CSCI 145 · Introduction to Computing · Fall 2026

Files and File Formats

Lecture 14 — how a program remembers

Outline

- Reading files
- Writing files
- File formats: CSV and JSON

Get set up

1. Download day14.zip
2. Drag `day14.zip` to your `class` folder and extract it there
   Mac — double-click it. **Windows** — right-click it and choose "Extract All..."
   Once you see the `day14` folder, delete `day14.zip`
3. Launch Thonny and open `cs145/class/day14/day14_live.py`
   Click Run once, so Python looks for files in the `day14` folder
4. Turn on the Files pane: View → Files

Where did Ana and Ben go?

SIGNINS.PY

```
signins = []
name = input("Name: ")
while name != "":
    signins.append(name)
    name = input("Name: ")
print(signins)
```

RUN 1

```
>>> %Run signins.py
Name: Ana
Name: Ben
Name:
['Ana', 'Ben']
```

RUN 2

```
>>> %Run signins.py
Name: Cy
Name:
['Cy']
```

- Variables last only while the program runs
- To remember anything, a program saves it in a **file**

Part 1 of 3

Reading files

> A **file** is text or data saved on a disk under a name
> It's still there after your program stops

What's in a text file?

IN A TEXT EDITOR

signins.txt

Ana
Ben
Cy

WHAT'S REALLY THERE

signins.txt

Ana `\n`
Ben `\n`
Cy `\n`

AS ONE PYTHON STRING

```
'Ana\nBen\nCy\n'
```

- `\n` is the **newline** character
  One character, typed as two
- A text file is one long string
  with a `\n` at the end of each line
- `\` starts an escape: `\t` is a tab,
  `\"` a quote, `\\` a backslash

```
>>> print("Ana\nBen")
Ana
Ben
```

- `print` shows `\n` as a line break
- Images and sound are files too,
  stored differently (Lectures 15–17)

Opening a file

AS IN NOTES 12

```
f = open("signins.txt", "r")
text = f.read()
f.close()
print(text)
```

USING WITH

```
with open("signins.txt", "r") as f:
    text = f.read()
print(text)
```

- `open(filename, mode)` opens a file
  `"r"` means read, and it's the default: `open("signins.txt")`
- `f` names the open file
- Read from `f` inside the indented block
- When the block ends, Python closes the file for you
  No `f.close()` to forget, so we'll use `with` from now on

Three ways to read

```
with open("signins.txt") as f:
    ...
```

signins.txt

Ana `\n`
Ben `\n`
Cy `\n`

INSIDE THE BLOCK	GIVES	
`f.read()`	`'Ana\nBen\nCy\n'`	One string
`f.readlines()`	`['Ana\n', 'Ben\n', 'Cy\n']`	A list of lines
`for line in f:`	`line` is `'Ana\n'`, then `'Ben\n'`, then `'Cy\n'`	One line at a time

- Every line keeps its `\n`
- `for line in f` reads one line per pass, so it works for huge files too

What does this print?

```
with open("signins.txt") as f:
    for line in f:
        print(line)
```

signins.txt

Ana \n
Ben \n
Cy \n

A

```
Ana
Ben
Cy
```

B

```
Ana

Ben

Cy
```

C

```
AnaBenCy
```

D

```
Ana\nBen\nCy\n
```

What does this print?

```
with open("signins.txt") as f:
    for line in f:
        print(line)
```

signins.txt

Ana \n
Ben \n
▶ Cy \n

`line` → `'Cy\n'`

OUTPUT

```
Ana

Ben

Cy
```

A

```
Ana
Ben
Cy
```

B

```
Ana

Ben

Cy
```

C

```
AnaBenCy
```

D

```
Ana\nBen\nCy\n
```

Answer: B

Each `line` ends with `\n`, and `print` adds another

Why not the others?

A: forgets the `\n` in each line
C: `print` always ends with a newline
D: `print` shows `\n` as a line break

Cleaning up a line

One line of a file of Vermont peaks and their heights

```
>>> line = "Mansfield,4393\n"
>>> line.strip()
'Mansfield,4393'
>>> parts = line.strip().split(",")
>>> parts
['Mansfield', '4393']
>>> parts[1] + 7
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not
"int") to str
>>> int(parts[1]) + 7
4400
```

- `.strip()` removes spaces and `\n` from both ends (Lecture 4)
- `.split(",")` splits at each comma into a list of strings
- Everything read from a file is a string
- `int(...)` or `float(...)` converts it
- The fix for the poll: `print(line.strip())`
- Skip a blank line: `if line.strip():` `int("")` would crash

