Lab 6

In this lab, we'll practice accessing CSVs, sorting, and using sets.

To start, familiarize yourself with the dataset for P6 on GitHub here: wine.csv.
Download the `wine.csv` to a new `lab6` directory, and start a new notebook in that directory for this lab.

CSVs

Chapter 14 of
Automate the Boring Stuff introduces CSV files and provides a code snippet we can reuse. We showed how to turn this code into a function in class. Copy the following into a cell in your notebook and run it:

```
import csv

# copied from https://automatetheboringstuff.com/chapter14/
def process_csv(filename):
    exampleFile = open(filename, encoding="utf-8")
    exampleReader = csv.reader(exampleFile)
    exampleData = list(exampleReader)
    return exampleData

# use process_csv to pull out the header and data rows
csv_rows = process_csv("wine.csv")
csv_header = csv_rows[0]
csv_data = csv_rows[1:]
```

We recommend you also copy the above code into your P6 notebook when you start the project.

Try running the following and thinking about the results (thinking about the results involves locating the values produced in the `wine.csv` dataset):

- `csv_header`
- `len(csv_data)`
- `csv_data[:5]`
- `csv_data[0]`
- `csv_data[0][3]`
- `csv_header.index("variety")`
- `csv_data[0][csv_header.index("variety")]`

Note on last two: you can find the index of a value in a list by using the `.index(val)` method. For example, the following prints 2:

```
letters = ["A", "B", "C", "D"]
print(letters.index("C"))
```

While looking at the wine data, now try to write Python expressions to extract the following:

- `Tinta de Toro`
- `Ponzi`
- `90.0`

Also try to complete the following:

- `csv_data[0][csv_header.index(????)]` to get "US"
- `csv_data[1][csv_header.index(????)]` to get "Bodega Carmen Rodríguez"
- `csv_data[2][csv_header.index(????)]` to get "Sauvignon Blanc"

You'll use the following function as the basis for accessing data in P6, but first you need to fill in some missing pieces using the parameters (ignore the option part for now and ONLY change the ???? parts):

```
def cell(row_idx, col_name):
    col_idx = csv_header.index(????)
    val = csv_data[????][col_idx]
    if val == "":
        return None
    # optional: convert types based on column name?
    return val
```

Is your implementation correct? Test it with the following:

1. `cell(0, "country")` should return "US"
2. `cell(1, "points")` should return "96"
3. `cell(2, "price")` should return "90.0"
4. `cell(3, "variety")` should return "Pinot Noir"

Optional: it will save you time in the long run if `cell(1, "points")` returns an `int` (example 2) and `cell(2, "price")` returns a `float` (example 3) instead of strings. Consider improving the `cell` function so it automatically converts the result to the desired value based on the column name (e.g., the value for any price might be cast to a float).

Important Reminder: while you and your lab partner (if you have one) can collaborate on writing the `cell` function, you may not collaborate with your lab partner on other parts of the project, unless of course the person you're doing the lab with is also your project partner.

Sorting

There are two major ways to sort lists in Python: (1) with the `sorted` function or (2) with the `.sort` method. You should experiment with both and understand their effects. More generally, when encountering a new method, you should learn (a) how it modifies existing structures, and (b) what new values it returns, if any.

Try running the following:

```
letters = ["B", "C", "A"]

result = letters.sort()

print("original list:", letters)
print("returned value:", result)
```

You should record your observations in your notes. What does `.sort` do to existing structures? What does it return?

Now let's try `sorted`:

```
letters = ["B", "C", "A"]

result = sorted(letters)

print("original list:", letters)
print("returned value:", result)
```

What does `sorted` do to existing structures? What does it return?

While `.sort` only works on lists, `sorted` works on other sequences, such as strings. Can you guess why there's not the equivalent of a `.sort` method for strings? Hint: remember strings are immutable.

Let's try `sorted` on a string:

```
s = "BCA"

result = sorted(s)

print("original str:", s)
print("returned value:", result)
```

Note that `sorted` always returns a list sequence, even if the input is a string sequence.

Some methods both change existing structures AND return something. Try `pop`:

```
letters = ["B", "C", "A"]

result = letters.pop(0)

print("original list:", letters)
print("returned value:", result)
```

Sets

In class, we learned about the Python `list`. Another simpler structure you'll sometimes find useful is the `set`. You can create sets the same way as lists, just replacing the square brackets with curly braces. Try it!

```
example_list = ["A", "B", "C"]
print(example_list)
example_set = {"A", "B", "C"}
print(example_set)
```

`in` operator

Some things are similar between lists and sets, like checking if they contain something. Try this:

```
"A" in example_list
```

And this:

```
"A" in example_set
```

Note: if you have a LOT of values, the `in` operator is MUCH faster for Python to execute with a `set` than with a `list`.

Order (or lack thereof)

Sets have no inherent ordering, so they don't support indexing. Try and watch it fail:

```
example_list[0]  # works
example_set[0]   # crashes
```

The lack of order also matters for comparisons. Try evaluating this boolean expression:

```
["A", "B", "C"] == ["C", "B", "A"]
```

And now try this:

```
{"A", "B", "C"} == {"C", "B", "A"}
```

Type Conversions

You can switch back and forth between lists and sets with ease. Let's try it:

```
items = [3,2,1]
items_set = set(items)
print(items_set)
```

Or in the other direction:

```
items = {3,2,1}
items_list = list(items)
print(items_list)
```

Be careful! When going from a set to a list, Python has to choose how to order the previously unordered values. If you run the same code, there's no guarantee Python will always choose the same way to order the set values in the new list.

Sometimes people convert from a more complicated type (like a float) to a less complicated type (like an int) and back. Can you think why they might do this? Run this code and think about it:

```
x = 3.8
y = float(int(x))
print(y)
```

In the same way, sometimes people convert from a list (more complicated type) to a set (simpler type) and back to a list again. Explore the resulting effect:

```
list_1 = ["A", "A", "B", "B", "C", "B", "A"] # try playing with different values here
list_2 = list(set(list_1))
print(list_2)
```

Project

Hopefully this will help you get off to a good start on the project! Here's how the lab relates:

1. you should use the `cell` function from the lab to access data in the project
2. sorting strings will help you detect anagrams
3. converting a list to a set and then back to a list will remove duplicates from your list

Good luck!