# Working with Files in Python

By the end of this guide, you'll know how to read from files, write to files, navigate the file system, and work with different file formats.

## Part 1: Reading Data from Files

### Getting Started with File Input

Instead of always using keyboard input, you can retrieve data from files using Python's built-in `open()` function.

**Step 1: Understanding the open() function**

The `open()` function needs at least one argument—the path to your file:

- **Same directory**: `open("myfile.txt")` opens a file in the same folder as your script
- **Full path**: `open("C:\\Users\\BWayne\\tax_return.txt")` specifies the complete location

**Step 2: Closing files**

Always close your file when you're done with it using the `.close()` method. After closing, you can't read or write to that file anymore.

```
file = open("myfile.txt")
# Do something with the file
file.close()  # Important: close when done
```

### Three Methods for Reading Text

Python gives you three main ways to read file contents. Let's explore each one:

#### Method 1: file.read()

This method returns the entire file contents as a single string.

```
file = open("myfile.txt")
contents = file.read()
print(contents)
file.close()
```

**Optional parameter**: You can specify how many bytes to read:

```
first_100_chars = file.read(100)
```

#### Method 2: file.readlines()

This method returns a list of strings, where each element is one line from the file.

```
file = open("myfile.txt")
lines = file.readlines()
# lines[0] contains the first line
# lines[1] contains the second line
# and so on...
file.close()
```

### Method 3: file.readline()

This method returns one line at a time. It's particularly useful for large files that might not fit into memory.

```
file = open("myfile.txt")
first_line = file.readline()
second_line = file.readline()
file.close()
```

**Note**: All three methods automatically stop reading when they reach the end-of-file (EOF).

### Processing File Data Step-by-Step

One of your most common tasks will be reading data from a file and processing it. Here's the typical workflow:

1. Read the contents of the file
2. Iterate over each line to process data values
3. Compute results (like averages, totals, etc.)

**Example: Processing numeric data from a file**

```
# Step 1: Open and read the file
file = open("numbers.txt")

# Step 2: Iterate over each line
for line in file:
    # Step 3: Process the data
    number = float(line)
    print(f"Processing: {number}")

file.close()
```

**Pro tip**: File objects support iteration directly with `for...in` syntax, making line-by-line processing very simple!

---

## Part 2: Writing Data to Files

Programs write to files when you need to store data permanently. Let's learn how!

### The file.write() Method

This method writes a string to your file.

**Important**: `write()` only accepts strings. You must convert numbers first using `str()`:

```
file = open("output.txt", "w")
file.write("Hello, world!")
file.write(str(5.75))  # Convert number to string first
file.close()
```

### Understanding File Modes

When opening a file, you can specify a **mode** that determines what operations are allowed. The mode is the second argument to `open()`.

#### The Three Essential Modes

| Mode | Description | Read? | Write? | Creates File? | Overwrites? |
|---|---|---|---|---|---|
| "r" | Open for reading | Yes | No | No | No |
| "w" | Open for writing | No | Yes | Yes | Yes |
| "a" | Open for appending | No | Yes | Yes | No |

**Step-by-step guide to each mode:**
**Read mode "r"** (default if you don't specify):

```
file = open("myfile.txt", "r")  # Opens for reading
# If file is missing, you'll get an error
```

**Write mode "w"**:

```
file = open("myfile.txt", "w")  # Opens for writing
# Creates file if missing
# CAUTION: Overwrites existing content!
```

**Append mode "a"**:

```
file = open("myfile.txt", "a")  # Opens for appending
# Creates file if missing
# Adds new content to the end of existing content
```

**Advanced: Update modes**
Add a "+" to any mode to enable both reading and writing:

- "r+" allows reading and writing
- "w+" allows reading and writing (but still overwrites)

## Understanding Output Buffering

When you write to a file, Python doesn't immediately save it to disk. Instead, it uses a **buffer**.
**Default behavior**: Data is line-buffered, meaning it's written to disk only when a newline character appears.
**Controlling buffering**:

```
# Disable buffering (binary files only)
file = open("myfile.txt", "wb", buffering=0)

# Enable default line-buffering
file = open("myfile.txt", "w", buffering=1)

# Set custom buffer size (100 bytes)
file = open("myfile.txt", "w", buffering=100)
```

**Force writing immediately**:

```
file.flush()  # Forces buffered data to disk
# Note: Some systems also need os.fsync()
```

**Remember**: Closing a file automatically flushes the buffer!

# Part 3: Navigating the File System with the OS Module

## Why You Need the OS Module

Your program needs to interact with the computer's file system to perform tasks like:

- Getting the size of a file
- Opening files in different directories
- Checking if a file or directory exists

The OS module provides an interface to operating system functions.

```
import os
```

