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 later retrieve it. Instead of entering data manually each time a program runs, we can read information directly from a file.

Learning goals:

- Understand how to open, read, and close text files in Python.
- Explore the difference between `read()`, `readline()`, and `readlines()`.
- Learn to process data from files (e.g., computing averages).

—

2.1 Reading from a File

The most basic way to read data from a file is with `open()` and `read()`.

Listing 2.1: Reading text from a file.

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

# Open the file in read mode
myjournal = open("journal.txt")

# Read the entire file into a single string
contents = myjournal.read()

# Display what was read
print(contents)

# Close the file after use
myjournal.close()
```

Key points:

- `open("filename")` creates a file object.

- `read()` reads all text at once and returns it as a string.
- Always close the file using `close()` when done.

—

2.2 A More Complete Example

This version adds print statements and clarifies program flow.

Listing 2.2: Creating a file object and reading text.

```
print("Opening file myfile.txt.")
f = open("myfile.txt")  # create file object

print("Reading file myfile.txt.")
contents = f.read()     # read text into a string

print("Closing file myfile.txt.")
f.close()               # close the file

print("\nContents of myfile.txt:")
print(contents)
```

Tip: The file must be in the same directory as your Python script unless you specify a full path (e.g., `C:\Users\evertj\myfile.txt`).

—

2.3 Reading Line by Line

The `readlines()` method reads each line into a list of strings.

Listing 2.3: Reading all lines into a list.

```
# Example 2: Read lines from a file
my_file = open("readme.txt")
lines = my_file.readlines()

# Print the second line (remember, Python starts counting at 0)
print(lines[1])

my_file.close()
```

Note: Each element of `lines` includes the newline character `"\n"`.

—

2.4 Processing Data from a File

Programs often read data from files to compute a result, such as an average.

Listing 2.4: Calculating the average value of integers stored in a file.

```
# Example 3: Calculating an average from a file

print("Reading in data...")
```

```
f = open("mydata.txt")
lines = f.readlines()
f.close()

# Process data
print("\nCalculating average...")
total = 0
for ln in lines:
    total += int(ln)

avg = total / len(lines)
print(f"Average value: {avg}")
```

This example demonstrates:

- How to iterate through file lines.
- Converting strings to integers using `int()`.
- Computing an average from numeric data.

—

2.5 Iterating Directly Over a File Object

Python lets you loop through a file directly, one line at a time.

Listing 2.5: Iterating over the lines of a file.

```
"""Echo the contents of a file."""
f = open("myfile.txt")

for line in f:
    print(line, end="")  # end="" avoids double newlines

f.close()
```

This approach is memory-efficient and ideal for large files.

—

2.6 Practice Exercise

Challenge: Create a Python program that reads a filename from user input, opens that file, and prints its contents in uppercase.

Listing 2.6: Challenge Activity: Read and modify file contents.

```
# Example 4: Read and transform file content
filename = input("Enter filename: ")

with open(filename) as f:        # 'with' auto-closes the file
    contents = f.read()

print(contents.upper())
```

—

2.7 Explore More

For additional reading and examples:

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

—

Summary

- Use `open()` to access a file.
- `read()`, `readline()`, and `readlines()` offer flexibility.
- Always close files, or use the `with` statement.
- Practice reading, processing, and displaying file data.

Chapter 3

12.2 Writing Files

Overview

Programs write to files to store data permanently. The `file.write()` method writes a string argument to a file. This section teaches how to open files for writing, the difference between modes, and why errors occur when you write the wrong data type.

Helpful documentation:

- Python Docs: Reading and Writing Files
- `open()` built-in function
- `io.TextIOBase.write()`
- `os.fsync()`

3.1 Writing to a File

Listing 3.1: Basic example: Writing text to a file.

```
def write_basic_example():
    """Write two lines to a new file."""
    with open("myfile.txt", "w", encoding="utf-8") as f:
        f.write("Example string.\n")
        f.write("test....\n")
    print("File written successfully!")

write_basic_example()
```

Key idea: Opening a file in mode `"w"` creates it if missing and overwrites it if it already exists.

—

3.2 Why Some Writes Fail

Listing 3.2: Example: Handling write errors gracefully.

```
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(10.0)
    except TypeError as e:
        print("Caught error:", e)
        print("Explanation: write() in text mode needs a string, not a
            number.")

