
    shutil.copy2(d / "demo.txt", d / "copy.txt")
    os.replace(d / "copy.txt", d / "moved.txt")
```

```
    # Delete safely
    (d / "moved.txt").unlink(missing_ok=True)
    print("Sandbox contents:", list(d.iterdir()))

safe_demo()
```

Listing 4.5: Safe create, copy, rename, delete.

Atomicity tip: `os.replace()` is atomic on the same drive— either the old file stays or the new one appears, never half a file.

—

4.6 Split, Join, and Inspect Paths

```
import os

def split_examples(p):
    head, tail = os.path.split(p)
    root, ext = os.path.splitext(p)
    print("\nSplit:", p)
    print(" head:", head)
    print(" tail:", tail)
    print(" root:", root)
    print(" ext :", ext)

split_examples(os.path.join("C:\\", "Users", "Demo", "batsuit.jpg"))
```

Listing 4.6: Split and reassemble file paths.

If you ever need to rename every `.jpg` to `.png`, `os.path.splitext()` is your sidekick.

—

4.7 Challenge: Counting Files

```
import os

def count_ext(root, ext=".txt"):
    total = 0
    for dirpath, _, files in os.walk(root):
        for name in files:
            if name.lower().endswith(ext):
                total += 1
    return total

print("Number of .txt files under logs:", count_ext("logs"))
```

Listing 4.7: Count all files with a given extension.

—

4.8 Atomic Writes (for the Perfectionists)

```python
import os, tempfile
from pathlib import Path

def atomic_write_text(path, text):
    target = Path(path)
    target.parent.mkdir(parents=True, exist_ok=True)
    with tempfile.NamedTemporaryFile("w", encoding="utf-8",
                                     dir=target.parent, delete=False) as tmp:
        tmp.write(text)
        tmp.flush()
        os.fsync(tmp.fileno())
        tmp_name = tmp.name
    os.replace(tmp_name, target)
    print("Atomically wrote:", target)

atomic_write_text("sandbox/report.txt", "Final, correct, no-half-file version.\n")
```

Listing 4.8: Write safely using a temporary file.

—

4.9 Windows-Specific Quirks

- Old APIs limited paths to 260 characters; modern Python can exceed this.
- Windows is case-insensitive but case-preserving—`File.txt` and `file.txt` refer to the same file.
- Text mode converts newlines automatically; use binary mode for raw bytes.

—

4.10 Practice Prompts

1. Create a directory structure `logs/YYYY/MM/DD/` and write a file for today's date.
2. Walk a directory tree and print the three largest files.
3. Implement `safe_replace(path, text)` that uses a temp file then `os.replace()`.
4. On Windows, demonstrate the difference between a raw string and an escaped string path.

—

Summary

- Use `pathlib` for readability and safety.
- Always handle `FileNotFoundError`, `PermissionError`, and `IsADirectoryError`.
- Remember: flushing and atomic replace are your insurance policies.

- The file system is a conversation—speak politely, close files, and check before you delete.

Chapter 5

12.4 Binary Data

Why Binary Files Feel Like Secret Code

Most files you touch daily (text, CSV, Python scripts) are human-readable. Binary files, on the other hand, are the silent operators — full of raw bytes that mean nothing until decoded correctly. Think of them as data that only machines (or very patient humans) can love.

Examples: images, videos, PDFs, ZIP archives. Open them in Notepad and you'll see digital hieroglyphs.

—

5.1 Bytes in Python

A `bytes` object represents a sequence of byte values (0–255). They are immutable, much like strings.

```python
# From a text string
b1 = bytes("A text string", "ascii")

# From a size (filled with zeros)
b2 = bytes(10)

# From numeric values
b3 = bytes([12, 15, 20])

print(b1)
print(b2)
print(b3)
```

Listing 5.1: Creating bytes in different ways.

—

5.2 Bytes Literals and Escapes

Prefix a string with `b` to make a bytes literal. You can also use hexadecimal escapes (`\xHH`) to show specific values.

```
print(b"123456789" == b"\x31\x32\x33\x34\x35\x36\x37\x38\x39")
```

Listing 5.2: Byte literals and hex escapes.

Output:

```
True
```

Yes — the string "123456789" and the exact same bytes in hex are identical. One just wears a fancier coat.

—

5.3 Reading and Writing Binary Files

When reading or writing binary data, use mode 'rb' or 'wb'. Binary mode skips newline conversions and encodings.