## Writing Portable Code

**Challenge**: Different operating systems use different path separators:

- Windows: "subdir\bat_mobile.jpg"
- Mac/Linux: "subdir/bat_mobile.jpg"

**Solution**: Use `os.path.join()` to create paths that work on any system!

### Step-by-Step: Using os.path.join()

**Don't do this** (hardcoded paths):

```
path = "subdir\\bat_mobile.jpg"  # Only works on Windows!
```

**Do this instead**:

```
import os
path = os.path.join("subdir", "bat_mobile.jpg")
# Automatically uses correct separator for current OS
```

**Windows full paths** (note the drive letter separator):

```
path = os.path.join("C:\\", "subdir1", "myfile.txt")
# Returns: "C:\\subdir1\\myfile.txt"
```

## Splitting Paths

You can split a path into its components:

```
import os
path = "C:\\Users\\BWayne\\tax_return.txt"
tokens = path.split(os.path.sep)
# Returns: ["C:", "Users", "BWayne", "tax_return.txt"]
```

**Note**: `os.path.sep` stores the correct separator for your operating system.

## Other Useful os.path Functions

The os.path module offers many helpful functions:

- Check if a path is a directory or file
- Get the size of a file
- Obtain file extensions (.txt, .doc, .pdf)
- Create and delete directories

## Walking Directory Trees with os.walk()

`os.walk()` lets you visit every subdirectory in a directory tree.

**Example usage**:

```
import os

for dirname, subdirs, files in os.walk("path/to/directory"):
    print(f"Current directory: {dirname}")
```

```
    print(f"Subdirectories: {subdirs}")
    print(f"Files: {files}")
```

**Understanding the three-tuple**:

4. `dirname`: Path to the current directory
5. `subdirs`: List of all subdirectories in the current directory
6. `files`: List of all files in the current directory

**Common use cases**:

- Searching for specific files when you don't know the exact path
- Filtering files by extension (.pdf, .txt, etc.)

---

# Part 4: Working with Binary Data

## What is Binary Data?

Some files store data as sequences of raw bytes, not as readable text. These include:

- Images
- Videos
- PDF files

Opening these files in a text editor shows incomprehensible text because the editor tries to interpret raw bytes as characters.

## The bytes Object

Python uses `bytes` objects to represent sequences of byte values.

**Creating bytes objects**:

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

# Create 100 zero-valued bytes
b2 = bytes(100)

# From a list of byte values
b3 = bytes([12, 15, 20])
```

**Using raw byte values** with the \x escape character:

```
# Hexadecimal byte values
raw_bytes = b"\x31\x32\x33"  # Represents ASCII '1', '2', '3'
print(raw_bytes)  # Displays: b'123'
```

## Binary File Modes

Open files in binary mode by adding "b" to the mode string:

```
# Read binary
file = open("image.jpg", "rb")

# Write binary
file = open("output.bin", "wb")
```

**Important difference on Windows**: Binary mode prevents automatic newline conversion ("\n" to "\r\n"), which would corrupt binary data.

**Key behaviors in binary mode**:

- `file.read()` returns a `bytes` object (not a string)
- `file.write()` expects a `bytes` argument (not a string)

---

# Part 5: The struct Module

## Packing and Unpacking Binary Data

The struct module helps you convert values (like integers) into bytes and vice versa.

```
import struct
```

## Using struct.pack()

Convert values into sequences of bytes:

```
# Pack a 2-byte integer
packed = struct.pack("<h", 1234)
```

**Understanding the format string**:

- **"<"**: Little-endian (least significant byte first)
- **">"**: Big-endian (most significant byte first)
- **"h"**: 2-byte integer

**Common format characters**:

- "b": 1-byte integer
- "h": 2-byte integer
- "I": 4-byte integer
- "s": String

**Example with multiple values**:

```
packed = struct.pack("<hhs", 100, 200, b"text")
```

## Using struct.unpack()

Convert bytes back into values:

```
# Unpack returns a tuple
result = struct.unpack("<h", packed_bytes)
value = result[0]  # Extract the first (and only) value
```

**Important**: `unpack()` always returns a tuple, even for single values!

---

# Part 6: Command-Line Arguments

You can make your program accept file locations as command-line arguments, allowing users to specify input files when running your program.

