University of Sunderland

Technology Management for Organisations
Workshop 3
File IO, CSV, and JSON File Formats

Aims of the workshop

Previously we have looked at functions and classes. We looked at how we might build up reusable code sections using these concepts, as well as breaking down programming problems into smaller, easier to manage sub-problems.

An example of this was the exercise from Workshop 2 where we converted grades into classifications. We built a simple function that accepts one number, and produces one string. (70 -> “First Class”). Using our knowledge of list comprehensions we can apply this function we made to each element of grades.

This workshop is based off of Lectures on File I/O. In those lecture we’ve looked at how Python can interact with any operating system which is running it (Windows/Mac/Linux) and manipulate files with reading and writing (or both). We explored how we might save data to files such that it can be easily understood by other parties, and why data formats are beneficial in this regard.
Notable examples of this were the Comma-Separated Values (CSV) and Javascript Object Notation (JSON) data formats. These are the common ingest data format for many business operations.
Along the way we introduced some new vocabulary and keywords such as context managers (using the **with** keyword), as well as **continue** (used in for loops).

In this workshop, we will also briefly consider Numpy which is a library for numerical computing within Python. These are used to provide some nice functionality for calculation purposes. Most of the numerical computing within python either uses this directly, or builds on-top of it.

Please see ‘Useful Information’ below on how to lookup certain Python functionality. The concept behind this workshop is about discovery, and experimentation surrounding topics covered so far.

Feel free to discuss the work with peers, or with any member of the teaching staff.

Useful Information

Throughout this workshop you may find the following useful.

Python Documentation

https://docs.python.org/3.8/

This allows you to lookup core language features of Python 3.8 as well as tangential information about the Python Language. We mentioned this at the end of the introductory lecture.

Note: Depending on the Python version you have installed, the documentation may not be appropriate. Ensure you're on the correct version. E.g Python 3.8 in the case here. You can change this at the top of the page.

Jupyter Notebook Basics

Jupyter itself offers some basic documentation for people new to the editor. These can be found on https://nbviewer.org/github/ipython/ipython-in-depth/blob/master/examples/Notebook/Notebook%20Basics.ipynb

Jupyter can also use markdown cells for text input to describe things. If you wish to annotate each cell as to which Exercise it belongs to, you may find https://nbviewer.org/github/ipython/ipython-in-depth/blob/master/examples/Notebook/Working%20With%20Markdown%20Cells.ipynb useful.

Reminder

We encourage you to discuss the contents of the workshop with the delivery team, and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

The contents of this workshop are not intended to be 100% complete within the session; as such it's expected that some of this work be completed outside of the session. Exercises herein represent an example of what to do; feel free to expand upon this.

Running A Jupyter/IPython Notebook

A jupyter notebook is a server which runs on the local machine. It will use whichever directory it is invoked within as the root folder. It is then able to see all sub-folders within that. This is currently installed on the lab machines; however, if you are wishing to install this at home, you may need to follow pip instructions for installing jupyter.

Jupyter Notebook can be launched from the command line by invoking:

```
jupyter notebook
```

This will start a server, typically on http://localhost:8888/tree, and should automatically open a new tab in the browser.

Jupyter notebook server uses a token, generated at launch, to authenticate access. By default, the notebook server is accessible to anybody with network access to your machine over the port, with the correct token.

If you are asked for a password or token, you can always invoke:

```
jupyter notebook list
```

This will provide you a list of currently active Notebook servers (you can run multiple, on different ports, for different folders), along with their token, all within a single URL.

PIP and packages

PIP is python's packaging tool, it enables the installation of libraries. If you tried to import a library (e.g numpy) when starting your notebook, you may have noticed an error. This is because, by default, python doesn't include pandas as part of the default libraries. Anything not included by Python needs to be installed before it can be utilised.

Python has a rich community backing, with several libraries available over at PyPI which users can install. Users can even make their own libraries and host them online for others to use.