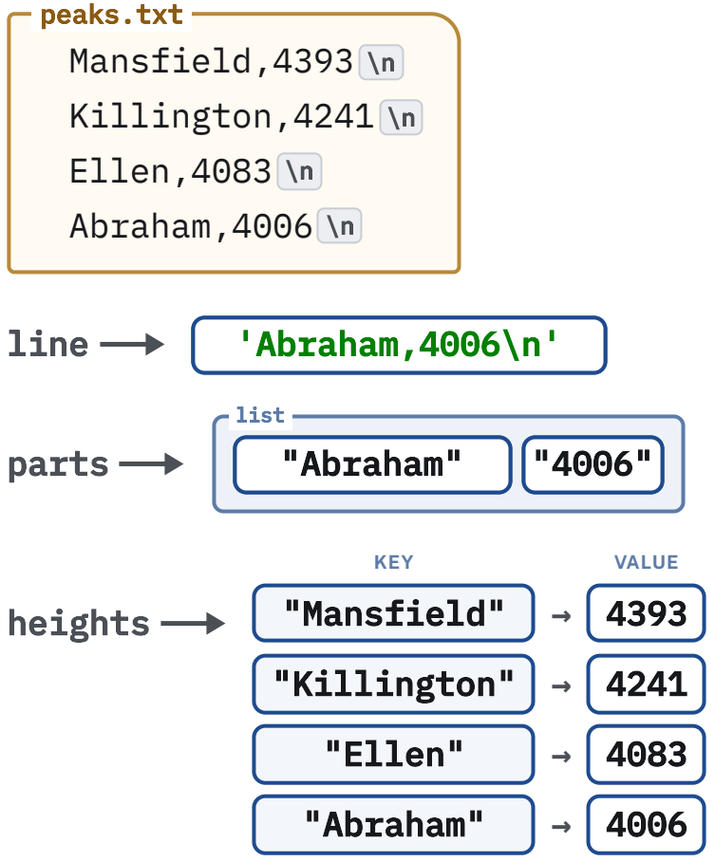
Reading a file into a dictionary

```
def read_peaks(filename):
    heights = {}
    with open(filename) as f:
        for line in f:
            parts = line.strip().split(",")
            heights[parts[0]] = int(parts[1])
    return heights

print(read_peaks("peaks.txt"))
```

OUTPUT

```
{'Mansfield': 4393, 'Killington': 4241, 'Ellen': 4083,
'Abraham': 4006}
```



Where does Python look?

📁 **cs145/class/day14**
day14_live.py
day14_files.py
peaks.txt
scores.csv
signins.txt
...

```
>>> read_peaks("peaks.txt")
{'Mansfield': 4393, 'Killington': 4241, 'Ellen': 4083,
'Abraham': 4006}
>>> read_peaks("peak.txt")
Traceback (most recent call last):
  ...
FileNotFoundError: [Errno 2] No such file or directory:
'peak.txt'
```

- A name alone is a **relative path**: Python looks in the folder your program runs from
- Thonny runs a file from that file's own folder
  Keep data files next to your `.py` file
- A **full path** works from any folder, but only on your computer:
  `/Users/ana/cs145/class/day14/peaks.txt`

Activity A

In `day14_files.py`, write `read_scores`

PATTERNS

Open a file to read	`with open(filename) as f:`
Get all the lines as a list	`lines = f.readlines()`
Skip the first line	`for line in lines[1:]:`
Remove the `\n`	`line.strip()`
Split at commas	`line.split(",")`
Text to a number	`int(text)`

Part 2 of 3

Writing files

> Open a file in **write** mode to save your program's results
> They're still there the next time it runs

Three modes

```
with open(filename, mode) as f:
```

Mode	Means	If the file exists	If it doesn't
`"r"`	Read (the default)	Reads it from the start	`FileNotFoundError`
`"w"`	Write	Erases it first	Creates it
`"a"`	Append	Adds to the end	Creates it

- Careful: `"w"` erases the file as soon as it opens, even if you never write anything
- Never open a file you need with `"w"`

Writing to a file

NO NEWLINES

```
with open("names.txt", "w") as f:
    f.write("Ana")
    f.write("Ben")
```

names.txt

AnaBen

NEWLINES ADDED

```
with open("names.txt", "w") as f:
    f.write("Ana\n")
    f.write("Ben\n")
    f.write(str(92) + "\n")
```

names.txt

Ana `\n`

Ben `\n`

92 `\n`

- `print` adds a newline
  `f.write` adds nothing, not even a space
- `f.write` takes one string: `f.write(92)` is a `TypeError`
  Convert with `str(92)` first

