Logic, Loops and Input/IO

Looping through data and reading in files or writing out to files.

There are many Python functions available to do many things. Here is a link to list of many of these https://docs.python.org/3/library/functions.html.

Logic

In python if statements end with `:` and Indentation is required. This indentation can be with spaces or tabs, but not a mix of each.

```
if TRUE:
    THEN DO X
elif OTHER_THING_TRUE:
    THEN DO Y
else:
    THEN DO Z
```

Loops

Loops let you

```
while ( TRUE ):
    Do X
for ITERATOR in LIST:
    Do Y
for i in range(5):
    print(i)
```

Python Indentation

Most editors do this automatically but you might find it hard to detect errors looking at the code until you try to run it if there are tabs in one part and sapces in another.

This indentation can be either spaces or tabs. Emacs/Atom/Vi allow customizing your editor so that when [tab] is typed it will either insert 4 spaces or a tab. Switching between editors while editing a file can sometimes cause problems.

Solution: delete the indendentation and enter the tab (or spaces) again.

After this slide you will now understand this scene from Silicon Valley.

Logic operators

To test if somehting is equal use `==` or `is`.

`==` test numerical equality, `is` checks if both the variables point to the same object. In some cases this can be different things so avoid using `is` for strings and numbers.

```
x=10
if x == 10:
        print("this == 10")
if x == "10":
        print("this == '10'")
if x is 10:
        print("this is a 10")
if x is "10":
        print("This is '10'")
a = b = [1,2,3]
c = [1,2,3]
print( a == b)   # this is true
print( a == c )  # this is true
print( a is b )  # this is true
print( a is c )  # this is false

# another example of testing for equality of string content
# and the actual object
a="fiat"
b="".join(['f','i','a','t'])
print("a is ",a)
print("b is ",b)
if a is b:
   print("strings are same") # this will NOT print
if a == b:
   print("these are the same string: %s %s"%(a,b)) # this will print
```

`!=` for not equal and `is not` for correlary test with `is`.

```
if a is not c:
   print("a is not c")
if 10 != 20:
   print("10 is not equal to 20")
```

`<`, `<=`, `>`, `>=` for less than, less than or equal to, greater than, greater than or equal to.

```
a=7
b=20
if b > 20:
```

```
   print("b is greater than a")
```

Combining logical statements

`not`, `or`, `and` logical operators.

`True and True` = `True True or True` = `True not True` = `False`

`False and False` = `False False or False` = `False not False` = `True`

`True and False` = `False True or False` = `True`

```
a=10
b=12
c=13
if a < b and b < c:
    print("a is smaller than c")

if not a == b:
    print("a is not equal to b")
else:
   print("a is equal to b")
```

Loops

`while` loops iterate as long as a condition is true.

```
x=1
while ( x < 10 ):
      print("x is ",x)
      x += 1
```

`for` loops through a set of items

```
DNA="AACGCA"
for base in DNA:
    print("base is ",base)
```

range operator (like `seq` in UNIX)

Simple way to setup a counter. See the range

```
for i in range(1,10): # forwards counting
    print(i)
for i in range(10,0,-1): #backwards counting
    print(i)
```

```
for i in range(2,16,2): # count by twos
    print(i)
```

quick reminder about data types in Python

Lists are sets of values, strings, numbers. Initialize with a `[ ]` around the values. **Tuples** are sets of values but they are immutable (cannot change) and are initialized with `( )`.

```
mylist = ['a', 'b','c']
mylist.append('z')
mylist[2]='C'
mylist.insert(0,'Start')
print(mylist)
# will print
# ['Start', 'a', 'b', 'C', 'z']
```

A **tuple** cannot be changed

```
mylist = ('a', 'b','c')
# this next line will throw an error
mylist.append('z')

print(mylist)
# will print (note the parens not [ ] )
# ('a', 'b', 'c')
```

