Working with Files in Python

So far we just have explored the interpreter and some native data structures...

Now we can start to actually work with data!

Here we will talk about IO operations, let's starts with the *I* in *IO*, the input.

Let's start first by creating a mock file with 4 lines: a header and 3 lines of "data". We will do this in the bash terminal (outside Python)

In [ ]:
```
!echo "Column1,Column2,Column3\n1,2,3\none,two,three\nuno,dos,tres" > afile.txt
!cat afile.txt
```

Note that we have used the character `\n` to represent an new line (frequently called line ending, end of line (EOL), line feed, or line break), as is for Linux systems. Other operating systems might differ, and having either extra characters or different characters.

Now let's get into python and try to read this file. Just like a book, you need to first open the file before reading it:

In [ ]:
```
handle = open('afile.txt')
print(handle)
```

Now to actually access the contents, we need to tell the handle to read them:

In [ ]:
```
text = handle.read()
print(text)
```

In [ ]:
```
...
handle.close()
```

As you could see, every line read from the file, is represented as a string type variable. We also explored the handle and saw that it is not text, it is a pointer to the opened file. As we read information from the handle, it exhausts it. Once empty, you would need to reopen and redefine the handle before use. Finally, always remember to close the file. Remaining open files can interfere with the normal execution of a program.

Context Manager

Since you always have to remember (or should!!) to close files, let the context manager do it for you:

In [ ]:
```
with open('afile.txt') as handle:
    text = handle.read()
print(text)
```

In [ ]:
```
...
```

Once the handle is closed, you can no longer access its contents. The context manager allows you to guarantee that the handle is closed once you have performed tasks on its contents.

What does printing a file handle do?

- Errors (yes)
- Prints the memory address of the object (no)
- Prints the contents of the file (go slower)
- Prints the first line of the file (go faster)

Reading by Line

If your file is too big to fully buffer it on memory or if you only need a few lines, the `read` function might be an overkill.

Let's imagine that we only want the headers of the file (or the first line):

In [ ]:
```
with open('afile.txt') as handle:
    header = handle.readline()
print(header)
...
```

As we have mentioned before, the handle is exhausted, so with every call of the `readline` method, the next line will be printed instead.

What about reading the full file one line at a time?

In [ ]:
```
with open('afile.txt') as handle:
    for line_num, line in enumerate(handle):
        print("Line {} is {}".format(line_num, line))
```

In [ ]:
```
...
```

To recap, there are multiple ways to read a file in python:

1. read: Read full content into a string variable.
2. readlines: Read full content into a list, one line per entry
3. readline: reads the first line in the handle

What happens if you use a file handler outside of the context block?

- yes: Nothing - Python is nice
- no: Error!

Writing to files

Before we go to processing, lets see how can we write to a file. So let's code a way make a copy of `afile.txt` :

In [ ]:
```
with open('afile.txt', 'r') as rh, open('afile2.txt', 'w') as wh:
    for line in rh:
        wh.write(line)
```

As when reading, we first need to open the target file. Using the writing mode, Python automatically creates the new file (or re-write an existing file with the same name) and will write in it.

In [ ]:
```
!echo afile.txt
!cat afile.txt
!echo afile2.txt
!cat afile2.txt
```

Modes of opening file

As you saw, the function `open` can be:

1. Read mode (`r`): This is the default to read from a file
2. Write mode (`w`): This will create **(or recreate)** a file to write to
3. Append mode (`a`): This will open a file to write, but will append to the end of it instead of recreating

These are the basic modes.

Additionaly Python has the modifiers `b` to read/write in binary format and `+` to open the file for updating (reading and writing).

The last two modes are a bit more advanced, and we will not cover them.

What does append will look like and why would you like to use it? Say you have the `afile.txt` that you have created either upstream or by a completely different program, and you want to add a line:

In [ ]:
```
new_line = "ichi,ni,san"
with open('afile2.txt', 'w') as wh, open('afile.txt', 'a') as ah:
    wh.write(new_line)
    ah.write(new_line)
...
```

Differing from the writing mode, the append mode will not erase our previous file if it already exists, but will still create a new file if it does not exist.

Processing Data From Files

We now know how to read and write to files in Python.

This is the main way to get data into our programs, and output meaningful information.

But to really make use of it, we need to be able to manipulate and interact with the data.