If you require to install python packages on-campus, the image we have is not an administrator. Therefore installing packages to the system level is not permitted. However, pip provides a --user flag for installing packages to the user-profile instead.

Most packages can be installed by using:

```
pip install packageName
```

and for the user-flag:

```
pip install --user packageName
```

Remember, you can always use `pip help install` for specific help information on installing.

Table of Contents

File IO

For this workshop, as always, create a new notebook. Name this something memorable. E.g “Workshop 3”.

Make sure to fill in your ePortfolio for Teaching Week 3 / Workshop 3 after the workshop.

Reading Files

Exercise 1

Create a text file in the same directory as your python notebook for this workshop session. Fill this text file with content of your choice. These could be your favourite song lyrics. “Dear Theodosia” from Hamilton will be my choice here.
Name this file **ex1_data.txt**.

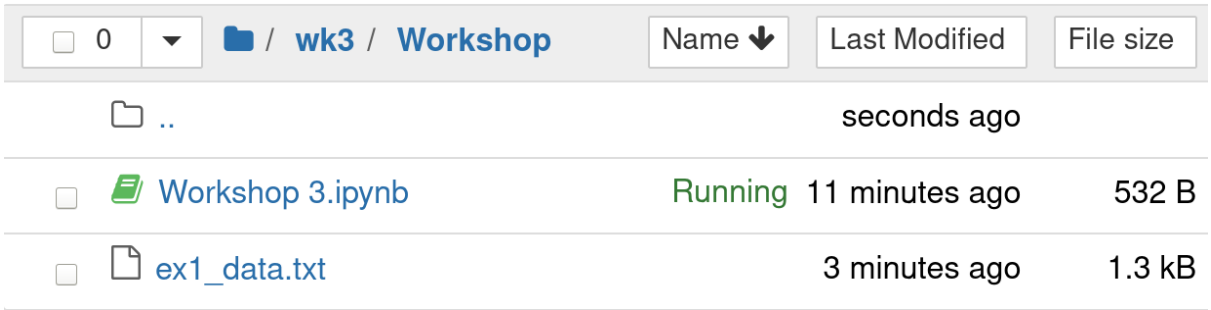
Note: I will upload my ex1_data.txt with my “Dear Theodosia” lyrics to Canvas if you wish to follow along with my examples.

Example:

```
Dear Theodosia, what to say to you?
You have my eyes
You have your mother's name
When you came into the world, you cried and it broke my heart
I'm dedicating every day to you
Domestic life was never quite my style
```

At the moment, we are not interested in the format of what is in the file. Just the raw text within it.

When you are on Jupyter Notebook (the main page which lists all the files, etc), you should see something like the following:



Exercise 2
We can now use our notebook to read this data in via Python.

From the lectures, we know that we can use `open()` to open any file path we provide. Remember, `open()` returns a file object itself. Assign this to a sensibly named variable. For now, we will specify the file path as the local file path, and we will explicitly pass the argument `mode='r'` even though this is the default value.

Once we have this assigned to a variable, we can print the type of the variable (it's an object), to see what it is.

```
my_file = open('ex1_data.txt', mode='r')
print( my_file )
print( type(my_file) )
```

This should look something similar to the following:

```
1 my_file = open('ex1_data.txt', mode='r')
2 print(my_file)
3 print(type(my_file))
```

```
<_io.TextIOWrapper name='ex1_data.txt' mode='r' encoding='UTF-8'>
<class '_io.TextIOWrapper'>
```

Additionally, try replacing the `'ex1_data.txt'` with the full path equivalent instead. For me this might look like `'/home/ashley/Documents/CETM50/Wk3/Workshop/ex1_data.txt'` and yours may look like `'C:/Users/...'` etc. This depends on your Operating System (Windows/Mac/Linux). Re-run the cell and make sure it also works.

Hint: You might want to look up local file-paths vs absolute file-paths.