Find the bugs

Should copy the grades to a new file, leaving out one lowest grade

```
def drop_lowest_grade(filename_in, filename_out):
    with open(filename_in, "r") as f_in:
        grades = []
        for line in f_in:
            grades.append(int(line))
    with open(filename_out, "w") as f_out:
        for grade in grades:
            if grade != min(grades):
                f_out.write(grade)
```

- `write` needs a string: `TypeError`
- Even as strings, the grades would run together with no `\n` between them
- Leaves out both 80s, not one

grades.txt

80 \n
100 \n
80 \n
90 \n

dropped.txt

100 \n
80 \n
90 \n

```
def drop_lowest_grade(filename_in, filename_out):
    with open(filename_in, "r") as f_in:
        grades = []
        for line in f_in:
            grades.append(int(line))
    grades.remove(min(grades))
    with open(filename_out, "w") as f_out:
        for grade in grades:
            f_out.write(str(grade) + "\n")
```

- `int(line)` skips the `\n`: `int("80\n")` is 80

Now the program remembers

SIGNINS.PY

```
with open("signins.txt", "a") as f:
    name = input("Name: ")
    while name != "":
        f.write(name + "\n")
        name = input("Name: ")

signins = []
with open("signins.txt") as f:
    for line in f:
        signins.append(line.strip())
print(signins)
```

- `"a"` adds to the end of the file (and creates it if it isn't there)
- With `"w"`, each run would erase the names from earlier runs

RUN 1

```
>>> %Run signins.py
Name: Dee
Name:
['Ana', 'Ben', 'Cy', 'Dee']
```

RUN 2

```
>>> %Run signins.py
Name: Eli
Name:
['Ana', 'Ben', 'Cy', 'Dee', 'Eli']
```

signins.txt

Ana\n
Ben\n
Cy\n
Dee\n
Eli\n

Activity B

Write `write_honor_roll`
Run the file, then open `honor_roll.txt` to check it

PATTERNS

Open a file to write

```
with open(filename, "w") as f:
```

Write one line

```
f.write(text + "\n")
```

Part 3 of 3

File formats

> A **file format** is an agreement about how a file's text is laid out so that different programs can read it

CSV: a table as text

IN A SPREADSHEET

planet	dist_sun_au	diameter_km
Mercury	0.39	4879
Venus	0.72	12104
Earth	1.0	12756
Mars	1.52	6792
Jupiter	5.2	142984
Saturn	9.54	120536
Uranus	19.2	51118
Neptune	30.06	49528

IN A TEXT EDITOR

planets.csv

```
planet,dist_sun_au,diameter_km
Mercury,0.39,4879
Venus,0.72,12104
Earth,1.0,12756
Mars,1.52,6792
Jupiter,5.2,142984
Saturn,9.54,120536
Uranus,19.2,51118
Neptune,30.06,49528
```

- A **header** row names the columns
- Each line after it is one row
- Commas separate the fields: **comma-separated values**
- Same file, two views: a spreadsheet shows a table, a text editor shows commas

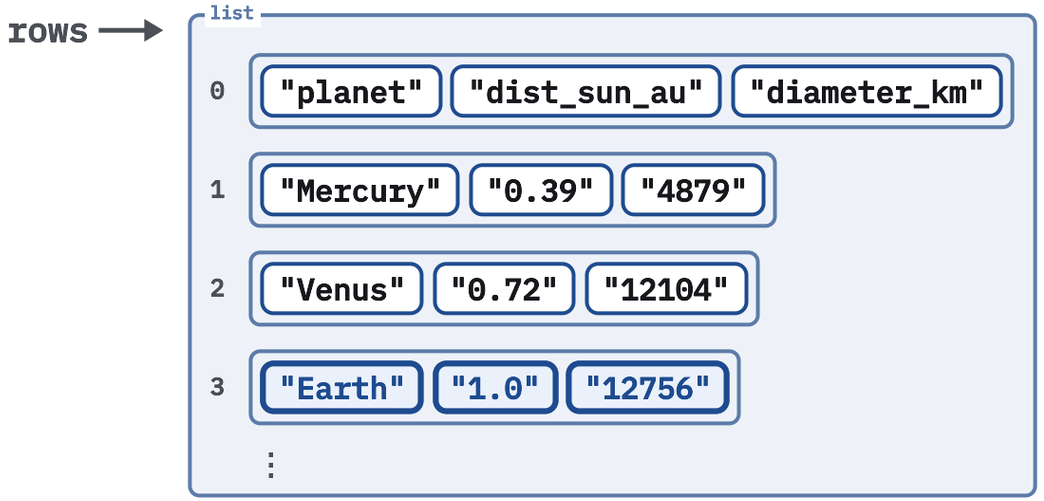
`peaks.txt` was a CSV file with no header