    # Correct way: convert numbers to strings.
    with open("right_write.txt", "w", encoding="utf-8") as f:
        num1, num2 = 5, 7.5
        total = num1 + num2
        f.write(str(num1))
        f.write(" + ")
        f.write(str(num2))
        f.write(" = ")
        f.write(str(total))
        f.write("\n")
    print("Fixed version works correctly!")

demonstrate_wrong_write()
```

—

3.3 File Modes

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

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

```
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("Explanation: 'r' is read-only; writing is not allowed.")

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

demonstrate_modes()
```

—

3.4 Buffered Output

Python buffers output before writing to disk. This means data may not appear immediately in the file system until a newline, `flush()`, or `close()`.

Listing 3.4: Example: Forcing a buffer flush.

```
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 (no newline)")  # stays in memory
    print("Before flush: file may still be empty.")
    time.sleep(1)
    f.flush()            # Push data from Python to OS
    os.fsync(f.fileno())  # Ask OS to sync to disk
    f.close()
    print("After flush: file contents written.")

buffering_demo()
```

—

3.5 Quiz Demonstrations

Listing 3.5: Mini-quiz code with real behavior.

```
def quiz_behavior():
    print("\n1) f.write(10.0) produces error:")
    try:
        with open("q1.txt", "w", encoding="utf-8") as f:
            f.write(10.0)
    except TypeError as e:
        print("True:", e)

    print("\n2) write() does NOT always write immediately:")
    with open("q2.txt", "w", encoding="utf-8") as f:
```

```
        f.write("hi")  # buffered
        print("Immediate read:", open("q2.txt").read())
        f.flush()
    print("After flush: data is saved.")

    print("\n3) flush()/fsync() forces output to disk:")
    import os
    with open("q3.txt", "w", encoding="utf-8") as f:
        f.write("sync me")
        f.flush()
        os.fsync(f.fileno())
    print("True: Data synced to disk.")

quiz_behavior()
```

—

3.6 Safe File Creation

Listing 3.6: Use 'x' mode to avoid accidental overwrite.

```
def create_once(filename="create_once.txt"):
    try:
        with open(filename, "x", encoding="utf-8") as f:
            f.write("File created successfully.\n")
        print("Created new file:", filename)
    except FileExistsError:
        print("File already exists; not overwritten.")

create_once()
create_once()
```

—

Summary

- Mode `"w"` overwrites, `"a"` appends, `"x"` creates new.
- `write()` requires strings; convert numbers with `str()` or f-strings.
- Output may be buffered; use `flush()` or close the file.
- Use `with open(...)` to ensure the file closes automatically.

Chapter 4

12.3 Interacting With File Systems

Big Picture Mental Model

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

1. **Your Python code** calls functions like `open()`, `os.stat()`, `os.remove()`, `pathlib.Path(...)`.

2. **CPython implementation** translates those calls into C functions that use the operating system's native API. On Linux/macOS this is usually POSIX calls (open, read, write, stat, unlink). On Windows it goes through the Win32 layer (CreateFileW, ReadFile, WriteFile, GetFileInformationByHandle, DeleteFileW).

3. **The operating system kernel** checks permissions, updates metadata, and interacts with the file system driver. It also uses caches and scheduling to read/write blocks on storage devices.

4. **Hardware and firmware** (disk controller, SSD firmware, DMA) actually move bytes. **CPUs (Intel/AMD/ARM)** execute instructions, switch between user mode and kernel mode on system calls, and provide memory management and caching. The CPU does not know about "files" directly; it executes the OS code that does.

Takeaway: Python gives a friendly interface, but durability, permissions, and path rules come from the OS and file system.

Docs to Bookmark

- Python Tutorial: Reading and Writing Files – docs.python.org
- `os` module – docs.python.org
- `os.path` module – docs.python.org
- `pathlib` – docs.python.org
- `shutil` (copy, move) – docs.python.org
- File object methods (`flush`) – docs.python.org

4.1 Portable Paths: os.path.join and pathlib

Hard-coding backslashes (Windows) or slashes (Linux/macOS) makes code fragile. Use joiners.

Listing 4.1: Right vs. wrong for building file paths.

```
import os
from pathlib import Path

def build_paths():
    # WRONG on non-Windows and brittle even on Windows:
    p_bad = "logs\\2025\\01\\log.txt"   # backslashes are Windows-only
    print("Brittle:", p_bad)

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

    # RIGHT: Path objects are even nicer
    p = Path("logs") / "2025" / "01" / "log.txt"
    print("Pathlib:", str(p))

build_paths()
```

Windows note: inside Python string literals, a single backslash begins an escape (like `"\n"`). Use raw strings like `r"C:\users\me"` or double the backslashes.

4.2 Existence, File vs. Directory, and Getting Size

Listing 4.2: Check existence and type with both os.path and pathlib.