Exercise 3
Let's read some data out from this file object. We can use a for loop which iterates directly over lines in the file itself.
For each line in the file, print the line.

```
In [2]:
1 for line in my_file:
2     print(line)
```

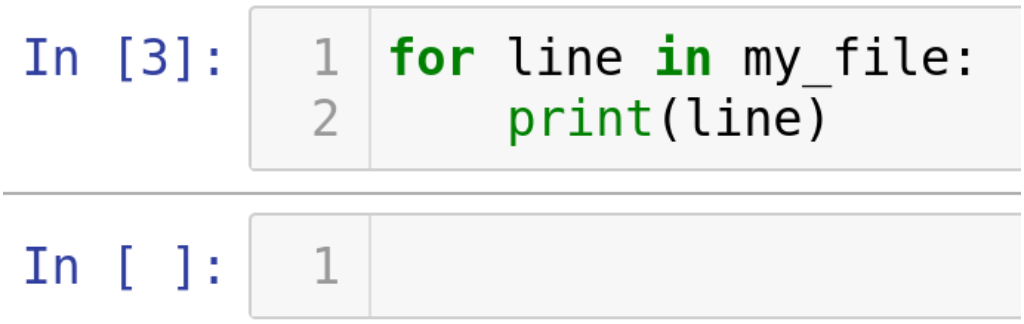
```
Dear Theodosia, what to say to you?

You have my eyes

You have your mother's name

When you came into the world, you cried and it broke my heart
```

Note: If things don't seem to print, read the beginning of Ex 4 and it might make sense.

Exercise 4

Execute the for loop again.

We have a problem with the previous exercise. If we try and execute the for loop again, just printing the line, we won't get anything out!



This is because we've already hit the end of the file. Python is doing some 'tracking' in the background which keeps track of where in the file we are. Once we iterate over the file once, we're at the end of it (in Ex3). When we try to iterate again, no new lines have been added to our file. Thus it will not print anything.

This isn't ideal. So everytime we want to iterate through the file from the beginning, we need to reopen the file!

Combine the Notebook cell which opens the file, with the iterating code from Exercise 3. From our File I/O lectures, we know that it's important to close a file once we've finished with it. Add that to the end of the cell.

To Merge cells: Select a Cell, go to Edit -> Merge Cell (Above/Below).

Once this is done we should have something which looks like the following:

In [3]:
```
1 my_file = open('ex1_data.txt', mode='r')
2 print(my_file)
3 print(type(my_file))
4
5 for line in my_file:
6     print(line)
7
8 my_file.close()
```

This is good, as we know all of that code will execute in-order, at once. If they are in separate notebook cells we can technically execute them in any order! Now this cell will open, read, and close the file.

Exercise 5

Now that we have everything together, and it's not going to behave strangely when printing lines, we can begin to change our simple print expression to something more complex.

We can search for strings within other strings by executing the following boolean expression:

```
<string> in <other string>
```

E.g: In my example, I want to check if the line we're checking contains the word Theodosia. Note: "theodosia" != "Theodosia" (Capital letters are different to lowercase in Python programming!).

```
if "Theodosia" in line:
    print("Found!")
```

If I execute this cell again, it should open the file, iterate through it, and print "Found" for the line in which this comes up.

```
Found!
```

Depending on what is in your text document, search for some string which is present.

Exercise 6

At the moment, Ex 5 isn't very useful for us. I want to know the line number where this lyric comes up. We can use enumerate, covered in previous workshops, to provide line numbers alongside the elements themselves.

1. Modify your file object for loop, make it use enumerate.
2. Replace your print statement, to now print the line numbers which your string appears.

```
Found! At:  0
```

In my example, Theodosia only appears once, on line 0 (the first line).
If I wanted to find a more common word. I'm going to search for "We".

```
Found! At:  9
Found! At:  11
Found! At:  32
Found! At:  34
```

Note: I can call the method lower() on a string to convert it all to lowercase. This might make checking easier. I can check for "We" and "we" with a single check now.

