Warmup (L13)

Create a function to sum *every* column in a CSV file, returning a dictionary associating each column name with the sum. You may assume that every column contains only numbers.

```
def sumEveryColumn(filename: str) -> dict[str, float]:
```

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”

Unlabeled CSV

- The `csv` module also provides a CSV reader that just splits into cells (no dictionaries)
- `csv.reader(fileHandle)`
- Returns a `typing.Iterable[list[str]]`
- Let's use nino34.csv to demonstrate
  - Most real data has labels, so it's hard to find a useful example that doesn't!

Unlabeled CSV

```
import csv

with open("nino34.csv") as ninoCSV:
    nino = csv.reader(ninoCSV)
    for line in nino:
        print(line)
```

Writing files

CS114 (M6)

Writing

- So far we've been consuming data produced by others
- We've output data to the screen (`print`) and into variables, but not yet into files
- Let's change that!

Opening in writing mode

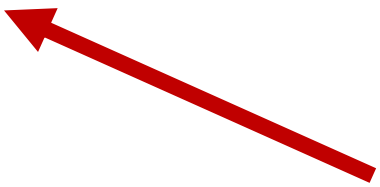
- `open` has an optional second parameter for the *mode*
  - Actually, `open` has tons of optional parameters, but we just care about this one
- `open("output.txt", "w")` means "open output.txt for writing"
- WARNING: Python will happily overwrite files you already have! Make sure you know what you're opening!

Reverse every line

- Let’s write a function to copy text from one file to another, but reverse every line

Reverse every line

```
def reverseLines(
    outputFile: str, inputFile: str
) -> None:
    with open(outputFile, "w") as o:
        with open(inputFile) as i:
            for line in i:
                o.write(
                    line.strip()[::-1] +
                    "\n"
                )
```



What's this thing???

Line-break Hell!

- Remember: `print` automatically adds a line break
- `write` doesn't
- If we'd done `write` without the `\n`, it would've written one *very* long line to the file!
- This weird `\n` is an *escape sequence*: it means "interpret this as a line break, even though I've literally written a backslash and then an n"
  - (n stands for "new line")

`print` is so smart

- print takes any number of arguments, adds spaces, prints any type… it's very smart
- `.write` takes a string. Not smart.
- Python has a way of formatting values into strings nicely: *format strings*
- To demonstrate, let's write a function like `reverseLines`, but writing some more

Formatted lines

```
def lineInfo(
    outputFile: str, inputFile: str
) -> None:
    with open(outputFile, "w") as o:
        with open(inputFile) as i:
            for line in i:
                o.write(
f"Line length: {len(line)}. Line
palindrome: {line}{line[::-1]}\n"
                )
```

New syntax!

- Format strings are written like strings (in quotes), but with an '`f`' before the opening quote
- In format strings, everything in braces ({}) is run as Python code, and its result put in the string
- Extremely useful for writing to files, but useful anywhere you want something string'd

In-lecture quiz (L13)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: I tried to fix it but it's still printing double-spaced! Why?

```
with open("file.txt") as rdr:
    for line in rdr:
        line.strip()
        print(line)
```

A. `strip` is the wrong method
B. A **`for`** loop isn't the right way to get lines
C. `line.strip()` doesn't change `line`
D. Need `rdr.strip()`, not `line.strip()`
E. Python is cruel and vindictive

In-lecture quiz (L13)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What is the type of `x`?

```
with open("nino34.csv") as rdr:
    someCSV = csv.DictReader(rdr)
    x = list(someCSV)
```

A. This code has an error (no type)
B. `list[str]`
C. `list[dict[str, float]]`
D. `list[dict[str, str]]`
E. `dict[str, list[float]]`

Writing CSV files

CS114 (M6)

Writing CSVs

- Just like you can create a CSV reader for a reading file handle, you can create a CSV writer for a writing file handle
- `DictWriter` is a bit more fussy than `DictReader`
- To demonstrate, let's add a Fahrenheit field from nino34.csv into nino34f.csv

