File I/O and a few more notes about string formatting, along with some NumPy for binary files and the os module for dir info

- How to read/write different file types - .txt, binary, json

useful link to Python Software Foundation doc

link to colab specific file I/O info

Quick notes on string formatting because we'll use this to generate some data files to read/write

- With the str.format(), you pass in the index of the argument that we want to insert into each { }
- The second number determines how many places we want to set aside for spacing the output.
- Using this convention makes it easy to make neatly organized output using print or writing to a file

In [ ]:
```
for x in range(0, 11):
  print('{0:2d} {1:4d} {2:5d}'.format(x, 2**x, 3**x))
```

Quick note about file I/O on Google Colab

- Usually when we write a new file on a local install of Python (running e.g. Jupyter), the file will write directly to the current working directory
- However, on Google Colab it will write to your 'content' folder, which can only be mounted in the virtual machine via a special authentication process
- You can do this if you'd like, but I'm not sure I'd suggest it just for the purpose of this class because it will open all your drive files to the google file stream service - its probably fine, but its a decision you might want to investigate a bit more on your own.
- If you ever do want to do this, use the code in the cell below - it will generate an authentication link and then you'll be able to mount the content drive and you can access google drive just like a local hard drive.

In [ ]:
```
# read the text cell above FIRST before running this code
from google.colab import drive
drive.mount('/content/drive')

# %ls #line magic to list out the contents of the current working directory.
```

An alternate to mounting the drive locally...this will open a download dialog box that you can use to download any text files that you create if you want to view them

In [ ]:
```
from google.colab import files
# usage: see below, but here is the basic syntax
files.download('test.txt')
```

Open a file

- open() creates a file object, and usually takes two arguments - a filename and the read mode
- The first argument is the filename. The second argument describes how the file will be used - read mode ('r'), write mode('w'), read/write mode ('r+') or append mode ('a').
  - read mode 'r' will be assumed if the second arg is omitted
- By default, files are opened in text mode, so you're reading and writing strings to the file.
- Binary mode is enabled by appending 'b' to the read/write/append arg (e.g. 'rb' is read binary).
- In binary mode, you're reading/writing in units of bytes - this will often be the case for non txt files like image files and so forth

In [ ]:
```
# for writing to a txt file
# 'w' will overwrite the file with that name!
f = open('test.txt', 'w')
f.close()
%ls
```

In [ ]:
```
# for reading
f = open('test.txt', 'r')
f.close()
```

In [ ]:
```
# for appending
f = open('test.txt', 'a')
f.close()
```

In [ ]:
```
# for reading or writing
f = open('test.txt', 'r+')
f.close()
```

In [ ]:
```
# for writing binary file
f = open('test.bin', 'wb')
f.close()
```

Now lets try it out by actually writing something to the text files.

- Note the use of the newline character in format statement!

In [ ]:
```
f = open('test.txt', 'w')
for x in range(0, 11):
  # include the \n newline character - the text file will need that specified
  # to properly know what line to put the text on
  f.write('{0:2d} {1:4d} {2:5d}\n'.format(x, 2**x, 3**x))

f.close()

# download and take a look!
# NOTE - if you're running windows then use WordPad instead of Notepad - Notepad
# ignores newline chars
files.download('test.txt')
```

QUESTION: what happens if you don't close it?

- Leaving too many things open is bad...lots of overhead
- You're letting python clean up your mess...not a good idea.
- Changes to the file may not into effect until you close it. This is big problem if you write to a file, then try to read from it...you may not see the stuff that you wrote.

In [ ]:
```
# a better way...this will ensure that the file is properly closed when you're
# done dealing with it (as many errors are caused by failing to close a file after op
with open('test.txt', 'w') as f:
  for x in range(0, 11):
    # include the \n newline character
    f.write('{0:2d} {1:4d} {2:5d}\n'.format(x, 2**x, 3**x))

#check to see if its closed - should be beacuse we're outside the 'with' block
print(f.closed)

# download the file
files.download('test.txt')
```

The 'read' method - f.read(size)

- Will read in **size** data from the file, where size is in terms of text or in terms of bytes (for binary read, more on that later)
- If you leave this blank, then it will read the entire file. That can be very problematic if the file is REALLY big and explodes your computer.

