File IO

Table of contents

Outline . . . 1
Memory hierarchy . . . 2
Why files? . . . 2
Opening a file . . . 3
Reading a file . . . 3
The file-pointer is exhausted after one pass . . . 5
Closing files . . . 5
Exercise: counting empty lines . . . 6
Reading numbers from a file . . . 7
Exercise: the highest-rated band . . . 8
Exercise: reading a Sudoku board . . . 10
Writing to files . . . 11
Appending to files . . . 12
Summary . . . 12

Lecture 6B. See csse1001[1] for course logistics and python-file-io[2] for the full reference on file modes and reading/writing patterns.

Outline

- Memory hierarchy and why files
- Opening, reading, and closing files
- The file-pointer, and why it can only be spent once
- Reading numbers out of a file, and casting
- Exercises: counting empty lines, 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

Five kinds of memory, with the capacity and the persistence of each:

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

As you go down the list you get **more space** and, past Random Access Memory, you also get **permanence**.

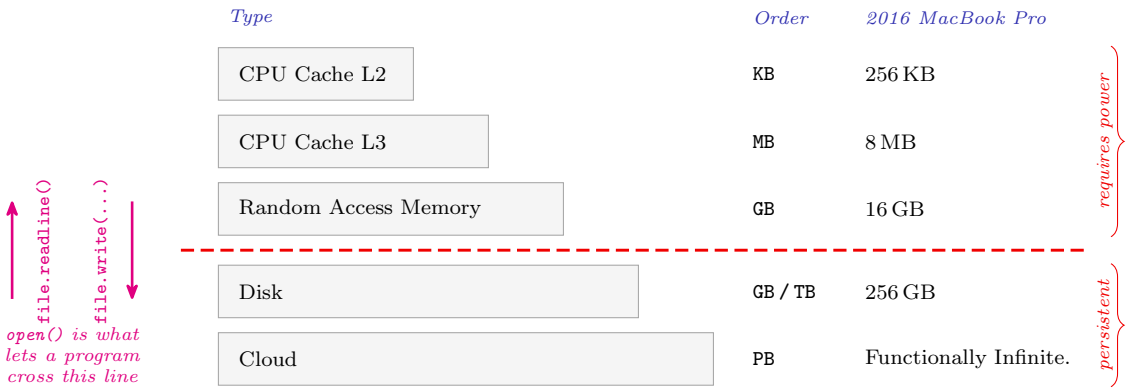

Figure 1: The memory hierarchy and its one boundary: everything above the dashed line is gone when the power goes, everything below it survives. That boundary is why files are needed. A Python program lives in RAM, and `open` is the one thing that lets it reach across. The stepped bar widths stand for the growing Order column, and the two arrows are a write and a read crossing the line.

Why files?

When Python (i.e. IDLE) launches it gets **allocated space in RAM**, which is possible to fill up. This is a problem if we want to solve memory-intensive problems — analyzing tweets, payroll, scientific computing. We may also want to *save* our data with **persistence**.

In either case, it is necessary to instruct Python to use the disk memory one level up.

Opening a file

To open a file assumed to be in the **same directory** as your running Python:

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

Mode	Description
`r`	read
`w`	write
`a`	append

Append means write at the end of the file.

`os.getcwd()` and `os.chdir()` are what you reach for when the file is **not** in the same directory as the running program.

Reading a file

The example file, `hello.txt`, is three lines:

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

where ¶ is the **new-line character**, which normally is not displayed. The third line carries none.

```
>>> file = open("hello.txt", "r")     # open for read.
>>> file.readline()
'What a\n'
>>> file.readline()
'wonderful\n'
>>> file.readline()
'hello world.'      # note lack of newline character
>>> file.readline()
''
>>> file.readline()
''
```

