

Lecture 12

File Handling: TXT, CSV and JSON

Files and text

File Handling: TXT, CSV and JSON · Lecture 12

Learning outcomes

Explain why variables do not survive after a program exits

Read and write text files using `with open(...)`

Distinguish read, write and append modes

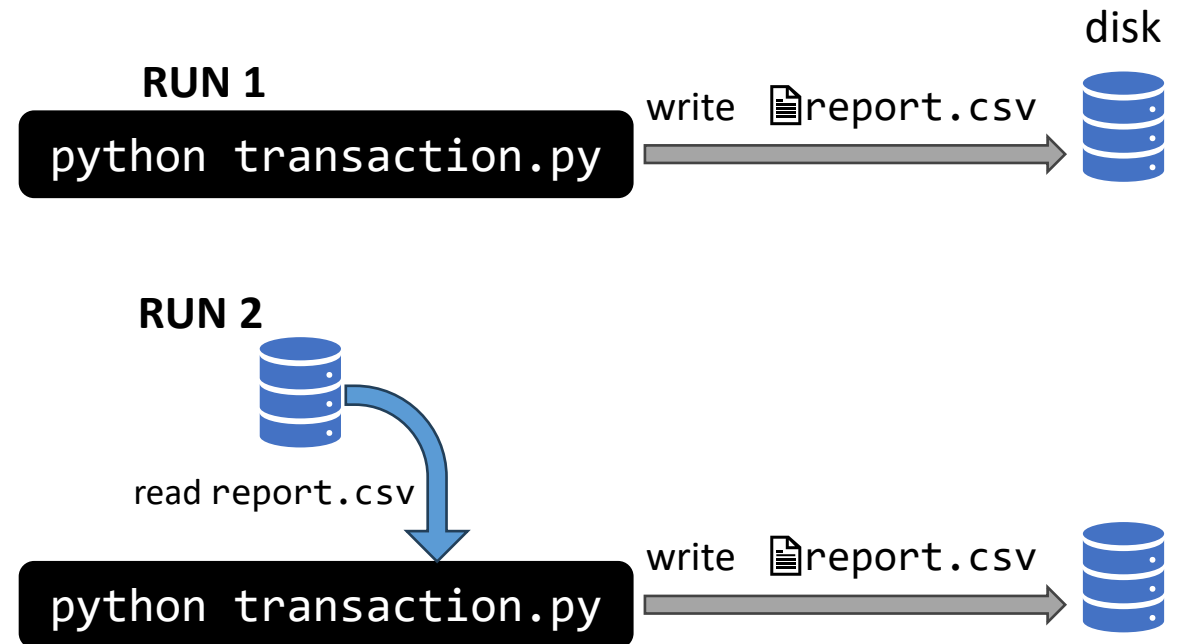
Work with CSV rows and typed values

Save and load nested Python-compatible JSON data.

A file is persistent storage, not a Python object kept between runs

Memory (RAM) is temporary; files persist

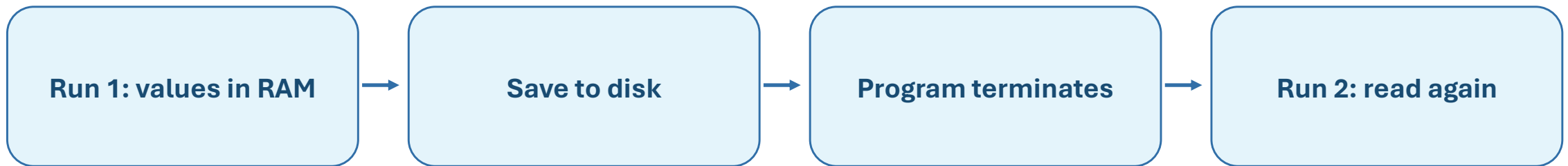
- A variable stores data while the program runs
- After program termination, ordinary variables are lost
- Unlike variables in memory, **files can preserve data** after a program terminates (Mass Storage)
- If a program modifies a file, future executions can read the updated content
 - As a consequence, the behaviour of your program may change depending on the current state of the file.



Persistent data must be converted between text/bytes and Python values.

What persistence means

Data inside the running program disappears, but saved files remain.



To retrieve old values, the second run must explicitly open and parse the file.

File Path; Structure and Syntax

- A **path** is a sequence of directory names, optionally followed by a file name, that identifies a location in the file system
- Directories are organized hierarchically, forming a tree-like structure
 - **Separators** distinguish consecutive directories: `/` on Linux/macOS, typically `\` on Windows
- **Root:** the starting point of an absolute path (`/` on Linux/macOS; typically a drive such as `C:\` or `D:\` on Windows)
- If the path identifies a **file**, the name must include the extension such as `.txt` or `.csv`, otherwise it is a **directory**
- Special components:
 - `.` refers to the current directory
 - `..` refers to its parent directory.

