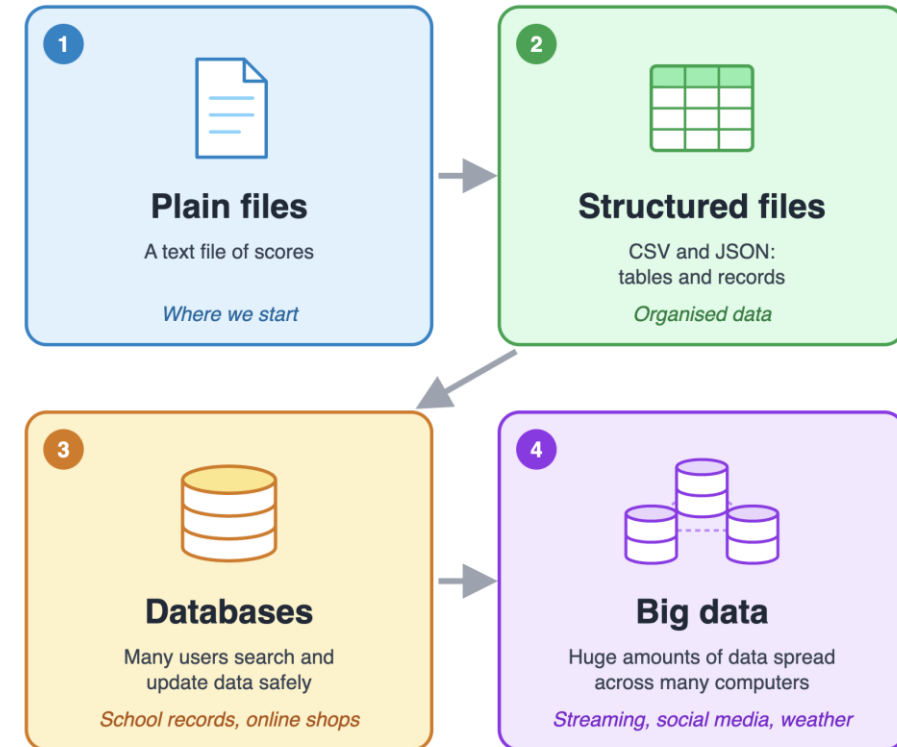


Persistence (File-I/O)

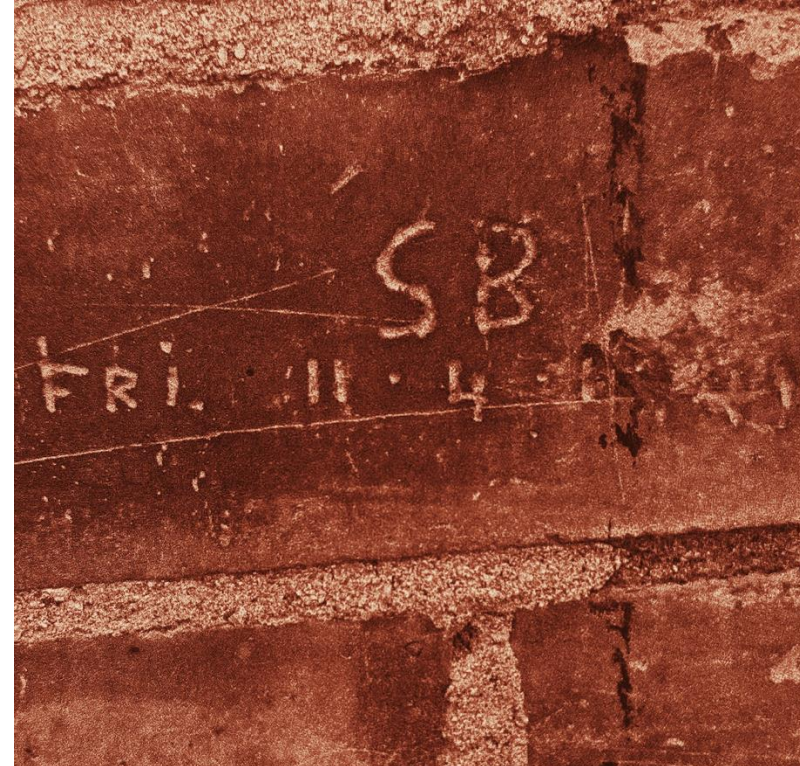
The big picture: memory

- **Short-term (RAM):**
 - where your program's variables live. Fast.
- **Long-term (storage):**
 - files on a disk or SSD. Slow.
- **'Persistence': data that lasts after the program ends.**
- **Saving to files:**
 - open the file → read or write → close the file.
- **Files:**
 - Plain text files
 - Structured CSV / JSON
- **Databases:**
 - many users concurrently...
 - ...search, update and share data safely
- **Big data**
 - huge amounts of data spread across many computers.
 - Streaming services, social media etc.
- **The same idea all the way up:**
 - keeping data so it outlives the program



