Module 5 – Definite and Indefinite Loops

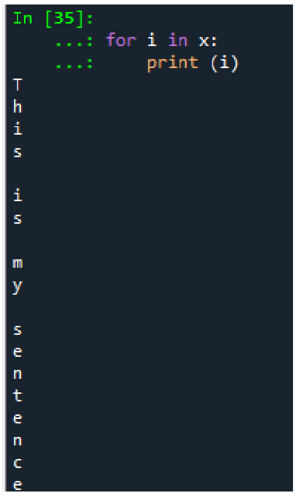
Instructions

1) If not already open, from the Anaconda Navigator Home screen, launch Sypder

2) In Spyder, go to "File", then "Open", then navigate to the location where you saved: "5_Looping.py", and select it to open the script file.

3) Introduction

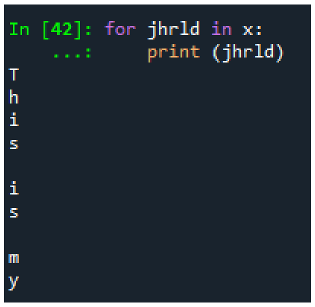
The previous modules gave you an introduction to some of the basic data types in Python, but in order to start building more useful programs, looping is inevitably necessary. Two kinds of loops we will cover in this module are **definite** loops and **indefinite** loops. **Definite loops** include a set number of iterations to complete, or in other words: the loop will execute a *definite* number of times. We saw an example of one of these near the end of the previous module.



```
In [35]:
    ...: for i in x:
    ...:     print (i)
T
h
i
s

i
s

m
y

s
e
n
t
e
n
c
e
```

This example of a definite loop involved printing out a value at (i) index position each time a value is encountered. Whenever a value is encountered at a given index position for any object, this loop then prints the value at that index position.

Notice, though, that the loop terminated when reaching the last value stored in the string ('e'). The nice thing about the structure of this loop is that the number of index positions stored in the object sets the boundaries of the definite loop. A note here: 'i' is a common way to note an **iterator** in a definite loop, but you could really store any string character here that acts as a temporary variable for use in this loop, so long as you are consistent with its use. See, for example:



```
In [42]: for jhrld in x:
    ...:     print (jhrld)
T
h
i
s

i
s

m
y
```

Setting ‘i’ as the iterator variable for temporary use in the definite loop is just common notation, and certainly simpler than that jumble of consonants that I put together.

In contrast, **indefinite loops** continue to iterate until a condition is met, which terminates the iteration. We actually saw one of these in the first program we developed in the introduction. Many indefinite loops include **conditional statements**, which we will explore a bit in this module, and focus on some more in the next.

Definite loops can readily be identified by the “for (variable) in (sequence):” structure, and indefinite loops can likewise be identified by the “while (condition): (evaluation)” structure.

4) Introduction to Indefinite Loops

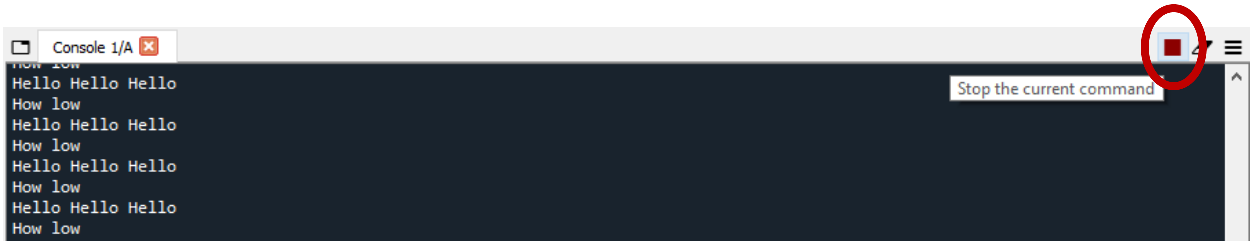
First, **view** lines 9-13. Don’t run this block, at least not yet. Take a look at the logical structure. First, a constant numeric variable is set on line 9, and stored in the variable **n**. This variable – and its value – is entered into a conditional expression that evaluates whether or not an iteration in an indefinite loop should proceed. If true, it proceeds until a set condition is met where it breaks, or it is simply no longer true. Of course, n=5, and this is greater than 0, so per lines 10-12, two lines of print statements should then proceed to run.