Linux/macOS

- `/home/user/project/data.csv`
- `./project/data.csv` (if you are working in “user”)

Windows

- `C:\home\user\project\data.csv`
- `.\project\data.csv`

Both use:

- `.` (current directory)
- `..` (parent directory refers to “home”)

Relative and Absolute paths

- A path can be **relative** or **absolute**
- Absolute paths describe the full path from root to the directory/file (e.g. C:\)
- Relative paths locates a file relative to the *current working directory* (aka, the folder where you are)
 - Example: C:\home\user\project\data.csv
 - Assuming the current working directory is project:

Path	Meaning
data.csv	File inside the current directory
data/data.csv	File inside the data subdirectory
../data.csv	File in the parent directory
./data.csv	File in the current directory (explicit notation)

Python note

In Python strings, “\” can represent escape sequences (e.g., \n for new line). Be careful when using Windows paths.

There are three possible solutions:

- **Raw strings:** using “r” before the string, inhibit the interpretation of special chars e.g. `r"C:\Users\Alice\data.csv"` avoiding the problem using directly Windows style
- **Escaped \:** using “\\” to escape the special symbol
- **Use the Linux/macOS style** with “/”, Python interprets automatically

```
# Backslashes in a raw string
path = r"C:\Users\Alice\Documents\data.csv"

# Escaped backslashes
path = "C:\\Users\\Alice\\Documents\\data.csv"

# Forward slashes (recommended for simple examples)
path = "C:/Users/Alice/Documents/data.csv"
```

Working with files: open, use, close

Before Python can read or modify a file, the program must open it.



Opening a file creates a file object that the program uses to access the file.

File contents may persist on disk; an open file object is a temporary program resource

open(): path, mode and encoding

- `open()` is a built-in Python function
- Its first parameter, **file**, identifies the **path to the file**
- **mode** tells Python what you intend to do ("r", "w", "a", "r+", "w+ "r+"...), it is a string
- other parameters, like `encoding` to specify how text characters are encoded (default often UTF-8 but may depend on the OS, in Windows better specify)
- `open()` returns an **object of type file**, which we can store in a variable.

```
file = open("notes.txt", "r", encoding="utf-8")

# Typical call structure:
# open(file, mode="r")

# "r" = read (the default, can be omitted)
# "w" = write (replace existing contents)
# "a" = append
```

Docs: <https://docs.python.org/3/builtins/functions.html#open>

Read, write and append modes

- **r** reads an existing file, failing if it is missing
- **w** creates or replaces a file: existing contents are destroyed
- **a** appends new content to an existing file (or creates it)
- **x** creates a new file
- with **+** we can obtain mixed behaviors

```
# read file and add a new line at the end
file = open("data.txt", "r+", encoding="utf-8")
content = file.read()
file.write("\nNew line")

# Write/overwrite a file with a string
file = open("results.txt", "w", encoding="utf-8")
file.write("Hello!")
```

Mode	Read	Write	If file does not exist	If file exists
"r"	✓	✗	Error	Preserves content
"w"	✗	✓	Creates file	Erases existing content
"a"	✗	✓	Creates file	Appends at the end
"r+"	✓	✓	Error	Preserves content initially
"w+"	✓	✓	Creates file	Erases existing content
"a+"	✓	✓	Creates file	Appends and reads
"x"	✗	✓	Creates file	Error

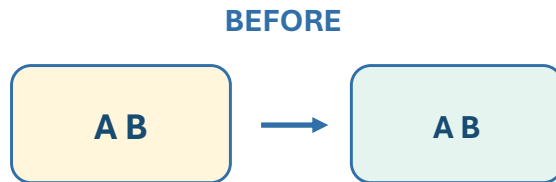
Never use w if you mean to keep existing records.

File modes change what happens to existing data

A simple timeline shows how opening a file affects earlier contents.

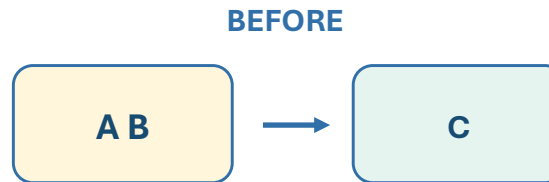
"r" · Read only

Existing file required



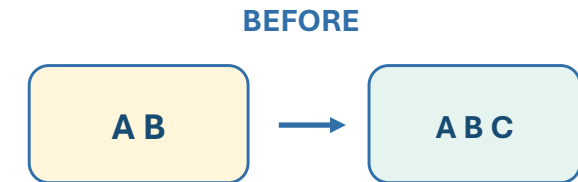
"w" · Write

Replaces old content



"a" · Append

Adds after existing data



Always choose the opening mode deliberately: "w" destroys the previous file contents

The file object and `close()`

- The file object provides operations such as `read()` and `write()`
- When finished, call its `.close()` method
- `.close()` takes no arguments; it closes the file and releases resources
- For writing, closing also flushes buffered data before closing
- After `close()`, do not try to read or write using that object.

```
f = open("notes.txt", "r", encoding="utf-8")

text = f.read()
print(text)

f.close()    # No parameters

# f.read()  # Error: file is closed
```

[Official close\(\) documentation ↗](#)

Remember: **a file object is *not* the file content itself**

A complete example: open → process → close

- Open the file in read mode using its path
- Read its contents into a Python string
- Process the string (here: count its lines)
- Close the file when no longer needed
- **Problem:** if an error interrupts the code before `close()`, the closing statement may not run, and the resource is not freed

```
# Assume notes.txt exists
f = open("notes.txt", "r", encoding="utf-8")

text = f.read()
lines = text.splitlines()
print("Number of lines:", len(lines))

f.close()
```