Example:

```
if "we" in line.lower():
      print("Found! At: ", line_no)
```

I've now found some more entries which I missed before! (I was only checking for "We" prior and missed all the cases of 'we').

```
Found! At:  9
Found! At:  10
Found! At:  11
Found! At:  26
Found! At:  32
Found! At:  33
Found! At:  34
```

Exercise 7
Instead of just printing the lines where we've found "we" (lowercase and uppercase variants), let's append these lines to a fresh List. This is similar to what we did with numbers in Workshop 2.

1. Create an empty list, giving it a suitable variable name. Make sure this is done before your for loop! Otherwise you'll constantly make empty ones.
2. If your chosen word is in the line, append the line to this List.
3. After the file close line, print the List (both ways, see below).

We can see that it is indeed a List of strings (Print the List object):

```
Found! At:  33
Found! At:  34
["We'll bleed and fight for you, we'll make it right for you\n", '
If we lay a strong enough foundation\n', "We'll pass it on to you,
we'll give the world to you\n", 'I swear that\n', "We'll bleed and
fight for you, we'll make it right for you\n", 'If we lay a strong
enough foundation\n', "We'll pass it on to you, we'll give the wor
ld to you\n"]
```

Or a nicer way, we can iterate through this List we made, and print each individual item (Iterate over the list and print items):

```
Found! At:  34
We'll bleed and fight for you, we'll make it right for you

If we lay a strong enough foundation

We'll pass it on to you, we'll give the world to you
```

Exercise 7: Notice how we seem to be getting an extra space between prints. This is due to that "\n" which appears at the end - or rather, all of the \n.

We can remove the trailing newlines and spaces by calling rstrip() on our strings. To prove this we can check the following:

```
nightmare = "This is excessive\n\n\n\n\n\n\n\n\n\n          "
print(nightmare)
```

```
1 nightmare = "This is excessive\n\n\n\n\n\n\n\n\n\n          "
2 print(nightmare)
```

```
This is excessive
```

Yes, that is a lot of blank space…
If we print `nightmare.rstrip()` it will return a string which is rid of those special characters and all the spaces! The original variable is **NOT** modified.

```
2 print(nightmare.rstrip())
```

```
This is excessive
```

```
2 print(nightmare.rstrip() + "test")
```

```
This is excessivetest
```

It is immediately obvious that the newlines are gone (left), however, spaces are trickier. How can I check for trailing whitespace characters? We can do some string concatenation if we really wanted to prove the spaces are also gone (right)

Note: The existence of rstrip does indeed suggest the existence of a normal strip function, as well as an lstrip. Feel free to look these up.

Writing Files

Exercise 8

In a new cell, we need to open a file to write. Let's write to the same file we've been working with. This time we will specify `mode='w'`. We know we need to close the file, so let's write that in at the bottom before we forget.

```
ex8_file = open('ex1_data_copy.txt', mode='w')
print( ex8_file )
print( type(ex8_file) )

# Do stuff

ex8_file.close() # Might as well write this in before we forget.
```

As before, we should see it's a valid file, and the type is correct.

Even though we've not told Python to write anything, you may notice that a file has been created! (Check your file explorer) It's just empty.

Exercise 9

We can call `ex8_file.write()` passing it a string to write to the file. The behaviour of this depends on the file mode. In our case, we're on write, which will **erase** any existing content with what we put, if the file doesn't exist it will create it.

Write some content to this file by modifying the `#Do Stuff` section of our code.

After this open up the file and check what you intended to write. (Open in Notepad).

```
We're taking the hobbits to Isengard!
```

Exercise 10

Some time later we realised that we missed off some information. Let's append that to our file now.

