

Day 14 Files and data formats

Self-Study Guide · About 80 minutes · With answer key

By the end you will be able to

- Read a text file all at once or line by line, and write or add to one, using `with` so the file is always closed.
- Build file paths with `pathlib.Path` and open files with `encoding="utf-8"` so text looks the same on every computer.
- Read and write CSV files with the `csv` module, and save and load lists and dictionaries as JSON.
- Make a program keep its data between runs by loading a file when it starts and saving after every change.

What to remember from earlier

- `strip()` gives back a copy of a string without spaces, tabs, or new lines at either end. ([Day 3: Strings, 3.6 String Methods](#))
- `split()` breaks a string into a list of pieces at whitespace, or at a separator you give it. ([Day 3: Strings, 3.7 Splitting and Joining](#))
- `enumerate(items, 1)` gives each item together with a number that starts at 1. ([Day 5: Loops, 5.11 enumerate\(\): Index and Value Together](#))
- A dictionary maps keys to values, and a list of dictionaries can hold many records of the same shape. ([Day 8: Dictionaries, 8.8 Lists of Dictionaries](#))
- A function can return a value to its caller, and a docstring says what the function does. ([Day 9: Functions, part 1](#))
- The Day 12 to-do manager keeps its tasks in a list of dictionaries, each with a title and a done flag. ([Day 12: Checkpoint project 1](#))
- Day 13 split the to-do manager into `todo_app/tasks.py`, which holds the task functions, and `main.py`, which runs the menu loop. ([Day 13: Modules, packages and environments](#))
- `import module` and `from module import name` load code from another file, including the standard library. ([Day 13: Modules, packages and environments](#))

Make a `day14` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days. Several programs today read a small data file, such as `poem.txt` or `class_scores.csv`. Each one is shown on the page; type it into a file with that exact name **in the same folder** as the script, or download it with the example.

Every program you have written so far forgets everything when it stops. The Day 12 to-do manager is a good example: add five tasks, choose Quit, run it again, and the list is empty. Variables live in the computer's **memory**, which is wiped when the program ends. To keep anything, a program has to write it somewhere that lasts: a file. Today you learn to read files, write them, find them with paths, and store data in two formats that other programs understand, CSV and JSON. By the end, the to-do manager will remember its tasks.

14.1 Opening and Reading a File

You already use files every day: documents, photos, and the `.py` scripts you have written since Day 1. A **file** is a named block of data saved on a disk, where it stays after programs stop and even after the computer is switched off. The part of the name after the last dot, such as `.txt` or `.py`, is the **extension**; it hints at what kind of data is inside. Today's programs work with **text files**: files that hold characters you could open and read in any editor, such as `.txt`, `.csv`, and `.json` files, and your own `.py` scripts.

Here is the text file used in the next few sections. Save it as `poem.txt` in your `day14` folder:

```
Roses are red,  
violets are blue,  
Python reads files,  
and so can you.
```

Reading a file takes three steps, like reading a book: open it, read it, close it.

1. The built-in function `open()` takes a file name and returns a **file object**: a value that is connected to the file and has methods for reading from it or writing to it. Opening does not read anything yet.
2. The file object's `read()` method reads the whole file and returns it as one string.
3. The `close()` method disconnects the file object from the file. You will see why that matters in 14.2.

WORKED EXAMPLE Reading a whole file

`read_poem.py`

```
1 file = open("poem.txt", encoding="utf-8")  
2 text = file.read()  
3 file.close()  
4 print(text)  
5 print("Characters:", len(text))
```

```
Roses are red,  
violets are blue,  
Python reads files,  
and so can you.
```

```
Characters: 69
```

- Line 1 opens `poem.txt` and stores the file object in `file`. The second argument, `encoding="utf-8"`, tells Python how the characters are stored on the disk. Always include it; 14.7 explains what it means and what goes wrong without it.
- Line 2 reads the whole file into `text`, a normal string. From here on, every Day 3 string tool works on it.
- Line 3 closes the file. `text` still holds the string: closing ends the connection to the file, not the string you already read.
- Line 4 prints the poem. Notice the blank line before `Characters`. The file ends with a new line character, `"\n"` (Day 3), after `you.`, so the string ends with one too, and then `print` adds its own new line.

- Line 5 counts 69 characters. The four lines have 14, 17, 19, and 15 visible characters, which makes 65, and each line ends with a `"\n"`: 4 more. A new line is a character like any other.

How does `open("poem.txt")` know where `poem.txt` is? A name with no folder in front of it is looked up in the program's **current working directory**: the folder the terminal is in when you type `python read_poem.py`. Since you open the terminal in your `day14` folder, that is where Python looks. (14.6 shows how to find a file next to the script wherever you run it from.)

If no file has that name in that folder, `open()` stops the program. Here the name has a typo, `.text` instead of `.txt`:

`missing_file.py`

```
1 file = open("poem.text", encoding="utf-8")
2 text = file.read()
3 file.close()
4 print(text)
```

Traceback (most recent call last):

```
File ".../day14/missing_file.py", line 1, in <module>
    file = open("poem.text", encoding="utf-8")
FileNotFoundError: [Errno 2] No such file or directory: 'poem.text'
```

Read the last line first, as on Day 1: a `FileNotFoundError` names the file Python could not find. (`[Errno 2]` is the operating system's code number for "no such file".) Check three things: the spelling, including the extension; that the file is in the folder you ran the program from; and the capitals, because on Linux `Poem.txt` and `poem.txt` are two different files. Day 15 shows how a program can catch this error and carry on.

KEY IDEA

`open(name, encoding="utf-8")` returns a file object, `read()` returns the whole file as one string, and `close()` ends the connection. A plain file name is looked up in the folder you ran the program from.

WATCH OUT

Forgetting that the file's text ends with a new line. Most text files end with `"\n"`, so `text` ends with one too. Printing it adds a blank line, and comparing it with `"and so can you."` fails because of the hidden character. When you need clean text, `text.strip()` (Day 3) removes whitespace, including new lines, from both ends.

PAUSE AND TRY

Question 1 Tracing

What does this program print? Use `poem.txt` above.

first_line.py

```
1 file = open("poem.txt", encoding="utf-8")
2 text = file.read()
3 file.close()
4 print(text[:5])
5 print(text.count("\n"))
```

14.2 The with Statement

`read_poem.py` called `close()` by hand. That works until something goes wrong between `open` and `close`. If line 2 stopped the program with an error, line 3 would never run, and the file would stay open. Leaving files open causes real problems: the operating system limits how many files one program may have open at once; on Windows, an open file often cannot be renamed or deleted; and when you **write** a file, the text may not reach the disk until the file is closed (14.4 shows this happening). You need a way to say "close this file when I am done with it, whatever happens".

The **with statement** opens a file, runs an indented block, and closes the file automatically when the block ends, even if the block ends with an error:

```
1 with open("poem.txt", encoding="utf-8") as file:
2     statement
3     statement
```

- `with` is a keyword, followed by the expression that opens the file.
- `as file` names the file object, exactly as `file = open(...)` did.
- The colon and the indented block follow the same rules as `if` and `for`. The block is where you use the file.
- When Python leaves the block, by reaching its end or because of an error, it closes the file for you. There is no `close()` to forget.

A file object has an **attribute** called `closed`: a value that belongs to the object, written after a dot like a method but with no parentheses. It is `True` once the file is closed. Watch it change:

WORKED EXAMPLE with closes the file for you

with_poem.py

```
1 with open("poem.txt", encoding="utf-8") as file:
2     text = file.read()
3     print("Inside:", file.closed)
4
5 print("After:", file.closed)
6 print(text.upper())
```

```
Inside: False
After: True
ROSES ARE RED,
VIOLETS ARE BLUE,
PYTHON READS FILES,
AND SO CAN YOU.
```

- Line 1 opens the file and names it `file`. Line 2 reads it into `text`.
- Line 3 is inside the block, so the file is still open: `False`.
- Line 5 is back at the left edge, after the block, so Python has already closed the file: `True`.
- Line 6 still works, because `text` is an ordinary string that was read while the file was open. Everything the program needs from the file is read **inside** the block; everything done with the result can happen after it.

From now on, every program in this course opens files with `with`. It is shorter than `open` plus `close`, and the file is closed even when something goes wrong.

KEY IDEA

`with open(...) as file:` closes the file as soon as its block ends, even after an error. Do all your reading or writing inside the block.

WATCH OUT

Reading after the block has ended. Once the block is over, the file is closed, and a closed file cannot be read:

`closed_read.py`

```
1 with open("poem.txt", encoding="utf-8") as file:
2     first = file.read()
3
4 again = file.read()
5 print(again)
```

```
Traceback (most recent call last):
  File ".../day14/closed_read.py", line 4, in <module>
    again = file.read()
ValueError: I/O operation on closed file.
```

"I/O" means input/output: reading and writing. Line 4 is outside the block, so `file` is closed. Either move the line into the block, or use the string you already read, `first`.