Say you want to write a program that reads a comma-delimited file with the numbers 1-3 in English, Spanish or Japanese. And let's say we want to output a different file with all in numerical versions in a **TAB**-delimited file:

In [ ]:
```
translate = dict(one=1, two=2, three=3,
                 uno=1, dos=2, tres=3,
                 ichi=1, ni=2, san=3)
with open('afile.txt', 'r') as infile, open('output.txt', 'w') as outfile:
    for line_num, line in enumerate(infile):
        ...
```

As we seen before, you can actually open two file in different modes within a context manager. Also, we can interact with our own variables to create the desired output.

Now, let's say that instead of the tab-delimited numerical output, we just want to have the original file into a list of lists:

In [ ]:
```
with open('afile.txt', 'r') as infile:
...
```

We could solve the problem with a simple loop, but here is a more efficient solution, list comprehension. List comprehensions provide a concise way to create lists, and avoids using the method append in a list or defining the empty container.

What does the following list comprehension produce?

```
In [ ]: [x for x in range(2, 200) if all(x % y != 0 for y in range(2, x))]
```

- yes — All multiples of 2 from 2 to 200
- no — All prime numbers from 2 200
- go slower — Some sort of error

Pre-existing Data Structures

CSV files are extremely common, and there are packages that will allow you to read and write easier. You need to import them though...

```
In [ ]: import csv
        with open('afile.txt') as csvfile, open('afile3.txt', 'w') as outcsv:
            reader = csv.reader(csvfile, delimiter=',', quotechar='"')
            writer = csv.writer(outcsv, delimiter=',')
            for row in reader:
                print(row)
                writer.writerow([row])
```

In [ ]:
```
import pandas as pd
fi = pd.read_csv('afile.txt', sep=',', header=None)
print(fi)
fo = fi.write('afile4.txt', sep=',', header=False, index=False)
```

Installing Packages

The Python Package Index or Pypi by:

```
pip install <package>
```

In [ ]:
```
!python3 -m pip install pandas
```

Packages of Note

1. Pandas: Python Data Analysis Library
2. Numpy: the fundamental package for scientific computing with Python
3. Scipy: Scientific Library for Python
4. Scikit-learn: Python's main machine learning library
5. argsparse: Python's command line argument parser

...

Interacting with the System

Often times you want to be able to query the system, or get input from the command line. In Python, this is done through the packages `os` and `sys` of the standard library (no need to install them). For example, let's say that you would like to ask the system if a file is present in the path, then:

In [ ]:
```
import os
print(os.path.isfile('afile.txt'))
...
```

Imports in python can be done relative to the module name. For example, the `os` module has a sub-module called `path` (as in the example). we can invoke its functions as before, or import functions from the path submodule directly:

```
from os.path import isfile
print(isfile('afile.txt'))
```

or

```
from os import path
print(path.isfile('afile.txt'))
```

Likewise we can get inputs from the terminal using the python module `sys`, and its attribute `argv`. `argv` is a list of command line input, and will gather everything when you invoked a script:

1. `sys.argv[0]` : name of the script
2. `sys.argv[1]` : first argument etc.

To better understand this, let's create a simple script. Let's create a file called `hello_world.py`, and write in it:

```
print('Hello world')
```

We can execute it on the terminal by typing:

```
python hello_world.py
```

Now, let's modify the file so it will take as an argument a name, and then output a greeting to that person:

```
import sys
name = sys.argv[1]
print('Hello', name)
```

and then we can try:

```
python hello_world.py Sergio
```

If I wanted to use the shorthand `pd` to access `pandas` package functions, what would my import look like?

yes `import pandas as pd`

no `from pandas import pd`

go slower `import pd by pandas`

go faster `import pandas`

Base problem

Scenario:

We will be analyzing the data of the project "A large-scale COVID-19 Twitter chatter dataset for open scientific research - an international collaboration", specifically the term count. We will be creating a script that will return the most frequent terms.

Aim:

Create a script where you can input the minimum count of terms, read a csv file, and output a filtered version.