1. Open the file in **append** mode
2. Create a List, and put some strings you want to write in it. (Note: These can also be numbers converted to strings - If we try numbers, we'll get an error).

```
some_jibberish = ["Doe, a deer", "a female deer", "far", "a
long long way to run!" ]
```

3. Iterate through this list, and for each element write this to the file.
4. Don't forget to close the file!

Open your file once finished, and check if the original content is present, with your additions added to the bottom in sequence order.

```
We're taking the hobbits to Isengard!Doe, a deera female deerfara long long way to run!Doe,
a deera female deerfara long long way to run!
```

5. **Whoops!** They're all on the same line in the file! Remember to add that special \n to each string you want to add! We can either change each string element literal. Or do some string concatenation in the write line itself!
    a. E.g Instead of `.write(s)` we can do `.write( s + "\n" )`. Saves us a lot of typing!

```
We're taking the hobbits to Isengard!Doe, a deer
a female deer
far
a long long way to run!
```

6. Ahh! Almost. We need to make sure we write a newline at the end of the Exercise 9 bit, or before the first line written for this exercise. Try fixing this so you get the following output (swings and roundabouts on this one):

```
We're taking the hobbits to Isengard!
Doe, a deer
a female deer
far
a long long way to run!
```

Exercise 11

In the lectures we introduced Context Managers. Let's use those now.
Recall:

```
f = open(..)
# Do stuff
f.close()
```

Becomes:

```
with open(..) as f:
```

```
        # Do Stuff.
```

Duplicate your solutions to the previous answers involving file opening, replacing the clunky open, and close mechanisms with some Context Managers.

Exercise 12

Try the following in a new cell.

```
with open("./no_exist.txt") as f:
      pass
```

You should get the following output:

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-53-faeeba27d39a> in <module>
----> 1 with open("./no_exist.txt") as f:
      2     pass

FileNotFoundError: [Errno 2] No such file or directory: './no_exist.txt'
```

This week we also introduced `try` and `except` for helping with Error Handling which follows the following format:

```
try:
      # Do stuff here.
except:
      # Any Issues, do this block.
```

1. Put the context manager, in its entirety, within the `try` block.
2. Add a nice print message into the `except` block which alerts you that something happened.

```
1 try:
2     with open("./no_exist.txt") as f:
3         pass
4 except:
5     print("Uh oh, we're in trouble.")
```

```
Uh oh, we're in trouble.
```

Exercise 13

We can improve this, currently we have no idea what the error is. We only know that the print we put in the except block executed - so there is an error.

Modify the `except` line to catch the general exception `Exception`. Using the `as` keyword, give it a useful name. Then use this in your print statement.

```
Uh oh, we're in trouble:  [Errno 2] No such file or directory: './no_exist.txt'
```

Exercise 14

As we have already encountered this error, we can add a more specific `except` clause above the general one. Make this catch the `FileNotFoundError` exception, and get it to print something different.

```
except FileNotFoundError as not_found:
    print( "Didn't find the file, see message below" )
    print( not_found )
```

```
Didn't find the file, see message below
[Errno 2] No such file or directory: './no_exist.txt'
```

This try, except structure will prevent the whole of python from erroring out. Any code after this structure (try.... except...) will continue to be executed as normal. If this code is unrelated then it will execute just fine. Be cautious of putting code outside this try structure, if it relates to what's inside.

For example, if we open our file and it errors, but we try/except it and carry on afterwards to then attempt writing to the file... we're going to get another error as our file object will likely not be correct.

The behaviour of errors being swallowed by these `except` clauses is known as Error Hiding

CSV

Exercise 15

Alongside this workshop you will find a wind_data.csv file. Download this to the same directory as your python notebook. Just like the text files from earlier.

Exercise 16

Import csv, and read the wind_data.csv file using a context manager. The file will be in 'read' mode.

Create a csv_reader, passing it the file we've just opened.

Print each line of this reader to see what data we're dealing with.