PAUSE AND TRY

Question 2 Multiple choice

Why is `with open("poem.txt", encoding="utf-8") as file:` better than `file = open("poem.txt", encoding="utf-8")` followed later by `file.close()`?

- A. `with` reads the whole file automatically, so you do not need `read()`

- B. `with` closes the file when its block ends, even if an error stops the block partway
- C. `with` keeps the file open for the rest of the program, so you can read it again later
- D. `with` is the only way to open a file that is not in the current folder

14.3 Reading Line by Line

`read()` gives you the whole file as one big string. Many files are made of **lines** that each mean something on their own: one task, one score, one log entry. You want to handle them one at a time, and for a large file you do not want to hold the whole thing in memory at once.

A file object works like a sequence of lines: a **for loop over a file** runs its body once per line, with the loop variable holding that line as a string. Python reads the file a piece at a time as the loop goes, so even a huge file uses little memory. The REPL shows what each line really contains, because it displays strings with their escape characters (Day 3):

```
>>> file = open("poem.txt", encoding="utf-8")
>>> file.readline()
'Roses are red,\n'
>>> file.readline()
'violets are blue,\n'
>>> file.close()

>>> with open("poem.txt", encoding="utf-8") as file:
...     lines = file.readlines()
...
>>> lines
['Roses are red,\n', 'violets are blue,\n', 'Python reads files,\n', 'and so can you.\n']
>>> len(lines)
4
```

- `readline()` reads **one** line each time you call it. The REPL shows `'Roses are red,\n'`: each line keeps the `"\n"` that ended it in the file.
- `readlines()` reads every line into a list (Day 6), one string per line, each still ending in `"\n"`.

That `"\n"` is the thing to watch. Here is the most natural loop:

`lines_double.py`

```
1 with open("poem.txt", encoding="utf-8") as file:
2     for line in file:
3         print(line)
```

```
Roses are red,
violets are blue,
Python reads files,
and so can you.
```

The poem comes out double-spaced. Each `line` already ends with `"\n"`, and `print` adds a second one. Remove the line's own new line with `strip()` (Day 3) before using it. Together with `enumerate()` from Day 5, this numbers the lines:

WORKED EXAMPLE Numbering the lines of a file

lines_numbered.py

```
1 with open("poem.txt", encoding="utf-8") as file:
2     for number, line in enumerate(file, 1):
3         print(f"{number}: {line.strip()}")
```

```
1: Roses are red,
2: violets are blue,
3: Python reads files,
4: and so can you.
```

Pass	number	line	line.strip()
1	1	"Roses are red,\n"	"Roses are red,"
2	2	"violets are blue,\n"	"violets are blue,"
3	3	"Python reads files,\n"	"Python reads files,"
4	4	"and so can you.\n"	"and so can you."

`enumerate(file, 1)` gives each line with a number starting at 1, exactly as it did for the characters of a string. The loop ends by itself after the last line.

Three ways to read, side by side:

Code	Gives you	Use it when
<code>file.read()</code>	The whole file as one string	The file is small and you want it all
<code>for line in file:</code>	One line at a time, each ending in <code>"\n"</code>	You handle each line on its own (usual)
<code>file.readlines()</code>	A list of all the lines, each ending in <code>"\n"</code>	You need the lines as a list, for example to count or index them
<code>file.read().splitlines()</code>	A list of all the lines, without <code>"\n"</code>	You want a list of clean lines

The last row uses the string method `splitlines()`, which splits a string at its line breaks and drops them. It is like `split("\n")`, but it does not leave an empty string at the end when the text ends with a new line.

KEY IDEA

`for line in file:` reads a file one line at a time. Each line keeps its `"\n"` ending, so call `line.strip()` before printing or comparing it.

WATCH OUT

Reading the file twice. A file object remembers how far it has read, like a bookmark. After `read()` or a full `for` loop, the bookmark is at the end, so a second `read()` inside the same

block returns the empty string "" and a second loop over the same file runs zero times. Read the data once into a variable (a string or a list) and use that variable as often as you need, or open the file again.

PAUSE AND TRY

Question 3 Tracing

What does this program print? Count carefully: `line.strip()` has no "\n".

long_lines.py

```
1 with open("poem.txt", encoding="utf-8") as file:
2     for number, line in enumerate(file, 1):
3         if len(line.strip()) > 15:
4             print(number, line.strip())
```

14.4 Writing a File

Reading lets a program use data someone else saved. To keep its **own** data, a program must write. `open()` takes a second argument, the **mode**: a short string that says what you plan to do with the file. Without it, the mode is "r", read. For writing, the mode is "w":

```
1 with open("shopping.txt", "w", encoding="utf-8") as file:
2     file.write("eggs\n")
```

Mode "w" does two things at once. If the file does not exist, it **creates** it. If it does exist, it **empties it first**, throwing away everything that was in it. Then the file object's `write()` method adds the string you give it to the file.

WORKED EXAMPLE Writing a shopping list, then reading it back

write_shopping.py

```
1 with open("shopping.txt", "w", encoding="utf-8") as file:
2     file.write("eggs\n")
3     file.write("bread\n")
4     file.write("milk\n")
5
6 with open("shopping.txt", encoding="utf-8") as file:
7     print(file.read())
```

```
eggs
bread
milk
```

- Line 1 opens `shopping.txt` for writing, creating it in your `day14` folder. Open the folder after running the program and you will find it there; open it in your editor and you will see the three lines.

- Lines 2 to 4 each write one string. Each string ends in `"\n"`, so each item lands on its own line.
- The block ends, so the file is closed and everything written is safely on the disk.
- Lines 6 and 7 open the same file again, this time for reading, and print what it holds. Reading back what you wrote is a good habit while you learn: it shows exactly what is in the file.

Run the program twice. The output is the same both times, because `"w"` empties the file before writing.

`write()` does not work like `print()`. It writes exactly the characters you give it, nothing more. In particular, it does **not** add a new line:

`write_no_newline.py`

```
1 with open("shopping.txt", "w", encoding="utf-8") as file:
2     file.write("eggs")
3     file.write("bread")
4     file.write("milk")
5
6 with open("shopping.txt", encoding="utf-8") as file:
7     print(file.read())
```

```
eggsbreadmilk
```

The three words run together into one line, because no `"\n"` was ever written. Each `write()` continues right where the last one stopped.

`write()` also accepts only strings. To save numbers, turn them into text first, and an f-string (Day 3) is the neatest way. A loop over a dictionary's items (Day 8) writes one line per entry:

`write_prices.py`

```
1 prices = {"apples": 3.5, "rice": 2.25, "tea": 4}
2
3 with open("prices.txt", "w", encoding="utf-8") as file:
4     for item, price in prices.items():
5         file.write(f"{item}: ${price:.2f}\n")
6
7 with open("prices.txt", encoding="utf-8") as file:
8     print(file.read())
```

```
apples: $3.50
rice: $2.25
tea: $4.00
```

Each pass builds one line with the price shown to two decimal places and a `"\n"` at the end.

If you try `write()` in the REPL, it also shows a number after each call: `write()` returns how many characters it wrote, and a script simply ignores it.

KEY IDEA

Mode `"w"` creates the file, or empties it if it already exists. `file.write(text)` adds exactly text: it takes only strings and never adds a new line, so end each line with `"\n"` yourself.

WATCH OUT

Reading back before the file is closed. To be fast, Python collects small writes in memory, in a **buffer**, and sends them to the disk in bigger pieces, at the latest when the file is closed. Until then, another file object may not see them:

unclosed_write.py

```
1 file = open("draft.txt", "w", encoding="utf-8")
2 file.write("Dear Sam,\n")
3
4 reader = open("draft.txt", encoding="utf-8")
5 print("Read back:", len(reader.read()), "characters")
6 reader.close()
7
8 file.close()
9
10 reader = open("draft.txt", encoding="utf-8")
11 print("After close:", len(reader.read()), "characters")
12 reader.close()
```

```
Read back: 0 characters
After close: 10 characters
```

The first read, on line 5, finds an empty file: "Dear Sam,\n" is still waiting in the buffer. After `file.close()` on line 8, the same read finds all 10 characters. This is one more reason to write inside a `with` block and read only after it ends, as `write_shopping.py` does.

PAUSE AND TRY**Question 4** Debugging

This program should save three scores, one per line. It stops with an error instead. Quote the error, explain it, and fix the program.

write_scores_bug.py

```
1 scores = [72, 88, 95]
2
3 with open("scores_out.txt", "w", encoding="utf-8") as file:
4     for score in scores:
5         file.write(score)
```

14.5 Appending and Choosing a Mode

A training log gets one new line every day. Mode `"w"` is wrong for that job: each run would empty the file and keep only today's entry. You need to add to the end and keep what is already there.

Mode `"a"`, **append**, opens a file for writing at its end. Nothing is erased; each `write()` adds after the last character already in the file. If the file does not exist yet, `"a"` creates it, just as `"w"` does.