In [ ]:
```
# open our file for reading...
with open('test.txt', 'r') as f:
  # go ahead and read the entire file...
  out = f.read()

# print it out
print(out)
```

In [ ]:
```
# open our file for reading...just grab 13 (one line in this case) elements and print
with open('test.txt', 'r') as f:
  out = f.read(13)

# print it out
print(out)
```

A better way to read a line of text...

- Importance of newline character!

In [ ]:
```
with open('test.txt', 'r') as f:
  # read a line
  out = f.readline()

# print it out
print(out)
```

Loop line by line and print out...

- "end" keyword defines what ends each line in the print statement - default is the newline char, and we already have that so we don't want to add it in again

In [ ]:
```
with open('test.txt', 'r') as f:
  # loop over all lines
  for line in f:
    print(line, end='')
```

Append mode

In [ ]:
```
# open our test.txt file and append to it - will just pick up where you left off!
with open('test.txt', 'a') as f:
  for x in range(0, 11):
    # include the \n newline character
    f.write('{0:2d} {1:4d} {2:5d}\n'.format(x, 2*x, 3*x))

#confirm that its closed
print(f.closed)
files.download('test.txt')
```

What happens when you try to write an integer to a text file?

In [ ]:
```
# open a file for writing
with open('test.txt', 'w') as f:
  for x in range(0, 11):
    f.write(x)
```

Binary file I/O

- So far we've just been dealing with text files where everything is a string (of characters)
- Binary files are written in "machine language" that is denser and easier to interpret (for the machine, not for you!)
- Can use bytearray to convert numbers over the range 0-255 to binary format

In [ ]:
```
# open a file for writing binary
with open('test.bin', 'wb') as f:
  # generate a list of numbers, use bytearray to convert
  # numbers over the range 0-255 to binary format
  bytes_to_write = bytearray([0,1,2,3,4,5])

  # write to file!
  f.write(bytes_to_write)

# have a look!
files.download('test.bin')
!hexdump test.bin
```

In [ ]:
```
# now read it back in
with open('test.bin', 'rb') as f:
  # remember that f.read() reads in the entire file...
  bytes_read = f.read()

# notice that f.read() returns the byte array as a string
print(bytes_read)
```

using the numpy "fromfile" method for easier binary read operations

In [ ]:
```
import numpy as np
```

In [ ]:
```
with open('test.bin', 'rb') as f:
  bytes_read = np.fromfile(f, dtype=np.int8)
  #bytes_read = np.fromfile(f, dtype=np.int16)

print(bytes_read)
```

JSON (JavaScript Object Notation) format

- straightforward and standardized way of storing and exchanging data files
- kind of like a csv or a txt file in nature, but more sophisticated
- developed as a way of tranferring JavaScript objects between browsers and servers, but now frequently used for all types of data and languages
- takes one of several data formats:
  - objects (like dictionaries)
  - arrays (like lists)
  - values (string in double quotes or a number)
  - strings (sequence of characters)

link to main page

In [ ]:
```
# import json module
import json
```

In [ ]:
```
# build a dictionary with a bunch of different data types, including a sub list
# of dictionaries
user_profile = {
  "name": "John",
  "age": 30,
  "bicycle": "Giant",
}
```

Now write a .json file to disk - very similar to file creating/writing that we did above

In [ ]:
```
with open('test.json', 'w') as f:
  json.dump(user_profile, f)

files.download('test.json')
```

Now load the json file back in!

In [ ]:
```
with open('test.json', 'r') as f:
  x = json.load(f)

# and you get back a dictionary
x['bicycle']
```