```
['\ufeffDate/Time', 'LV ActivePower (kW)', 'Wind Speed (m/s)', 'Theoretical_Power_Curve (KW
h)', 'Wind Direction (°)']
['01 01 2018 00:00', '380.047790527343', '5.31133604049682', '416.328907824861', '259.99490
3564453']
['01 01 2018 00:10', '453.76919555664', '5.67216682434082', '519.917511061494', '268.641113
28125']
['01 01 2018 00:20', '306.37658691406...', '5.21603679656982', '390.900015810951', '272.56478
```

Exercise 17

Extract the first line (as this contains the headers) and store this as a variable. Take note of the type of this line, compared to our lines from the raw file reading earlier in the workshop. What do you notice?

Exercise 18

1. Which index would we need to take to extract the "Wind Speed" attribute?
2. For each line, obtain the Wind Speed (index the correct index).
3. This will be a string! Convert/cast it to a number (which type of number would be appropriate? An int? or a float? Decimal precision?)
4. Append it to a list of wind speeds.

Note: Be careful not to include the headers in this! Just data values we want. Numbers to do analyses on.

```
12 print( speed )
```

```
[5.31133604049682, 5.67216682434082, 5.21603679656982, 5.65967416763305, 5.57794094085693,
5.60405206680297, 5.79300785064697, 5.30604982376098, 5.58462905883789, 5.52322816848754,
5.72411584854125, 5.93419885635375, 6.54741382598876, 6.19974613189697, 6.50538301467895,
6.63411617279052, 6.37891292572021, 6.4466528892517, 6.41508293151855, 6.43753099441528, 6.
22002410888671, 6.89802598953247, 7.60971117019653, 7.28835582733154, 7.94310188293457, 8.3
7616157531738, 8.23695755004882, 7.87959098815917, 7.10137605667114, 6.95530700683593, 7.09
829807281494, 6.95363092422485, 7.24957799911499, 7.29469108581542, 7.37636995315551, 7.448
55403900146, 7.2392520904541, 7.32921123504638, 7.13970518112182, 7.47422885894775, 7.03317
403793334, 6.88645505905151, 6.88782119750976, 7.21643209457397, 7.0685977935791, 6.9382958
4121704, 6.53668785095214, 6.18062496185302, 5.81682586669921, 5.45015096664428, 5.81814908
```

Exercise 19

1. How many wind speed records/entries do we have? How might we find this out from the List we've just created.
2. What is the average Wind Speed recorded in these data? Hint: You may find the `sum` function useful here.
3. What is the minimum Wind Speed? (Hint: max(some_list))
4. What is the maximum Wind Speed? (Hint: min(some_list))

Exercise 20

Using a new Context Manager, this time set for **write** mode. Create a CSV Writer, and write out your Wind Speed List you created in Ex18.

View this file in a notepad to verify.

Small Introduction to Numpy

Exercise 21

Numpy is used for numerical computing, and allows us to convert from Lists to Numpy Arrays. These types have some useful functionality defined on them!
Have a look at the following code. Numpy arrays behave very similarly to Python standard Lists; however, they have more defined functions available to them for computational purposes. Previously you had to create an expression to get the summation and then the length and calculate the mean. Numpy has a function to do this. Remember: Functions are useful for utilities and routines we may often require. Therefore, numpy bundled a lot of these as behaviours on their own Data Type numpy ndarray.

For scientific computing, Numpy is written with several optimisations in mind. This makes it significantly faster for calculating on large sets of data. Ideal for crunching large amounts of data.

```
import numpy as np # Just an alias, I type np instead of numpy
x = np.array( speed )
print( type(speed) )
print( type(x) )

print(x[0] == speed[0]) # Standard Indexing. So far equivalent in
how we use them!
print(x[-1] == speed[-1]) # Reverse Indexing.

# useful statistics
print("Mean:", x.mean() )
print("Max:", x.max() )
print("Min:", x.min() )

# More complex statistical measures of spread/dispersion
print(x.std()) # Standard Deviation
print(x.var()) # Variance = std ** 2
try:
      print(mean(speed)) # Try either of these.
      #print(speed.mean())
except Exception as e:
      print(e)
```

JSON Read/Write

Exercise 22

Recall Davey McDuck from this week's lectures. I have decided to add a new data attribute for our ducks, the number of people they follow on twitter. I have chosen to call this: 'following'.