Manual `close()` works, but requires care if something goes wrong

with: automatic file closing

- **with** - **as** opens a managed block for working with the file
- The variable after **as** refers to the open file object; it defines an alias you can use as the variable name.
- Read or write inside the indented block
- Leaving the block, automatically closes the file, even if an exception/error occurs inside it
- Important: **with** does not catch or hide the exception; it still propagates unless handled.

```
# Same program, shorter and safer
with open("notes.txt", "r", encoding="utf-8") as f:
    text = f.read()
    lines = text.splitlines()
    print("Number of lines:", len(lines))
# The file has been closed here.
# No explicit f.close() required.
```

[Python tutorial: reading and writing files ↗](#)

Use with open(...) as f: as the normal pattern for file operations.

Why use `with` instead of manual close?

- `with` manages opening and closing as one block
- The file object remains valid inside the indented suite
- **Resources are released** when the suite ends, *even if an exception happens*
- For simple programs prefer this pattern over `open()` followed by `close()`.

```
with open("report.txt", "w") as f:  
    f.write("Completed")  
# automatically closed f  
print("Operation completed")
```

Opening a file is a resource-management operation.

Reading a file: where does Python start?

An open file has a **current position**: the location of the next data to be read

notes.txt

↑ Apple	1
Banana	2
Cherry	3

Imagine a **cursor** showing where the next read begins

At first, it is at the beginning of the file

The position is not limited to line boundaries: it may also be in the middle of a line.

Key idea: reading normally moves the cursor position forward.

A read moves the current position

The next operation continues from where the previous one stopped.

notes.txt (same example)

↑ Apple\nBanana\nCherry\n



```
with open("notes.txt", "r") as f:
    first = f.readline()
    second = f.readline()

print(first.strip()) # Apple
print(second.strip()) # Banana
```

`readline()` reads one line and moves the position to the following line.

A read moves the current position

The next operation continues from where the previous one stopped.

notes.txt (same example)

Apple\nBanana\nCherry\n

Stops at the next
new line symbol \n

```
with open("notes.txt", "r") as f:  
    → first = f.readline()  
    second = f.readline()  
  
print(first.strip()) # Apple  
print(second.strip()) # Banana
```

`readline()` reads one line and moves the position to the following line.

A read moves the current position

The next operation continues from where the previous one stopped.

notes.txt (same example)

Apple\nBanana\nCherry\n



```
with open("notes.txt", "r") as f:  
    first = f.readline()  
    → second = f.readline()  
  
print(first.strip()) # Apple  
print(second.strip()) # Banana
```

`readline()` reads one line and moves the position to the following line.

What happens when we reach the end?

The **end of file (EOF)** means there is no more content to read.

notes.txt (same example)

↑ Apple\nBanana\nCherry\n

→

```
with open("notes.txt", "r") as f:  
    all_text = f.read()  
    again    = f.read()  
  
print(all_text)  
print(again)
```

A second read does not automatically restart at the beginning.

What happens when we reach the end?

The **end of file (EOF)** means there is no more content to read.

notes.txt (same example)

Apple\nBanana\nCherry\n



```
with open("notes.txt", "r") as f:  
→ all_text = f.read()  
  again    = f.read()  
  
print(all_text)  
print(again)
```

A second read does not automatically restart at the beginning.

What happens when we reach the end?

The **end of file (EOF)** means there is no more content to read.

notes.txt (same example)

Apple\nBanana\nCherry\n

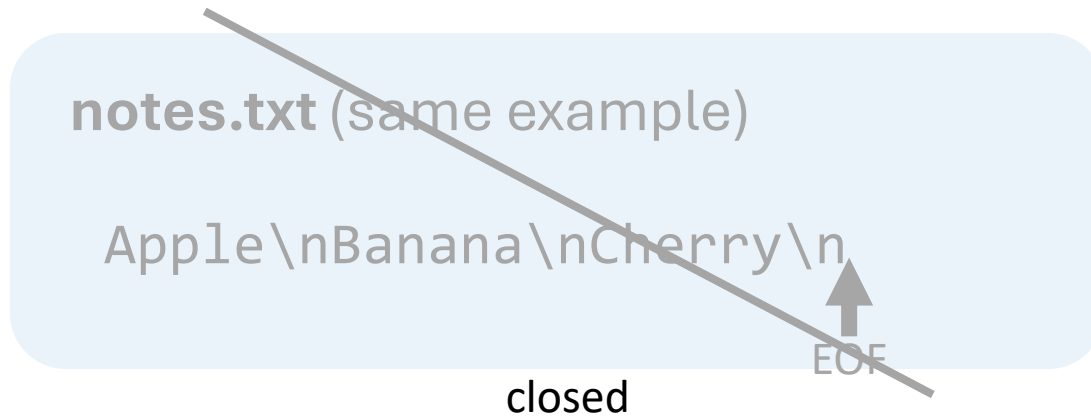


```
with open("notes.txt", "r") as f:  
    all_text = f.read()  
    → again = f.read()  
  
print(all_text)  
print(again)
```

A second read does not automatically restart at the beginning.

