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`.

```
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

```
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

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.

```
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()
```

Listing 4.1: Right vs. wrong for building file 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

```
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")
```

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

4.3 Metadata: os.stat and datetime

```
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")
```

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

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

4.4 Walking a Directory Tree

```
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")
```

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

`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

```
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)

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")
```

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

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

```
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()
```

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

4.7 Splitting Paths and Getting Extensions

```
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"))
```

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

4.8 Challenge: Use os.walk to Count Specific Files

```
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"))
```

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

4.9 Safe and Durable Writes: Temp File + Atomic Replace

```
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")
```

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

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

```
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()
```

Listing 4.10: Demonstrate typical mistakes and show the fixes.

4.12 Pathlib Cheatsheet

```
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)
```

Listing 4.11: Common pathlib operations, very readable.

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`:

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

Listing 5.1: Creating a bytes object using a literal.

```
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.

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

Listing 5.2: Byte string literals.

```
True
```

Reading and Writing Binary Files

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

```
# 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()
```

Listing 5.3: Opening binary files.

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.

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

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

Listing 5.4: Inspecting binary contents of an image.

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

```
import struct

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

# BMP header stores pixel data offset in bytes 1014
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)
```

Listing 5.5: Modifying pixels in a BMP image.

Using the struct Module

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

```
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)
```

Listing 5.6: Packing and unpacking bytes.

```
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.

```
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!")
```

Listing 5.7: Comparing binary and text file speeds.

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 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.