WORKED EXAMPLE A log that grows

append_log.py

```

1 with open("log.txt", "w", encoding="utf-8") as file:
2     file.write("Training log\n")
3
4 with open("log.txt", "a", encoding="utf-8") as file:
5     file.write("Monday: ran 3 km\n")
6
7 with open("log.txt", "a", encoding="utf-8") as file:
8     file.write("Tuesday: swam 20 laps\n")
9
10 with open("log.txt", encoding="utf-8") as file:
11     print(file.read())

```

```

Training log
Monday: ran 3 km
Tuesday: swam 20 laps

```

Lines	Mode	File afterwards
1-2	"w"	Training log
4-5	"a"	Training log, Monday: ran 3 km
7-8	"a"	Training log, Monday: ran 3 km, Tuesday: swam...
10-11	"r"	Unchanged; printed

Lines 1 and 2 start the log fresh with "w", so this example prints the same thing every time you run it. A real log program would leave that part out and only append, so that each run adds one more line.

Change the two appends to "w" and watch the difference:

w_erases.py

```

1 with open("log.txt", "w", encoding="utf-8") as file:
2     file.write("Training log\n")
3     file.write("Monday: ran 3 km\n")
4
5 with open("log.txt", "w", encoding="utf-8") as file:
6     file.write("Tuesday: swam 20 laps\n")
7
8 with open("log.txt", encoding="utf-8") as file:
9     print(file.read())

```

```
Tuesday: swam 20 laps
```

The second open with "w" emptied the file, so the title and Monday's entry are gone. Nothing warns you: Python assumes you meant it.

Mode	If the file exists	If it does not exist
"r", read	Reads it from the start	FileNotFoundError
"w", write	Empties it , then writes	Creates it

"a", append	Keeps it, writes at the end	Creates it
-------------	-----------------------------	------------

"r" is the default, which is why the reading examples could leave the mode out. A file opened with "r" can only be read, and one opened with "w" or "a" can only be written; you will see the error in the review questions.

KEY IDEA

Choose the mode by asking what should happen to what is already in the file: read it ("r"), replace it ("w"), or keep it and add more ("a").

WATCH OUT

Using "w" to update a file you meant to keep. Opening an important file with "w" erases it at the moment of the `open`, before you write anything. If the program then crashes, the file stays empty. Use "a" to add lines; to change a file's contents, read it first, change the data in your program, and then write it all back with "w". 14.10 shows that pattern.

PAUSE AND TRY**Question 5** Multiple choice

A program contains only these two lines, and `visits.txt` does not exist yet. You run the program three times. What does `visits.txt` contain afterwards?

```
1 with open("visits.txt", "a", encoding="utf-8") as file:
2     file.write("visit\n")
```

- A. One line, `visit`
- B. Three lines, each `visit`
- C. Nothing: the program stops with `FileNotFoundError` on the first run
- D. One line, `visitvisitvisit`

14.6 Paths with `pathlib.Path`

So far every file name has been a plain name in the current folder. Real programs keep files in folders: `reports/march/sales.csv`. You could glue names together with strings, `"reports" + "/" + "march"`, but Windows writes folder separators as `\` while macOS and Linux use `/`, and string-gluing makes it easy to get doubled or missing separators.

A **path** is the location of a file or folder written as a chain of folder names ending in a name, such as `reports/march/sales.csv`. The standard library module `pathlib` (Day 13 showed how to import from the standard library) has a type, `Path`, for working with paths as values instead of strings. Build a path by joining parts with the `/` operator; Python uses the right separator for your computer.

WORKED EXAMPLE Taking a path apart

path_parts.py

```

1 from pathlib import Path
2
3 report = Path("reports") / "march" / "sales.csv"
4 print(report)
5 print(report.name)
6 print(report.stem)
7 print(report.suffix)
8 print(report.parent)

```

```

reports/march/sales.csv
sales.csv
sales
.csv
reports/march

```

- Line 1 imports Path from pathlib.
- Line 3 starts with Path("reports") and joins two more parts with /. On a Path, / means "inside this folder", not division.
- Line 4 prints the whole path. On Windows it shows reports\march\sales.csv: same path, written the Windows way.
- Lines 5 to 8 read four attributes (values with no parentheses, like closed in 14.2):

Attribute	Means	Here
.name	The last part: the file name	sales.csv
.stem	The name without its extension	sales
.suffix	The extension, including the dot	.csv
.parent	The path of the folder that holds it	reports/march

Making a Path never touches the disk. reports/march/sales.csv does not have to exist for these lines to work.

A Path also has methods that **do** use the disk. exists() asks whether a file or folder is really there. read_text() and write_text() open, read or write, and close a file in one call, which is handy for small files. And path.open() works just like open(path), with the same modes; open() itself also accepts a Path.

path_read_write.py

```
1 from pathlib import Path
2
3 poem = Path("poem.txt")
4 print(poem.exists())
5 print(Path("poems.txt").exists())
6
7 text = poem.read_text(encoding="utf-8")
8 print(text.splitlines()[0])
9
10 copy = Path("poem_copy.txt")
11 copy.write_text(text.upper(), encoding="utf-8")
12 print(copy.read_text(encoding="utf-8"))
```

```
True
False
Roses are red,
ROSES ARE RED,
VIOLETS ARE BLUE,
PYTHON READS FILES,
AND SO CAN YOU.
```

`poem.txt` is in the folder, so line 4 prints `True`; there is no `poems.txt`, so line 5 prints `False` (no error: `exists()` only asks). Line 7 reads the whole poem, and line 8 prints its first line using `splitlines()` from 14.3. Line 11 writes a capitalized copy to a new file, `poem_copy.txt`, replacing it if it exists, like mode `"w"`. `exists()` lets a program check for a file before opening it, which the to-do manager will use in 14.10.

Relative and absolute paths. `Path("poem.txt")` is a **relative path**: it is looked up starting from the current working directory, the folder you ran `python` from. An **absolute path** starts at the top of the disk (on macOS, something like `/Users/ana/day14/poem.txt`; on Windows, `C:\Users\ana\day14\poem.txt`) and means the same file wherever you run from. Writing your own absolute paths into a program is a bad idea, because the program then works only on your computer.

Relative paths cause a classic surprise. If your terminal is in the folder **above** `day14` and you run `python day14/read_poem.py`, Python finds the script, but `open("poem.txt")` looks in the folder you are in, not in `day14`, and you get `FileNotFoundError`. Some editors' Run buttons do the same. The fix is to build the path from the script's own location. When Python runs a script, it stores the script's path in a special variable, `__file__` (two underscores on each side, like `__name__` from Day 13). `Path(__file__).parent` is therefore the folder the script is in:

here.py

```
1 from pathlib import Path
2
3 here = Path(__file__).parent
4 poem = here / "poem.txt"
5
6 with poem.open(encoding="utf-8") as file:
7     lines = file.readlines()
8
9 print(poem.name, "has", len(lines), "lines")
```

poem.txt has 4 lines

Line 3 gets the script's folder, and line 4 joins `poem.txt` onto it. Now the program finds the poem whether you run it as `python here.py` from `day14` or as `python day14/here.py` from the folder above.

KEY IDEA

`Path("folder") / "file.txt"` builds a path that works on every operating system. `.name`, `.stem`, `.suffix`, and `.parent` take it apart, and `exists()`, `read_text()`, `write_text()`, and `open()` use the disk. `Path(__file__).parent / "name"` finds a file next to the script.

WATCH OUT

Expecting `exists()` or `/` to create anything. `Path("reports") / "sales.csv"` only describes a location. Writing to it with `"w"` creates the **file**, but not missing **folders**: if there is no `reports` folder, the open fails with `FileNotFoundError`. Create the folder first in your file manager or editor, or keep the file in an existing folder.

PAUSE AND TRY**Question 6** Tracing

What does this program print?

photo_path.py

```
1 from pathlib import Path
2
3 photo = Path("photos") / "2026" / "beach.jpeg"
4 print(photo.name)
5 print(photo.stem)
6 print(photo.suffix)
7 print(photo.parent.name)
```

14.7 Text Encodings and `encoding="utf-8"`

Every example so far has passed `encoding="utf-8"`. Here is why, and what happens without it.

A disk does not store characters. It stores **bytes**: small numbers from 0 to 255. To save text, each character must be turned into bytes, and to read text, the bytes must be turned back into characters. A **text encoding** is the rule for doing that. **UTF-8** is the encoding almost all files and web pages use today: it can store every character in every language, plus emoji, and plain English letters take one byte each, while other characters take two to four.

A string's `encode()` method shows the bytes an encoding would produce. Python displays bytes as `b'...'`, with each byte that is not a plain English character written as `\x` and two hexadecimal (base-16) digits:

```
>>> len("cafe")
4
>>> len("café")
4
>>> "cafe".encode("utf-8")
b'cafe'
>>> "café".encode("utf-8")
b'caf\xc3\xa9'
>>> len("café".encode("utf-8"))
5
```

"café" is 4 characters but 5 bytes in UTF-8: `c`, `a`, and `f` take one byte each, and `é` takes two, `\xc3\xa9`. That is the important point: the **same bytes mean different characters under different encodings**. If a file is written with one encoding and read with another, the text comes back wrong.