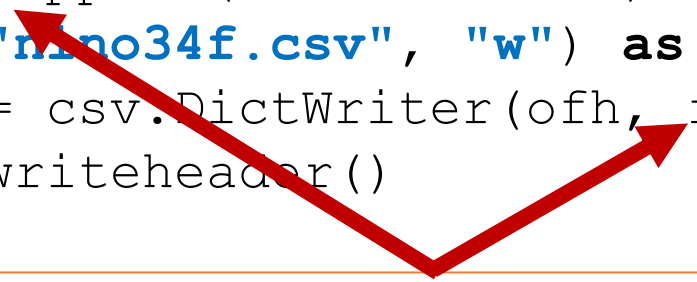
Writing CSVs

```
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    fieldnames = list(nino.fieldnames or [])
    fieldnames.append("Fahrenheit")
    with open("nino34f.csv", "w") as ofh:
        ninoF = csv.DictWriter(ofh, fieldnames)
        ninoF.writeheader()

        for row in nino:
            row["Fahrenheit"] = (
                float(row["TOTAL"])*9/5+32
            )
            ninoF.writerow(row)
```

Writing CSVs

```
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    fieldnames = list(nino.fieldnames or [])
    fieldnames.append("Fahrenheit")
    with open("nino34f.csv", "w") as ofh:
        ninoF = csv.DictWriter(ofh, fieldnames)
        ninoF.writeheader()
                float(row["TOTAL"])*9/5+32
            )
            ninoF.writerow(row)
```



`DictWriter` needs the field names to use

Writing CSVs

```
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    fieldnames = list(nino.fieldnames or [])
    fieldnames.append("Fahrenheit")
    with open("nino34f.csv", "w") as ofh:
        ninoF = csv.DictWriter(ofh, fieldnames)
        ninoF.writeheader()

        for row in nino:
            ninoF.writerow(row)
```

You must explicitly ask to write the first row (The row with labels, the *header*)

Writing CSVs

```
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    fieldnames = list(nino.fieldnames or [])
    fieldnames.append("Fahrenheit")
    ninoF = csv.DictWriter(ofh, fieldnames)
    ninoF.writeheader()

    for row in nino:
        row["Fahrenheit"] = (
            float(row["TOTAL"])*9/5+32
        )
        ninoF.writerow(row)
```

Dictionaries are mutable, so we can simply add Fahrenheit right into the row we already have

Writing CSVs

```
with open("nino34.csv") as ifh:
    nino = csv.DictReader(ifh)
    fieldnames = list(nino.fieldnames or [])
    fieldnames.append("Fahrenheit")
```

Since this is a *dictionary* writer, each row should be a *dictionary*. It knows how to write the row from the `fieldnames` you specified when creating it.

```
    for row in nino:
        row["Fahrenheit"] = (
            float(row["TOTAL"])*9/5+32
        )
        ninoF.writerow(row)
```

CSVs without labels

- `DictWriter` can write a file without labels: just don't use `.writeheader()`
- It still writes dictionaries though
- If you just want to use lists to write your rows, you want `csv.writer(fileHandle)`
- It behaves the same, but no `.writeheader()`, and `.writerow()` expects a list instead of a dictionary

Other file formats

CS114 (M6)

JSON

- CSV is structured, but two dimensional
- What if you have complex data with lists and dictionaries and all sorts of stuff in it?
- JSON is a standard format for complicated data

Example JSON

```
{
    "surname": "Richards",
    "given name": "Gregor",
    "height": 1.76,
    "places lived": ["USA", "Canada"]
}
```

- What an unfamiliar way of storing data! Never seen anything like that!