What happens when we reach the end?

The **end of file (EOF)** means there is no more content to read.



```
with open("notes.txt", "r") as f:  
    all_text = f.read()  
    again    = f.read()  
  
print(all_text)  
print(again)
```

Example output

```
'Apple\nBanana\nCherry\n'  
''
```

The second read returns an empty string because the position is already at EOF

A second read does not automatically restart at the beginning.

Moving the File Position: seek()

- Python provides the `.seek()` method to change the current position within an open file
- The simplest use is `file.seek(0)`, which moves the position **back to the beginning** of the file
- This allows us to read the same content again, without closing and reopening the file
- Moving to arbitrary positions is subject to restrictions; for now, we will only use `seek(0)`.

```
with open("notes.txt", "r") as f:
    print(f.readline().strip()) # Apple
    print(f.readline().strip()) # Banana

    f.seek(0)                  # Restart
    print(f.readline().strip()) # Apple
```

`seek(0)` sets the position back to the beginning of the file.

Output

Apple
Banana
Apple

For beginners: prefer one reading pass whenever possible; reopen the file only if needed

Four ways to read text files

Each method **starts at the current position**, not necessarily at the beginning.

read()

```
text = f.read()
```

One **string** containing all remaining text

readlines()

```
lines = f.readlines()
```

A **list** of the remaining lines (strings)

readline()

```
single_line = f.readline()
```

A **string** containing text until next new line

for line in f:

```
for line in f:  
    print(line.strip())
```

Processes remaining lines one at a time

Example: `readlines()` and iteration

Consider `notes.txt` containing Apple, Banana, Cherry on separate lines.

Option A – `readlines()`

```
with open("notes.txt", "r") as f:
    lines = f.readlines()

for line in lines:
    print(line.strip())
```

Apple
Banana
Cherry

Option B – direct iteration

```
with open("notes.txt", "r") as f:
    for line in f:
        print(line.strip())
```

Both print the same three names; direct iteration does not first build a list of all lines

Loads all remaining lines into a list

File can be closed before processing

Higher memory consumption

Convenient for **small files** and repeated processing

All lines are immediately accessible

Processes lines progressively

File remains open during processing

Lower memory consumption

Suitable for **large files** and selective processing

Can stop early when the desired information is found

Use `readlines()` when you need to keep all lines available in memory. Use direct file iteration when you want to process lines progressively, **especially for large files** or when you may stop reading early.

Writing a file

- File objects have a method `.write()`
- `.write()` expects a **string**; convert numbers with `str()` or f-strings
- Writes subsequent strings, as if you are writing a normal text with the cursor moving forward
- New lines must be included explicitly using `\n`
- After writing, open the file to verify its contents.

```
with open("summary.txt", "w") as f:  
    f.write("Count: 3")  
    f.write(f"\nTotal: {46.5:.2f}")
```

The written file becomes a testable program output

The terminal has a current directory

The current working directory (CWD) is the folder from which the terminal runs commands

- **Display** the CWD with `pwd` (PowerShell, Linux and macOS)
- **change** directory with `cd` (Windows Command Prompt) to navigate the file system from the terminal
 - Accepts absolute or relative paths
 - `cd ..` moves to the parent folder
- A terminal starts in some folder, but you can navigate elsewhere before running Python

PowerShell (Windows)

```
PS> pwd
PS> cd C:/course
```

Linux / macOS

```
$> pwd
$> cd /home/alice/course
```

In Command Prompt (cmd), `cd` prints the CWD; use `cd /path/to/directory` to switch drives and folders

Run a Python script from any directory

The path after `python` identifies the script to run: it may be relative or absolute

Example directory structure

```
C:/course/  
  scripts/  
    report.py  
  data/  
    results.txt
```

Run from C:/course/

```
# Relative path  
PS C:\course> python scripts/report.py  
  
# Absolute path  
PS C:\course> python C:/course/scripts/report.py
```

The script path determines WHAT is executed, not the process working directory

The script folder and CWD may differ

- Python normally **inherits the CWD of the terminal** where the command is launched
- The script may be stored in another directory
- Import module `os` and use `os.getcwd()` to inspect the CWD from inside a Python program

Terminal: CWD is C:/course/

```
python scripts/report.py
```

```
# Inside report.py:
```

```
import os
```

```
print(os.getcwd())    # C:/course
```

`report.py` is located in `C:/course/scripts/`, but its CWD is `C:/course/`

Why relative file paths can be surprising

The same script can access different locations depending on where it is launched. This may happen both reading or writing with unexpected “File not found” or creating/overwriting files in unexpected positions.

```
# Inside report.py  
with open("data/results.txt", "r") as file:
```

Launched from C:/course/

```
python scripts/report.py
```

Opens: C:/course/data/results.txt

Launched from C:/course/scripts/

```
python report.py
```

Looks for:
C:/course/scripts/data/results.txt

The code is unchanged; only the CWD is different

Connecting CWD to file handling

Before using a relative path in `open()`, ask:

- Where is the terminal currently working? Check with `pwd` or `cd`
- Which path am I using to launch the Python script?
- Where will `open("data.csv")` look for the file? Start from the CWD

```
Quick Python check: import os; print(os.getcwd())
```

A `FileNotFoundError` may indicate an unexpected CWD, not incorrect Python syntax

File paths independent of the working directory

A relative path such as "reports/summary.txt" normally starts from the current working directory (CWD). If the same script is launched from another folder, Python may look for that file in the wrong place.

