COMP101 — Week 10
File Handling in Python

Reading, writing, and cleaning external data

UM6P — SASE

December 19, 2025

Today's Goals

- Open, read, and write text files safely in Python.

Today's Goals

- Open, read, and write text files safely in Python.
- Use context managers (with open(...) as f) to avoid resource leaks.

Today's Goals

- Open, read, and write text files safely in Python.
- Use context managers (with open(...) as f) to avoid resource leaks.
- Process line-based data: strip, split, convert types, validate.

Today's Goals

- Open, read, and write text files safely in Python.
- Use context managers (with open(...) as f) to avoid resource leaks.
- Process line-based data: strip, split, convert types, validate.
- Handle common file errors (FileNotFoundError, malformed lines).

Today's Goals

- Open, read, and write text files safely in Python.
- Use context managers (with open(...) as f) to avoid resource leaks.
- Process line-based data: strip, split, convert types, validate.
- Handle common file errors (FileNotFoundError, malformed lines).
- Think about performance: reading whole files vs streaming line by line.

Why File Handling?

- Programs rarely work only with data you type at the keyboard.

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.
- Files let us:

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.
- Files let us:
    - Persist results between program runs.

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.
- Files let us:
    - Persist results between program runs.
    - Reuse existing datasets.

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.
- Files let us:
  - Persist results between program runs.
  - Reuse existing datasets.
  - Exchange data with other tools (Excel, databases, etc.).

Why File Handling?

- Programs rarely work only with data you type at the keyboard.
- Real tasks: logs, configuration files, CSV exports, reports.
- Files let us:
    - Persist results between program runs.
    - Reuse existing datasets.
    - Exchange data with other tools (Excel, databases, etc.).
- Input/output is the first point where your code meets the messy outside world.

What is a File?

Definition

A file is a named sequence of bytes stored by the operating system on a device (disk, SSD, USB, …). Python gives you a higher-level interface to read and write these bytes.

- The OS controls access to files (permissions, paths, modes).

What is a File?

Definition

A file is a named sequence of bytes stored by the operating system on a device (disk, SSD, USB, ...). Python gives you a higher-level interface to read and write these bytes.

- The OS controls access to files (permissions, paths, modes).
- Python wraps this with a file object.

What is a File?

Definition

A file is a named sequence of bytes stored by the operating system on a device (disk, SSD, USB, . . .). Python gives you a higher-level interface to read and write these bytes.

- The OS controls access to files (permissions, paths, modes).
- Python wraps this with a file object.
- For text files, Python converts between bytes and strings using an encoding.

The open() Function

- Basic usage:

```
f = open("data.txt", "r")  # "r" = read text
text = f.read()
f.close()
```

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):
    - ”r”: read (file must exist)

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):
  - "r": read (file must exist)
  - "w": write (overwrite if file exists)

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):
  - "r": read (file must exist)
  - "w": write (overwrite if file exists)
  - "a": append (add to end)

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):
    - ”r”: read (file must exist)
    - ”w”: write (overwrite if file exists)
    - ”a”: append (add to end)
    - ”rb”/”wb”: binary modes

The open() Function

- Basic usage:

```
f = open("data.txt", "r")   # "r" = read text
text = f.read()
f.close()
```

- Filename: string, relative or absolute path.
- Mode (common):
  - ”r”: read (file must exist)
  - ”w”: write (overwrite if file exists)
  - ”a”: append (add to end)
  - ”rb”/”wb”: binary modes
- Calling close() releases the resource; forgetting it is a bug.

The Safe Pattern: with open(...)

Definition

A context manager automatically acquires and releases a resource, even if an error occurs inside the block.

```
# Recommended pattern
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
    # work with text here

# f is automatically closed here
```

- Always prefer with for files.

The Safe Pattern: with open(...)

Definition

A context manager automatically acquires and releases a resource, even if an error occurs inside the block.

```
# Recommended pattern
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
    # work with text here

# f is automatically closed here
```

- Always prefer with for files.
- Avoids leaking open file handles.

The Safe Pattern: with open(...)

Definition

A context manager automatically acquires and releases a resource, even if an error occurs inside the block.

```
# Recommended pattern
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
    # work with text here

# f is automatically closed here
```

- Always prefer with for files.
- Avoids leaking open file handles.
- Makes the lifetime of the file object explicit.