JSON is from programming

- JSON looks like how you would store data in a programming language, because… that's what it is
- It's *JavaScript Object Notation*, and JavaScript is a different programming language
- Python's syntax for data is nearly the same
  - This is partially because they share heritage, and partially Python being influenced by JSON

What can JSON do

- Strings
- Numbers
- Booleans
- Lists of anything it can store (including lists)
- Dictionaries
  - Key must be a string
  - Values can be anything JSON can store

Using JSON

- Python has a module for handling JSON: `import json`
- `json.load`: Load JSON from a file handle
- `json.loads`: Load JSON from a string
- `json.dump`: Write JSON from a file handle
- `json.dumps`: Write JSON as a string
- Let's demonstrate using our own ipynb notebook, which is stored as JSON!

Using JSON

```
import json

with open("exercise.ipynb") as ifh:
    ipynb = json.load(ifh)
print(ipynb["cells"][0]["source"])
ipynb["cells"][0]["source"] = [
    "print('Hello, world!')"
]
with open("output.ipynb") as ofh:
    json.dump(ipynb, ofh)
# Now we can look in output.ipynb!
```

JSON foibles

- Can’t store anything with cycles
  - (Remember when we put a list reference in the list? Yeah, can’t store that.)
- Can’t store every type
  - Python will convert lots of things, but this changes, e.g., tuples to lists
- References not preserved
  - ```
    x = [[0], [0]]
    x[0] = x[1]
    x[0] is x[1] # True
    x = json.loads(json.dumps(x))
    x[0] is x[1] # False
    ```
    ```

JSON vs. CSV

- JSON is way harder to read and understand than CSV
- CSV: Great when you have data that fits (2D-ish?) and human readability is a critical concern
- JSON: Useful when the data doesn't fit CSV, and the files are mainly for programs
- JSON preserves types better: because strings are written in JSON with quotes, the JSON loader can load numbers as numbers (not strings)

JSON

JSON is very useful, but there's little more to say about it, because it directly relates to programming data types

Remember that the filename extension is just a hint. JSON files aren't always named .json.

SSV

- Occasionally you'll see quasi-CSV files where the data is separated by spaces instead of commas
- Luckily, the csv module has lots of parameters, and can handle this:

```
with open("nino34.ssv") as ifh:
    nino = csv.DictReader(ifh, delimiter=" ")
    for row in nino:
        print(row)
```

SSV

- Occasionally you'll see quasi-CSV files where the data is separated by spaces instead of commas
- Lucki
parameters, and can handle this.

("Delimiter" is a synonym of "separator")

```
with open("nino34.ssv") as ifh:
    nino = csv.DictReader(ifh, delimiter=" ")
    for row in nino:
        print(row)
```

TSV

- Third verse, same as the first: CSV, but tabs instead of commas
  - Tabs are whitespace used to keep indentation consistent. Rather than a tab always being the same width, it always moves text to the same place.
- The escape for tab is `\t`. (So, delimiter=`"\t"`)
- It's unlikely you'll ever encounter a TSV, but they're out there

XML

- Pray you never encounter XML
  - You probably won't, but it's used in, e.g., PowerPoint files
- There's an `xml` module. We shall discuss it no further.

Binary formats

- There *are* file formats that aren't text
- Mostly used when the amount of data is huge
  - A 5MP picture is worth about 2.5M words
- If you find yourself needing to use one, search for the module that handles it; there'll usually be one!

Fun with files

CS114 (M6)

Word distribution CSV

- Let’s do another long demonstration
- We’ll continue our word distribution example from A Tale of Two Cities, but
  - fix our issues with non-word stuff,
  - sort the result by frequency, instead of by word, and
  - store the result in a CSV file.

Step one: find the words

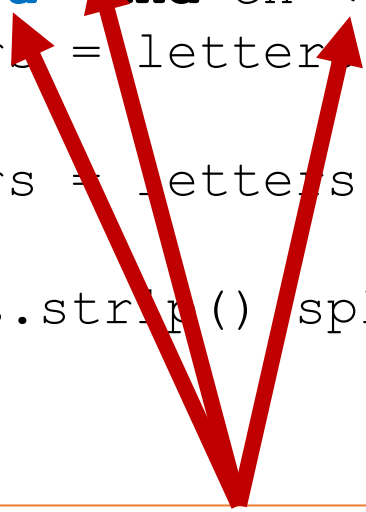
- We want a function that gets only the words from a string, and drops all the non-word symbols
- We actually have everything we need, but it may not be obvious how
- Biggest trick here: string comparison

Step one: find the words

