

ZyBooks Chapter 12: Files in Python

Jeremy Evert

October 20, 2025

Contents

1	12.1 Reading Files	1
1.1	Reading from a File	1
1.2	A More Complete Example	2
1.3	Reading Line by Line	2
1.4	Processing Data from a File	2
1.5	Iterating Directly Over a File Object	3
1.6	Practice Exercise	3
1.7	Explore More	4
2	12.2 Writing Files	5
2.1	Writing to a File	5
2.2	Why Some Writes Fail	5
2.3	File Modes	6
2.4	Buffered Output	7
2.5	Quiz Demonstrations	8
2.6	Safe File Creation	8
3	12.3 Interacting With File Systems	11
3.1	Portable Paths: os.path.join and pathlib	12
3.2	Existence, File vs. Directory, and Getting Size	12
3.3	Metadata: os.stat and datetime	13
3.4	Walking a Directory Tree	13
3.5	Creating, Renaming, Copying, Deleting	14
3.6	Portable File Path Building Activity	15
3.7	Splitting Paths and Getting Extensions	15
3.8	Challenge: Use os.walk to Count Specific Files	15
3.9	Safe and Durable Writes: Temp File + Atomic Replace	16
3.10	Windows-Specific Notes	16
3.11	Right vs. Wrong: OS Operations With Explanations	17
3.12	Pathlib Cheatsheet	18

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.

```
1 # Example 1: Reading the entire contents of a file
2
3 # Open the file in read mode
4 myjournal = open("journal.txt")
5
6 # Read the entire file into a single string
7 contents = myjournal.read()
8
9 # Display what was read
10 print(contents)
11
12 # Close the file after use
13 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.

```
1 print("Opening file myfile.txt.")
2 f = open("myfile.txt") # create file object
3
4 print("Reading file myfile.txt.")
5 contents = f.read() # read text into a string
6
7 print("Closing file myfile.txt.")
8 f.close() # close the file
9
10 print("\nContents of myfile.txt:")
11 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\everetj\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.

```
1 # Example 2: Read lines from a file
2 my_file = open("readme.txt")
3 lines = my_file.readlines()
4
5 # Print the second line (remember, Python starts counting at 0)
6 print(lines[1])
7
8 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.

```
1 # Example 3: Calculating an average from a file
2
3 print("Reading in data...")
```



```
4 f = open("mydata.txt")
5 lines = f.readlines()
6 f.close()
7
8 # Process data
9 print("\nCalculating average...")
10 total = 0
11 for ln in lines:
12     total += int(ln)
13
14 avg = total / len(lines)
15 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.

```
1 """Echo the contents of a file."""
2 f = open("myfile.txt")
3
4 for line in f:
5     print(line, end="") # end="" avoids double newlines
6
7 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.

```
1 # Example 4: Read and transform file content
2 filename = input("Enter filename: ")
3
4 with open(filename) as f: # 'with' auto-closes the file
5     contents = f.read()
6
7 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.

```
1 def write_basic_example():
2     """Write two lines to a new file."""
3     with open("myfile.txt", "w", encoding="utf-8") as f:
4         f.write("Example string.\n")
5         f.write("test....\n")
6     print("File written successfully!")
7
8 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.

```
1 def demonstrate_wrong_write():
```

```

2  """Show why writing numbers directly causes a TypeError."""
3  try:
4      with open("wrong_write.txt", "w", encoding="utf-8") as f:
5          # This will fail: write() only accepts strings.
6          f.write(10.0)
7  except TypeError as e:
8      print("Caught error:", e)
9      print("Explanation: write() in text mode needs a string, not a
10         number.")
11
12  # Correct way: convert numbers to strings.
13  with open("right_write.txt", "w", encoding="utf-8") as f:
14      num1, num2 = 5, 7.5
15      total = num1 + num2
16      f.write(str(num1))
17      f.write(" + ")
18      f.write(str(num2))
19      f.write(" = ")
20      f.write(str(total))
21      f.write("\n")
22  print("Fixed version works correctly!")
23  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.

```

1  from pathlib import Path
2  import io
3
4  def show_file(path):
5      p = Path(path)
6      print(f"\n[{p.name}] contents:")
7      print(p.read_text(encoding="utf-8") if p.exists() else "<missing>")
8
9  def demonstrate_modes():
10     path = "modes_demo.txt"
11     Path(path).write_text("START\n", encoding="utf-8")
12
13     # Wrong: open in read mode, try to write
14     try:
15         with open(path, "r", encoding="utf-8") as f:
16             f.write("APPEND\n")

```

```

17     except io.UnsupportedOperation as e:
18         print("Error:", e)
19         print("Explanation: 'r' is read-only; writing is not allowed.")
20
21     # Correct: append mode
22     with open(path, "a", encoding="utf-8") as f:
23         f.write("APPEND\n")
24     show_file(path)
25
26 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.