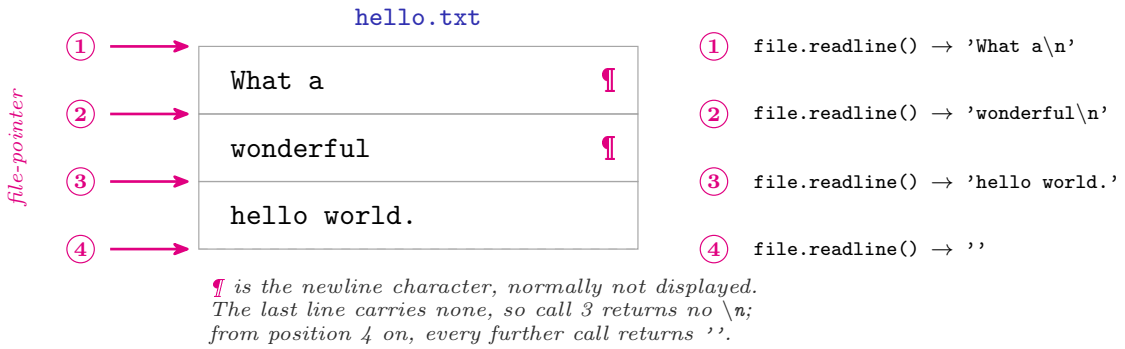

Figure 2: How readline moves the file-pointer. Each call to readline returns everything from the file-pointer up to and including the next newline, and then leaves the pointer past it. So the five calls in the transcript are not five readings of the same file, but reads taken from four successive pointer positions, the fifth call finding the pointer exactly where the fourth left it. Once the pointer reaches position 4 there is nothing left and every further call returns the empty string, which is why only four positions are drawn. This is why, to read a line twice, the file must be opened twice. The pilcrow stands for the newline character.

Past the end of the file `readline` returns `''` every time. That is the empty string, not a newline, and it is what makes `readline` a usable loop-termination test.

A for-loop iterates line by line:

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

The three ¶ in that output are inserted by Python's `print`: every `print` call ends with a newline of its own. For the first two lines that added newline lands on top of the `\n` the line already carries, so the terminal shows a blank line between outputs. The third line carries no `\n`, so its ¶ is the only newline printed and nothing blank follows it.

The file-pointer is exhausted after one pass

Run the same loop again on the same `file`:

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

Nothing prints. The **file-pointer** `file` has already reached the end of the file during the last loop. To reset the file pointer to the beginning, open the file again:

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

This loop uses a bare `line` rather than `print(line)`, which is why the newline characters are visible here and were invisible in the previous transcript.

Closing files

If you do not close your files they remain open and vulnerable to side-effects. That is, you may find some of your file is missing, or extra bits in your file, when you neglect to close after use.

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

Alternative to close

Using the `with` construct has the advantage of **closing your file for you** — even if the program crashes while it is executing its code block.

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

Exercise: counting empty lines

Write a function `def num_empty_lines(file: str) -> int:` that counts the number of **empty lines** (those that *only* contain `\n`) in a file.

```
>>> def num_empty_lines(path: str) -> int:
...     """
...     Count 'empty' lines.
...     """
...     ans = 0
...     with open(path, 'r') as file:
...         for line in file:
...             if line == '\n':
...                 ans = ans + 1
...     return ans
>>> num_empty_lines("filename.txt")
```

The answer calls the parameter `path` rather than `file`. `path` is the better name: the argument is a *file name*, not an open file, and the next exercise turns on exactly that distinction.

The equivalent written with the compound assignment operator, and with the file variable named so it cannot be confused with the parameter:

```
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
```

The same function over a file-pointer

Write a function that counts empty lines, but takes an already-open file:

```
from typing import TextIO

def num_empty_lines(file_pointer: TextIO) -> int:
```

This is different from the last one, which took a **name** and not a **pointer** (to a file).

`TextIO` is a new doc-hint type, imported from `typing`.

```
from typing import TextIO

def num_empty_lines(file_pointer: TextIO) -> int:
    """
    Count 'empty' lines.
    """
    ans = 0
    for line in file_pointer:
        if line == '\n':
            ans = ans + 1
    return ans

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

Two consequences of the change, both visible in the code:

- There is no `open` and no `with` inside the function any more — **the caller** now owns opening (and closing) the file.
- The function is handed a pointer that may already have been partly consumed. It counts from wherever the pointer happens to be, not from the start of the file.

Reading numbers from a file

Given `numbers.dat`, one number per line (the file ends without a newline after the `6`):