Steps

1. Download the input file to your guest account:

```
wget https://raw.githubusercontent.com/Andesha/sharcnet-python/master/inputs/frequent_terms.csv
```

1. Create a file for the script where you import the `sys` module to allow input the threshold and the name of the output file:

In [ ]:
```
import sys
program = sys.argv[0]
threshold = sys.argv[1]
outfn = sys.argv[2]
```

1. Print a greeting including the name of the program, and the inputs:

In [ ]:
```
welcome_msg = 'Welcome to {}. Your threshold is {} an the outputfile is {}'
print(welcome_msg.format(program, threshold, outfn))
```

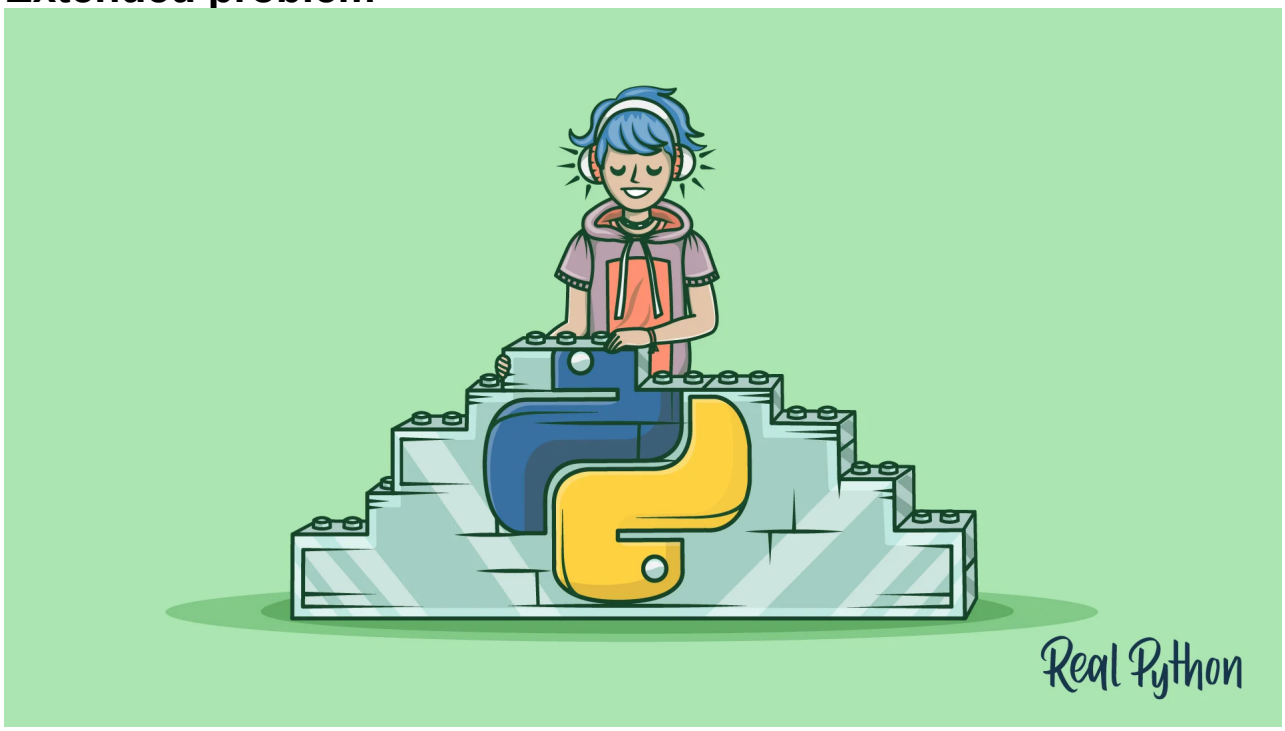
1. Using the context manager, read the file and split it by comma

In [ ]:
```
with open('frequent_terms.csv') as infile:
      text = [line.strip().split(',') for line in infile]
```

1. Write to file the entries where the count is greater than the input from `sys`.

In [ ]:
```
with open(outfn, 'w') as outfile:
    for index, entry in enumerate(text):
        if index == 0:
            line = '{}\n'.format(','.join(entry))
            outfile.write(line)
        elif float(entry[1]) >= threshold:
            line = '{}\n'.format(','.join(entry))
            outfile.write(line)
```

Extended problem



Now you will modify the script so that it will only output terms that start with an `e` **OR** that end in `n`. Additionally you will write a second output file (the name has to be provided by the command line as the third argument) with the 10 least frequent terms. **HINT:** a numeric list (has to be of type int or float) can be sorted using the function `sorted`.