## Warmup (L14)

I think your midterm yesterday was enough of a warmup ☺

# Files

M6

## Reading files

CS114 M6

## Data

- We’ve seen lots of types of data
- But, so far, everything we’ve seen was *in* the Python code
- Usually we want to deal with data created elsewhere
- E.g., you get the results of an experiment, and then, later, want to do some computation on that

## Files

- Files contain data
- Generally permanent (until deleted)
- Can be written and read by different programs
- Allow exchange of data between programs
- Have names, so the *filesystem* is like a string→data dictionary

## Files

- There are lots of kinds of data, and so lots of kinds of files
  - .txt, .py, .ipynb, .csv, .json, .png, .webp, .webm, .mp4, .pptx, …
  - The *filename extension* is just a hint though. Nothing stops you from using a misleading extension.
- We've seen a few of these in this course

## Files

- You need to know how the data is organized in your file to make use of it
- Often need bespoke code for each kind of data
- We'll focus on text files for now

# Getting a file onto Jupyter

- Y'know the "`!wget`" command you've been using to get starter code?
- That's a tool to get a file from the web (web-get), and it puts the file in your Jupyter directory
- We'll use it to get other files
- You can also upload your own files ⬆

# FileLib

- Because code to read and write data from/to files is very boilerplate, we provide a library of functions to do it

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/filelib.ipynb
```

- We will not go into detail about how each function is implemented (but they're very short, feel free to read them)
- You can use these functions in assignments and exams

## Fetch a file

- Let’s get a big text file to play with

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/a-tale-of-two-cities.txt
```

## Read a file

- All of filelib's functions are named *x*`_to_`*y*, and convert in one direction or the other between file data and Python data
- For right now, we're concerned with plain text files, so we want `txt_to_`*:
    - `txt_to_str`: Reads in a text file as one (maybe long) string
    - `txt_to_ls`: Reads in a text file as a **l**ist of **s**trings, one string per line in the file

## Read a file

- Let’s just print each line of a-tale-of-two-cities.txt:

```
cites = txt_to_ls("a-tale-of-two-cities.txt")
for line in cites:
    print(line)
```

## Text files

We got all the lines, but it printed weird… what’s happening?

## Text files

```
It was the best of times, it was the worst of times, it was the age of

wisdom, it was the age of foolishness, it was the epoch of belief, it

was the epoch of incredulity, it was the season of Light, it was the
```

- Why is it double-spaced?
- txt_to_ls *includes* the line break in each line
- `print` always put a line break anyway, so we end up with two!

# Text files

- String methods to the rescue: `str.strip()` will strip all the whitespace from the beginning and end of the string

```
cities = txt_to_ls("a-tale-of-two-cities.txt")
for line in cities:
    print(line.strip())
```

## Using file data

CS114 M6

## Using the data

- Let’s do a distribution chart of word use in A Tale of Two Cities
- First let’s remember our distribution chart functions from earlier…

```python
import typing

def distribution(
    lst: typing.Sequence
) -> dict[typing.Any, int]:
    r = {}
    for val in lst:
        if not (val in r):
            r[val] = 0
        r[val] = r[val] + 1
    return r
```

```python
def distributionChart(
    lst: typing.Sequence
) -> None:
    dist = distribution(lst)
    for key in sorted(dist):
        print(key, "*" * dist[key])
```

## Using the data

- For this, we'll need two more features:
    - `str.split()`: Split a string by whitespace (we've seen this)
    - `str.lower()`: Get the lower-case version of a string (capitals will confuse our distribution)

## Using the data

```python
def distributionChart(
    lst: typing.Sequence
) -> None:
    dist = distribution(lst)
    def distCount(key):
        return dist[key]
    for key in sorted(dist, key=distCount, reverse=True):
        print(key, "*" * dist[key])

cities = txt_to_str("a-tale-of-two-cities.txt")
words = cities.strip().lower().split()
distributionChart(words)
```

## That… kinda worked

- Common words were common, but by just splitting on whitespace, we considered “but,” to be a word (etc.)
- We could keep adding exceptions to our rules until it solves the problem, but wouldn’t it be simpler if our data was in a more consistent format in the first place?
- Seems like our problem was that text files are messy. Let’s look at other formats for data.

# In-lecture quiz (L14)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q1: What is `len(x)` after this code runs?

```
x = txt_to_ls("a-tale-of-two-cities.txt")
```

A. 10 (the number of words in the last line)
B. 59 (the number of characters in the last line)
C. 16,282 (the number of lines in the file)
D. 776,877 (the number of characters in the file)

# In-lecture quiz (L14)

- https://student.cs.uwaterloo.ca/~cs114/quiz/
- Q2: What is `len(x)` after this code runs?

```
for line in txt_to_ls("a-tale-of-two-cities.txt"):
    x = list(line)
```

A. 10 (the number of words in the last line)
B. 59 (the number of characters in the last line)
C. 16,282 (the number of lines in the file)
D. 776,877 (the number of characters in the file)

