Module 6: Files

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-06-start.ipynb
```

Discuss the previous module with your neighbour.

- What is the different between `list.sort` and `sorted`? How do we use them?
- What can we do with dictionaries?

Getting Started: Download Data Files

We are going to be working with some sample data files.

Run the first cell of the start code, which contains lines that begin `!wget`, to download these data files.

You should now have files entitled `jabberwocky.txt`, `Lake_Partners3.csv`, `sample.json`, and others.

Working with Data

To do Science, we need to be able to work with data sets.

To get this into our Python code, we need to be able to work with **files**.

In its simplest form, a **file** is just a sequence of numbers, each between 0 and 255, called a **byte**.

The meaning of these bytes depends on the **format** of the file.

These numbers happen to correspond to a particular image:

```
data = [137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73, 72, 68, 82, 0, 0, 0,
        7, 0, 0, 0, 7, 8, 0, 0, 0, 0, 225, 57, 8, 15, 0, 0, 0, 67, 73, 68, 65,
        84, 8, 29, 1, 56, 0, 199, 255, 1, 255, 0, 0, 0, 0, 0, 0, 2, 0, 0, 1,
        0, 1, 0, 0, 1, 255, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 2, 0, 1,
        0, 255, 0, 1, 0, 1, 255, 0, 1, 0, 0, 255, 0, 1, 255, 0, 0, 0, 0, 0, 0,
        147, 151, 6, 11, 111, 201, 52, 232, 0, 0, 0, 0, 73, 69, 78, 68, 174,
        66, 96, 130]
```

(They might also happen to be the result of rolling a lot of dice.)

Text Files

We often want to work with **text files**.

Here, each byte or group of bytes corresponds to a number, letter, punctuation mark, or other special character.

For example, 97 is interpreted as the letter `'a'`, and 65 is interpreted as the letter `'A'`.
32 is interpreted as the space character `' '`.
10 is the "new line" character, which we often write `'\n'`. It starts a new line of text.

We don't need to work with the numbers ourselves because Python knows how to work with text. If we look at a text file the viewer interprets it for us. So `jabberwocky.txt` looks like:

```
'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.

...
```

Reading Text Files

To read a text file, we need to **open** the file, and later `close` it.

While the file is open, it is an iterable that gives us the contents of the file, a line at a time.

To ensure we don't forget to `close` the file, we will always use the following pattern:

1. Write `with open(filename, 'r') as myfile:` to open the file, then
2. in a code block, use the iterable `myfile`.

```
with open('jabberwocky.txt', 'r') as myfile:
    i = 0
    ## Inside this `with`, we can iterate through myfile:
    for line in myfile:
        print("line", i, "is", line)
        i = i + 1
```

! This process automatically closes the file for us, as it reaches the end of the `with`. We will always use `with`, and never directly close our files.

Syntax of `open`

The help on `open` is rather long. All we need is two arguments:

1. The first is a `str` indicating the name of the file to open.
2. The second is a `str`.
   For now, use `'r'` to indicate we want to **read** the file. (Later we'll use `'w'` to write.)

```
with open('jabberwocky.txt', 'r') as myfile:
    i = 0
    ## Inside this `with`, we can iterate through myfile:
    for line in myfile:
        print("line", i, "is", line)
        i = i + 1
```

Exercise

Write a function `count_lines(filename: str) -> int` that takes the name of a text file, and returns an `int` indicating the number of lines in it. For example,

```
check.expect("Jabberwocky", count_lines('jabberwocky.txt'), 34)
count_lines('pg2097.txt') ⇒ ?
count_lines('19033.txt') ⇒ ?
```

More with `str`

Whenever we have a `str`, we can use the `str.split` method to break it into pieces.

```
line = "Twas brillig, and the slithy toves"
line.split() ⇒ ['Twas', 'brillig,', 'and', 'the', 'slithy', 'toves']
```

Since we can now get a `str` from each line of a text file, we can use this to answer many important questions. For example:

Exercise

Write a function `count_words(filename: str) -> int` that takes the name of a file, and returns a count of the words in the file named `filename`.

```
check.expect("Jabberwocky count", count_words('jabberwocky.txt'), 166)
```

Exercise

Write a function `count_word(filename: str, target: str) -> int` that takes the name of a file, and determines how many times the word `target` appears in the file named `filename`.