WORKED EXAMPLE Reading with the wrong encoding

wrong_encoding.py

```
1 from pathlib import Path
2
3 menu = Path("menu.txt")
4 menu.write_text("Café: crème brûlée\n", encoding="utf-8")
5
6 print(menu.read_text(encoding="utf-8"))
7 print(menu.read_text(encoding="cp1252"))
```

Café: crème brûlée

CafÃ©: crÃme brÃlÃe

- Line 4 writes the menu as UTF-8: each accented letter becomes two bytes.
- Line 6 reads it back as UTF-8, and the text is perfect.
- Line 7 reads the **same bytes** as `cp1252`, an older encoding that many Windows computers use by default. In `cp1252`, every byte is one character, so the two bytes of `é` turn into the two characters `Ã©`. No error is reported: the program simply shows nonsense. Garbled text like this is called **mojibake** (a Japanese word for "character transformation").

Sometimes the bytes do not make sense in the encoding at all, and Python stops. Here a file is written in `cp1252`, where `é` is the single byte `\xe9`, and read as UTF-8:

decode_error.py

```
1 from pathlib import Path
2
3 menu = Path("menu_old.txt")
4 menu.write_text("Caf  \n", encoding="cp1252")
5
6 print(menu.read_text(encoding="utf-8"))
```

Traceback (most recent call last):

...

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in position 3: invalid
continuation byte
```

A **codec** is Python's word for an encoding. The message says that byte number 3 (counting from 0: c, a, f, then   ) is not valid UTF-8. Older Python versions may word the end of the message a little differently.

So why does the encoding have to be written out? When you leave `encoding` out, `open()` uses your computer's **default** encoding. On macOS and Linux that is usually UTF-8, but on many Windows computers it is not. A program without `encoding="utf-8"` can work perfectly on one computer and show mojibake, or crash, on a friend's. Writing the encoding makes the program behave the same everywhere.

KEY IDEA

Files store bytes, and an encoding turns characters into bytes and back. Always pass `encoding="utf-8"` to `open()`, `read_text()`, and `write_text()`, so that your program reads and writes text the same way on every computer.

WATCH OUT

Thinking the encoding only matters for other languages. A file of plain English letters is the same in UTF-8 and cp1252, so a missing `encoding` can go unnoticed for months, until one name with an accent, a curly quote pasted from a word processor, or an emoji arrives. Then the file breaks. Include `encoding="utf-8"` every time, not only when you expect special characters.

PAUSE AND TRY**Question 7** Short answer

A classmate's program saves student names to a file. When the teacher opens the file, the name Jos   shows as Jos    . Explain what happened in terms of bytes and encodings, and say how to fix it.

14.8 CSV Files with the csv Module

A teacher's gradebook, a bank statement, a list of cities and their temperatures: data shaped like a **table**, with rows and columns, is everywhere. Spreadsheet programs can save a table as a CSV file, and almost every data tool can read one.

CSV stands for comma-separated values. A **CSV file** is a text file in which each line is one **row** of a table and commas separate the values in the row. The first row is usually a **header**: the names of the columns. Save this as `class_scores.csv`:

```
name,score
Ana,88
Ben,92
"Lopez, Maya",79
Sam,95
```

Look at the fourth line. A value that contains a comma is wrapped in double quotes, so the comma inside it is not a separator. That is the CSV rule, and it is why splitting each line at commas is not enough:

`split_csv_wrong.py`

```
1 with open("class_scores.csv", encoding="utf-8") as file:
2     for line in file:
3         print(line.strip().split(","))
```

```
['name', 'score']
['Ana', '88']
['Ben', '92']
['"Lopez', ' Maya"', '79']
['Sam', '95']
```

`split(",")` breaks "Lopez, Maya" into two pieces, with the quote marks still attached, and the row now has three values instead of two. Quotes inside quoted values, and line breaks inside values, make the rules more complicated still. The standard library module `csv` knows all of them.

`csv.reader(file)` goes through an open file and gives each row as a **list of strings**, one per column, with quotes handled. When you open a file for the `csv` module, pass `newline=""`: it tells `open()` to hand the line endings to the `csv` module untouched, so the module can deal with them itself. The `csv` documentation asks for this whenever you open a CSV file.

WORKED EXAMPLE Reading rows with `csv.reader`

`csv_reader.py`

```
1 import csv
2
3 with open("class_scores.csv", encoding="utf-8", newline="") as file:
4     rows = list(csv.reader(file))
5
6 print(rows[0])
7 for row in rows[1:]:
8     name = row[0]
9     score = int(row[1])
10    print(f"{name:12} {score:3}")
```

```
[ 'name', 'score' ]
Ana      88
Ben      92
Lopez, Maya 79
Sam      95
```

- Line 4 reads every row into a list with `list()`, so `rows` is a list of lists. Line 3's `with` block can then end, closing the file.
- Line 6 prints `rows[0]`, the header row.
- Line 7 loops over `rows[1:]`, a Day 6 slice of every row after the header.
- Lines 8 and 9 take the columns by index. "Lopez, Maya" arrives as one value with no quotes. The score arrives as the **string** "79": a CSV file is text, and the `csv` module never guesses that a value is a number. `int()` converts it.
- Line 10 lines up the columns with widths (Day 5), 12 characters for the name and 3 for the score.

Using `row[1]` means remembering which column is which. `csv.DictReader(file)` reads the header for you and gives each later row as a **dictionary** (Day 8) whose keys are the column names:

csv_dictreader.py

```
1 import csv
2
3 total = 0
4 count = 0
5 with open("class_scores.csv", encoding="utf-8", newline="") as file:
6     for row in csv.DictReader(file):
7         print(row)
8         total += int(row["score"])
9         count += 1
10
11 print(f"Average: {total / count:.1f}")
```

```
{'name': 'Ana', 'score': '88'}
{'name': 'Ben', 'score': '92'}
{'name': 'Lopez, Maya', 'score': '79'}
{'name': 'Sam', 'score': '95'}
Average: 88.5
```

The header row is not given to the loop; it supplied the keys. `row["score"]` says what it means, and the program keeps working if someone adds a column in the middle of the file. The values are still strings, so line 8 converts each score before adding it. The four scores total 354, and $354 / 4$ is 88.5.

Writing works the other way round. `csv.writer(file)` makes a writer object whose `writerow(values)` method writes one list as one row, adding commas, quotes where needed, and the line ending:

csv_write.py

```
1 import csv
2
3 honor_roll = [("Ben", 92), ("Lopez, Maya", 91), ("Sam", 95)]
4
5 with open("honor_roll.csv", "w", encoding="utf-8", newline="") as file:
6     writer = csv.writer(file)
7     writer.writerow(["name", "score"])
8     for name, score in honor_roll:
9         writer.writerow([name, score])
10
11 with open("honor_roll.csv", encoding="utf-8") as file:
12     print(file.read())
```

```
name,score
Ben,92
"Lopez, Maya",91
Sam,95
```

- Line 3 holds the rows as a list of tuples (Day 7), and line 8 unpacks each tuple into `name` and `score`.
- `writerow` accepts numbers and turns them into text itself; unlike `write()`, it is not limited to strings.
- `"Lopez, Maya"` comes out with quotes, because it contains a comma. The other names need none.
- Line 5 opens the file with `newline=""`. The `csv` module ends each row with its own line ending; without `newline=""`, Windows would add a second one, and the file would show a blank line between rows.

KEY IDEA

Open CSV files with `newline=""` and let the `csv` module split and join rows: `csv.reader` gives each row as a list, `csv.DictReader` as a dictionary keyed by the header, and `csv.writer` writes lists as rows. Every value read is a string; convert numbers yourself.

WATCH OUT

Doing math with a value straight from a CSV file. It looks like a number, but it is a string:

csv_sum_bug.py

```
1 import csv
2
3 total = 0
4 with open("class_scores.csv", encoding="utf-8", newline="") as file:
5     for row in csv.DictReader(file):
6         total += row["score"]
7
8 print("Total:", total)
```

```
Traceback (most recent call last):
  File ".../day14/csv_sum_bug.py", line 6, in <module>
    total += row["score"]
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

`total` is the int 0 and `row["score"]` is the string "88", and Python will not add the two. Convert with `int()` (or `float()` for decimals):

csv_sum_fixed.py

```
1 import csv
2
3 total = 0
4 with open("class_scores.csv", encoding="utf-8", newline="") as file:
5     for row in csv.DictReader(file):
6         total += int(row["score"])
7
8 print("Total:", total)
```

Total: 354

PAUSE AND TRY

Question 8 Tracing

What does this program print? Use `class_scores.csv` above.

csv_over_90.py

```
1 import csv
2
3 names = []
4 with open("class_scores.csv", encoding="utf-8", newline="") as file:
5     for row in csv.DictReader(file):
6         if int(row["score"]) >= 90:
7             names.append(row["name"])
8
9 print(len(names), names)
```

14.9 JSON Files with `json.dump` and `json.load`

CSV is perfect for one flat table, but much of a program's data is not flat. The to-do manager has a list of dictionaries; a game might save a dictionary holding a list of players, each with a list of items. Squeezing that into rows and columns is awkward, and every value would come back as a string.