Overview

- **Why files?**
 - Programs forget
- **Writing files**
 - `open()`, `write()`, `close()` and `with`
- **Reading files:**
 - line by line, `rstrip()` and sorting
- **CSV files:**
 - one book per line
 - `csv.reader`, `csv.writer`
- **When files go wrong:**
 - `try` and `except`



Why files?

- Our programs (so far) do short-term memory
- ...but they easily forget things

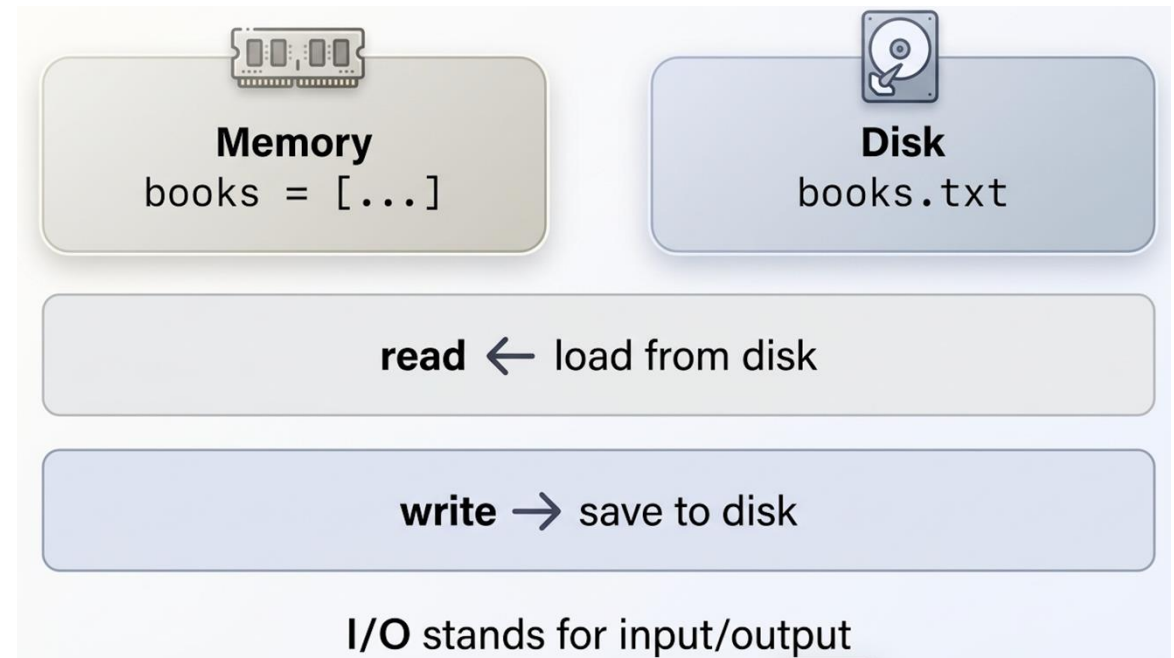
A reading log with no memory

- **Program:**
 - ask for three books
 - stores them in a list
 - prints them
- **Program: run it again**
 - list is empty
- **Stored lists**
 - live in the computer's memory
 - memory lost when the program ends

```
1  def main():
2      ... # Reading log that forgets everything
3      ... books = []
4
5      ... for _ in range(3):
6          ... books.append(input("Book? "))
7
8      ... for title in books:
9          ... print(f"You read: {title}")
10
11
12  if __name__ == "__main__":
13      ... main()
14
15
16
```

Computer memory

- **RAM: memory wiped**
 - Random Access
 - Variables and lists.
 - Fast, but wiped when program end
- **Hard Disk: stored**
 - Files. Still there tomorrow
- **File I/O means:**
 - writing to a file (save)
 - reading from a file (load)