A different approach is to first **determine where the Python file itself is located**, and then construct all project file paths **from that fixed location**. This does not change the CWD.

THE SAME SCRIPT, TWO LAUNCH LOCATIONS

```
cd project  
python report.py
```

```
cd ..  
python project/report.py
```

Both can write into project/reports/ when the program builds paths from the script directory.

Key idea: the CWD depends on how the process is launched; the script location does not.

Locating the script: `__file__` and `pathlib`

Python provides `__file__` in ordinary `.py` scripts and imported modules. It identifies **the path of that particular Python file**. The `pathlib` module represents paths as special `Path` objects and lets us manipulate them without assembling path strings manually.

```
from pathlib import Path

SCRIPT_PATH    = Path(__file__).resolve() # get abs. path of this file
SCRIPT_DIR     = SCRIPT_PATH.parent      # get path of parent folder
DATA_FILE_PATH = SCRIPT_DIR / "data" / "sales.csv" # construct a target path
```

`Path(__file__)` → create a `Path` object for this file
`.resolve()` → return its absolute, resolved path
`.parent` → path to the folder containing this file
`/ "data"` → joins a new path component (not division)

Reading a file relative to the script

Suppose `sales.csv` is stored in a `data` folder next to `report.py`. Construct its full path from `SCRIPT_DIR` and then pass that `Path` object directly to `open()`

PROJECT STRUCTURE

```
project/  
├── report.py  
└── data/  
    └── sales.csv
```

PYTHON CODE

```
from pathlib import Path  
  
SCRIPT_DIR = Path(__file__).resolve().parent  
file_path = SCRIPT_DIR / "data" / "sales.csv"  
  
with open(file_path, "r") as f:  
    contents = f.read()
```

Unlike `open("data/sales.csv")`, this path is *absolute*, therefore it is not interpreted relative to the process CWD.

The same technique works when opening files for writing in a known position.

Creating an output folder next to the script

A program can also create its own output folder beside the Python file. First build a Path to that folder; then call `mkdir()` to create it if necessary

```
from pathlib import Path

SCRIPT_PATH = Path(__file__).resolve()
SCRIPT_DIR  = SCRIPT_PATH.parent
REPORT_DIR  = SCRIPT_DIR / "reports"

REPORT_DIR.mkdir(parents=True, exist_ok=True)
```

WHAT THE NEW OPERATIONS DO

`SCRIPT_DIR / "reports"` → builds the desired destination path; it does not create a folder

`mkdir(...)` → creates the folder on disk

`parents=True` → also creates missing parent directories

`exist_ok=True` → does not raise an error when the folder already exists

Example: one report, several output files

The same `REPORT_DIR` can be reused for different output files. Every generated file goes into `project/reports/`, regardless of the CWD. Existing files with the same names are overwritten when opened in "w" mode.

COMPLETE EXAMPLE – `report.py`

```
from pathlib import Path
import csv
import json

REPORT_DIR = Path(__file__).resolve().parent / "reports"
REPORT_DIR.mkdir(parents=True, exist_ok=True)

with open(REPORT_DIR / "summary.txt", "w") as f:
    f.write("Sales report\nProducts: 2\n")

with open(REPORT_DIR / "results.txt", "w") as f:
    f.write("Details about sales\nTotal: 15\n")
```

Notice: `open()` also accepts `Path` objects, not only strings.

Result and practical limitations

OUTPUT FOLDER

```
project/  
├─ report.py  
└─ reports/  
    ├─ summary.txt  
    ├─ sales.csv  
    └─ metadata.json
```

TWO WAYS TO LAUNCH

```
# From inside project/  
> python report.py  
  
# From its parent folder  
> python project/report.py
```

Both commands produce the same output folder. The process CWD remains unchanged.

Scope and limitations: `__file__` normally exists in `.py` files, but not in ordinary Jupyter notebook cells. Imported modules have their own `__file__`. Writing beside the script also requires permission to modify that folder.

Tabular CSV

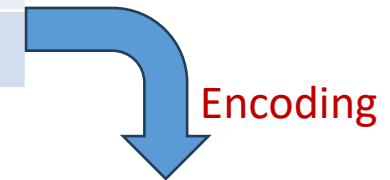
File Handling: TXT, CSV and JSON · Lecture 12

CSV is a table stored as text

- CSV (Comma Separated Values) stores rows of related fields separated by a delimiter (usually a comma)
- A **header** row names columns, making input meaning explicit.
- A spreadsheet can display CSV as a table, but the file itself is **plain text**
- Use the `csv` module, not naive `split(",")`, for general CSV handling.