Notice the structure of the statement, though. You may have heard that Python is picky about indentation, and that is not incorrect. Line 10 contains the ‘while’ statement, followed by the condition and then the necessary colon (:)

Then, Lines 11-12 are indented, and in so doing, are executed each time the condition is met to begin an iteration for the loop set on Line 10. So long as the condition is met (true) to begin another iteration of the loop in Line 10, whatever is supposed to happen in Lines 11-12 will execute, until a condition that makes Line 10 no longer true is encountered. Once that happens, the loop iterations will stop, the code steps out of the loop, and then proceeds to Line 13.

Logically, though, what will happen if you run lines 9-13?

If you really must, feel free to run lines 9-13 (though do not feel compelled to). If you choose to, make a note of the location of the ‘stop’ command in the console and use it when you are ready.



```
How low
Hello Hello Hello
How low
Hello Hello Hello
How low
Hello Hello Hello
How low
Hello Hello Hello
How low
```

On a lighter note, good luck getting that song out of your head, too.

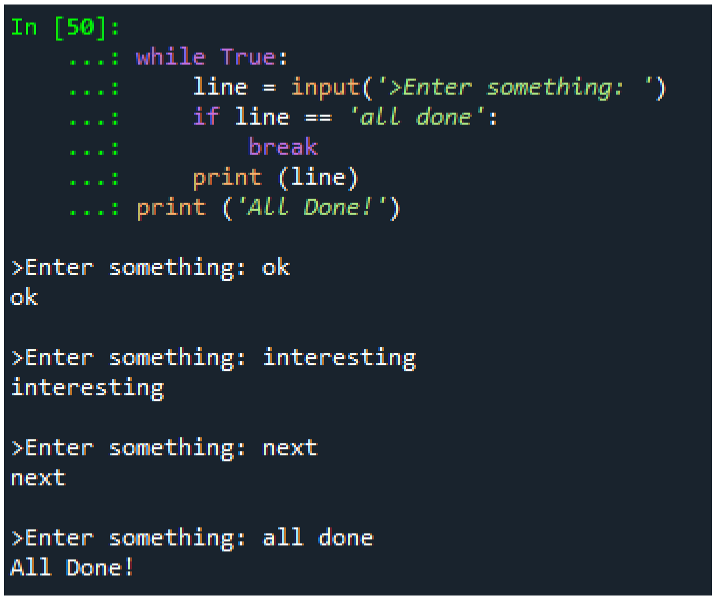
In contrast, now run **Lines 17-21 together**. What changed from running (if you did) lines 9-13? In this case, the variable value entered into the conditional statement for the indefinite loop was not true, so nothing in the loop evaluated. The code then proceeded to Line 21, which simply executed a print statement.

Running Lines 23-29 will help you see what happens when you enter **conditional statements** that allow the indefinite loop to break when a condition is met. Lines 24 initiates an indefinite loop, but now, a conditional statement is set to simply note whether or not a condition is met (True). Since we are starting with 'True' to initiate the loop, the condition is met, and into the loop we go.

We step into the loop, where I have set a variable to capture user input as string data. If the user enters a statement equivalent to ('==')* 'all done', then the loop will break, at which point the code steps out of the indefinite loop and goes straight to the code on Line 29, printing 'All done!' It is worth noting that, if that were to happen, the code on Line 28 will not be evaluated, since the loop is broken before we get there.

Note: in Python, = is an assignment operator, and == is an equality operator

If the user enters a statement that is not equivalent to 'all done', then the break condition is not met. So now, the program prints out whatever the user's input was for that iteration (Line 28), and the loop iterates again. This continues until the break condition is met. Basically, until someone enters 'all done', this program will run indefinitely, so make sure you eventually enter that text to exit the loop. See below for three iterations that I entered in the console that allowed the loop to continue until finally, we met the break condition.



```
In [50]:
    ...: while True:
    ...:     line = input('>Enter something: ')
    ...:     if line == 'all done':
    ...:         break
    ...:     print (line)
    ...: print ('All Done!')

>Enter something: ok
ok

>Enter something: interesting
interesting

>Enter something: next
next

>Enter something: all done
All Done!
```