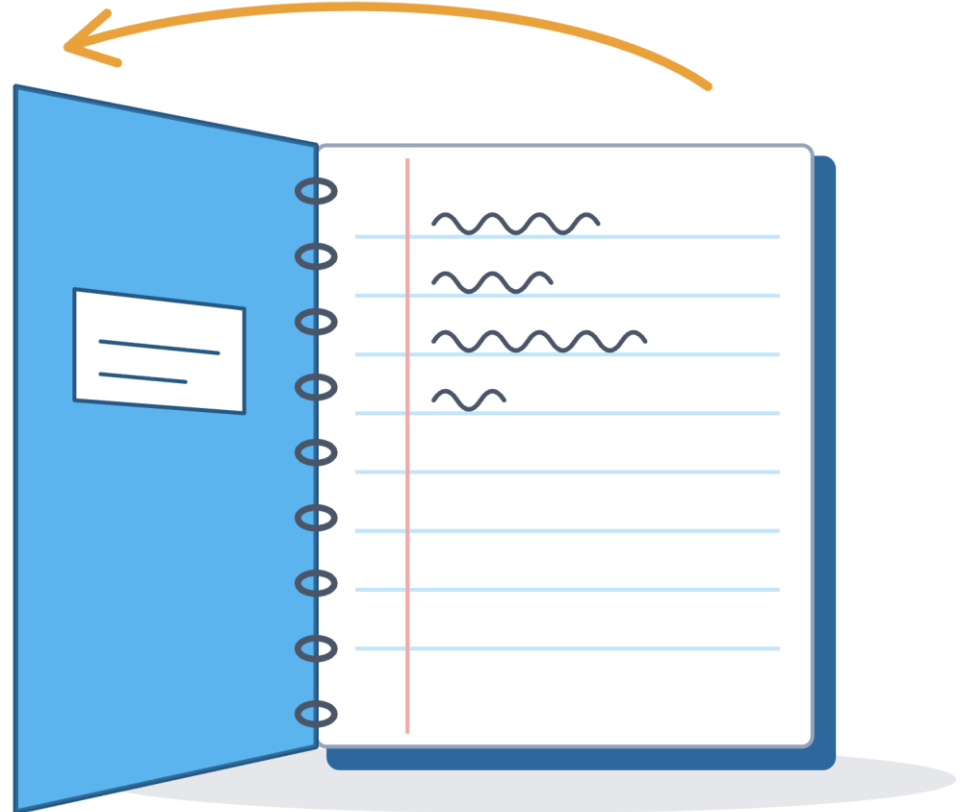


Writing (*w*, *a*)

- Programs memorise
- ...but they forget things
- ...writing to file (long term memory)

Open a file

- `open ()`
 - like opening a physical notebook
 - nothing written yet
- `open ()`
 - provides file *handle*
 - programmatic interaction
- **Interaction context / 'mode':**
 - `"r"` read (default)
 - `"w"` write: fresh empty file
 - `"a"` append: to the end of file



Write to and close file (TODO)

- **Three steps:**
 - `open()` finds or creates the file
 - `write()` puts text into it
 - `close()` finishes and saves it
- **File is there after the program ends.**
 - Contains the title you input

```
1  def main():
2      ... # Save one book title to file (mode "w")
3      ... title = input("Book? ")
4      ... file = open("books.txt", "w")
5      ... file.write(title)
6      ... file.close()
7
8
9  if __name__ == "__main__":
10     ... main()
11
```

The danger of "w"

- "w"
 - makes a brand-new, empty file every time it opens.
- **No error, no warning!**
 - Things get overwritten!
- **Some of the worst file bugs don't crash**
 - they quietly lose data.



Appending with "a" and \n

- "a"
 - adds to the end of the file instead of replacing it.
- **But the titles**
 - get glued together.
- `print()`
 - adds a new line
- `write()`
 - doesn't.
 - `\n`, required!

```
1  def main():
2      ... # Add to end of file i(mode="a" = append)
3      ... title = input("Book? ")
4
5      ... file = open("books.txt", "a")
6      ... file.write(f"{title}\n")
7      ... file.close()
8
9
10 if __name__ == "__main__":
11     ... main()
12
```

Closing automatically with `with`

- **Easy to forget `close()`**
 - Your writing might not be saved, especially if the program crashes first
- **Use `with`**
 - closes the file for you as soon as the indented block ends, even if an error happens.
 - cleaner code
 - clear intent

```
1  def main():
2      ... # "with" closes the file for us
3      ... # even if something goes wrong
4      ... title = input("Book? ")
5
6      ... with open("books.txt", "a") as file:
7          ... file.write(f"{title}\n")
8
9
10 if __name__ == "__main__":
11     ... main()
12
```

Reading (r)