```
import os
from pathlib import Path

def existence_and_type(path_str: str):
    print("\n-- Using os.path --")
    print("exists:", os.path.exists(path_str))
    print("isfile:", os.path.isfile(path_str))
    print("isdir:", os.path.isdir(path_str))

    print("\n-- Using pathlib --")
    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("modes_demo.txt")
existence_and_type("logs")
```

4.3 Metadata: os.stat and datetime

Listing 4.3: Inspect file metadata and pretty-print timestamps.

```
import os, datetime
from pathlib import Path

def show_stat(path_str: str):
    p = Path(path_str)
    if not p.exists():
        print(f"{path_str!r} does not exist")
        return
    st = p.stat()  # same as os.stat(path_str)
    print("\n-- stat for", path_str, "--")
    print("size:", st.st_size, "bytes")
    print("mode (permission bits):", oct(st.st_mode))
    print("modified:", datetime.datetime.fromtimestamp(st.st_mtime))
    print("created (platform dependent):", datetime.datetime.
        fromtimestamp(st.st_ctime))

show_stat("modes_demo.txt")
```

Platform note: `st_ctime` is creation time on Windows, but on POSIX it is "metadata change" time.

4.4 Walking a Directory Tree

Listing 4.4: Walk with os.walk and filter by extension.

```
import os
from pathlib import Path

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

list_py_files("logs", ".txt")
```

`os.walk` yields a 3-tuple per directory. The heavy lifting (reading directory entries) is done by the OS; Python iterates and filters.

4.5 Creating, Renaming, Copying, Deleting

Listing 4.5: Safe create, rename, copy, and delete with error handling.

```
import shutil
from pathlib import Path

def safe_create_dir(path: str):
    Path(path).mkdir(parents=True, exist_ok=True)
    print("Ensured directory exists:", path)
```

```python
def safe_rename(src: str, dst: str):
    try:
        # os.replace is atomic when src and dst are on the same filesystem
        os.replace(src, dst)
        print(f"Renamed {src!r} -> {dst!r}")
    except FileNotFoundError:
        print("Cannot rename: source not found.")
    except PermissionError:
        print("Cannot rename: permission denied.")

def safe_copy(src: str, dst: str):
    try:
        shutil.copy2(src, dst)  # preserves timestamps and metadata where possible
        print(f"Copied {src!r} -> {dst!r}")
    except FileNotFoundError:
        print("Cannot copy: source not found.")
    except PermissionError:
        print("Cannot copy: permission denied.")

def safe_delete(path: str):
    try:
        Path(path).unlink()
        print("Deleted file:", path)
    except FileNotFoundError:
        print("Nothing to delete:", path)
    except IsADirectoryError:
        print("Path is a directory; use rmdir or shutil.rmtree.")
    except PermissionError:
        print("Cannot delete: permission denied.")

safe_create_dir("sandbox")
Path("sandbox/demo.txt").write_text("hello\n", encoding="utf-8")
safe_copy("sandbox/demo.txt", "sandbox/demo_copy.txt")
safe_rename("sandbox/demo_copy.txt", "sandbox/demo_moved.txt")
safe_delete("sandbox/demo_moved.txt")
```

Atomicity note: `os.replace` is designed to be atomic on the same filesystem volume. If you move across drives, use `shutil.move` which may copy then delete.

4.6 Portable File Path Building Activity

Listing 4.6: Demonstrate os.path.join results on different OSes.

```python
import os

def join_examples():
    a = os.path.join("subdir", "output.txt")
    b = os.path.join("sounds", "cars", "honk.mp3")
    print("Example join A:", a)
    print("Example join B:", b)
    print("Path separator on this OS:", os.path.sep)

join_examples()
```

4.7 Splitting Paths and Getting Extensions

Listing 4.7: Split with os.path.split and get extension with splitext.

```
import os

def split_examples(p: str):
    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"))
```

4.8 Challenge: Use os.walk to Count Specific Files

Listing 4.8: Count .txt files and handle permissions gracefully.