```
import sys

# sys.argv contains command-line arguments
# sys.argv[0] is the script name
# sys.argv[1] is the first argument, etc.

if len(sys.argv) > 1:
    filename = sys.argv[1]
    file = open(filename)
    # Process file...
```

```
    file.close()
else:
    print("Please provide a filename")
```

**Running your script**:

```
python myscript.py myfile1.txt
```

# Part 7: The "with" Statement (Best Practice!)

## Why Use "with"?

The `with` statement automatically closes files when you're done—no need to remember to call `.close()`!

**Basic syntax**:

```
with open("myfile.txt") as myfile:
    contents = myfile.read()
    print(contents)
# File automatically closes here!
```

## Understanding Context Managers

The `with` statement creates a **context manager** that:

7. Sets up the resource (opens the file)
8. Lets you use the resource
9. Tears down the resource (closes the file) automatically

## Why This is Important

Forgetting to close files can cause problems:

- A file opened in write mode can't be written to by other programs
- System resources may be wasted

**Best practice**: Always use `with` statements when working with files to guarantee proper closure.

**Multiple files**:

```
with open("input.txt") as infile, open("output.txt", "w") as
outfile:
    data = infile.read()
    outfile.write(data.upper())
# Both files automatically close
```

# Part 8: Working with CSV Files

## What are CSV Files?

Comma-separated values (CSV) files are text files that organize data in rows and columns using commas as separators.

**Example CSV file (students.csv)**:

```
name,hw1,hw2,midterm,final
Petr Little,9,8,85,78
Sam Tarley,10,9,90,92
Joff King,4,5,60,55
```

- Each line is a **row**
- Commas separate **fields** (columns)
- First row often contains column headers

## Reading CSV Files

Use Python's csv module for easy CSV handling.

**Step 1: Import the module and create a reader**:

```
import csv

with open("students.csv") as file:
    reader = csv.reader(file)

    # Step 2: Iterate over rows
    for row in reader:
        print(row)  # Each row is a list of strings
```

**Accessing specific fields**:

```
import csv

with open("students.csv") as file:
    reader = csv.reader(file)

    for row in reader:
        name = row[0]
        hw1 = row[1]
        hw2 = row[2]
        print(f"{name}: HW1={hw1}, HW2={hw2}")
```

## Using Different Delimiters

Some CSV files use semicolons or other characters instead of commas.

```
# If your file uses semicolons
reader = csv.reader(file, delimiter=';')
```

**Why different delimiters?** If a field itself contains a comma (like "Little, Petr"), you can either:

10. Use a different delimiter: `Little, Petr;9;8;85;78`
11. Use quotes around the field: `"Little, Petr",9,8,85,78`

## Converting Field Data

CSV fields are read as strings. Convert them to numbers when needed:

```
import csv

with open("students.csv") as file:
    reader = csv.reader(file)
    next(reader)  # Skip header row

    for row in reader:
        name = row[0]
        hw1 = int(row[1])       # Convert to integer
        hw2 = int(row[2])       # Convert to integer
        midterm = int(row[3])   # Convert to integer
        final = int(row[4])     # Convert to integer
```

```
        average = (hw1 + hw2 + midterm + final) / 4
        print(f"{name}: Average = {average}")
```

## Writing CSV Files

Create a writer object to write data to CSV files.

**Step 1: Create a writer**:

```
import csv

with open("output.csv", "w", newline='') as file:
    writer = csv.writer(file)

    # Step 2: Write a single row
    writer.writerow(["name", "score", "grade"])

    # Step 3: Write multiple rows
    writer.writerows([
        ["Alice", "95", "A"],
        ["Bob", "87", "B"],
        ["Carol", "92", "A"]
    ])
```

**Important**: Use `newline=''` when opening CSV files for writing to avoid extra blank lines on Windows.

---

## Essential File Operations

```
# Reading
with open("file.txt") as f:
    contents = f.read()          # Entire file as string
    lines = f.readlines()        # List of lines
    line = f.readline()          # Single line

# Writing
with open("file.txt", "w") as f:
    f.write("text")              # Write string
    f.write(str(123))            # Convert numbers first

# Appending
with open("file.txt", "a") as f:
    f.write("more text")         # Add to end of file
```

## Working with Paths

```
import os

# Create portable paths
path = os.path.join("folder", "subfolder", "file.txt")

# Split paths
parts = path.split(os.path.sep)
```

```
# Walk directories
for dirname, subdirs, files in os.walk("path"):
    # Process directories and files
    pass
```

### CSV Files

```
import csv

# Reading
with open("data.csv") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)  # List of strings

# Writing
with open("data.csv", "w", newline='') as f:
    writer = csv.writer(f)
    writer.writerow(["col1", "col2"])
    writer.writerows([["a", "b"], ["c", "d"]])
```

## Practice Exercises

1- **Basic File Reading**: Create a text file with several lines of text. Write a program that reads and prints each line with line numbers.

2- **Data Processing**: Create a file with numbers (one per line). Write a program that calculates and prints the average.

3- **File Copying**: Write a program that copies the contents of one file to another using the `with` statement.

4- **CSV Analysis**: Create a CSV file with student grades. Write a program that calculates each student's average and writes the results to a new CSV file.

5- **Directory Search**: Use `os.walk()` to find all .txt files in a directory tree and print their full paths.