Reading a CSV file

```
import csv
with open("planets.csv") as f:
    rows = list(csv.reader(f))
print(rows[0])
print(rows[3])
```

OUTPUT

```
['planet', 'dist_sun_au', 'diameter_km']
['Earth', '1.0', '12756']
```



- Python's `csv` module reads CSV files
- `csv.reader` turns each line into a list
  No `strip` or `split` needed
- `rows[0]` is the header
  `rows[1:]` holds the planets
- Still strings: `float(rows[3][1])` is 1.0

Writing a CSV file

The planets more than 50,000 km across, from `rows`

```
with open("big_planets.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["planet", "diameter_km"])
    for row in rows[1:]:
        if int(row[2]) > 50000:
            writer.writerow([row[0], int(row[2])])
```

big_planets.csv

```
planet,diameter_km\n
Jupiter,142984\n
Saturn,120536\n
Uranus,51118\n
```

- `csv.writer(f)` writes rows to the file
- `writerow` takes a list
  It adds the commas and the newline
- Numbers are fine: `writerow` converts them
- `newline=""` keeps Windows from adding blank lines

Why use the csv module?

cities.csv

```
city,population\n
"Washington, D.C.",689545\n
Burlington,44743\n
```

SPLITTING LINE 2 AT COMMAS

```
['"Washington', ' D.C."', '689545']
```

Three pieces: the name is split in two

READING LINE 2 WITH THE CSV MODULE

```
['Washington, D.C.', '689545']
```

Two pieces, quotes removed

- A field with a comma inside goes in quotes
- `csv.reader` and `csv.writer` handle the quotes for you
- CSV has no types: every field comes back a string

JSON: Python-like data in a file

settings.json

```
{
  "player": "Ana",
  "level": 3,
  "sound": true,
  "high_scores": [120, 95, 80]
}
```

planets.json

```
[
  {"planet": "Mercury", "diameter_km": 4879},
  {"planet": "Venus", "diameter_km": 12104},
  ...
]
```

- **JSON**: JavaScript Object Notation
  Used by most languages and websites
- Looks like Python dictionaries and lists
- Keeps types: numbers, strings, `true`
  JSON's `true` is Python's `True`
- Data can nest: a list inside a dictionary
- A table becomes a list of dictionaries

The json module

```
import json
with open("settings.json") as f:
    settings = json.load(f)
print(settings["high_scores"][0])
settings["level"] = 4
with open("new_settings.json", "w") as f:
    json.dump(settings, f)
```



OUTPUT

```
120
```

new_settings.json

```
{"player": "Ana", "level": 4, "sound": true, "high_scores": [120, 95, 80]}
```

- `json.load(f)` : file to Python data
- `json.dump(data, f)` : Python data to file
- `json.dump(data, f, indent=2)` puts each item on its own line

CSV or JSON?

	CSV	JSON
Looks like	A table: rows and columns	Dictionaries and lists
Types	Every field is a string	Numbers, strings, true/false
Shape	Every row has the same fields	Any nesting
Easy to open in	A spreadsheet	Any programming language

Every student's score on each homework **CSV**

A game's saved settings **JSON**

Each state's name, capital, and list of counties **JSON**

The words of a song, line by line **Plain text**

Activity C

Write `save_json`
If you have time, write `count_words`

PATTERNS

Open a file to write	`with open(filename, "w") as f:`
Save as JSON	`json.dump(data, f)`
Open a file to read	`with open(filename) as f:`
Go through the lines	`for line in f:`
Split into words	`line.split()`
Lowercase	`line.lower()`

To do

- Review this slide deck and Notes 12
- Homework 4 is due Thursday 10/29
- Prepare for Lab 4 this Friday
- Quiz 7 (Lectures 13–14) is at the start of next class
- Exam 3 (Lectures 10–14) is Friday 11/6 in lab

See practice materials

Next time

Lecture 15 — representing values