Description	amount
Coffee	-3.5
Salary	1200
Rent	-450

Representation



Encoding

transactions.csv

description,amount

Coffee,-3.5

Salary,1200

Rent,-450

CSV cells are read as strings: amount must be converted

Opening a CSV file correctly

A CSV file is a text file. We open it with the familiar `open()` function, then give the file object to the `csv` module so it can interpret the records and fields.

OPENING A FILE

```
import csv

with open("sales.csv", "r", newline="") as file:
    reader = csv.reader(file)
```

WHY `newline=""`?

Text files can use different characters to mark line endings (e.g. depending on the Operating System). Setting `newline=""` avoids automatic newline translation, allowing the `csv` module to manage CSV record endings correctly. It also prevents extra blank lines in CSV output on some systems.

Encoding note: If characters look corrupted or reading fails, check the file encoding; setting the parameter `encoding="utf-8"` in `open()` may help when the file is UTF-8.

From an open file to CSV records

Opening the file does not read all its CSV records. `csv.reader(file)` creates a reader object; the for loop then reads and parses one record at a time.

FILE: sales.csv

```
Product,Quantity,Price
Book,3,12.50
Pen,10,1.20
```

WHAT THE LOOP RECEIVES

```
['Product', 'Quantity', 'Price']
['Book', '3', '12.50']
['Pen', '10', '1.20']
```

PYTHON CODE

```
import csv
with open("sales.csv", newline="") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Each row is a **list of strings**, including numeric-looking fields. Use `int()` or `float()` before doing calculations.

Why plain split can misread CSV

- A field can legally contain a comma if quoted
- The csv module understands quoting and delimiters
- A simple `line.split(",")` does not understand quotation marks
- Always prefer the standard parser for CSV files.

```
name,comment
```

```
Ana,"good, clear result"
```

```
# Proper parsed row:
```

```
["Ana", "good, clear result"]
```

CSV is simple to use, but not trivial to parse correctly.

Accessing CSV columns: position or name

With `csv.reader`, fields are accessed by their **positions** (since it returns lists). This is convenient when the column order is known and fixed; otherwise, **header names** are easier to maintain → Dictionaries.

`csv.reader` – INDEXES

```
with open("sales.csv", newline="") as file:
    reader = csv.reader(file)
    next(reader) # skip header
    for row in reader:
        product = row[0]
        price = float(row[2])
```

`csv.DictReader` – HEADER NAMES

```
with open("sales.csv", newline="") as file:
    reader = csv.DictReader(file)
    for row in reader:
        product = row["Product"]
        price = float(row["Price"])
```

`DictReader` uses the first CSV row as field names (to be used as keys). Each following record becomes a dictionary, for example:

```
{"Product": "Book", "Quantity": "3", "Price": "12.50"}
```

Now you can access the value using the field name as a key.

If columns are reordered but their headers remain unchanged, `DictReader` code still accesses the correct fields.

Accessing CSV columns: position or name

With `csv.reader`, fields are accessed by their **positions** (since it returns are lists). This is convenient when the column order is known and fixed; otherwise, **header names** are easier to maintain → Dictionaries.

`csv.reader` – INDEXES

```
with open("sales.csv", newline="") as file:
    reader = csv.reader(file)
    next(reader) # skip header
    for row in reader:
        product = row[0]
        price = float(row[2])
```

`csv.DictReader` – HEADER NAMES

```
with open("sales.csv", newline="") as file:
    reader = csv.DictReader(file)
    for row in reader:
        product = row["Product"]
        price = float(row["Price"])
```

`DictReader` uses the first CSV row as field names (to be used as keys). Each following record becomes a dictionary, for example:

```
{"Product": "Book", "Quantity": "3", "Price": "12.50"}
```

Now you can access the fields using the field name as a key.

If columns are reordered but their headers remain unchanged, `DictReader` code still accesses the correct fields.

Remember: CSV values are stored as strings until you convert them

Accessing CSV columns: position or name

With `csv.reader`, fields are accessed by their **positions** (since it returns are lists). This is convenient when the column order is known and fixed; otherwise, **header names** are easier to maintain → Dictionaries.

`csv.reader` – INDEXES

```
with open("sales.csv", newline="") as file:
    reader = csv.reader(file)
    next(reader) # skip header
    for row in reader:
        product = row[0]
        price = float(row[2])
```

`csv.DictReader` – HEADER NAMES

```
with open("sales.csv", newline="") as file:
    reader = csv.DictReader(file)
    for row in reader:
```

`DictReader` uses the first CSV row as field names (to be used as a dictionary, for example:

```
{"Product": "Book", "Quantity": "3", "Price": "1.20"}
```

Now you can access the value using the field name as a key.

- NOTE: the first row, using `.reader()`, is the header with the column names
- We have to find a way to skip it and process data
- `next()` is a built-in function to skip an element of an iterable
- This is the shortest way, but you may find another e.g. keep a counter/flag to skip the first loop iteration or create a list and `l[1:]`
- Since `reader` is not a list we cannot do directly `reader[1:]`

Writing CSV: `csv.writer` vs `csv.DictWriter`

We have the same possibilities with writing: `csv.writer` VS `csv.DictWriter`. Both tools write rows into a CSV file. The difference is how we provide each record: ordered values or named fields.

`csv.writer` – rows as lists