Reading From Files: 3 Patterns

- Whole file at once (read()):

```
with open("data.txt", "r", encoding="utf-8") as f:
    data = f.read()   # one big string
```

```
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:          # streaming iteration
        line = line.strip()
        # process line
```

```
with open("data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()  # list of strings
```

Reading From Files: 3 Patterns

- Whole file at once (read()):

```
with open("data.txt", "r", encoding="utf-8") as f:
    data = f.read()   # one big string
```

- Line by line (readline() or iteration):

```
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:          # streaming iteration
        line = line.strip()
        # process line
```

```
with open("data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()  # list of strings
```

Reading From Files: 3 Patterns

- Whole file at once (read()):

```
with open("data.txt", "r", encoding="utf-8") as f:
    data = f.read()   # one big string
```

- Line by line (readline() or iteration):

```
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:          # streaming iteration
        line = line.strip()
        # process line
```

- Read all lines into a list:

```
with open("data.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()  # list of strings
```

Writing To Files

```
# Overwrite or create file
with open("results.txt", "w", encoding="utf-8") as f:
    f.write("Average score: 15.7\n")
    f.write("Median score: 14.0\n")

# Append to existing file
with open("results.txt", "a", encoding="utf-8") as f:
    f.write("New run finished.\n")
```

- ”w” truncates the file if it exists.

Writing To Files

```
# Overwrite or create file
with open("results.txt", "w", encoding="utf-8") as f:
    f.write("Average score: 15.7\n")
    f.write("Median score: 14.0\n")

# Append to existing file
with open("results.txt", "a", encoding="utf-8") as f:
    f.write("New run finished.\n")
```

- ”w” truncates the file if it exists.
- ”a” keeps existing content and adds at the end.

Writing To Files

```
# Overwrite or create file
with open("results.txt", "w", encoding="utf-8") as f:
    f.write("Average score: 15.7\n")
    f.write("Median score: 14.0\n")

# Append to existing file
with open("results.txt", "a", encoding="utf-8") as f:
    f.write("New run finished.\n")
```

- ”w” truncates the file if it exists.
- ”a” keeps existing content and adds at the end.
- Always control your newlines ”\n” explicitly.

Encodings and Newlines

- Text in Python is Unicode; files are bytes.

```
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

Encodings and Newlines

- Text in Python is Unicode; files are bytes.
- Encoding tells Python how to translate between them:

```python
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

Encodings and Newlines

- Text in Python is Unicode; files are bytes.
- Encoding tells Python how to translate between them:

```
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

- Use encoding=”utf-8” unless you have a strong reason not to.

Encodings and Newlines

- Text in Python is Unicode; files are bytes.
- Encoding tells Python how to translate between them:

```
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

- Use encoding=”utf-8” unless you have a strong reason not to.
- Different systems use different newline conventions; Python normalizes them to ”\n” when reading in text mode.

Encodings and Newlines

- Text in Python is Unicode; files are bytes.
- Encoding tells Python how to translate between them:

```
with open("data.txt", "r", encoding="utf-8") as f:
    text = f.read()
```

- Use encoding="utf-8" unless you have a strong reason not to.
- Different systems use different newline conventions; Python normalizes them to "\n" when reading in text mode.
- Encoding mistakes show up as strange symbols or UnicodeDecodeError.

From Raw Lines to Usable Data

```
with open("scores.txt", "r", encoding="utf-8") as f:
    scores = []
    for line in f:
        line = line.strip()
        if not line:            # skip empty lines
            continue
        parts = line.split(",")    # e.g. "Ali,17"
        name = parts[0]
        score = int(parts[1])
        scores.append((name, score))
```

- strip() to remove extra spaces and newlines.

From Raw Lines to Usable Data

```python
with open("scores.txt", "r", encoding="utf-8") as f:
    scores = []
    for line in f:
        line = line.strip()
        if not line:        # skip empty lines
            continue
        parts = line.split(",")   # e.g. "Ali,17"
        name = parts[0]
        score = int(parts[1])
        scores.append((name, score))
```

- strip() to remove extra spaces and newlines.
- split() to break structured text into fields.

From Raw Lines to Usable Data

```
with open("scores.txt", "r", encoding="utf-8") as f:
    scores = []
    for line in f:
        line = line.strip()
        if not line:          # skip empty lines
            continue
        parts = line.split(",")   # e.g. "Ali,17"
        name = parts[0]
        score = int(parts[1])
        scores.append((name, score))
```

- strip() to remove extra spaces and newlines.
- split() to break structured text into fields.
- Type conversion (e.g. int(), float()) with basic validation.

Handling File Errors

```python
filename = "scores.txt"

try:
    with open(filename, "r", encoding="utf-8") as f:
        data = f.read()
except FileNotFoundError:
    print(f"File {filename} not found.")
except PermissionError:
    print(f"No permission to read {filename}.")
```

- Expect missing files and permission issues.

