Warmup (L12)

- You have a `dict[str, float]`
- Sort the *keys* of the dictionary in decreasing order, sorting by the *values* the keys are associated with in the dictionary

```
x = {"A": 42, "B": -3, "C": math.pi}
# We want ["A", "C", "B"]
```

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 

Step one: 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
```

Step two: opening a file

- (Note: We're going to see a better way to do this in a moment)

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

Text files

- Text files are just very long strings
- Often useful to view the text one line at a time
- So, when you loop over the file, you get one line at a time
- But… it’s printing weird…

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

- What's with the extra line breaks? Why is it double-spaced?
- When we loop through a text file, the strings we get *include* the line break
- `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 = open("a-tale-of-two-cities.txt")
for line in cities:
    print(line.strip())
cities.close()
```

File handles and cleanup

CS114 (M6)

`open`

- `open` returns a file *handle*
  - It lets you get a *handle* on the file data
- A file handle is an *iterable*
- Iterables are like sequences, but you can't use `len` on them
  - Python doesn't know how long the file is until it's finished reading it
  - More precisely, sequences are sized iterables

New type: iterable

- `typing.Iterable`
- `typing.Iterable[str]` for, e.g., an iterable of strings
- (We've seen enough types now that hopefully this is enough!)

Python cleanup

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

- We've never had to do anything like `cities.close()` before. Why now?

Python cleanup

- Python cleans up after itself, but files are outside of Python
- *You* must close the file when you're done
- What if your code crashes, or you have complicated code paths?
- Cleanup is surprisingly error prone, so Python provides a built-in way to close a file after a block of code has used it

Step two: opening a file

- (Now the right way!)

```
with open("a-tale-of-two-cities.txt") as cities:
    for line in cities:
        print(line.strip())
```

New syntax!

- **`with`** *(anything)* **`as`** *(variable name)***`:`**
- A built-in way to do cleanup automatically
- Equivalent to assigning to a variable, but has a block of code like **`if`** or **`while`**
- When the block is done, automatically cleans up (in this case, closes the file)
- Even cleans up if the code crashes, early returns, etc.

Step three: 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…

```
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
```

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

Step three: 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)

Step three: using the data

```
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])

words = []
with open("a-tale-of-two-cities.txt") as cities:
    for line in cities:
        words = words + line.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 (L12)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: What is `len(x)` after this code runs?

```
with open("a-tale-of-two-cities.txt") as rdr:
    x = list(rdr)
```

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 (L12)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What is `len(x)` after this code runs?

```
with open("a-tale-of-two-cities.txt") as rdr:
    for line in rdr:
        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)