Reading line by line

- **Open "r" mode**
 - to read information
- **for line in file**
 - reads one line at a time, top to bottom.
- **Output from program**
 - altered with `rstrip()`
 - otherwise blank line after every title?
 - Remember the defaults of `print()`

```
1  def main():
2      ... # Read every line back (mode "r" = read)
3      ... with open("books.txt", "r") as file:
4          ...     for line in file:
5              ...         print("You read:", line.rstrip())
6
7
8  if __name__ == "__main__":
9      ... main()
10
```

Sorting: read everything first

- You can't sort books while putting them on a list
- Read every line into a list
 - then use `sorted()`
- No mode given?
 - `open()` reads, `(r)` is default.

```
1  def main():
2      ... # Read everything first,
3      ... # then sort...tidying a bookshelf!
4      ... books = []
5      ... with open("books.txt") as file:
6          ...     for line in file:
7              ...         books.append(line.rstrip())
8
9      ... for title in sorted(books):
10         ...     print("You read:", title)
11
12
13  if __name__ == "__main__":
14      ... main()
15
```

Structured data

CSV, comma (or character) separated values

Reading CSV with `csv.reader`

- **Python**
 - Built-in `CSV` module.
- **In CSV**
 - double quotes mean "keep this together".
- **`csv.reader`**
 - knows the rules and gives back a list for each line.
- **Coding by hand more cumbersome**
 - You will get to learn which packages are useful for you projects over time

```
1  import csv
2
3
4  def main():
5      ... # Let Python's csv module handle the tricky
6      ... # commas and quotes
7      ... with open("books.csv") as file:
8          ... reader = csv.reader(file)
9          ... for title, author, rating in reader:
10             ... print(f"{title} by {author}: {rating}/5")
11
12
13  if __name__ == "__main__":
14      ... main()
15
```

Writing CSV with `csv.writer`

- `writerow()`
 - takes a list: one item per column.
 - It adds quotes when a value contains a comma.
- `newline=""`
 - stops blank lines appearing on Windows.

```
1  import csv
2
3
4  def main():
5      ... # Add a book to the CSV file -- csv.writer
6      ... # adds quotes when needed
7      ... title = input("Title? ")
8      ... author = input("Author? ")
9      ... rating = input("Rating? ")
10
11     ... with open("books.csv", "a", newline="") as file:
12         ...     writer = csv.writer(file)
13         ...     writer.writerow([title, author, rating])
14
15
```

Exceptions

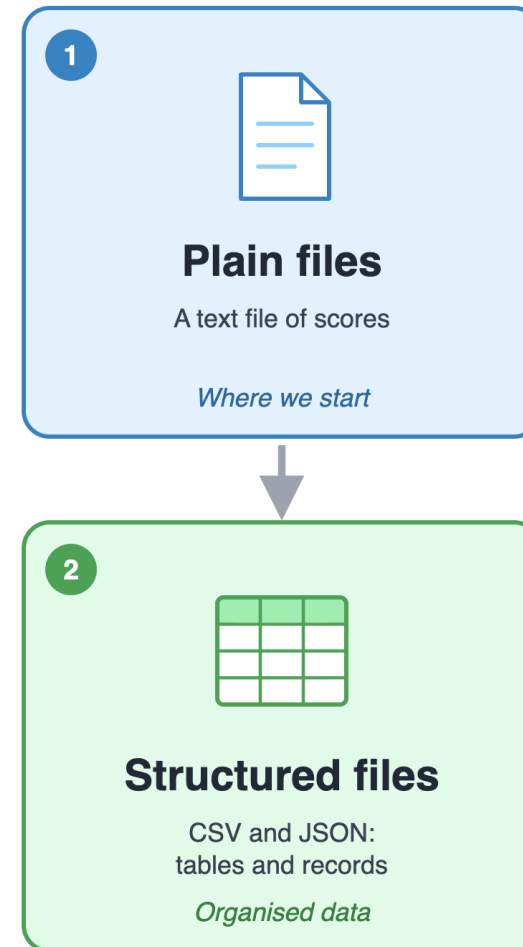
Missing file? `try / except`

- **The very first run:**
 - no `books.csv` yet.
- **How it works:**
 - Python runs the code in the `try` block.
 - If a `FileNotFoundError` occurs, it jumps to the `except` block.
- **A missing file is a brand-new user, not a bug**
 - We use the exception to the rule (a file is available) to allow the program to run sensibly, rather than crash.