JSON (JavaScript Object Notation, usually said "jay-son") is a text format for nested data. It looks almost exactly like Python's lists and dictionaries, which makes it easy to read:

JSON	Python	Notes
------	--------	-------

object {"key": value}	dict	Keys must be strings
array [1, 2, 3]	list	A Python tuple is saved as an array too
"text"	str	Always double quotes in JSON
42, 3.5	int, float	Numbers stay numbers
true, false	True, False	Lowercase in JSON
null	None	

The standard library module `json` turns Python values into JSON text and back. Turning a value into text that can be stored or sent is called **serializing** it. Two functions work with strings, which makes them easy to try in the REPL: `json.dumps(value)` returns the JSON text for a value, and `json.loads(text)` turns JSON text back into Python values (the `s` stands for string).

```
>>> import json
>>> book = {"title": "Holes", "pages": 233, "read": True, "rating": None}
>>> text = json.dumps(book)
>>> text
'{"title": "Holes", "pages": 233, "read": true, "rating": null}'
>>> type(text)
<class 'str'>
>>> back = json.loads(text)
>>> back
{'title': 'Holes', 'pages': 233, 'read': True, 'rating': None}
>>> back == book
True
```

`text` is a plain string. Inside it, `True` became `true` and `None` became `null`, as the table says. `json.loads` reads it back into a dictionary equal to the original.

Not everything survives the trip unchanged, and some values cannot make it at all:

```
>>> import json
>>> data = {"point": (3, 4), 7: "lucky"}
>>> json.loads(json.dumps(data))
{'point': [3, 4], '7': 'lucky'}
>>> json.dumps({"red", "blue"})
Traceback (most recent call last):
...
TypeError: Object of type set is not JSON serializable
```

- The tuple `(3, 4)` comes back as the **list** `[3, 4]`: JSON has only one kind of sequence.
- The int key `7` comes back as the **string** `'7'`: JSON object keys are always strings.
- A set cannot be saved at all. JSON has nothing like a set, so `dumps` stops with a `TypeError`. Convert it to a list first, `list(tags)`.

For files there are two matching functions without the `s`. `json.dump(value, file)` writes the JSON for `value` into an open file, and `json.load(file)` reads a whole JSON file and returns the Python value. Here are two separate programs. The first saves a reading log:

WORKED EXAMPLE One program saves, another loads

save_log.py

```
1 import json
2
3 reading_log = [
4     {"title": "Holes", "pages": 233, "finished": True},
5     {"title": "Wonder", "pages": 310, "finished": False},
6 ]
7
8 with open("reading_log.json", "w", encoding="utf-8") as file:
9     json.dump(reading_log, file, indent=2)
10
11 print("Saved", len(reading_log), "books")
```

Saved 2 books

`json.dump(reading_log, file, indent=2)` writes the list into the file. The keyword argument (Day 10) `indent=2` puts each item on its own line, indented by two spaces, so a person can read the file too. Without it, the whole list goes on one long line, which works just as well for Python. Open `reading_log.json` in your editor and you will see:

```
[
  {
    "title": "Holes",
    "pages": 233,
    "finished": true
  },
  {
    "title": "Wonder",
    "pages": 310,
    "finished": false
  }
]
```

Now a **different** program, run afterwards, reads it:

show_log.py

```
1 import json
2
3 with open("reading_log.json", encoding="utf-8") as file:
4     reading_log = json.load(file)
5
6 for book in reading_log:
7     mark = "x" if book["finished"] else " "
8     print(f"[{mark}] {book['title']} ({book['pages']} pages)")
```

```
[x] Holes (233 pages)
[ ] Wonder (310 pages)
```

`json.load(file)` returns a list of two dictionaries, exactly the shape `save_log.py` saved. `book["pages"]` is the int 233, not a string, and `book["finished"]` is the bool `True`: JSON keeps

numbers and `true/false` as they were. The data outlived the program that created it, which is the whole point of saving.

KEY IDEA

`json.dump(data, file, indent=2)` saves lists, dictionaries, strings, numbers, booleans, and `None` as readable JSON text; `json.load(file)` reads them back as the same Python types. Tuples come back as lists, and keys come back as strings.

WATCH OUT

Swapping the arguments of `json.dump`. The data comes first and the file second, as in "dump this data into that file". Reversed, Python tries to turn the file object into JSON:

`json_dump_order.py`

```
1 import json
2
3 settings = {"theme": "dark", "volume": 7}
4
5 with open("settings.json", "w", encoding="utf-8") as file:
6     json.dump(file, settings)
```

Traceback (most recent call last):

...

TypeError: Object of type TextIOWrapper is not JSON serializable

`TextIOWrapper` is the type of a text file object. The message means "I was asked to save a file object as JSON, and I cannot". Write `json.dump(settings, file)`.

PAUSE AND TRY

Question 9 Tracing

`team.json` holds this:

```
{
  "name": "Riverside Otters",
  "players": [
    { "name": "Ana", "goals": 4 },
    { "name": "Ben", "goals": 7 },
    { "name": "Kai", "goals": 2 }
  ]
}
```

What does this program print?

json_nested.py

```
1 import json
2
3 with open("team.json", encoding="utf-8") as file:
4     team = json.load(file)
5
6 print(team["name"])
7 print(len(team["players"]))
8 print(team["players"][1]["goals"])
9 total = sum(player["goals"] for player in team["players"])
10 print(total)
```

14.10 Keeping Data Between Runs

You now have every piece needed to make a program remember. The question is **when** to read and write. A program that keeps its data in a file follows a simple pattern:

1. **Load when the program starts.** If the save file exists, read the data from it. If it does not (the very first run), start with an empty or default value.
2. **Work with the data in memory**, in ordinary lists and dictionaries, exactly as before.
3. **Save after every change.** Write the whole data structure back to the file with mode `"w"`, replacing the old copy.

Putting load and save into two small functions keeps the rest of the program free of file details. The program below plays three runs one after another, so you can watch each run pick up where the last one stopped:

WORKED EXAMPLE Save, load, change, save

book_store.py

```
1 import json
2 from pathlib import Path
3
4
5 def save_books(books, path):
6     with path.open("w", encoding="utf-8") as file:
7         json.dump(books, file, indent=2)
8
9
10 def load_books(path):
11     with path.open(encoding="utf-8") as file:
12         return json.load(file)
13
14
15 path = Path("books.json")
16
17 # First run: start with one book and save it.
18 save_books([{"title": "Holes", "pages": 233}], path)
19
20 # Next run: a fresh program loads what was saved, adds a book, saves again.
21 books = load_books(path)
22 books.append({"title": "Wonder", "pages": 310})
23 save_books(books, path)
24
25 # The run after that sees both books.
26 for book in load_books(path):
27     print(book["title"], book["pages"])
```

```
Holes 233
Wonder 310
```

- `save_books` (lines 5–7) opens the file with "w" and dumps the whole list. Each save replaces the previous copy, which is what you want: the file should hold the **current** list, not a history.
- `load_books` (lines 10–12) returns whatever the file holds. It returns from inside the `with` block; leaving the function also leaves the block, so the file is still closed.
- Line 18 is the first run: it saves a list with one book.
- Lines 21–23 are the next run: `books` is filled **from the file**, not from anything the program remembered. It adds a book and saves the list of two.
- Line 26 is a later run: loading gives both books.

Why save after every change instead of once, when the user quits? Because programs do not always get to quit politely. A crash, Ctrl+C (Day 5), a closed terminal window, or a power cut ends the program at once, and anything changed since the last save is lost. Saving after each change means the file is never more than one change behind. The lists in today's programs are small, so writing the whole list each time is fast.

On the first run there is no file yet, and `load_books` would stop with `FileNotFoundError`. Use `exists()` from 14.6 to choose:

```
1 if path.exists():  
2     books = load_books(path)  
3 else:  
4     books = []
```

(Day 15 shows another way to handle a missing file, and how to cope with a save file that has been damaged.)

KEY IDEA

To keep data between runs, load it from a file when the program starts (or start empty if there is no file yet) and save the whole data structure after every change.

PAUSE AND TRY

Question 10 Short answer

A notes program loads `notes.json` when it starts and saves only when the user types `quit`. A user adds ten notes, and then the laptop's battery dies. Explain what is lost and why, and describe the change that prevents it.

