File IO

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 6B's technical content. See python-file-io[2] for the full reference on file modes and reading/writing patterns.

Today's outline

- Memory hierarchy and why files
- Opening, reading, and closing files
- Exercises: counting empty lines, reading numbers, finding the highest-rated band, reading a Sudoku board
- Writing and appending to files

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-file-io.pdf

Memory hierarchy

Type	Order	2016 MacBook Pro	Persistence
CPU Cache L2	KB	256KB	Requires power
CPU Cache L3	MB	8MB	Requires power
Random Access Memory	GB	16GB	Requires power
Disk	GB/TB	256GB	Persistent
Cloud	PB	Functionally infinite	Persistent

Why files?

When Python launches, it's allocated space in RAM, which can fill up — a problem for memory-intensive problems (analyzing tweets, payroll, scientific computing). We may also want to *save* our data with persistence. Either way, we need to instruct Python to use the disk, one level up the hierarchy.

Opening a file

```
file = open("file.dat", "<mode>")
```

Mode	Description
`r`	read
`w`	write
`a`	append (write at end of file)

See also `os.getcwd()` and `os.chdir()` (a file assumed to be in the same directory as the running script).

Reading a file

Given `hello.txt` containing:

```
What a
wonderful
hello world.
```

```
>>> file = open("hello.txt", "r")
>>> file.readline()
'What a\n'
>>> file.readline()
'wonderful\n'
>>> file.readline()
'hello world.'      # note: no trailing newline (last line of the file)
>>> file.readline()
''                  # empty string once exhausted, returned indefinitely
```

A for-loop iterates line by line:

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     print(line)
What a

wonderful

hello world.

>>>
```

(The extra blank lines above come from `print`'s own newline stacking on top of each line's own trailing `\n`.)

The file-pointer is exhausted after one pass

```
>>> for line in file:
...     print(line)
>>>
```

Nothing prints — the file-pointer already reached the end of the file during the loop above. We need to reopen the file (or seek back to the start) to read it again:

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     line
'What a\n'
'wonderful\n'
'hello world.'
```

Closing files

Files left open are vulnerable to side-effects — you may find data missing, or extra bytes, if you neglect to close after use:

```
>>> file = open("hello.txt", "r")
>>> for line in file:
...     line
>>> file.close()
```

The `with` construct closes the file for you automatically, even if the code block crashes:

```
with open("hello.txt", "r") as file:
    for line in file:
        print(line)
```

Exercise: counting empty lines

```
def num_empty_lines(path: str) -> int:
    """ Count the number of empty lines (those that only contain \n)
    in the file at <path>.
    """
```

```
def num_empty_lines(path: str) -> int:
    ans = 0
    with open(path, "r") as the_file:
        for line in the_file:
            if line == "\n":
                ans += 1
    return ans
```

A second version takes a file *pointer* directly, rather than a path — note the different parameter type (the caller is now responsible for opening the file):

```
from typing import TextIO

def num_empty_lines(file_pointer: TextIO) -> int:
    ans = 0
    for line in file_pointer:
        if line == "\n":
```

```
            ans += 1
    return ans

fp = open("filename.txt")
num_empty_lines(fp)
```

Reading numbers from a file

Given `numbers.dat` containing one number per line (`1` through `6`), reading always gives back **strings** — cast to the appropriate type when needed:

```
>>> with open("numbers.dat", "r") as file:
...     ans = []
...     for line in file:
...         ans.append(int(line))
>>> ans
[1, 2, 3, 4, 5, 6]
```

Exercise: the highest-rated band

Given a CSV file `bands.txt` with a header row (`Band,Rating,Plays`), find the highest-rated band:

```
def highest_rated(path: str) -> str:
    """ Return the highest-rated band in the file at <path>.
    Precondition: the file has a header row.
    """
    current_most_popular_band = ""
    current_highest_rating = -float('inf')   # guarantees the first row updates it

    with open(path, "r") as file:
        file.readline()   # skip the header
        for line in file:
            band, rating, _ = line.split(',')   # don't name values you won't use
            rating = int(rating)
            if rating > current_highest_rating:
                current_highest_rating = rating
                current_most_popular_band = band

    return current_most_popular_band
```

> The lecture slide names this function `higest_rated` (missing a `t`) — corrected to `highest_rated` above. The companion source file has a differently-named `most_played` function with the *same shape* of code, but it unpacks the *third* CSV column (`Plays`) instead of the second (`Rating`) — so despite its docstring claiming to return "the highest rated band", it actually returns the band with the most plays. Reproduced here as `highest_rated` (ranking by rating, matching the slide and this section's title) rather than perpetuating that docstring/behaviour mismatch.

Exercise: an arbitrary attribute

Extend the previous answer to take an arbitrary file with a header of attributes (e.g. `name,grade,age`) and a function `def most(path: str, attribute: str) -> str` that finds the maximum of that attribute.

> No worked solution is given in the source for this exercise — left as an open exercise rather than invented here. (The companion source file also includes a `least_rated_band` function that is an intentional joke stub — `return "Drake"` with the comment `# This is a joke` — rather than a real implementation.)

Exercise: reading a Sudoku board

Given a partially-filled Sudoku board as a text file (`|` separates 3x3 blocks column-wise, a row of `-` separates them row-wise, and spaces mark empty cells):

```
685|13 | 47
7  |   | 1
 1 |764| 5
-----------
9  | 7 |5 4
8 1|  9| 72
4 3|  6|
-----------
   |427|39
 4 |9  | 68
1 7|   |4
```

```
def read_board(path: str) -> list[list[int | None]]:
    """ Return a board representation for the Sudoku board in the
    file at <path>. None denotes an empty position.
    """
```

Character-by-character version:

```
def read_board_v1(path: str) -> list[list[int | None]]:
    with open(path, "r") as the_file:
        board = []
        for line in the_file:
            row = []
            if "-" in line:
                continue   # skip horizontal dividers
            for x in line:
                if x.isdigit():
                    row.append(int(x))
                if x == " ":
                    row.append(None)
            board.append(row)
    return board
```

Equivalent using a list comprehension (filtering out `|` characters, rather than simply not appending on them):

```
def read_board_v2(path: str) -> list[list[int | None]]:
    with open(path, "r") as the_file:
        board = []
        for line in the_file:
            if "-" in line:
                continue   # skip horizontal divider
            row = [int(x) if x.isdigit() else None for x in line if x != '|']
            board.append(row)
    return board
```

Exercise: has the Sudoku been won?

```
def has_won(board: list[list[int]]) -> bool:
    """ Return True when the Sudoku board is solved (contains the
    digits 1 through 9 in each row, column, and 3x3 grid).
    """
```

No worked solution is given in the source for this exercise — left as an open exercise rather than invented here.

Writing to files

```
>>> with open("numbers.dat", "w") as file:
...     file.write("Hello World.\n")            # write a single string
...     file.writelines(["Hello\n", "World\n"])  # write a list of strings
```

Careful! Opening a file for write (`"w"`) creates the file if it doesn't exist, or **overwrites** it if it does.

Appending to files

Appending opens a file without overwriting, instead adding to the end (creating the file if it doesn't exist):

```
>>> with open("numbers.dat", "a") as file:
...     file.write("Hello World.\n")
```

Summary

We can read strings from files and write strings to files. A file-pointer moves forward every time a line is read — to read a line (or the whole file) twice, the file must be reopened.

Next: 2025-09-01-testing[3] (Lecture 6C).

[3] courses/csse1001/2025-09-01-testing.pdf