Python Refresher
Software Engineering for Scientists
https://github.com/swe4s/lectures/tree/master/python_refresher

Python programs are typically saved in files that end with the `.py` extension. The code in a python file can be executed using the `python` command. Python files that are executed directly are typically called scripts, and library files are called modules. Both end in `.py`.

`hello_world.py`

```
print('Hello, World!')
```

```
$ python hello_world.py
Hello, World!
```

Variables store values. Python does not require that you specify what type (integer, double, string, array) a variable will be, and you can switch types by reassigning the variable.

`variables.py`

```
a = 10
b = 11
print(a, b, a + b)
b = '11'
print(a, b, str(a) + b)
```

```
$ python variables.py
(10, 11, 21)
(10, '11', '1011')
```

Indentation (leading spaces) is (unfortunately) very important. In python, code blocks (e.g., all of the code that will execute when an `if` statement resolves to `True`) are at the same indentation level. While python will allow any number leading spaces for a given block, you should use four.

`indent.py`

```
i = 10
j = 11

if i < j:
 print( str(i) + ' is less than ' + str(j))
 print( str(j) + ' is not less than ' + str(i))

if i > j:
    print( str(i) + ' greater than ' + str(j))
print( str(j) + ' is not greater than ' + str(i))
```

```
$ python indent.py
10 is less than 11
11 is not less than 10
11 is not greater than 10
```

Functions are a useful way of simplifying code and allowing portions to be reused. In Python, function declarations begin with the `def` statement, then the function name, and last the parameters. Parameters can be positional, named, or a mix.

functions.py

```
def is_equal(a, b, delta=0.001):
    return abs(a - b) < delta

print(my_imports.is_equal(0.001,0.002))
print(my_imports.is_equal(0.001,0.002,0.005))
```

```
$ python functions.py
False
True
```

Import statements allow Python scripts to use code defined in other modules. The Python Standard Library (https://docs.python.org/3/library/) has many modules that can be imported directly. If a module is not in the standard library, then it must be downloaded and installed (e.g. `conda install`, `pip install`) before it can be imported and used your scripts.

imports.py

```
import math
import random

a = 1
b = 100
r1 = random.randint(a, b)
r2 = random.randint(a, b)
print(r1, r2, math.gcd(r1, r2))
```

```
$ python imports.py
74 62 2
```

You can define and import your own libraries by moving functions defined in a script to their own module file. Modules increase reusability and their use is highly encouraged. For `import my_import` to work, a file named `my_import.py` must be in the same directory as the file attempting the import or in the *path*.

my_imports.py

```
def is_equal(a, b, delta=0.001):
    return abs(a - b) < delta
```

use_my_imports.py

```
import my_imports

print(my_imports.is_equal(0.001,0.002))
print(my_imports.is_equal(0.001,0.002,0.005))
```

Scalars are individual values (e.g., `1`, `3.1415`, `'SWE4S'`). Arrays, Lists, Sets, and Tuples are data structures that hold multiple values. Arrays can only hold one type of object (i.e. all ints, all floats), but store things efficiently. Lists can store different types of values, but less efficiently. Sets ensure items are unique. Tuples are efficient but are immutable (you cannot add or remove elements).

array_list_tuple_set.py

```
from array import array
from sys import getsizeof as size

a = array('f',[1.0, 2.0, 3.14])
l = [1.0, 2.0, 3.14]
s = set([1.0, 2.0, 3.14])
t = (1.0, 2.0, 3.14)
print (a, l, s, t)

f = 1.0
print(size(f)*3)
print (size(a), size(l), size(s), size(t))
```

```
$ python array_list_tuple_set.py
array('f', [1.0, 2.0, 3.140000104904175]) [1.0, 2.0, 3.14] {1.0, 2.0, 3.14} (1.0, 2.0,3.14)
72
76 88 224 72
```

Looping is an efficient way to execute the same code many times considering different values (e.g., reading a file, calculating the median of different sets of numbers, etc.). The most common loop construct is the `for` loop. In Python, `for` loops are performed over sequences (lists, sets, strings, etc). Most other programming languages iterate a variable from some initialize value and is incremented (or decremented) at each loop until a terminating condition is met (i.e., `for(i=1;i<=3;++i)` in C). Since Python loops over sequences, a similar loop can be expressed if, for example, a list is populated with the values 1, 2, and 3 (`range(1,4)`).