```
# Write binary bytes
with open("data.bin", "wb") as f:
    f.write(b"\x01\x02\x03\x04")

# Read them back
with open("data.bin", "rb") as f:
    contents = f.read()
    print(contents)
```

Listing 5.3: Opening files in binary mode.

Output:

```
b'\x01\x02\x03\x04'
```

—

5.4 Peeking Inside Binary Files

You can read a few bytes to see what a binary file looks like. This works well for identifying file headers.

```
with open("ball.bmp", "rb") as f:
    header = f.read(32)

print("First 32 bytes of ball.bmp:")
print(header)
```

Listing 5.4: Reading the first bytes of a file.

Example output:

```
b'BM\xf6\x00\x00\x00\x00\x00\x00\x00\x36\x04\x00\x00...'
```

The first two bytes BM mark this as a BMP file — the ancient but still cheerful "bitmap."

—

5.5 Editing Binary Data (Carefully!)

Binary files are like Jenga towers — one wrong byte and the whole thing collapses. Still, here's a playful example using the `struct` module to peek at and modify data.

```
import struct, pathlib

# Read entire file into memory
data = pathlib.Path("ball.bmp").read_bytes()

# Bytes 10 through 13 store the pixel data offset
pixel_data_offset = struct.unpack("<I", data[10:14])[0]
print("Pixel data starts at:", pixel_data_offset)

# Let's change the first 3000 pixels to a repeating red-green-yellow pattern
pattern = (b"\xff\x00\x00" + b"\x00\xff\x00" + b"\xff\xff\x00") * 1000
new_data = data[:pixel_data_offset] + pattern + data[pixel_data_offset + len(pattern)
    :]

pathlib.Path("new_ball.bmp").write_bytes(new_data)
print("Created modified image: new_ball.bmp")
```

Listing 5.5: Altering part of a BMP image.

Note: Don't use this on family photos unless you like abstract art.

—

5.6 Packing and Unpacking with struct

`struct.pack()` and `struct.unpack()` convert Python data to binary and back.

```
import struct

# Pack two short integers (big-endian)
data = struct.pack(">hh", 5, 256)
print("Packed:", data)

# Unpack them again
numbers = struct.unpack(">hh", data)
print("Unpacked:", numbers)
```

Listing 5.6: Packing and unpacking values.

Output:

```
Packed: b'\x00\x05\x01\x00'
Unpacked: (5, 256)
```

This trick is essential for reading file headers, network protocols, or microcontroller data.

—

5.7 Binary vs. Text Performance

Binary writes skip encoding overhead, often making them faster for large data dumps.

```
import time, os

N = 10_000_000
text_data = "A" * N
binary_data = b"A" * N

start = time.time()
with open("text_test.txt", "w") as f:
    f.write(text_data)
text_time = time.time() - start

start = time.time()
with open("binary_test.bin", "wb") as f:
    f.write(binary_data)
binary_time = time.time() - start

print(f"Text write:   {text_time:.4f}s")
print(f"Binary write: {binary_time:.4f}s")
print(f"Binary was {text_time / binary_time:.2f}x faster!")
```

Listing 5.7: Comparing write speeds.

Your results may vary, but binary writing usually wins the sprint.

—

Summary

- Binary files store raw bytes — text encoding is not involved.
- Use `rb` and `wb` for binary reads and writes.
- The `struct` module is your translator between Python types and byte layouts.
- Always test on copies — once a byte is gone, no "undo" button will save it.

Chapter 6

12.5 Command-Line Arguments and Files

Overview

Command-line arguments let your program take input file names or other parameters directly from the shell. It's how you tell your code: *"Work with this file today, not the one you hard-coded last night."*

Docs to Bookmark:

- sys module
- os.path module

—

6.1 Example: Using an Input File from the Command Line

```
import sys
import os

if len(sys.argv) != 2:
    print(f"Usage: python {sys.argv[0]} input_file")
    sys.exit(1)  # 1 = error

file_name = sys.argv[1]
print(f"Opening file {file_name}...")

if not os.path.exists(file_name):
    print("File does not exist.")
    sys.exit(1)

with open(file_name, "r", encoding="utf-8") as f:
    print("Reading two integers.")
    num1 = int(f.readline())
    num2 = int(f.readline())
```

```python
print(f"Closing file {file_name}.")
print(f"\nnum1: {num1}")
print(f"num2: {num2}")
print(f"num1 + num2 = {num1 + num2}")
```

Listing 6.1: listing_5_1_command_line_args.py

Run it like this:

```
> python listing_5_1_command_line_args.py myfile1.txt
> python listing_5_1_command_line_args.py myfile2.txt
> python listing_5_1_command_line_args.py missing.txt
```

Output examples:

```
Opening file myfile1.txt...
Reading two integers.
Closing file myfile1.txt.

num1: 5
num2: 10
num1 + num2 = 15
```

6.2 Files for Testing

myfile1.txt

```
5
10
```

myfile2.txt

```
-34
7
```

6.3 Extending the Idea

- Add a second argument for an *output file*.
- Add try/except to handle non-numeric data.
- Use `argparse` for fancier options later in the course.

Practice Prompts

1. Modify the program so it accepts both an input and output filename: `python myscript.py infile.txt outfile.txt`
2. For a run as `python scriptname data.txt`, what is `sys.argv[1]`?
3. What happens if you forget the argument? Why does Python show a usage message?

Chapter 7

12.6 The `with` Statement

Overview

A **with** statement is Python's built-in way of managing resources — such as files — safely and elegantly. When used with `open()`, it guarantees that the file is closed automatically when the block of code completes, even if an exception occurs inside the block.

Think of it as a friendly librarian: you borrow a book (open a file), do your reading, and no matter what happens — whether you finish or drop your coffee — the librarian takes the book back and shelves it neatly.

Docs to Bookmark:

- The with statement (Python Docs)
- File objects in Python

—

7.1 Basic Example: Safe File Reading

```
print("Opening myfile.txt")
with open("myfile.txt", "r", encoding="utf-8") as f:
    contents = f.read()
    print(contents)
print("File closed automatically after this block!")
```

Listing 7.1: Opening and reading a file safely.

The **with** statement ensures that once you exit the block, `f.close()` is called automatically. Even if an error occurs inside the block, Python still closes the file properly.

—

7.2 Why Not Just Call `close()` Yourself?

```
f = open("oops.txt", "w", encoding="utf-8")
f.write("This file might stay open forever...")
# Oops! We forgot f.close()
```

Listing 7.2: Manual open and close – easy to forget.

If you forget to close a file, it can remain locked or unflushed in memory — causing confusion, corrupted files, or grumpy system administrators.

```python
with open("oops.txt", "w", encoding="utf-8") as f:
    f.write("This version closes itself. Much better!")
```

Listing 7.3: Fixed using a with statement.

—

7.3 Reading and Writing Together

```python
print("Opening numbers.txt for reading and writing...")
with open("numbers.txt", "r+", encoding="utf-8") as f:
    num1 = int(f.readline())
    num2 = int(f.readline())
    product = num1 * num2
    f.write(f"\nProduct: {product}\n")
print("Closed numbers.txt automatically.")
```

Listing 7.4: Using the same file object for reading and writing.

—

7.4 Cautionary Tale: The File That Refused to Close

```python
try:
    f = open("chaos.txt", "w", encoding="utf-8")
    f.write("Everything is fine so far...")
    raise RuntimeError("  Oops! Chaos strikes!")
    f.close()
except Exception as e:
    print("Caught error:", e)
    print("Did we close the file? Nope!")

# Now the safe way:
try:
    with open("calm.txt", "w", encoding="utf-8") as f:
        f.write("Tranquility restored. File will close no matter what.")
        raise RuntimeError("  A wild exception appears!")
except Exception as e:
    print("Caught safely:", e)
```

Listing 7.5: Why with is safer.

Even though both blocks raise errors, only the second one guarantees that the file is properly closed.

—

7.5 Challenge Example: Writing Logs

```python
from datetime import datetime

def log_event(message):
    """Append an event to a log file with timestamp."""
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    with open("log.txt", "a", encoding="utf-8") as log:
        log.write(f"[{timestamp}] {message}\n")

log_event("Program started successfully.")
log_event("User pressed the red button.")
```

Listing 7.6: Appending a timestamped log entry safely.

—

7.6 Quiz Questions

1. When does Python automatically close a file opened with `with`? **Answer:** When the indented block ends, even if an error occurs.
2. What is a `context manager`? **Answer:** An object that sets up and tears down resources automatically.
3. What happens if you forget `close()`? **Answer:** The file may remain locked or data may not be saved.

—

Summary

- Use `with open(...)` to ensure automatic closure of files.
- It's cleaner, safer, and preferred in all modern Python code.
- Works great for other resources too (network sockets, database connections, etc.).

  Moral of the story: If your program opens files without `with`, it's like leaving the refrigerator door open — it still "works," but you're wasting power and asking for trouble.

Notes

This companion book was designed to accompany the zyBooks interactive textbook, Chapter 12: *Files in Python*. Each section includes clear examples that are ready to copy and paste directly from this PDF into your favorite code editor. The examples also include sample data files, demonstrations of modern best practices, and a touch of humor to keep learning lively.