Putting it together

A teacher keeps three quiz scores per student in `quiz_scores.csv`, saved from a spreadsheet:

```
name,quiz1,quiz2,quiz3  
Ana,8,9,10  
Ben,7,6,9  
"Lopez, Maya",10,9,9  
Sam,5,8,7
```

This program works out each student's average, saves the results as JSON for another program to use, and then reads the JSON back to print a report:

grade_report.py

```

1 import csv
2 import json
3 from pathlib import Path
4
5 here = Path(__file__).parent
6 scores_path = here / "quiz_scores.csv"
7 report_path = here / "grade_report.json"
8
9 averages = {}
10 with scores_path.open(encoding="utf-8", newline="") as file:
11     for row in csv.DictReader(file):
12         quizzes = [int(row["quiz1"]), int(row["quiz2"]), int(row["quiz3"])]
13         averages[row["name"]] = round(sum(quizzes) / len(quizzes), 1)
14
15 with report_path.open("w", encoding="utf-8") as file:
16     json.dump(averages, file, indent=2)
17
18 with report_path.open(encoding="utf-8") as file:
19     report = json.load(file)
20
21 for name, average in report.items():
22     print(f"{name:12} {average:4}")
23 print("Saved", report_path.name)

```

```

Ana          9.0
Ben          7.3
Lopez, Maya  9.3
Sam          6.7
Saved grade_report.json

```

Lines	What happens	Idea
5–7	here is the script's folder; both file paths are built from it, so the program runs from any folder	14.6
10	Open the CSV with with, encoding="utf-8", and newline=""	14.2, 14.8
11	DictReader gives each student as a dictionary keyed by the header	14.8
12	Each quiz score is a string, so int() converts it	14.8
13	The average, rounded to one decimal place (Day 2), is stored under the student's name: Ben's is 22 / 3 = 7.33... → 7.3	14.8
15–16	Save the whole dictionary as JSON with mode "w"	14.4, 14.9
18–19	Read it back: a dictionary of names to floats	14.9
21–22	Print one line per student; {average:4} gives each number a width of 4	Day 5

The CSV file's quotes around Lopez, Maya never show up anywhere: the csv module removed them while reading, and the JSON file stores the name as an ordinary string. After running, open grade_report.json in your editor to see the saved averages.

Vocabulary

Term	Definition	Example
file	A named block of data on a disk that lasts after the program stops	poem.txt
extension	The end of a file name after the last dot, hinting at its kind	.csv
text file	A file holding characters you can read in any editor	.txt, .csv, .json, .py
open()	Opens a file and returns a file object	open("poem.txt", encoding="utf-8")
file object	A value connected to an open file, with methods to read or write it	file.read()
read() / close()	Read the whole file as one string / end the connection to the file	text = file.read()
current working directory	The folder you ran the program from; plain file names are looked up there	your day14 folder
FileNotFoundError	The error when a file to read does not exist at that path	open("poem.txt")
with statement	Opens a file for a block and closes it when the block ends, even after an error	with open(...) as file:
attribute	A value that belongs to an object, read with a dot and no parentheses	file.closed, path.name
readline() / readlines()	Read one line / all lines into a list, each keeping its "\n"	lines = file.readlines()
splitlines()	Splits a string at line breaks, dropping them	text.splitlines()
mode	What open() should do: "r" read, "w" write, "a" append	open(name, "a", ...)
write()	Writes exactly the string given, adding no new line	file.write("eggs\n")
buffer	Memory where writes wait before reaching the disk	emptied at close()
append	Write at the end of a file, keeping what is there	mode "a"
path	A file's location as folder names ending in a name	reports/march/sales.csv
Path	The pathlib type for paths; join parts with /	Path("reports") / "sales.csv"
relative path	A path looked up from the current working directory	Path("poem.txt")
absolute path	A path that starts at the top of the disk	C:\Users\ana\day14\poem.txt

<code>__file__</code>	A variable holding the running script's path	<code>Path(__file__).parent</code>
byte	A number from 0 to 255; what disks actually store	<code>b'caf\xc3\xa9'</code>
text encoding	The rule that turns characters into bytes and back	<code>encoding="utf-8"</code>
UTF-8	The encoding that can store every character; the usual choice today	<code>é</code> is 2 bytes
mojibake	Garbled text from reading bytes with the wrong encoding	<code>CafÃ©</code>
CSV file	A text table: one row per line, values separated by commas	<code>Ana,88</code>
header	The first row of a CSV file, naming the columns	<code>name,score</code>
<code>csv.reader</code>	Gives each row of a CSV file as a list of strings	<code>list(csv.reader(file))</code>
<code>csv.DictReader</code>	Gives each row as a dictionary keyed by the header	<code>row["score"]</code>
<code>csv.writer</code>	Writes lists as CSV rows with <code>writerow()</code>	<code>writer.writerow(["Ana", 88])</code>
JSON	A text format for nested lists, dictionaries, strings, numbers, and bools	<code>{"done": true}</code>
serialize	Turn a value into text (or bytes) that can be saved or sent	<code>json.dumps(book)</code>
<code>json.dump / json.load</code>	Save a value to a JSON file / read it back	<code>json.dump(tasks, file, indent=2)</code>
<code>json.dumps / json.loads</code>	The same, to and from a string	<code>json.loads('[1, 2]')</code>

Summary

- `open()` connects to a file; inside a `with` block, `read()` gives the whole text and `for line in file:` gives one line at a time, each ending in `"\n"` (strip it). The block closes the file, even after an error.
- The **mode** decides what happens to an existing file: `"r"` reads it, `"w"` empties and replaces it, `"a"` adds to the end. `write()` takes only strings and adds no new lines.
- `pathlib.Path` builds paths with `/` that work on every system, takes them apart with `.name`, `.stem`, `.suffix`, and `.parent`, and checks the disk with `exists()`. `Path(__file__).parent` finds files next to the script.
- Files hold bytes, and an **encoding** turns them into characters. Always pass `encoding="utf-8"`.
- Use the `csv` module (opened with `newline=""`) for tables, where every value is read as a string, and the `json` module for nested lists and dictionaries, which come back with their types.
- To keep data between runs, **load** when the program starts and **save** after every change.

Practice

Do these in your `day14` folder. Type each program yourself, run it, and only then open the answer.

Question 11 Short answer

Count lines, words, and characters. Write a program that reads a text file and prints how many lines, words, and characters it has. Test it on this file, saved as `sample.txt` (the third line is empty):

```
The quick brown fox
jumps over the lazy dog.

Then it naps.
```

Question 12 Short answer

Average score from a CSV file. Save this as `scores.csv`, then write a program that reads it and prints how many students there are and their average score:

```
name,score
Ana,88
Ben,92
Kai,79
Sam,95
Zoe,81
```

Question 13 Short answer

Save and load the to-do manager. Start from your Day 13 to-do manager (the `todo_app` folder with `tasks.py` and `main.py`). Add two functions to `tasks.py`: `save_tasks(tasks, path)`, which writes the task list to a JSON file, and `load_tasks(path)`, which reads it back. Change `main()` so that it loads `tasks.json` when it starts (or starts with an empty list if the file does not exist yet) and saves after every add, complete, and delete. Test it: add a task, quit, run it again, and list the tasks.

Review questions

Question 14 Tracing

What does this program print? Work out what `chores.txt` holds before the last loop.

review_log.py

```
1 with open("chores.txt", "w", encoding="utf-8") as file:
2     file.write("dishes\n")
3     file.write("laundry\n")
4     file.write("trash\n")
5
6 with open("chores.txt", "a", encoding="utf-8") as file:
7     file.write("plants\n")
8
9 count = 0
10 with open("chores.txt", encoding="utf-8") as file:
11     for line in file:
12         if "s" in line:
13             count += 1
14
15 print(count)
```

Question 15 Debugging

This program should save two numbers and then print each one doubled. It stops with an error. Quote the error, explain what went wrong (including what has happened to the file), and fix it.

reread_bug.py

```
1 with open("tally.txt", "w", encoding="utf-8") as file:
2     file.write("12\n")
3     file.write("30\n")
4
5 with open("tally.txt", "w", encoding="utf-8") as file:
6     for line in file:
7         print(int(line) * 2)
```

Question 16 Tracing

What does this program print? Use `class_scores.csv` from 14.8.

csv_row_lengths.py

```
1 import csv
2
3 with open("class_scores.csv", encoding="utf-8", newline="") as file:
4     for row in csv.reader(file):
5         print(len(row), row[0])
```

Question 17 Multiple choice

A quiz app must save, for each player, a name, a high score, and a list of the badges they have earned (a different number of badges for each player). Which format and code fit best?

- A. A CSV file with columns `name,score,badges`, written with `csv.writer`
- B. A JSON file holding a list of dictionaries, written with `json.dump(players, file, indent=2)`
- C. A plain text file, written with `file.write(players)`
- D. A JSON file, written with `json.dump(file, players)`

Question 18 Short answer

Ana's `report.py` opens `quiz_scores.csv` with `open("quiz_scores.csv", encoding="utf-8", newline="")`. Both files are in her `day14` folder.

Running `python report.py` from inside `day14` works. Pressing her editor's Run button, which runs the script from the project folder above `day14`, gives `FileNotFoundError`. Explain why, and show the change that makes both ways work.