```
import os

def count_ext(root: str, ext: str = ".txt") -> int:
    total = 0
    for dirpath, subdirs, files in os.walk(root, onerror=None):
        for name in files:
            if name.lower().endswith(ext):
                total += 1
    return total

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

4.9 Safe and Durable Writes: Temp File + Atomic Replace

Listing 4.9: Avoid partial writes by writing to a temp file and replacing.

```
import os, tempfile
from pathlib import Path

def atomic_write_text(path: str, text: str):
    target = Path(path)
    target.parent.mkdir(parents=True, exist_ok=True)

    # Create a temp file in the same directory to keep the replace
       atomic
    with tempfile.NamedTemporaryFile("w", encoding="utf-8", dir=str(
       target.parent), delete=False) as tmp:
        tmp.write(text)
        tmp.flush()
        os.fsync(tmp.fileno())  # push to disk as best as the OS can
```

```
        tmp_name = tmp.name

    # Replace is atomic on same filesystem; readers will see old or new
        , not partial
    os.replace(tmp_name, str(target))
    print("Atomically wrote:", target)

atomic_write_text("sandbox/report.txt", "final contents\n")
```

Durability note: `flush()` moves data from Python to the OS; `os.fsync()` asks the OS to persist to storage. On real hardware, disk caches and controllers also play a role. If the computer loses power, even fsync cannot guarantee survival on every device, but it is the standard tool for best-effort durability.

4.10 Windows-Specific Notes

- Path length: old Windows APIs had a 260-character limit. Modern Windows can support longer paths with configuration; the prefix

  ?
  can be involved under the hood. Pathlib and modern Python try to handle this for you.
- Drives and UNC: paths can be drive-based (C:\) or UNC (

  `server`
  `share`
  `path`). `pathlib.Path` handles both.
- Newlines: text mode translates newlines to the OS convention. Use binary mode (`"rb"`/`"wb"`) if you need raw bytes.
- Case: Windows file systems are usually case-insensitive but case-preserving; Linux is case-sensitive.

4.11 Right vs. Wrong: OS Operations With Explanations

Listing 4.10: Demonstrate typical mistakes and show the fixes.

```
import os, io
from pathlib import Path

def show(path):
    p = Path(path)
    print(f"[{path}] exists:", p.exists(), "is_file:", p.is_file(), "
        is_dir:", p.is_dir())

def wrong_then_right_remove():
    # Wrong: try to unlink a directory with unlink
```

```
    safe_create_dir("sandbox_dir")
    try:
        Path("sandbox_dir").unlink()
    except IsADirectoryError as e:
        print("Caught:", e)
        print("Reason: unlink removes files, not directories.")
    # Right:
    try:
        os.rmdir("sandbox_dir")
        print("Removed empty directory 'sandbox_dir'")
    except OSError as e:
        print("Directory not empty; use shutil.rmtree if needed.")

def wrong_then_right_open_dir():
    # Wrong: open() expects files, not directories
    safe_create_dir("sandbox_dir2")
    try:
        open("sandbox_dir2", "r")
    except IsADirectoryError as e:
        print("Caught:", e)
        print("Reason: open() cannot open directories in text mode.")
    # Right: list directory entries
    print("Entries:", list(os.scandir("sandbox_dir2")))
    os.rmdir("sandbox_dir2")

wrong_then_right_remove()
wrong_then_right_open_dir()
```

4.12 Pathlib Cheatsheet

Listing 4.11: Common pathlib operations, very readable.

```
from pathlib import Path

p = Path("logs") / "2025" / "01" / "log.txt"
print("Parent:", p.parent)            # logs/2025/01
print("Name:", p.name)                # log.txt
print("Stem:", p.stem)                # log
print("Suffix:", p.suffix)            # .txt

p.parent.mkdir(parents=True, exist_ok=True)
p.write_text("hello\n", encoding="utf-8")
print("Read back:", p.read_text(encoding="utf-8"))

for q in p.parent.rglob("*.txt"):
    print("Found:", q)
```

Why This Works The Way It Works

- **System calls**: File operations cross from user mode to kernel mode through system calls. The CPU handles this transition (for x86, via syscall/sysenter or legacy int 0x80), then resumes your code when the OS returns. The CPU is agnostic about files; it only runs instructions.

- **Caching layers**: The OS keeps a page cache to avoid slow disk I/O. That is why `write()` may not be visible until newline, flush, close, or after a delay. `os.fsync()` asks the OS to flush its cache to the storage driver.
- **File systems**: NTFS, APFS, ext4, and others decide naming rules, metadata, and durability guarantees. Python does not change these rules; it exposes them.
- **Portability**: Using `os.path.join` or `pathlib` and handling exceptions (`FileNotFoundError`, `PermissionError`, `IsADirectoryError`, `NotADirectoryError`) produces code that behaves well across OSes.