Handling File Errors

```python
filename = "scores.txt"

try:
    with open(filename, "r", encoding="utf-8") as f:
        data = f.read()
except FileNotFoundError:
    print(f"File {filename} not found.")
except PermissionError:
    print(f"No permission to read {filename}.")
```

- Expect missing files and permission issues.
- Catch specific exceptions, not a generic except:.

Handling File Errors

```python
filename = "scores.txt"

try:
    with open(filename, "r", encoding="utf-8") as f:
        data = f.read()
except FileNotFoundError:
    print(f"File {filename} not found.")
except PermissionError:
    print(f"No permission to read {filename}.")
```

- Expect missing files and permission issues.
- Catch specific exceptions, not a generic except:.
- Decide: fail with a clear message, or create a default file.

Mini Workflow: Cleaning a Scores File

Algorithm

Task: Read a file scores.txt with lines ”name,score”, ignore invalid lines, and write a cleaned file plus summary.

Mini Workflow: Cleaning a Scores File

```python
valid = []
with open("scores.txt", "r", encoding="utf-8") as f:
    for line in f:
        line = line.strip()
        if not line:
            continue
        parts = line.split(",")
        if len(parts) != 2:
            continue
        name, raw = parts
        try:
            score = float(raw)
        except ValueError:
            continue
        valid.append((name, score))
```

Mini Workflow: Cleaning a Scores File

```
with open("scores_clean.txt", "w", encoding="utf-8") as out:
    for name, score in valid:
        out.write(f"{name},{score}\n")
```

Performance: Whole File vs Streaming

- Small files: read() or readlines() are fine.

```
with open("big_log.txt", "r", encoding="utf-8") as f:
    for line in f:
        process(line)
```

Performance: Whole File vs Streaming

- Small files: read() or readlines() are fine.
- Large files: prefer iterating line by line:

```
with open("big_log.txt", "r", encoding="utf-8") as f:
    for line in f:
        process(line)
```

Performance: Whole File vs Streaming

- Small files: read() or readlines() are fine.
- Large files: prefer iterating line by line:

```
with open("big_log.txt", "r", encoding="utf-8") as f:
    for line in f:
        process(line)
```

- This keeps memory usage almost constant.

Performance: Whole File vs Streaming

- Small files: read() or readlines() are fine.
- Large files: prefer iterating line by line:

```
with open("big_log.txt", "r", encoding="utf-8") as f:
    for line in f:
        process(line)
```

- This keeps memory usage almost constant.
- Think about complexity: your algorithm may be fast, but IO can dominate.

Common Pitfalls

- Forgetting to close files (fixed by using with).

Common Pitfalls

- Forgetting to close files (fixed by using with).
- Assuming the file always exists and is well-formed.

Common Pitfalls

- Forgetting to close files (fixed by using with).
- Assuming the file always exists and is well-formed.
- Ignoring leading/trailing spaces and blank lines.

Common Pitfalls

- Forgetting to close files (fixed by using with).
- Assuming the file always exists and is well-formed.
- Ignoring leading/trailing spaces and blank lines.
- Confusing text and binary modes.

Common Pitfalls

- Forgetting to close files (fixed by using with).
- Assuming the file always exists and is well-formed.
- Ignoring leading/trailing spaces and blank lines.
- Confusing text and binary modes.
- Accidentally overwriting files with ”w” instead of ”a”.

Common Pitfalls

- Forgetting to close files (fixed by using with).
- Assuming the file always exists and is well-formed.
- Ignoring leading/trailing spaces and blank lines.
- Confusing text and binary modes.
- Accidentally overwriting files with ”w” instead of ”a”.
- Hard-coding absolute paths that only work on your machine.

Summary

- File handling connects your code to persistent data.

Summary

- File handling connects your code to persistent data.
- Use with open(..., mode, encoding) as your default pattern.

Summary

- File handling connects your code to persistent data.
- Use with open(..., mode, encoding) as your default pattern.
- Read and write text carefully: strip, split, convert, validate.

Summary

- File handling connects your code to persistent data.
- Use with open(..., mode, encoding) as your default pattern.
- Read and write text carefully: strip, split, convert, validate.
- Handle file-related exceptions explicitly.

Summary

- File handling connects your code to persistent data.
- Use with open(..., mode, encoding) as your default pattern.
- Read and write text carefully: strip, split, convert, validate.
- Handle file-related exceptions explicitly.
- For larger files, stream line by line instead of loading everything.