## It’s all just text?

CS114 M6

```
1984,8,26.6,26.64,-0.04
1984,9,26.38,26.56,-0.19
1984,10,26.04,26.53,-0.49
1984,11,25.52,26.52,-1
1984,12,25.26,26.51,-1.25
1985,1,25.39,26.57,-1.17
1985,2,26.04,26.75,-0.71
1985,3,26.5,27.17,-0.67
1985,4,26.65,27.59,-0.93
1985,5,26.91,27.66,-0.75
1985,6,26.81,27.46,-0.65
1985,7,26.55,27.02,-0.47
1985,8,26.29,26.64,-0.35
1985,9,26.02,26.56,-0.55
1985,10,26.23,26.53,-0.3
1985,11,26.33,26.52,-0.2
1985,12,26.19,26.51,-0.32
1986,1,25.89,26.46,-0.56
1986,2,26.06,26.66,-0.6
1986,3,26.88,27.14,-0.26
1986,4,27.49,27.58,-0.08
1986,5,27.41,27.68,-0.27
1986,6,27.42,27.43,-0.01
1986,7,27.18,27.01,0.17
1986,8,27.17,26.66,0.51
1986,9,27.24,26.59,0.65
1986,10,27.51,26.54,0.98
1986,11,27.7,26.5,1.2
1986,12,27.71,26.47,1.24
1987,1,27.68,26.46,1.22
1987,2,27.89,26.66,1.23
1987,3,28.27,27.14,1.13
1987,4,28.4,27.58,0.82
1987,5,28.56,27.68,0.88
1987,6,28.64,27.43,1.21
1987,7,28.58,27.01,1.57
1987,8,28.41,26.66,1.76
```

# Humans love language

- All computer data is just bits (base-2/**b**inary dig**its**)
- There are standards for using those bits to encode text
  - We don’t need to know the details, but if you’re curious, the keywords you’re looking for are “ASCII” and “Unicode”
- Because humans love language, we usually encode other data… as text

## Humans love language

- Think about Python itself
- The actual code the machine runs (called *machine code*) is just bits; it does not resemble text!
- Python (and every other programming language) is an attempt to textualize computation to make it more human!

# Wallowing in pedantry

- Be very careful about representing numbers
- Although we can’t see the bits, an int or a float are numbers stored as binary
- In a text file, the number is
    - Converted to base-10 for ape convenience (or was never bits in the first place),
    - written in the glyphs used for numbers (digits), then
    - encoded as a string.
- In short: `"42"` and `42` are very different things!

## Spreadsheets

- It’s my friend and yours, *spreadsheets!*
- Basic idea:
  - Different kinds of data form columns
  - Connected data form rows
  - Each piece of data is a cell in the row
- By convention, first row labels data
- Spreadsheet software is just a (very different) computer programming language

# Let’s look at data

- Some measurements of sea surface temperature from NOAA
  - (We’re going to use this CSV file as a demo *a lot*, but the actual data in it isn’t significant for us)

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/nino34.csv
```

# CSV

- *Comma-Separated Values*
- A common *interchange format* for spreadsheets
  - I.e., a shared format that everyone understands. All spreadsheet software can export as CSV.
- It’s just text!
- One row per line,
- cells separated by commas.

## Make a CSV

- When you create a file in Jupyter, it will create a text file by default (.txt)
- Just rename it to .csv to make a CSV file: remember, it’s just text!
- If you want to edit it, right click and open in “editor” (the spreadsheet viewer in Jupyter can’t edit)

# Reading a CSV

- The first row in a CSV file is usually labels for each column
- FileLib provides three CSV readers:
  - `csv_to_ld`: Converts a CSV file into a list of dictionaries of column name to data
  - `csv_to_dl`: Converts a CSV file into a dictionary of column name to lists of data
  - `csv_to_ll`: Converts a CSV file into a list of rows, each of which is a list of data
- CSV is text, so everything comes out as strings!

## Reading a CSV

Let's show the output of these `csv_to_*` functions with our example CSV

## Using CSV data

- Let’s find the monthly average temperature for each month over the recorded period in nino34.csv
- (That is, the monthly average for January, the monthly average for February, etc.)

## averageOf

- First we'll need to recall our `averageOf` function from earlier:

```
def averageOf(l: list[float]) -> float:
    sum = 0.0
    for val in l:
        sum = sum + val
    return sum / len(l)
```

# Using CSV data

Mind your ranges!
Upper-bound exclusive!

```
measurements = {}
for m in range(1, 13):
    measurements[m] = []
nino = csv_to_ld("nino34.csv")
for row in nino:
    measurements[int(row["MON"])].append(
        float(row["TOTAL"])
    )
averages = {}
for m in range(1, 13):
    averages[m] = averageOf(measurements[m])
print(averages)
```

# CSV labels

- Note that the first row of nino34.csv didn’t have any data, just the *labels* for the data (these became our keys):

```
YR,MON,TOTAL,ClimAdjust,ANOM
```

- It’s just a convention to label CSV columns. Sometimes CSV has no labels.

# Unlabeled CSV

- `csv_to_ll` treats each row the same (it has no special treatment for the labels row)

```
nino = csv_to_ll("nino34.csv")
for row in nino:
    print(row)
```