for_loop.py

```
print(range(1,4))
s = 0
for i in range(1,4):
   s += i
print(s)
```

```
$ python for_loop.py
range(1, 4)
6
```

Reading a File in Python is easy. Open it. Loop over the lines. Close it. The `open` function takes the path to the target file and a single character that sets the mode (how you want to use the file). Here we are reading so we use `'r'`. To open a file for writing use `'w'`. There are eight options for the mode that cover other needs. See the full list here https://docs.python.org/3.6/library/functions.html#open. From these parameters, `open` returns a file object that we can then loop over (assuming it is a text file) to get one line at a time. The character that separates one line from the next is called a *newline* character, which can be `'\n'`, `'\r'`, or `'\r\n'`, depending on how the file was created and on what operating system. The `for` loop will stop when it reaches the end of the file, at which point the file needs to be closed

The following example uses a file from the New York Times COVID-19 GitHub repository. To get that file, run

```
$ git clone https://github.com/nytimes/covid-19-data.git
```

read_file.py

```
file_name = 'covid-19-data/us-counties.csv'

f = open(file_name, 'r')
for l in f:
    print(l, end='')
f.close()
```

```
$ python read_file.py
date,county,state,fips,cases,deaths
2020-01-21,Snohomish,Washington,53061,1,0
2020-01-22,Snohomish,Washington,53061,1,0
2020-01-23,Snohomish,Washington,53061,1,0
2020-01-24,Cook,Illinois,17031,1,0
2020-01-24,Snohomish,Washington,53061,1,0
2020-01-25,Orange,California,06059,1,0
2020-01-25,Cook,Illinois,17031,1,0
2020-01-25,Snohomish,Washington,53061,1,0
2020-01-26,Maricopa,Arizona,04013,1,0
```

To Read from STDIN, just use `sys.stdin` as the open file handle.

`read_stdin.py`

```
import sys

for l in sys.stdin:
    print(l, end='')
```

```
$ cat covid-19-data/us-counties.csv | python read_stdin.py
date,county,state,fips,cases,deaths
2020-01-21,Snohomish,Washington,53061,1,0
2020-01-22,Snohomish,Washington,53061,1,0
2020-01-23,Snohomish,Washington,53061,1,0
2020-01-24,Cook,Illinois,17031,1,0
2020-01-24,Snohomish,Washington,53061,1,0
2020-01-25,Orange,California,06059,1,0
2020-01-25,Cook,Illinois,17031,1,0
2020-01-25,Snohomish,Washington,53061,1,0
2020-01-26,Maricopa,Arizona,04013,1,0
```

Data comes in many different forms and is rarely in the exact format that you need. Searching through a file to extract the desired fields requires String Parsing. The most basic and by far the most common string paring operation is to take a line from a data file and split it by its delimiters. Delimiters can be any character but are usually either a comma or a tab. Given a string, such as a line for a file, running `split(',')` on the string will return an array with the fields that were separated by commas.

One pesky detail is that lines from a file have a character at the end called a *newline*. We can type it out as `'\n'`, but when you print a string with `'\n'` it producers a line break

```
$ python -c "print('a\nb\nc')"
a
b
c
```

While you may not be able to see this character it is there and it can cause bugs, in particular with equality operations because `'value'` does not equal `'value\n'`. You can avoid all of these issues by using `rstrip()` to remove the newline character and any other trailing whitespace.

parse_string.py

```
file_name = 'covid-19-data/us-counties.csv'

f = open(file_name, 'r')

for l in f:
    A = l.rstrip().split(',')

    date = A[0]
    county_city = A[1]
    state = A[2]
    fips = A[3]
    cases = int(A[4])
    deaths = int(A[5])

    if state == 'Colorado' and county_city == 'Boulder':
        print(date, cases)

f.close()
```

```
$ python parse_string.py
2020-03-14 1
2020-03-15 7
2020-03-16 7
2020-03-17 8
2020-03-18 8
2020-03-19 11
2020-03-20 24
2020-03-21 30
2020-03-22 37
2020-03-23 39
```