```
duck_1 = {
      "first_name": "Davey",
      "last_name": "McDuck",
      "location": "Undercover",
      "insane": True,
      "followers": 12865,
      "following": 120,
      "weapons": ["wit", "steely stare", "devilish good looks"],
      "remorse": None
}
```

Let's build up our duck collection. We can represent this as a List of Dictionaries. Where each Dictionary follows the same pattern for outlining a Duck. First let's define some ducks. Feel free to add your own!

```
duck_2 = {
      "first_name": "Jim",
      "last_name": "Bob",
      "location": "Out at sea",
      "insane": False,
      "followers": 123,
      "following": 5000,
      "weapons": ["squeak"],
      "remorse": None
}

duck_3 = {
      "first_name": "Celest",
      "last_name": "",
      "location": "Throne Room",
      "insane" : True,
```

```
        "followers": 40189,
        "following": 1, # Her other account
        "weapons": ["politics", "dance moves", "chess grandmaster",
"immortality"]
}
```

We shall put these in a List called `duck_collection`

```
duck_collection = [ duck_1, duck_2, duck_3 ]
```

Go through the duck_collection, and make sure each element is in-place and all your ducks are accounted for.

Exercise 23

1. `import json`
2. Using a context manager, open a json file for writing
3. Using `json.dump`, write your `duck_collection` to the file you've opened. Remember that the arguments are almost backwards from what we're used to!

Open the file and we can copy its contents, which are rather unreadable at the moment, and use a pretty printing website to make it a bit easier on the eyes.

Paste your file contents into https://jsonformatter.org/json-pretty-print and see the output. Are all your ducks there? Do they all have their attributes?

Exercise 24
In a new cell. Using json.load, load your saved json file back in, assigning the output to a new list variable (other than duck_collection). We hope that the ducks loaded in are the same ducks we saved earlier. Therefore, they should be identical to your duck_collection. We can check for this by checking the equivalence between both lists.

Exercise 25
Write some code which, for each duck, will calculate the difference between the number of people following them and the number of followers of their twitter account. (Positive if more people follow them, than they follow).

1. Print these
2. Append them to an empty list called `trendy_ducks`

E.g for Davey, this would be his "followers" minus his "following" value. In this case 12865 - 120 = 12745. Note: This should be done programmatically! I might give you many more ducks. I want these resultant numbers for ALL ducks.

Hint: How do we index dictionaries? `some_dict["key"]` should give us the value. Our List we just got from json.load has a dictionary at every index of our List (Nested structures here).

```
12745
-4877
40188
Trendy Ducks:  [12745, -4877, 40188]
```

Exercise 26

Numpy has some useful functionality. If I wanted to find the trendiest duck (the one with the most 'net' followers (followers - following), I could use max or min. But this gives me the value back out, not necessarily which duck this relates to! I wanted an index so I could track the duck down.

Convert the trendy_ducks list to a numpy array

```
arr_trendy_ducks = np.array(trendy_ducks)
```

We can now call argmax() on this numpy ndarray object (or argmin() too) to show which duck has the most (or least).

```
print( arr_trendy_ducks.argmax() )
```

If I assign a variable to that function call, I now have the index of the trendiest duck. I can use this to go back to my original duck_collection List, which houses each duck dictionary, and pull things like their name.

Print the first name of the trendiest duck, programmatically, and print out their 'net' following count (the thing you calculated!).

No manual entry here. This code should work directly off of the List collection, so that I can add more ducks and your code would work exactly the same, and maybe a new Duck is crowned champion.

Hint: duck_collection[0] would get our first duck. We can replace 0 with any variable, so long as it returns an integer. `duck_collection[0]` would return a whole dictionary related to that duck. We can then reference the keys within that dictionary! `duck_collection[0]["some_key"]`

Hint 2: You may need to cast the net followers! Numpy likes to use it's own primitive data types for numbers. You will see :)