```
import csv

with open("sales.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Product", "Qty", "Price"])
    writer.writerow(["Book", 3, 12.5])
```

Columns follow the position of each value. The header is just another row, written explicitly using `writerow()`.

`csv.DictWriter` – rows as dictionaries

```
import csv

with open("sales.csv", "w", newline="") as f:
    writer = csv.DictWriter(f,
                           fieldnames=["Product", "Qty", "Price"])
    writer.writeheader()
    writer.writerow({"Product": "Book",
                    "Qty": 3, "Price": 12.5})
```

Columns follow `fieldnames` (declared in advance in calling `DictWriter`). Dictionary keys identify the values; `writeheader()` writes the header, `writerows()` writes values in the correct field.

Both produce the same file: Product,Qty,Price → Book,3,12.5

Writing CSV: csv.writer vs csv.DictWriter

We have the same possibilities with writing: `csv.writer` VS `csv.DictWriter`. Both tools write rows into a CSV file. The difference is how we provide each record: ordered values or named fields.

`csv.writer` – rows as lists

```
import csv

with open("sales.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Product", "Qty", "Price"])
    writer.writerow(["Book", 3, 12.5])
```

Column headers and row values must be in the same order. Up to the programmer taking care of this.

`csv.DictWriter` – rows as dictionaries

```
import csv

with open("sales.csv", "w", newline="") as f:
    writer = csv.DictWriter(f,
                           fieldnames=["Product", "Qty", "Price"])
    writer.writeheader()
    writer.writerow({"Product": "Book",
                    "Qty": 3, "Price": 12.5})
```

Column headers and row values are managed automatically, according to the declared fieldnames.

Both produce the same file: Product,Qty,Price → Book,3,12.5

CSV writers: methods and output

Remember: A writer object does not save a record until you call a writing method.

Both writer types support `writerow()` and `writerows()` (a full list/tuple of rows, one shot)

Writing one or multiple records

```
# case csv.writer: a sequence of rows
writer.writerow(["Book", 3, 12.5])
writer.writerows([
    ["Pen", 10, 1.2],
    ["Pencil", 5, 0.8]
])
```

The example is for `csv.writer`.

For `DictWriter`, `writerow()` takes one **dictionary** and `writerows()` takes **an iterable of dictionaries**. Only `DictWriter` has `writeheader()`.

Resulting sales.csv

```
Product,Qty,Price
Book,3,12.5
Pen,10,1.2
Pencil,5,0.8
```

Both writers produce ordinary CSV text. Lists and dictionaries are Python input structures, not formats stored in the CSV.

File mode "w" creates or overwrites the CSV. Use "a" to append records, avoiding a second header in an existing file.

Appending CSV data

Data collection programs often run repeatedly. Each execution **adds a new record** to the same CSV file, **preserving previous records** (data collection, logs...). This is what file mode "a" (append) is for.

In general, this implies a different behaviour depending on the state of the file (Exist/Do not exist)

FIRST RUN – FILE CREATED

```
Product,Quantity,Price  
Book,3,12.50
```

SECOND RUN – NEW ROW APPENDED

```
Product,Quantity,Price  
Book,3,12.50  
Pen,10,1.20
```

The header identifies the columns and belongs only at the beginning of the file. When appending later records, write only the data row.

If the header is appended again, a subsequent `csv.DictReader` will treat it as ordinary data: for example, "Quantity" would appear where a numerical quantity is expected.

Appending CSV records: a complete example

Before appending, check whether the CSV already exists. A newly created file needs a header; an existing file needs only the new record.

PYTHON CODE – add_sale.py

```
from pathlib import Path
import csv

path = Path("sales.csv")
new_file = not path.exists()

product = input("Product: ")
quantity = int(input("Quantity: "))
price = float(input("Price: "))

with open(path, "a", newline="") as file:
    writer = csv.writer(file)
    if new_file:
        writer.writerow(["Product", "Quantity", "Price"])
    writer.writerow([product, quantity, price])
```

WHAT HAPPENS

1. `.exists()` (method of Path objects) checks whether the file is already present
2. Mode "a" creates the file if missing; otherwise, it adds to its end
3. Write the header only for a new file
4. Write the new sales record on every run.

Assumption: an existing `sales.csv` already has the correct header. The path here is relative to the CWD.

Structured JSON

Storing persistent dictionaries · Lecture 12

JSON stores structured values

- JSON originated in web development and remains widely used for exchanging structured data between client-server applications, although it is now widely used beyond web applications
- JSON supports objects, arrays, strings, numbers, Booleans and null
- Python dictionaries and lists map naturally to objects and arrays
- JSON is text: it can be inspected outside Python
- Unlike CSV, nested structure is directly represented.