```
check.expect("vorpal", count_word('jabberwocky.txt', 'vorpal'), 2)
check.expect("snack!", count_word('jabberwocky.txt', 'snicker-snack!'), 1)
```

Writing Text Files

To write to a file we again need to `open` the file.

While the file is open, we can use the `file.write` method, with a `str`. When we run this:

```
with open('some_stuff.txt', 'w') as myfile:
    myfile.write("Hello")
    myfile.write(" World!\n")
    for i in range(10,0,-1):
        myfile.write(str(i))

    myfile.write("That's all folks.\n")
```

Look at file named `'some_stuff.txt'`; it now contains:

```
Hello World!
10987654321That's all folks.
```

To start a new line, we need to write `'\n'`, a special symbol called “new line”.

(The `print` function adds `'\n'` automatically, so we’re not used to adding it ourselves.)

Writing Text Files

Use this code as a model:

```
with open('some_stuff.txt', 'w') as myfile:
    myfile.write("Hello")
    myfile.write(" World!\n")
    for i in range(10,0,-1):
        myfile.write(str(i))

    myfile.write("That's all folks.\n")
```

Some things to remember:

- Use `'\n'` at the end of every line.
- Use `str` to convert anything to a `str`.

Exercise

Write a function `write_count(filename: str, n: int) -> None`. It counts down from `n` to `1`, and writes the output, one number per line, in the file named `filename`.

For example, after `write_count("count3.txt", 3)`, the file named `"count3.txt"` contains:

```
3
2
1
```

Look at the files you create to verify they contain what you expect.

Practice with Text files

Exercise

Modify `histogram` below, so instead of taking a `list[str]`, it takes a `str` containing the name of a text file.

```
def histogram(words:
              list[str]) -> dict[str, int]:
    d = {}
    for w in words:
        if w not in d:
            d[w] = 1
        else:
            d[w] = d[w] + 1

    return d

check.expect("H1",
    histogram(['a', 'man', 'a', 'plan',
               'a', 'canal', 'panama']),
    {'a': 3, 'man': 1, 'plan': 1,
     'canal': 1, 'panama': 1})
```

```
new_histogram('jabberwocky.txt') ⇒
{"'Twas": 2, 'brillig,': 2, 'and': 7,
 'the': 15, 'slithy': 2, 'toves': 2,
 'Did': 2, 'gyre': 2, 'gimble': 2,
 'in': 6, 'wabe;': 2, 'All': 2,
 'mimsy': 2, 'were': 2, 'borogoves,': 2,
 'And': 5, 'mome': 2, 'raths': 2,
 'outgrabe.': 2, '"Beware': 1,
 'Jabberwock,': 2, 'my': 3, 'son': 1,
 'The': 4, 'jaws': 1, 'that': 2, 'bite,': 1,
 'claws': 1, 'catch!': 1, 'Beware': 1,
 'Jubjub': 1, 'bird,': 1, 'shun': 1,
 'frumious': 1, 'Bandersnatch!"': 1,
 'He': 4, 'took': 1, 'his': 2, 'vorpal': 2,
 'sword': 1, 'hand;': 1, 'Long': ...: ... }
```

CSV Files

Sometimes we want to look at text. But for science we more often want to look at **data**, which is often created in a spreadsheet and exported in CSV format.

We start by looking at some real world data: open data measurements of water quality. You should already have this in Jupyter in the file `Lake_Partners3.csv`.

Jupyter can open the file; it shows it as a table with a few columns and over 12 000 rows.

Looking at CSV as text

A CSV file is a special text file, so we **can** look at it directly.

```
with open('Lake_Partners3.csv','r') as myfile:

    for line in myfile:
        print(line)
```

If we look at it as a text file, it's pretty simple:

```
Lake Name,Township,STN,Site ID,Site Description,DMS_1,DMS_2,Date,Calcium (mg/L)
ABERDEEN LAKE (BASS),ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,11-May-08,5.7
ABERDEEN LAKE (BASS),ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,29-Oct-08,4.9
ABERDEEN LAKE (BASS),ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,12-May-09,5.7
ABERDEEN LAKE (BASS),ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,19-May-10,5.8
ABERDEEN LAKE (BASS),ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,25-Jun-10,5.6
```

Looking at CSV as text is fussy

It looks like we could read this pretty easily, split on the commas, and go from there:

```
line = "Lake Name,Township,STN,Site ID,Site Description,DMS_1,DMS_2,Date,Calcium (mg/L)"
line.split(",") ⇒ ['Lake Name', 'Township', 'STN', 'Site ID', 'Site Description',
                   'DMS_1', 'DMS_2', 'Date', 'Calcium (mg/L)']
```

But it's not quite so easy:

```
line = 'ABERDEEN LAKE,ABERDEEN,4,1,"Mid Lake, deep spot",463030,834840,11-May-08,5.7'
line.split(",") ⇒ ['ABERDEEN LAKE', 'ABERDEEN', '4', '1',
                   '"Mid Lake', ' deep spot"', '463030', '834840', '11-May-08', '5.7']
```

Notice how it split the `"Mid Lake, deep spot"` piece on its comma. But in the visualizer, that is one cell.

The quotation marks in the file indicate that the commas inside should be ignored.

We could fix this bug, but then we'll find another, and another; it's fussy. We want to do science, not muck around with files.

Someone has already done the fussy part, and created the `csv` module.

Reading files using the `csv` module

To read CSV files more easily, we will use the `csv` module. As usual, we write: `import csv`

Now there is just one extra line to add. Immediately inside our `with open(...)`, we write:

```
myfile = csv.reader(myfile)
```

`csv.reader` is a function that takes an iterable (like a text file), and returns a new iterable that gives a "transformed" version of the data.

Reading files using the csv module

Let's run same code with this addition:

```
with open('Lake_Partners3.csv','r') as myfile:
    myfile = csv.reader(myfile)

    for line in myfile:
        print(line)
```

```
['Lake Name','Township','STN','Site ID','Site Description','DMS_1','DMS_2','Date','Calcium (mg/L)']
['ABERDEEN LAKE', 'ABERDEEN', '4', '1', 'Mid Lake, deep spot', '463030', '834840', '11-May-08', '5.7']
['ABERDEEN LAKE', 'ABERDEEN', '4', '1', 'Mid Lake, deep spot', '463030', '834840', '29-Oct-08', '4.9']
['ABERDEEN LAKE', 'ABERDEEN', '4', '1', 'Mid Lake, deep spot', '463030', '834840', '12-May-09', '5.7']
['ABERDEEN LAKE', 'ABERDEEN', '4', '1', 'Mid Lake, deep spot', '463030', '834840', '19-May-10', '5.8']
['ABERDEEN LAKE', 'ABERDEEN', '4', '1', 'Mid Lake, deep spot', '463030', '834840', '25-Jun-10', '5.6']
```

Instead of each line just being a `str`, it is a `list[str]`, split **properly**.

Reading files using the `csv` module

Use this code as a model:

```
with open('Lake_Partners3.csv','r') as myfile:
    myfile = csv.reader(myfile)
    for line in myfile:
        print(line)
```

Exercise

Write a function `count_lines_by_lake_name(lake: str) -> int` that returns the number of lines in the file `'Lake_Partners3.csv'` on a lake named `lake`.

```
check.expect("LN1",count_lines_by_lake_name("ABERDEEN LAKE (BASS)"), 24)
check.expect("LN1",count_lines_by_lake_name("BITTERN LAKE"), 2)
check.expect("LN1",count_lines_by_lake_name("JOE LAKE"), 4)
check.expect("LN1",count_lines_by_lake_name("LAKE OF THE WOODS"), 356)
```

Named Columns in `csv`

Using `csv.reader`, we get each line simply converted into a `list[str]`. This is OK, but it's a little untidy; we need to remember that that column 0 is the name of the lake, that column 4 is the site description, and so on.

Make one small change: instead of using `csv.reader`, use `csv.DictReader` (note capitalization!). Then it automatically uses the **first line as the names of the columns**, and turns the rest of the lines into **dictionaries**:

```
with open('Lake_Partners3.csv','r') as myfile:
    myfile = csv.DictReader(myfile)
    for line in myfile:
        print(line)
```

```
{'Lake Name': 'ABERDEEN LAKE (BASS)', 'Township': 'ABERDEEN', 'STN': '4', 'Site ID': '1',
 'Site Description': 'Mid Lake, deep spot', 'DMS_1': '463030', 'DMS_2': '834840',
 'Date': '11-May-08', 'Calcium (mg/L)': '5.7'}
{'Lake Name': 'ABERDEEN LAKE (BASS)', 'Township': 'ABERDEEN', 'STN': '4', 'Site ID': '1',
 'Site Description': 'Mid Lake, deep spot', 'DMS_1': '463030', 'DMS_2': '834840',
 'Date': '29-Oct-08', 'Calcium (mg/L)': '4.9'}
```

Named Columns in `csv`

Use this code as a model:

```
with open('Lake_Partners3.csv','r') as myfile:
    myfile = csv.DictReader(myfile)
    for line in myfile:
        print(line)
```

```
{'Lake Name': 'ABERDEEN LAKE (BASS)', 'Township': 'ABERDEEN', 'STN': '4', 'Site ID': '1',
 'Site Description': 'Mid Lake, deep spot', 'DMS_1': '463030', 'DMS_2': '834840',
 'Date': '11-May-08', 'Calcium (mg/L)': '5.7'}
```

Exercise

Write a function `calcium_by_lake_name(lake: str) -> list[float]` that returns all the calcium concentrations in the file `'Lake_Partners3.csv'` on a lake named `lake`.