Question 19 Multiple choice

A program writes a CSV file that will be shared with people on Windows, macOS, and Linux, and some names contain letters such as ñ and é. Which `open` call is right?

- A. `open("roster.csv", "w")`
- B. `open("roster.csv", "w", encoding="utf-8")`
- C. `open("roster.csv", "w", encoding="utf-8", newline="")`
- D. `open("roster.csv", "a", encoding="utf-8", newline="")`

Go further

The official Python Tutorial covers today's files in Input and Output: Reading and Writing Files, including a short section on saving structured data with `json`. The library reference pages for `pathlib` and `csv` list every method used today and many more.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can open a text file and read all of it into a string. Reread: 14.1 Opening and Reading a File
- ☐ I can explain why `with` is better than calling `close()` myself. Reread: 14.2 The `with` Statement
- ☐ I can loop over the lines of a file and strip their new lines. Reread: 14.3 Reading Line by Line
- ☐ I can write lines to a file and remember to add `"\n"` myself. Reread: 14.4 Writing a File
- ☐ I can choose between the modes `"r"`, `"w"`, and `"a"` for a job. Reread: 14.5 Appending and Choosing a Mode
- ☐ I can build a path with `Path` and `/` and find a file next to my script. Reread: 14.6 Paths with `pathlib.Path`
- ☐ I can explain what an encoding is and why I pass `encoding="utf-8"`. Reread: 14.7 Text Encodings and `encoding="utf-8"`
- ☐ I can read and write CSV files with the `csv` module. Reread: 14.8 CSV Files with the `csv` Module
- ☐ I can save lists and dictionaries as JSON and load them back. Reread: 14.9 JSON Files with `json.dump` and `json.load`
- ☐ I can make a program load its data when it starts and save after each change. Reread: 14.10 Keeping Data Between Runs

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
Roses
4
```

`text` holds the whole poem as one string. `text[:5]` is a Day 3 slice: the first five characters, `Roses`. `text.count("\n")` counts the new line characters, and there is one at the end of each of the four lines. If you answered 3, you counted only the breaks **between** lines; the last line has a `"\n"` too.

Reread *14.1 Opening and Reading a File* if your output differed.

Question 2

B. The `with` statement guarantees the file is closed when its block ends, whether the block finished normally or an error stopped it. With a hand-written `close()`, an error before that line skips it.

- **A** is wrong: `with` only opens and closes. You still call `read()` (or loop over the file) inside the block.
- **C** is the opposite of what happens: the file is closed as soon as the block ends, and reading it later gives `ValueError: I/O operation on closed file.`
- **D** is wrong: where the file is depends on the name or path you give `open()` (14.6), not on `with`.

Reread *14.2 The with Statement* if you chose differently.

Question 3

```
2 violets are blue,
3 Python reads files,
```

number	line.strip()	Length	> 15?
1	"Roses are red,"	14	No
2	"violets are blue,"	17	Yes
3	"Python reads files,"	19	Yes
4	"and so can you."	15	No

Line 4 is exactly 15 characters, and `15 > 15` is `False`. If you also printed line 4, you probably measured `len(line)`, which counts the `"\n"` and gives 16. That is why the program strips before measuring.

Reread *14.3 Reading Line by Line* if your output differed.

Question 4

The program stops with:

```
TypeError: write() argument must be str, not int
```

`score` is an `int`, and `write()` accepts only strings. Even with `str(score)` the scores would run together as `728895`, because `write()` adds no new line. Write an f-string that ends in `"\n"`:

`write_scores_fixed.py`

```
1 scores = [72, 88, 95]
2
3 with open("scores_out.txt", "w", encoding="utf-8") as file:
4     for score in scores:
5         file.write(f"{score}\n")
6
7 with open("scores_out.txt", encoding="utf-8") as file:
8     print(file.read())
```

```
72
88
95
```

`f"{score}\n"` turns the number into text and ends the line. The second block reads the file back to check it.

Reread 14.4 *Writing a File* if you missed either problem.

Question 5

B. The first run creates the file (mode `"a"` creates a missing file) and writes one line. Each later run keeps what is there and adds one more line at the end.

- **A** is what mode `"w"` would leave: each run would empty the file first.
- **C** is wrong: only mode `"r"` needs the file to exist already.
- **D** is wrong: each write ends with `"\n"`, so each visit is on its own line.

Reread 14.5 *Appending and Choosing a Mode* if you chose differently.

Question 6

```
beach.jpeg
beach
.jpeg
2026
```

`.name` is the last part, `beach.jpeg`. `.stem` drops the extension. `.suffix` is the extension **with** its dot. `.parent` is the path `photos/2026`, and the `.name` of that path is its last part, `2026`. If you printed `photos/2026` for the last line, you stopped at `.parent`; the extra `.name` takes the last part of the parent.

Reread 14.6 *Paths with pathlib.Path* if your output differed.

Question 7

Model answer: the file was written in UTF-8, where `é` is stored as two bytes, but it was read with a different encoding (such as `cp1252`) in which each byte is a separate character, so the two bytes appeared as the two characters `Ã©`. The text on the disk is fine; it was decoded with the wrong rule. Read the file with `encoding="utf-8"`, and make sure every program that writes or reads the file passes `encoding="utf-8"` too.

A correct answer says that the writer and the reader used **different** encodings, that `é` is more than one byte in UTF-8, and that the fix is the same explicit encoding on both sides.

Reread *14.7 Text Encodings and encoding="utf-8"* if your reasoning differed.

Question 8

```
2 ['Ben', 'Sam']
```

`DictReader` uses the header for keys, so the loop sees four rows. Ana (88) and Maya (79) are below 90; Ben (92) and Sam (95) are appended to `names`. `int()` is needed for the comparison: `"92" >= 90` would compare a string with an int and stop with a `TypeError`. The name is appended as the string from the file, without converting.

Reread *14.8 CSV Files with the csv Module* if your output differed.

Question 9

```
Riverside Otters
3
7
13
```

`json.load` returns a dictionary. `team["players"]` is a list of three dictionaries, so its length is 3. `team["players"][1]` is the **second** player (indexes start at 0), Ben, and his `"goals"` is 7. The last line sums a generator expression (Day 11) over all players: $4 + 7 + 2 = 13$. The goals are ints, because JSON keeps numbers as numbers, so `sum` works without any conversion.

Reread *14.9 JSON Files with json.dump and json.load* if your output differed.

Question 10

Model answer: all ten notes are lost. They were only in the program's memory, which is wiped when the program stops, and the program never reached the `quit` code that writes the file. On the next run, `notes.json` still holds the notes from before this session. The fix is to call the `save` function right after each note is added (and after any other change), so the file always matches the list in memory.

A correct answer says that the data lived only in memory, that the `save` never ran, and that saving after each change keeps the file up to date.

Reread *14.10 Keeping Data Between Runs* if your reasoning differed.

Question 11

One possible solution:

count_text.py

```
1 from pathlib import Path
2
3 path = Path("sample.txt")
4 text = path.read_text(encoding="utf-8")
5
6 lines = text.splitlines()
7 words = text.split()
8
9 print("Lines:", len(lines))
10 print("Words:", len(words))
11 print("Characters:", len(text))
```

```
Lines: 4
Words: 12
Characters: 60
```

`read_text` reads the whole file into one string. `splitlines()` gives one item per line, including the empty third line, so there are 4. `split()` with no argument splits at any whitespace, including new lines, and ignores the empty line, so it finds the 12 words. `len(text)` counts every character, and that includes the four `"\n"` characters: 19 + 24 + 0 + 13 visible characters, plus 4, is 60. Tools that count "characters" sometimes leave new lines out; say which you mean.

Reread 14.3 *Reading Line by Line* and 14.6 *Paths with `pathlib.Path`* if your result differed.

Question 12

One possible solution:

average_score.py

```
1 import csv
2
3 scores = []
4 with open("scores.csv", encoding="utf-8", newline="") as file:
5     for row in csv.DictReader(file):
6         scores.append(int(row["score"]))
7
8 average = sum(scores) / len(scores)
9 print(f"{len(scores)} students, average score {average:.1f}")
```

```
5 students, average score 87.0
```

`DictReader` skips the header and gives each student as a dictionary. Each score is a string, so `int()` converts it before it goes into the list. Then `sum` and `len` from Day 6 give $435 / 5 = 87.0$. Collecting the scores in a list keeps the count and the total in step automatically; a counter and a running total (Day 5) work too.

Reread 14.8 *CSV Files with the `csv` Module* if your result differed.

Question 13

One possible solution. `tasks.py` keeps the Day 13 functions unchanged, imports `json`, and adds the two new functions before the self-test:

todo_app/tasks.py