```

1 import os, time
2 from pathlib import Path
3
4 def buffering_demo(path="buffer_demo.txt"):
5     p = Path(path)
6     if p.exists():
7         p.unlink()
8
9     f = open(path, "w", encoding="utf-8")
10    f.write("Write me (no newline)") # stays in memory
11    print("Before flush: file may still be empty.")
12    time.sleep(1)
13    f.flush() # Push data from Python to OS
14    os.fsync(f.fileno()) # Ask OS to sync to disk
15    f.close()
16    print("After flush: file contents written.")
17
18 buffering_demo()

```

3.5 Quiz Demonstrations

Listing 3.5: Mini-quiz code with real behavior.

```

1 def quiz_behavior():
2     print("\n1) f.write(10.0) produces error:")
3     try:
4         with open("q1.txt", "w", encoding="utf-8") as f:
5             f.write(10.0)
6     except TypeError as e:
7         print("True:", e)
8
9     print("\n2) write() does NOT always write immediately:")
10    with open("q2.txt", "w", encoding="utf-8") as f:

```

```
11     f.write("hi")    # buffered
12     print("Immediate read:", open("q2.txt").read())
13     f.flush()
14     print("After flush: data is saved.")
15
16     print("\n3) flush()/fsync() forces output to disk:")
17     import os
18     with open("q3.txt", "w", encoding="utf-8") as f:
19         f.write("sync me")
20         f.flush()
21         os.fsync(f.fileno())
22     print("True: Data synced to disk.")
23
24 quiz_behavior()
```

3.6 Safe File Creation

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

```
1 def create_once(filename="create_once.txt"):
2     try:
3         with open(filename, "x", encoding="utf-8") as f:
4             f.write("File created successfully.\n")
5             print("Created new file:", filename)
6     except FileExistsError:
7         print("File already exists; not overwritten.")
8
9 create_once()
10 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.

```

1 import os
2 from pathlib import Path
3
4 def build_paths():
5     # WRONG on non-Windows and brittle even on Windows:
6     p_bad = "logs\\2025\\01\\log.txt"    # backslashes are Windows-only
7     print("Brittle:", p_bad)
8
9     # RIGHT: OS-appropriate separator via os.path.join
10    p_good = os.path.join("logs", "2025", "01", "log.txt")
11    print("Portable:", p_good)
12
13    # RIGHT: Path objects are even nicer
14    p = Path("logs") / "2025" / "01" / "log.txt"
15    print("Pathlib:", str(p))
16
17 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`.

```

1 import os
2 from pathlib import Path
3
4 def existence_and_type(path_str: str):
5     print("\n-- Using os.path --")
6     print("exists:", os.path.exists(path_str))
7     print("isfile:", os.path.isfile(path_str))
8     print("isdir:", os.path.isdir(path_str))
9
10    print("\n-- Using pathlib --")
11    p = Path(path_str)
12    print("exists:", p.exists())
13    print("is_file:", p.is_file())
14    print("is_dir:", p.is_dir())
15
16    if p.exists():
17        print("size:", p.stat().st_size, "bytes")
18
19 existence_and_type("modes_demo.txt")
20 existence_and_type("logs")

```


4.3 Metadata: os.stat and datetime

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

```

1 import os, datetime
2 from pathlib import Path
3
4 def show_stat(path_str: str):
5     p = Path(path_str)
6     if not p.exists():
7         print(f"{path_str!r} does not exist")
8         return
9     st = p.stat() # same as os.stat(path_str)
10    print("\n-- stat for", path_str, "--")
11    print("size:", st.st_size, "bytes")
12    print("mode (permission bits):", oct(st.st_mode))
13    print("modified:", datetime.datetime.fromtimestamp(st.st_mtime))
14    print("created (platform dependent):", datetime.datetime.
15          fromtimestamp(st.st_ctime))
16 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.

```

1 import os
2 from pathlib import Path
3
4 def list_py_files(root=".", ext=".txt"):
5     print(f"\nListing {ext} files under {root!r}")
6     for dirpath, subdirs, files in os.walk(root):
7         for name in files:
8             if name.lower().endswith(ext):
9                 print(os.path.join(dirpath, name))
10
11 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.