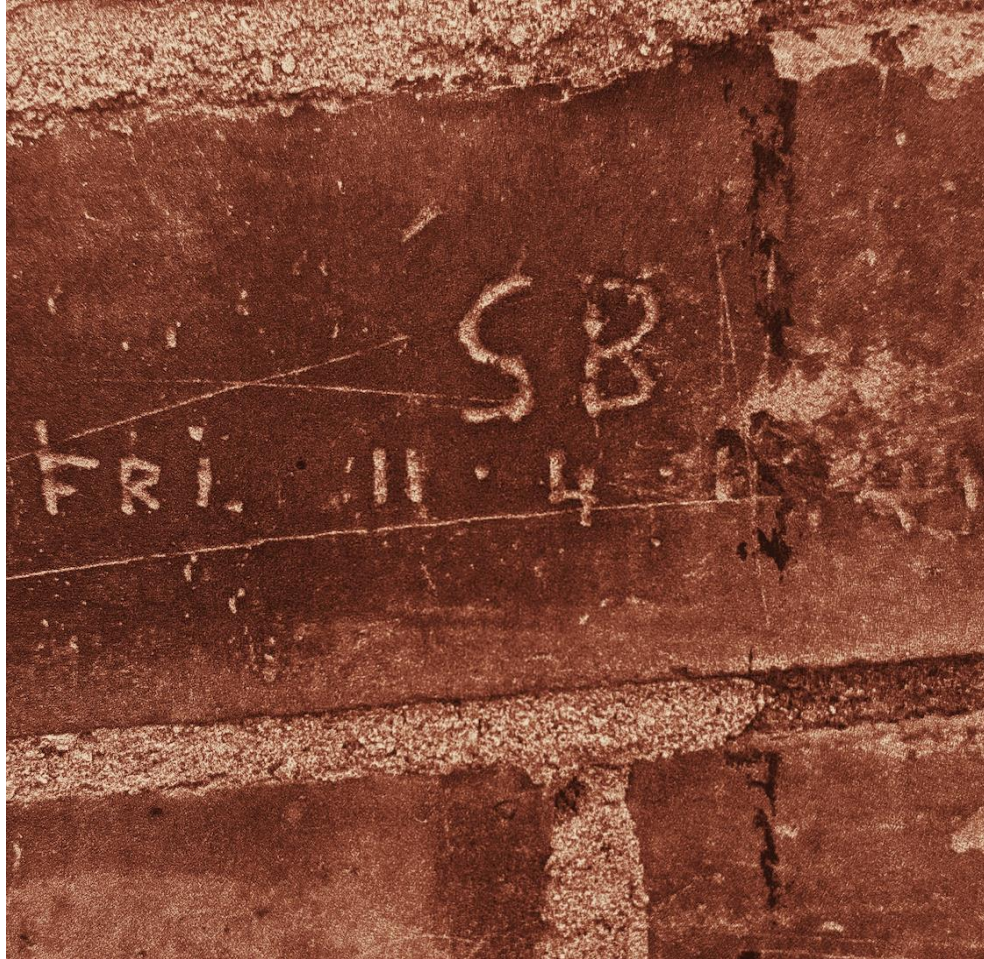
```
1  import csv
2
3  def main():
4      ... # The very first run: there is no file yet
5      ... try:
6          ... with open("books.csv") as file:
7              ... for title, author, rating in csv.reader(file):
8                  ... print(f"{title} by {author}: {rating}/5")
9      ... except FileNotFoundError:
10         ... print("Your shelf is empty!")
11
12
13 if __name__ == "__main__":
14     ... main()
15
```

Conclusions

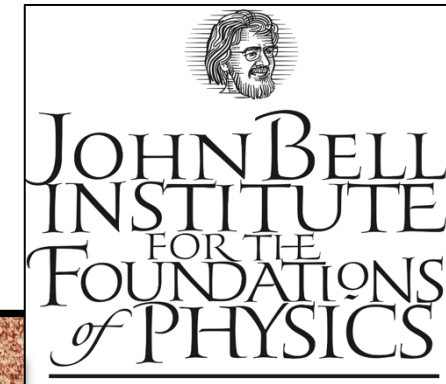
- **open(name, mode):**
 - "r" reads, "w" wipes, "a" adds.
- **write()**
 - needs `\n`;
 - `rstrip()` removes it.
- **with**
 - closes the file for you.
- **.CSV data**
 - `csv.reader` and `csv.writer`
- **try / except**
 - missing files



That brick



That brick



ON THE EINSTEIN PODOLSKY ROSEN PARADOX*

J. S. BELL†

Department of Physics, University of Wisconsin, Madison, Wisconsin

(Received 4 November 1964)

Campaign mounts to better recognise world-famous Belfast physicist who 'proved Einstein wrong'

John Stewart Bell has pedestrian walkway named after him close to childhood home, but locals are keen to increase awareness of scientist behind Bell's Theorem

Expand



Thanks!