Practice Prompts For Students

1. Build a portable path for today's date, such as: `logs/YYYY/MM/DD/log.txt`.
   Create any missing directories and write one line safely to that file.
2. Walk a directory tree and print the three largest files by size.
   Explain how you computed file sizes using the `os.stat()` function.
3. Write a function `safe_replace(path, text)` that writes to a temporary file and atomically replaces the target file, then verify that the contents were updated correctly.
4. On Windows, demonstrate the difference between a *raw string* (e.g., `r"C:\new\logs"`) and an *escaped string* (e.g., `"C:\\new\\logs"`).
   Explain what happens with backslashes in each case.

Chapter 5

Binary Data

Binary Data Basics

Some files consist of data stored as a sequence of bytes, known as **binary data**, that is not encoded into readable text using encodings like ASCII or UTF-8. Examples include images, videos, and PDFs.

When opened in a text editor, binary files often appear as random or unreadable symbols because the editor is trying to interpret raw byte values as text characters.

`bytes` objects are used in Python to represent sequences of byte values. They are immutable (cannot be changed after creation), similar to strings. A bytes object can be created using the built-in `bytes()` function or a bytes literal.

- `bytes("A text string", "ascii")` – creates bytes from a string using ASCII encoding
- `bytes(100)` – creates 100 zero-value bytes
- `bytes([12, 15, 20])` – creates bytes from numeric values

You can also create a bytes literal by prefixing a string with `b`:

Listing 5.1: Creating a bytes object using a literal.

```
my_bytes = b"This is a bytes literal"
print(my_bytes)
print(type(my_bytes))
```

```
b'This is a bytes literal'
<class 'bytes'>
```

Byte String Literals

You can represent specific byte values using hexadecimal escape codes. Each `\xHH` represents one byte in hexadecimal form.

Listing 5.2: Byte string literals.

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

```
True
```

Reading and Writing Binary Files

When working with binary files, use `'rb'` (read binary) or `'wb'` (write binary) modes.

Listing 5.3: Opening binary files.

```
# Open file for binary reading
f = open("data.bin", "rb")
contents = f.read()
f.close()

# Open file for binary writing
f = open("new_data.bin", "wb")
f.write(b"\x01\x02\x03\x04")
f.close()
```

In binary mode, Python does not translate newline characters. On Windows, this avoids converting `\n` to `\r\n`.

Inspecting Binary Contents of a File

Suppose we have an image `ball.bmp`. Reading it in binary mode allows us to inspect the raw byte values.

Listing 5.4: Inspecting binary contents of an image.

```
f = open("ball.bmp", "rb")
contents = f.read(32)
f.close()

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

This prints unreadable byte sequences like:

`b'BM\xf6\x00\x00\x00\x00\x00\x00\x00\x06\x04\x00\x00...'`

Example: Altering a BMP Image

Listing 5.5: Modifying pixels in a BMP image.

```
import struct

ball_file = open("ball.bmp", "rb")
ball_data = ball_file.read()
ball_file.close()

# BMP header stores pixel data offset in bytes 10 14
pixel_data_loc = struct.unpack("<I", ball_data[10-14])[0]

# Replace 3000 pixels with red, green, yellow pattern
new_pixels = b"\x01" * 3000 + b"\x02" * 3000 + b"\x03" * 3000

# Create new image data
new_data = ball_data[:pixel_data_loc] + new_pixels + ball_data[
    pixel_data_loc + len(new_pixels):]
```

```
# Save altered image
with open("new_ball.bmp", "wb") as f:
    f.write(new_data)
```

Using the struct Module

The `struct` module helps pack and unpack data into byte sequences.

Listing 5.6: Packing and unpacking bytes.

```
import struct

# Pack integers into binary data
data = struct.pack(">hh", 5, 256)
print("Packed:", data)

# Unpack binary back to integers
unpacked = struct.unpack(">hh", data)
print("Unpacked:", unpacked)
```

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

Performance Comparison: Binary vs Text Write Speed

Let's see how fast binary writing can be compared to text writing.

Listing 5.7: Comparing binary and text file speeds.

```
import time
import os

data_size = 10_000_000  # 10 MB
text_data = "A" * data_size
binary_data = b"A" * data_size

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

# Write binary data
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 is {(text_time/binary_time):.2f}x faster!")
```

Depending on your storage and system, binary writes are often slightly faster because fewer conversions are performed during I/O operations.

Key Takeaways

- Binary files store raw bytes, not human-readable text.
- Use `rb` / `wb` modes for reading and writing binary files.
- The `struct` module helps encode/decode structured binary data.
- Binary I/O can be faster than text I/O for large datasets.

Notes

This companion book was created to accompany the zyBooks interactive textbook, Chapter 12: *Files in Python*. Each section demonstrates the key concepts through well-structured LaTeX examples that are easy to copy and paste directly into a Python editor.