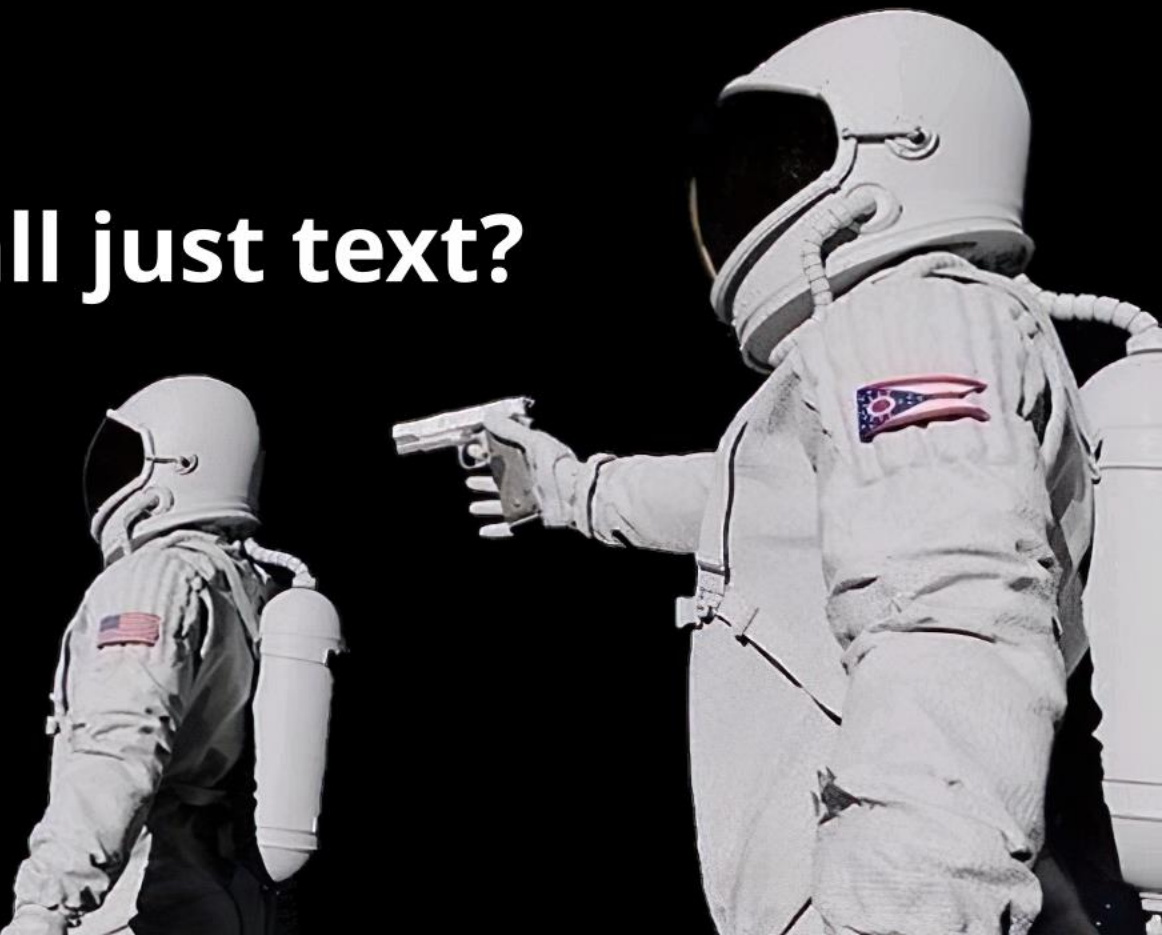
It’s all just text?

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

Always has been

It's all just text?



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

```
!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.

CSV

- You can just `line.split(",")`
  - `str.split` takes an optional argument, so you can choose how to split your lines
- Let's show that in code

```
with open("nino34.csv") as nino:
    for line in nino:
        for col in line.split(","):
            print(col)
```

CSV

- This worked for nino34.csv because NOAA is pretty careful with their CSV files
- But, some details of CSV are messy and annoying…
  - Cells *may* be in quotes… or not
  - Commas *may* have a space after them… or not
  - There *may* be the same number of cells every row/line… or not

CSV

- … luckily, there's a module in Python that already knows how to deal with all this!

```
import csv
with open("nino34.csv") as ninoCSV:
    nino = csv.DictReader(ninoCSV)
    for row of nino:
        print(row)
```

CSV readers

- `csv.DictReader` (note the capitalization!) gives us an *iterable* of *dictionaries*
  - `typing.Iterable[dict[str, str]]`
- The dictionary we got was string-to-string
- Most of the data was numbers, but how was the reader to know that?
- We must convert as we go!

Using CSV data

- Let’s find the monthly average temperature for each month over the recorded period
- (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

```
import csv
measurements = {}
for m in range(1, 13):
    measurements[m] = []
with open("nino34.csv") as ninoCSV:
    nino = csv.DictReader(ninoCSV)
    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)
```

Mind your ranges!
Upper-bound exclusive!

Examining the labels

- Each cell of a row is called a *field*, and the labels are the fields' *names*
- `DictReader`s let you see the list of fields:

```
with open("nino34.csv") as ninoCSV:
    nino = csv.DictReader(ninoCSV)
    for row in nino:
        for field in nino.fieldnames:
            print(field, "is", row[field])
```

`fieldnames` foibles

- The type of `fieldnames` is slightly strange, because it may fail to detect fields
- Use this incantation to get a sensible type out:

```
fieldnames = list(rdr.fieldnames or [])
```

  - This just means “if fieldnames are there, convert them into a list, otherwise use an empty list”