```
def words(line: str) -> list[str]:
    letters = ""
    for ch in line.lower():
        if ch >= "a" and ch <= "z":
            letters = letters + ch
        else:
            letters = letters + " " # So we'll
                                    # still split
    return letters.strip().split()
```

Step one: find the words

```
def words(line: str) -> list[str]:
    letters = ""
    for ch in line.lower():
        if ch >= "a" and ch <= "z":
            letters = letters + ch
        else:
            letters = letters + " " # So we'll
                                    # still split
    return letters.strip().split()
```



Because we lower-cased first, we didn't need to check the capitals case.

Aside: remember `distribution`

```
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
```

Step two: sort by frequency

- We did this in a warmup!

```
def dictValue(key: str) -> int:
    return dist[key]

byFreq = sorted(dist, reverse=True, key=dictValue)
```

Step three: write to CSV

- Let’s put it together, into one function that takes the output (CSV) and input (text) file as arguments

```
def wordDistributionCSV(
    outFilename: str, inFilename: str
) -> None:
    # 1: Words
    inWords = []
    with open(inFilename) as ifh:
        for line in ifh:
            inWords = inWords + words(line)

    # 2: Distribution and sorting
    dist = distribution(inWords)
    def dictValue(key: str) -> int:
        return dist[key]
    byFreq = sorted(dist, reverse=True, key=dictValue)

    # 3: Write CSV
    with open(outFilename, "w") as ofh:
        outCSV = csv.DictWriter(ofh, ["Word", "# of appearances"])
        outCSV.writeheader()
        for word in byFreq:
            outCSV.writerow({
                "Word": word,
                "# of appearances": dist[word]
            })
```

Another CSV demo

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

- We have two CSV files:
  - Employee ID,Name
  - Employee ID,Hourly wage
- We want a single CSV file that has Employee ID,Name,Hourly Wage
- But, some data may be missing!

```
import csv
import typing

employeeInfo: dict[int, dict[str, typing.Any]] = {}

def mkEmployeeInfo(id: int) -> dict[str, typing.Any]:
    if not (id in employeeInfo):
        employeeInfo[id] = {"Employee ID": id}
    return employeeInfo[id]

with open("employee_names.csv") as ifh:
    names = csv.DictReader(ifh)
    for row in names:
        id = int(row["Employee ID"])
        info = mkEmployeeInfo(id)
        info["Name"] = row["Name"]

with open("employee_wages.csv") as ifh:
    wages = csv.DictReader(ifh)
    for row in wages:
        id = int(row["Employee ID"])
        info = mkEmployeeInfo(id)
        info["Hourly wage"] = row["Hourly wage"]
```

```
with open("employee_info.csv", "w") as ofh:
    employeeCSV = csv.DictWriter(ofh, [
        "Employee ID", "Name", "Hourly wage"
    ])
    employeeCSV.writeheader()
    for employeeId in employeeInfo:
        info = employeeInfo[employeeId]
        if not ("Name" in info):
            info["Name"] = "!UNKNOWN!"
        if not ("Hourly wage" in info):
            info["Hourly wage"] = "!UNKNOWN!"
        employeeCSV.writerow(info)
```

This would be a pretty brutal way to add our unknowns if we had more columns than this. Can you think of another way?

Module summary

CS114 (M6)

Module summary

- Reading and writing text files line-by-line
- Reading and writing CSV files using dictionaries or lists
- Reading and writing JSON data
- Moving data back and forth between files and Python values