```
{  
  "owner": "Alice",  
  "assets": [  
    {"name": "Bond", "units": 3},  
    {"name": "Stock", "units": 5}  
  ]  
}
```

Think of JSON as a persistent nested dictionary/list.

Python and JSON types

- Python dict ↔ JSON object; list ↔ array
- Python str ↔ string; int/float ↔ number
- Python True/False ↔ true/false; None ↔ null.
- Tuples may serialize as arrays and come back as lists.

PYTHON	JSON
{"a": 1}	{"a": 1}
[1, 2]	[1, 2]
True	true
None	null

- when you serialize a tuple it becomes an array
- when you read it again it is converted into a list

Not every Python object is directly JSON serializable.

json.dump: save to a file

- `json.dump(object, file)` writes a JSON representation into an open text file
- `indent=4` makes nested data easier for humans to read (otherwise it is written in a single line)
- The file can be re-opened on another execution, it is persistent
- The structure is *serialized*; it is not a live reference to the original object.
- If you do not want to write a file, but only retrieve the json text as a string, use `.dumps()`

```
import json
with open("portfolio.json", "w",
          encoding="utf-8") as f:
    json.dump(portfolio, f, indent=4)

# get the json string
json_string = json.dumps(portfolio, indent=4)
print("written in 'portfolio.json':")
print(json_string)
```

`.dump` writes to a file; `.dumps` returns a string.

json.load: reconstruct Python data

- `json.load(file)` parses a JSON file and produces Python values
- After loading, access keys and iterate lists normally
- Automatically, types are converted in:
 - JSON object (key-value) → dict; array → list
 - string → str number → int/float
 - true/false → True/False; null → None
- Reading **does not automatically update the file**; save again to persist edits
- Inspect the loaded type when unsure.

```
with open("portfolio.json", encoding="utf-8") as f:  
    data = json.load(f)  
  
print(data["owner"])  
print(data["assets"][0]["name"])
```

load reads from a file; loads parses a string.

Choosing TXT, CSV or JSON

EXPLANATION

- TXT suits notes and short human-readable reports
- CSV suits repeated rows with the same flat columns
- JSON suits configurations and nested records with varied fields
- Choose by data shape and intended consumers, not fashion.

EXAMPLE / STRUCTURE

TXT → "Total: 18"

CSV → rows × columns

JSON → nested dicts/lists

There is no universally best format.

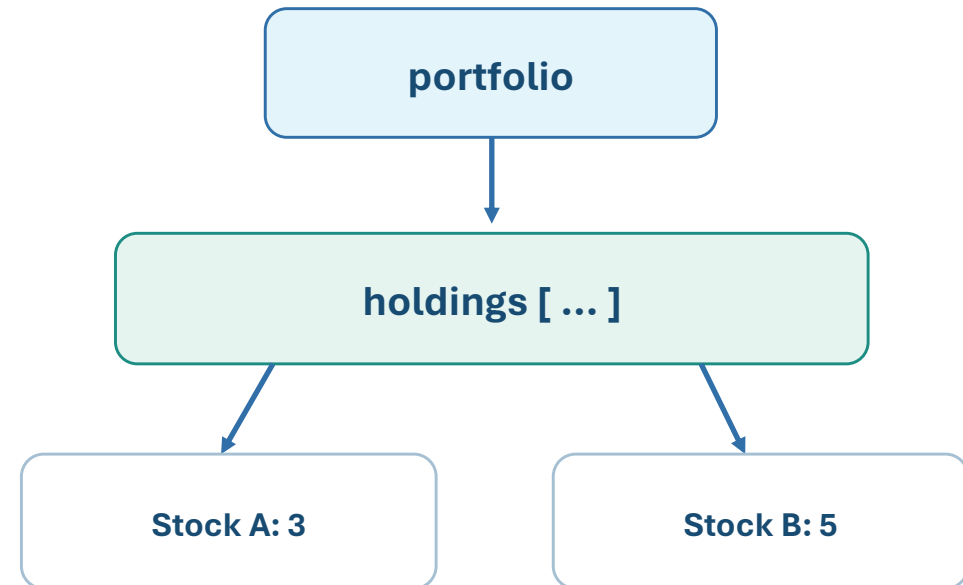
CSV vs. JSON: choose the shape that fits your data

CSV represents flat rows; JSON can preserve nested lists and dictionaries.

CSV · one row per record

Name	Balance	Opening date	Active
Ana	100	2026-06-11	True
Bob	75	2001-03-05	False
Clara	120	2003-03-13	False
Diogo	55	1996-01-02	True

JSON · hierarchical values



TXT for free-form notes, CSV for flat records, JSON for structured nested data.

Data-processing pipeline

- A robust program follows a clear data path
- Read external text → parse fields → convert types → compute → write output
- Each step has separate assumptions and possible failure conditions
- An output file is a deliverable: verify it, not just the console message.



Lecture 13 adds systematic error handling to each step.

Pathlib and FileSystem checks

Some details about Path from pathlib

- Path represents a filesystem path as an object
- `Path.exists()` and `Path.is_file()` can inspect files and directories:
 - `Path.is_file()` checks whether a path points to an existing regular **file**. It returns True if the file exists and is a file (not a directory), and False otherwise
 - Unlike `Path.exists()`, which checks whether a path exists regardless of its type (file or directory), `Path.is_file()` specifically checks for a file
- Paths can be combined with `/` instead of manual separators

```
from pathlib import Path
p = Path("data") / "sales.csv"
print(p.exists())

if p.is_file():
    print("File found!")
else:
    print("File not found (or not a regular file).")

# Path("data").is_file() -> False, it is a directory
```