```
Congratulations Celest. You are the trendiest duck!
You have a net following of 40188 followers!
```

Exercise 27

Your boss has given you the task of creating a separate JSON file. In this file, he only wants ducks who have a net follower count > 0.
You must filter out the ducks who follow more accounts than who follow them, and save these ducks back out to a JSON file just like your input was.

- Reusing your net follower calculations find indices of all the ducks you need.
- Store them in a separate data structure (create this, and add the correct ducks using the indices found)
- Open a new file in write mode to put this JSON data into. (Use a context manager).
- Use json.dump to convert your separate data structure into JSON and store it in the file.

This exercise will involve a lot of juggling of variables, and data structures, and logic. Values and Indices will need to be managed appropriately. Try doing this before moving onto Ex 28, where we'll look at a nice feature of Numpy which might be more helpful.

Exercise 28

Numpy provides some functionality outside of just its objects.
(https://numpy.org/doc/stable/reference/generated/numpy.where.html)

```
np.where( expression )
```

This returns a ndarray of indices where the expression holds True. For example, we can provide a whole array of values, and an expression which can be used to filter it. Numpy will then not only find where the condition is True or False, but then convert those into indices and provide them.

E.g
We can find all the indices of ducks, where their "net follower" count is even.

arr_trendy_ducks % 2 == 0

Normally we would pass an integer/number to this expression. However, with numpy arrays it can be applied to every value. If we print the output of this expression, we should get a List of True/False, where the condition holds!

We can pass this True/False List into `np.where()` and it will convert those into the indices for us.

Putting it together:

```
print(np.where(arr_trendy_ducks % 2 == 0))
```

```
#Ex 28
print(arr_trendy_ducks % 2 == 0)

print(np.where(arr_trendy_ducks % 2 == 0))
```

```
[False False  True]
(array([2]),)
```

This has returned a tuple. Remember, a tuple is defined as `(item, item, item)`. The trick here is that it's a single element, you can see by the trailing comma. To get the actual array/List which contains the indices you'll need to call [0] on the np.where result.
Note: If we wanted to convert a numpy array back into a List, we can cast it just like we did with numbers weeks ago.

```
actual_indices = np.where(arr_trendy_ducks % 2 == 0)[0]
print(actual_indices)
print(type(actual_indices))
print(type(list(actual_indices)))
```

```
[2]
<class 'numpy.ndarray'>
<class 'list'>
```

Technical Explanation: In our toy example here, we're using a single dimension (List of numbers). But np.where is very powerful and can actually be run over N-dimensional data. Therefore each entry in the tuple is a dimension. As we only have 1, we get a single one back.

As we can see, only one of our ducks has an even count. This is the duck at index 2. Surprise, surprise, this is Celest again.

Exercise 29

Copy your answer to Ex 27, and now use the `np.where` from Ex 28 to make your solution tidier, cleaner, more readable. In my example I did a boolean expression for even numbers, you need to find those > 0.

Hint: The array you get back, contains elements which are the index to look-up. A list comprehension is a perfect choice here. It sounds complicated, but it's just a List of indices you want to go through, and use them to reference the right ducks. Making a new list out of them.

```
[{'first_name': 'Celest',
  'last_name': '',
  'location': 'Throne Room',
  'insane': True,
  'followers': 40189,
  'following': 1,
  'weapons': ['politics', 'dance moves', 'chess grandmaster', 'immortality']}]
```

The Extended Exercises are optional, and are offered as an advanced supplement for those who have completed the existing work and wish to expand on their knowledge and challenge themselves further.

There is no extended exercise this week

This workshop is considered lengthy, and conceptually difficult enough as-is. If you do wish to learn more, after completing EVERYTHING, feel free to dive deeper into Numpy.

Create more ducks, make a duck army.
Use random number generators to generate this duck army. Numbers will be easy, but what about weapons? Maybe you want to sample the duck's weapons from a predefined list of weapons (an armory of sorts).

For more advanced data mocking, we have a library called **faker** which can do more than simple random ranges, but distributions, names, places, etc.

There is nothing stopping you now, you've learned Python syntax, as we continue you're just adding more to the knowledge you have. Play around with things, go crazy.