```
check.expect("Ca1", calcium_by_lake_name("BITTERN LAKE"), [1.7, 1.7])
check.expect("Ca2", calcium_by_lake_name("JOE LAKE"), [2.0, 2.0, 1.9, 2.1])
```

You'll need to use `line['Lake Name']` and `line['Calcium (mg/L)']`.

Each item from the file is still a `str`, even the ones like `'2.0'`. Convert with `int` or `float`:

`int('463030')` ⇒ `4603030`

`float('2.0')` ⇒ `2.0`

More complex data with JSON

Sometimes we can store a dataset simply using a table. But other times we might have more complex needs, or may want to organize our data better. We might want to store a dictionary in file, for example.

For this we can use the JSON (JavaScript Object Notation) file format.

Use is very simple: write `import json`. Then once we have opened the file, load all the data using `json.load`. Like so:

```
with open("Lake_Partners3.json", 'r') as myfile:
    data = json.load(myfile)
```

...That's it. Now the variable named `data` contains whatever is in the file.

Jupyter knows how to interpret JSON files. Open `Lake_Partners3.json` and look at it.

Storing more complex data with JSON

```
with open("Lake_Partners3.json", 'r') as myfile:
    data = json.load(myfile)
```

We can look in Jupyter and see that `data` should be a dictionary with two keys: `lakes` and `measurements`. We can also verify this:

```
for key in data:
    print(key)
```

```
## We see:
lakes
measurements
```

Looking further into `lakes`, we see that it's a dictionary, where each key is a `str` that looks like an `int`: the STN.

Each value is yet another dictionary with keys `'Lake Name'`, `'Township'`, and `'Sites'`:

```
print(data['lakes']['25'])
{'Lake Name': 'AEROBUS LAKE',
 'Township': 'UNORGANIZED',
 'Sites':{'1': {'Site Description': 'North Basin...', 'DMS_1': 502050, 'DMS_2': 932123},
          '2': {'Site Description': 'South Basin...', 'DMS_1': 501826, 'DMS_2': 932503}}}
```

Storing more complex data with JSON

```
with open("Lake_Partners3.json", 'r') as myfile:
    data = json.load(myfile)
print(data['lakes']['25'])
{'Lake Name': 'AEROBUS LAKE',
 'Township': 'UNORGANIZED',
 'Sites':{'1': {'Site Description': 'North Basin...', 'DMS_1': 502050, 'DMS_2': 932123},
          '2': {'Site Description': 'South Basin...', 'DMS_1': 501826, 'DMS_2': 932503}}}
```

Exercise

Write a function `name_to_stn(name: str)` that loops through the keys in `data['lakes']`, to determine the STN corresponding to the lake named `name`, or the empty string (`""`) if no such lake exists. For example,

```
check.expect("Aerobus: 25", name_to_stn('AEROBUS LAKE'), '25')
check.expect("Long: 1231", name_to_stn('LONG LAKE'), '1231')
check.expect("Evendim: None", name_to_stn('LAKE EVENDIM'), "")
```

Exercise

Write a function `names_by_township(town: str) -> list[str]` that takes the name of a township, and returns a list containing the names of all lakes in that township. E.g.

```
check.expect("N1", names_by_township('SPENCE'),
             ['BELLS LAKE (SILVER)', 'OLD MANS LAKE'])
check.expect("N2", names_by_township('WATT'),
             ['BRANDY LAKE', 'SKELETON LAKE', 'THREE MILE LAKE'])
check.expect("N3", names_by_township('THE SHIRE'),
             [])
```

Hint

A few examples:

```
data['lakes']['322']['Township']   ⇒ 'SPENCE'
data['lakes']['322']['Lake Name']  ⇒ 'BELLS LAKE (SILVER)'
data['lakes']['4138']['Township']  ⇒ 'SPENCE'
data['lakes']['4138']['Lake Name'] ⇒ 'OLD MANS LAKE'
```

Writing JSON files

Writing to a JSON file is as simple as reading. Use `json.dump(obj, fp)` to dump the value `obj` to the already-open file `fp`.

```
sampledata = {42: [2, 4, 6, 0, 1], 17: "foobar!"}

with open("new-file.json", 'w') as myfile:
    json.dump(sampledata, myfile)
```

We can then load the data back to see that it did what we want:

```
with open("new-file.json", 'r') as myfile:
    newdata = json.load(myfile)

sampledata ⇒ {42: [2, 4, 6, 0, 1], 17: "foobar!"}
newdata ⇒ {'42': [2, 4, 6, 0, 1], '17': 'foobar!'}
```

Notice the difference: the ints `42` and `17` were converted to the strs `'42'` and `'17'`.

Although Python dictionaries can have many types as keys (**`str`**, **`int`**, **`float`**, tuples, and more), in JSON files the keys can **only** be **`str`**.

Module summary

- Use `with open(filename, 'r') ...` to read text files, and `with open(filename, 'w') ...` to write to them.
- Use the `csv` module to read csv files created by a spreadsheet.
- Use the `json` module to read and write any kind of structured data.

Before we begin the next module:

- Read and complete the exercises in module 6 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 6 Review Quiz, due on Monday.