```
1¶
2¶
3¶
4¶
5¶
6
```

When reading from a file we are **always reading strings**. Do not forget to **cast** your strings to more appropriate types when necessary:

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

`int("1\n")` works because `int` tolerates surrounding whitespace. That is the reason no `strip()` is needed here (see python-string-methods[3]).

Exercise: the highest-rated band

Write a function that finds the highest-rated band in `bands.txt`:

```
Band,Rating,Plays          <- this is called a header
Metallica,7,512
The Black Keys,8,193
Amy MacDonald,5,290
The Killers,9,402
U2,9,789
Portishead,6,116
Wiggles,2,1849
Coldplay,7,0
Taylor Swift,11,1234567
Drake,0,1
```

Each line of the body goes through the same short pipeline:

`line` → `.split(',')` → `band, rating, _` → `int(rating)` → compare against the running best

```
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 passes the > test

    with open(path, "r") as file:
        file.readline()   # skip the header
```

[3] courses/csse1001/python-string-methods.pdf

```
        for line in file:
            band, rating, _ = line.split(',')   # don't give names to values you don't intend
            rating = int(rating)
            if rating > current_highest_rating:
                current_highest_rating = rating
                current_most_popular_band = band

    return current_most_popular_band
```

The three comments in that block mark the three ideas in it:

- `-float('inf')` as the starting rating **guarantees** the first row passes the `if rating > current_highest_rating:` test, so no special case is needed for "the first line".
- The bare `file.readline()` before the loop **skips the header** — it is called for its side effect on the file-pointer, and its return value is thrown away. This only works because the pointer moves (see Figure 2).
- `_` is the conventional name for a value you unpack but do not want: *don't give names to values you don't intend on using.*

The parameter `path` is a `str`, not a `TextIO`: the body passes it straight to `open(path, "r")`, exactly as in the earlier exercise.

The third column, `Plays`, is unpacked into `_` and thrown away. On this particular data the highest rating (11) and the highest play count (1234567) both belong to Taylor Swift, so a version that ranked by the wrong column would still return the right answer here.

Exercise: an arbitrary attribute

Extend the previous answer to take an arbitrary file **with a header of attributes** like

```
name,grade,age
```

and write a function

```
def most(path: str, attribute: str) -> str:
```

that finds the maximum of that attribute.

The header is what makes it possible: `attribute` has to be looked up in the header row to find *which* column to compare.

Exercise: reading a Sudoku board

> Write a function `def read_board(fname: str) -> list[list[int]]:` that, given a file containing a partially filled Sudoku board, returns a list of lists of integers with `None` in unfilled spots.

In `board.txt`, `|` separates the 3x3 blocks column-wise, a row of `-` separates them row-wise, and a space marks an empty cell:

```
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
```

A return type of `list[list[int]]` has no room for the `None`s the statement asks for, so the stub is annotated `list[list[int | None]]`:

```
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:

```
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
```

The same thing with a list comprehension. Where `v1` simply declines to append on `|` and on the trailing newline, a comprehension has to *filter* both out explicitly, and each omission costs a different amount: drop the `if x != '|'` and every row comes back with eleven entries instead of nine, drop the `rstrip("\n")` and every row but the file's last comes back with ten.

```
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.rstrip("\n") if x != '|']
            board.append(row)
    return board
```

See python-comprehensions[4] for the comprehension form.

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).
    """
```

Writing to files

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

Both calls write exactly the characters given. The newlines in the example are part of the strings themselves, not something the call adds.

Careful! Opening a file for write will create the `numbers.dat` file, or **overwrite the old one** if it exists.

[4] courses/csse1001/python-comprehensions.pdf

Appending to files

Appending will open a file **without over-writing**, instead **appending** to the end of the file. A file is created if one does not exist.

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

See python-file-io[5] for a side-by-side picture of what `r`, `w` and `a` each do to an existing file.

Summary

We can **read** strings **from** files and **write** strings **to** files. A **file-pointer** is moved every time a line from the file is read. Therefore, if we want to read a line twice the file must be opened twice.

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

[5] courses/csse1001/python-file-io.pdf
[6] courses/csse1001/2025-09-01-testing.pdf