Iterate on a list or tuple

```
list = [7, 10, 2, 2, 7]
for i in range(len(list)):
    print("list item ",i, "=",list[i])
    # print("list[%d] = %s"%(i,list[i])) # I like formatted printing too

list=(1,7,8)
for item in list:
    print("item is ",item)
# prints out
# item[0] = 1
# item[1] = 7
# item[2] = 8

#can use enumerate to get the index number
for idx,item in enumerate(list):
    print("item[%d] = %s"%(idx,item))
```

```
# prints out
# item [0] = 7
# item [1] = 10
```

Exiting a loop early

Sometimes we want to exit the loop if a condition is met. `continue` will jump back to the beginning of the loop and not execute any more of the code block. `break` will exit the loop all together.

```
list = ('a', 'b', 'c', 'C','d')
for l in list:
    if l == "b":
        continue

    if l == "C":
       break
    print(l)

# would print only the following
# a
# c
```

File handles

The open function is used to open file handles. Good reference can be found at https://en.wikibooks.org/wiki/Python_Programming/Input_and_Output

Data streams could be from cmdline (eg STDIN)

```
$ cat file | python myscript.py
```

Inside the Python code here we print every line that is sent in.

```
#!/usr/bin/env python

import sys # this tells python we need to use a package called sys
i=0
for line in sys.stdin:
    print("line[%d] is %s"%(i,line),
          end = '')
    i +=1
```

Can also open files for reading with `open`.

```
filehandle = open(myfile,"r")
```

Here is a program that will read in each line from a file and print it back out.

```
#!/usr/bin/env python
i=0
file = "data1.dat"
fh = open(file,"r")
for line in fh:
      print("line[%d] is %s"%(i,line),
          end = ''))
      i+=1
```

What if file wasn't there. Try it but python will throw an error and exit. You could protect your code a bit more with this construction.

```
with open(myfile,"r") as fh:
    for line in fh:
        print(line)
```

The `with` means only run the next lines if the filehandle can be opened. There are additional more robust error handling and catching options.

Writing data out

Besides using print to the OUTPUT stream we can write to a file

```
ofh = open("my_data.tab","w")
ofh.write("Species\tHabitat\tSize\n")
ofh.write("Crab\tBeach\tM\n")
ofh.write("Fish\tOcean\tS\n")
```

```
$ cat my_data.tab
Species    Habitat    Size
Crab    Beach    M
Fish    Ocean    S
```

Modules

Modules are collections of code routines. We will talk more about functions/routines in next lecture. Can use these as tools.

sys - System-specific parameters and functions urllib.request - URLs for opening web or network connections csv - Comma and Tab delimited data parsing

STDIN again

Remember we can pass data into a program via STDIN if we use the '|' "pipes" in UNIX. Here we just wrote a `wc -l` replacement.

```
import sys
counter = 0
for line in sys.stdin:
    counter += 1
print ("There are",counter, "lines")
```

Streams can be URL too

Can be a network connection (eg URL for web or FTP)

```
import urllib.request
orginfo = "https://raw.githubusercontent.com/biodataprog/GEN220_data/main/data/random_exons
info = urllib.request.urlopen(orginfo)
for line in info:
    linestrip = line.decode('UTF-8').strip()
    print(linestrip)
```

CSV files

Comma delimited files can be parsed this way. The module is smart enough to handle cases where the delimiter is embedded within quotes. So for example

```
Colorado State University,"Fort Collins, Colorado",CSU
```

```
import csv
file2 = "test2.csv"
with open(file2) as csvfile:
    reader = csv.reader(csvfile,delimiter=",")
    for row in reader:
        print("\t".join(row))
with open("outtest.csv","w") as csvfile:
    writer = csv.writer(csvfile,delimiter="\t")
    writer.writerow(["Name","Flavor","Color"])
    writer.writerow(["Apple","Sweet","Red"])
    writer.writerow(["Pretzel","Salty","Brown"])
```