```

1 import shutil
2 from pathlib import Path
3
4 def safe_create_dir(path: str):
5     Path(path).mkdir(parents=True, exist_ok=True)
6     print("Ensured directory exists:", path)

```

```

7
8 def safe_rename(src: str, dst: str):
9     try:
10         # os.replace is atomic when src and dst are on the same
11         # filesystem
12         os.replace(src, dst)
13         print(f"Renamed {src!r} -> {dst!r}")
14     except FileNotFoundError:
15         print("Cannot rename: source not found.")
16     except PermissionError:
17         print("Cannot rename: permission denied.")
18
19 def safe_copy(src: str, dst: str):
20     try:
21         shutil.copy2(src, dst) # preserves timestamps and metadata
22         # where possible
23         print(f"Copied {src!r} -> {dst!r}")
24     except FileNotFoundError:
25         print("Cannot copy: source not found.")
26     except PermissionError:
27         print("Cannot copy: permission denied.")
28
29 def safe_delete(path: str):
30     try:
31         Path(path).unlink()
32         print("Deleted file:", path)
33     except FileNotFoundError:
34         print("Nothing to delete:", path)
35     except IsADirectoryError:
36         print("Path is a directory; use rmdir or shutil.rmtree.")
37     except PermissionError:
38         print("Cannot delete: permission denied.")
39
40 safe_create_dir("sandbox")
41 Path("sandbox/demo.txt").write_text("hello\n", encoding="utf-8")
42 safe_copy("sandbox/demo.txt", "sandbox/demo_copy.txt")
43 safe_rename("sandbox/demo_copy.txt", "sandbox/demo_moved.txt")
44 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.

```

1 import os
2
3 def join_examples():
4     a = os.path.join("subdir", "output.txt")
5     b = os.path.join("sounds", "cars", "honk.mp3")
6     print("Example join A:", a)
7     print("Example join B:", b)
8     print("Path separator on this OS:", os.path.sep)
9
10 join_examples()

```

4.7 Splitting Paths and Getting Extensions

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

```

1 import os
2
3 def split_examples(p: str):
4     head, tail = os.path.split(p)
5     root, ext = os.path.splitext(p)
6     print("\nSplit:", p)
7     print(" head:", head)
8     print(" tail:", tail)
9     print(" root:", root)
10    print(" ext:", ext)
11
12 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.

```

1 import os
2
3 def count_ext(root: str, ext: str = ".txt") -> int:
4     total = 0
5     for dirpath, subdirs, files in os.walk(root, onerror=None):
6         for name in files:
7             if name.lower().endswith(ext):
8                 total += 1
9     return total
10
11 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.

```

1 import os, tempfile
2 from pathlib import Path
3
4 def atomic_write_text(path: str, text: str):
5     target = Path(path)
6     target.parent.mkdir(parents=True, exist_ok=True)
7
8     # Create a temp file in the same directory to keep the replace
9     # atomic
10    with tempfile.NamedTemporaryFile("w", encoding="utf-8", dir=str(
11        target.parent), delete=False) as tmp:
12        tmp.write(text)
13        tmp.flush()
14        os.fsync(tmp.fileno()) # push to disk as best as the OS can

```

```

13         tmp_name = tmp.name
14
15         # Replace is atomic on same filesystem; readers will see old or new
16         # , not partial
17         os.replace(tmp_name, str(target))
18         print("Atomically wrote:", target)
19
20 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.

```

1 import os, io
2 from pathlib import Path
3
4 def show(path):
5     p = Path(path)
6     print(f"[{path}] exists:", p.exists(), "is_file:", p.is_file(), "
7           is_dir:", p.is_dir())
8
9 def wrong_then_right_remove():
10     # Wrong: try to unlink a directory with unlink

```

```

10     safe_create_dir("sandbox_dir")
11     try:
12         Path("sandbox_dir").unlink()
13     except IsADirectoryError as e:
14         print("Caught:", e)
15         print("Reason: unlink removes files, not directories.")
16     # Right:
17     try:
18         os.rmdir("sandbox_dir")
19         print("Removed empty directory 'sandbox_dir'")
20     except OSError as e:
21         print("Directory not empty; use shutil.rmtree if needed.")
22
23 def wrong_then_right_open_dir():
24     # Wrong: open() expects files, not directories
25     safe_create_dir("sandbox_dir2")
26     try:
27         open("sandbox_dir2", "r")
28     except IsADirectoryError as e:
29         print("Caught:", e)
30         print("Reason: open() cannot open directories in text mode.")
31     # Right: list directory entries
32     print("Entries:", list(os.scandir("sandbox_dir2")))
33     os.rmdir("sandbox_dir2")
34
35 wrong_then_right_remove()
36 wrong_then_right_open_dir()

```

4.12 Pathlib Cheatsheet

Listing 4.11: Common pathlib operations, very readable.

```

1 from pathlib import Path
2
3 p = Path("logs") / "2025" / "01" / "log.txt"
4 print("Parent:", p.parent)           # logs/2025/01
5 print("Name:", p.name)               # log.txt
6 print("Stem:", p.stem)               # log
7 print("Suffix:", p.suffix)           # .txt
8
9 p.parent.mkdir(parents=True, exist_ok=True)
10 p.write_text("hello\n", encoding="utf-8")
11 print("Read back:", p.read_text(encoding="utf-8"))
12
13 for q in p.parent.rglob("*.txt"):
14     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.

Notes

This book was created to accompany the zyBooks interactive textbook, Chapter 12: Files. Each section demonstrates the concepts with well-structured LaTeX code examples and clear explanations.