```
1  """Task functions for the to-do manager. main.py imports them."""
2
3  import json
4
5
6  def show_menu():
7      """Print the menu of choices."""
8      print()
9      print("1) Add  2) List  3) Complete  4) Delete  5) Quit")
10
11
12  def add_task(tasks, title):
13      """Add a new, unfinished task to the end of tasks."""
14      tasks.append({"title": title, "done": False})
15
16
17  def list_tasks(tasks):
18      """Print every task with its number and a check box."""
19      if not tasks:
20          print("No tasks yet.")
21          return
22      for number, task in enumerate(tasks, 1):
23          mark = "x" if task["done"] else " "
24          title = task["title"]
25          print(f"{number}. [{mark}] {title}")
26
27
28  def complete_task(tasks, number):
29      """Mark task number (counting from 1) as done.
30
31      Return the task, or None if no task has that number.
32      """
33      if not 1 <= number <= len(tasks):
34          return None
35      task = tasks[number - 1]
36      task["done"] = True
37      return task
38
39
40  def delete_task(tasks, number):
41      """Remove task number (counting from 1) from tasks.
42
43      Return the removed task, or None if no task has that number.
44      """
45      if not 1 <= number <= len(tasks):
46          return None
47      return tasks.pop(number - 1)
48
49
50  def save_tasks(tasks, path):
51      """Write tasks to the JSON file at path, replacing what was there."""
52      with path.open("w", encoding="utf-8") as file:
53          json.dump(tasks, file, indent=2)
54
```

```

55
56 def load_tasks(path):
57     """Read the JSON file at path and return its list of tasks."""
58     with path.open(encoding="utf-8") as file:
59         return json.load(file)
60
61
62 if __name__ == "__main__":
63     test_tasks = []
64     add_task(test_tasks, "Test the module")
65     print(complete_task(test_tasks, 1))
66     print(complete_task(test_tasks, 2))
67     list_tasks(test_tasks)

```

main.py builds the save file's path next to the script, loads at the start, and calls `save_tasks` after each change:

todo_app/main.py

```

1  """The to-do manager's menu loop. Run this file to start the program."""
2
3  from pathlib import Path
4
5  from tasks import (
6      add_task,
7      complete_task,
8      delete_task,
9      list_tasks,
10     load_tasks,
11     save_tasks,
12     show_menu,
13 )
14
15 SAVE_FILE = Path(__file__).parent / "tasks.json"
16
17
18 def main():
19     """Run the menu loop until the user chooses Quit."""
20     if SAVE_FILE.exists():
21         tasks = load_tasks(SAVE_FILE)
22     else:
23         tasks = []
24     print("To-do list manager")
25     print(f"Loaded {len(tasks)} tasks from {SAVE_FILE.name}")
26     while True:
27         show_menu()
28         choice = input("Choose 1-5: ").strip()
29         if choice == "1":
30             title = input("Task title: ").strip()
31             if title:
32                 add_task(tasks, title)
33                 save_tasks(tasks, SAVE_FILE)
34                 print(f"Added: {title}")
35             else:
36                 print("A task needs a title.")

```

```

37         elif choice == "2":
38             list_tasks(tasks)
39         elif choice in ("3", "4"):
40             text = input("Task number: ").strip()
41             if not text or not all(ch in "0123456789" for ch in text):
42                 print("Please type a task number, such as 1.")
43                 continue
44             number = int(text)
45             if choice == "3":
46                 task = complete_task(tasks, number)
47                 action = "Completed"
48             else:
49                 task = delete_task(tasks, number)
50                 action = "Deleted"
51             if task is None:
52                 print(f"There is no task {number}.")
53             else:
54                 save_tasks(tasks, SAVE_FILE)
55                 title = task["title"]
56                 print(f"{action}: {title}")
57         elif choice == "5":
58             print("Goodbye!")
59             break
60         else:
61             print("Please choose a number from 1 to 5.")
62
63
64 if __name__ == "__main__":
65     main()

```

A run that starts with two saved tasks in `tasks.json`, adds a third, completes task 2, and tries to delete a task 7 that does not exist:

```

To-do list manager
Loaded 2 tasks from tasks.json

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [x] Return library books
2. [ ] Practice piano

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 1
Task title: Water the plants
Added: Water the plants

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 3
Task number: 2
Completed: Practice piano

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 4

```

```

Task number: 7
There is no task 7.

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 2
1. [x] Return library books
2. [x] Practice piano
3. [ ] Water the plants

1) Add  2) List  3) Complete  4) Delete  5) Quit
Choose 1-5: 5
Goodbye!

```

How it works:

- The tasks are already a list of dictionaries of strings and bools, so `json.dump` can save them as they are, and `json.load` gives back the same structure. No conversion is needed, which is why JSON fits this job better than CSV.
- `save_tasks` uses mode `"w"`: each save replaces the file with the current list.
- `SAVE_FILE = Path(__file__).parent / "tasks.json"` puts the save file next to `main.py`, however you start the program.
- `main()` checks `SAVE_FILE.exists()` so the first run, with no file yet, starts with an empty list instead of a `FileNotFoundError`.
- `save_tasks` is called only after a change actually happened: after a task is added, and after `complete_task` or `delete_task` returned a task rather than `None`. Listing and invalid numbers change nothing, so there is nothing to save.
- Only `main()` reads input and only the two new functions touch the file; the other functions, and the self-test, did not change at all.

Each change was saved as it happened, so the next run starts with

Loaded 3 tasks from `tasks.json`. Try it: run it, add a task, choose 5, and run it again; choice 2 lists the task you added. Then open `tasks.json` in your editor and read your tasks as JSON.

Reread 14.9 *JSON Files with `json.dump` and `json.load`* and 14.10 *Keeping Data Between Runs* if your program behaved differently.

Question 14

3

Mode `"w"` creates the file with three lines, and mode `"a"` adds a fourth, so the file holds `dishes`, `laundry`, `trash`, `plants`. The loop counts the lines that contain an `s`: `dishes`, `trash`, and `plants` do; `laundry` does not. If you answered 1, you thought `"a"` replaced the file, leaving only `plants`; that is what `"w"` does.

Reread 14.5 *Appending and Choosing a Mode* and 14.3 *Reading Line by Line* if your output differed.

Question 15

The program stops at line 6 with:

```
io.UnsupportedOperation: not readable
```

The second `open` uses mode `"w"`, so the file object can only be written, and the `for` loop tries to read from it. Worse, opening with `"w"` **emptied** `tally.txt` before the error, so the two numbers are gone. Reading needs mode `"r"`, which is the default, so leave the mode out:

`reread_fixed.py`

```
1 with open("tally.txt", "w", encoding="utf-8") as file:
2     file.write("12\n")
3     file.write("30\n")
4
5 with open("tally.txt", encoding="utf-8") as file:
6     for line in file:
7         print(int(line) * 2)
```

```
24
60
```

`int(line)` works even though each line ends with `"\n"`, because `int()` ignores whitespace at the ends of the string.

Reread the mode table in *14.5 Appending and Choosing a Mode* if you missed it.

Question 16

```
2 name
2 Ana
2 Ben
2 Lopez, Maya
2 Sam
```

`csv.reader` gives every row, the header included, as a list of strings. Every row has two values, including Maya's, because the comma inside the quotes is part of the value, not a separator. `row[0]` shows her name without the quotes. Compare `split_csv_wrong.py` in 14.8, where `split(",")` gave that row three pieces.

Reread *14.8 CSV Files with the csv Module* if your output differed.

Question 17

B. Each player is a record with a list inside it, which is nested data, and JSON stores lists inside dictionaries directly. The scores come back as numbers and the badges as a list.

- **A** is wrong: a CSV row is a flat list of values. A list of badges does not fit in one column without inventing your own rule for packing it into a string, and every value would come back as a string.
- **C** is wrong: `write()` accepts only a string, so passing a list stops with a `TypeError`.
- **D** has the arguments reversed: the data comes first. As written, it stops with `TypeError: Object of type TextIOWrapper is not JSON serializable`.

Reread *14.9 JSON Files with json.dump and json.load* if you chose differently.

Question 18

Model answer: "quiz_scores.csv" is a relative path, so Python looks for it in the current working directory, the folder the program was **started from**, not the folder the script is in. From day14 that is the right folder; from the project folder above it, there is no quiz_scores.csv, so open() fails. Build the path from the script's own location:

```
1 from pathlib import Path
2
3 here = Path(__file__).parent
4 scores_path = here / "quiz_scores.csv"
```

and open scores_path instead of the plain name. __file__ is the script's path, so here is always day14, wherever the program is run from.

A correct answer names the current working directory as where relative paths are looked up, and uses Path(__file__).parent to find the file next to the script.

Reread 14.6 Paths with pathlib.Path if your reasoning differed.

Question 19

C. It writes ("w"), names the encoding so that ñ and é are stored the same way on every computer, and passes newline="" so the csv module controls the line endings and Windows does not get blank lines between rows.

- **A** uses the computer's default encoding, which may not be UTF-8 on Windows, so the names could turn into mojibake for some readers. It also leaves out newline="".
- **B** gets the encoding right but leaves out newline="", which the csv module needs.
- **D** appends: every run would add the roster again below the old copy instead of replacing it.

Reread 14.7 Text Encodings and encoding="utf-8" and 14.8 CSV Files with the csv Module if you chose differently.