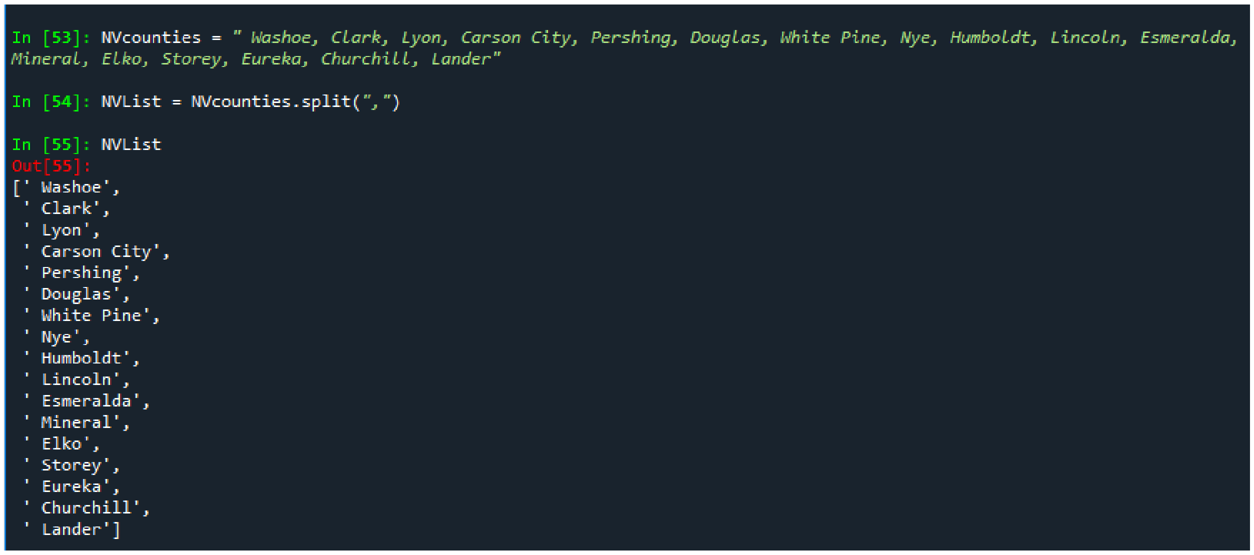
4) Introduction to Definite Loops

What follows are a series of examples of how to use definite loops. The first (Lines 37-40) works by storing all of Nevada's 17 county and county-equivalent populations (as integers) in a list object, which we will explore more in a later module. Then, a definite loop accesses each value in that list and prints it, one at a time (Lines 38-39). Line 40 is really just an optional statement that signals that the loop is done.

Run Lines 37-40 together, which initiate the loop and prints the population values. That's nice to see, but unless you really know your counties and populations, you probably don't know which population value belongs to which county in the state. Maybe you could guess Clark, Washoe, and one or two others, but all 17 would be tricky.

Line 43 creates a string object that includes all of the county and county-equivalent names. **Run this line**

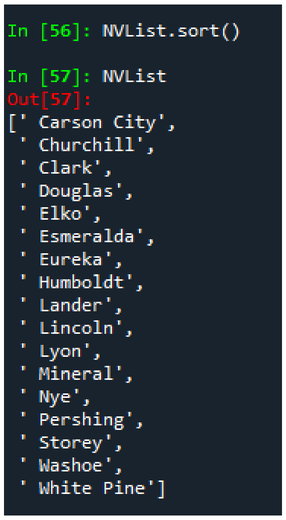
Next, take a look at **Line 44 and run this line**. It makes use of the **.split() method**, which searches throughout the length of a string object, specifically looking for whatever character you identify in the parenthesis. In this case, I'm searching for the comma. Hence the NVcounties.split(",") syntax. I store this output in a new variable called NVList. Take a look at NVList after running Line 44, and then entering NVList in the console



```
In [53]: NVcounties = " Washoe, Clark, Lyon, Carson City, Pershing, Douglas, White Pine, Nye, Humboldt, Lincoln, Esmeralda,
Mineral, Elko, Storey, Eureka, Churchill, Lander"

In [54]: NVList = NVcounties.split(",")

In [55]: NVList
Out[55]:
[' Washoe',
 ' Clark',
 ' Lyon',
 ' Carson City',
 ' Pershing',
 ' Douglas',
 ' White Pine',
 ' Nye',
 ' Humboldt',
 ' Lincoln',
 ' Esmeralda',
 ' Mineral',
 ' Elko',
 ' Storey',
 ' Eureka',
 ' Churchill',
 ' Lander']
```

The .split() command automatically stores each item encountered between the "," characters in NVcounties as a unique item in a list.

Line 45 now takes the county names and, through use of the **.sort()** method, places them in alphabetical order. **Run Line 45** and enter NVList in the console, as below



```
In [56]: NVList.sort()

In [57]: NVList
Out[57]:
[' Carson City',
 ' Churchill',
 ' Clark',
 ' Douglas',
 ' Elko',
 ' Esmeralda',
 ' Eureka',
 ' Humboldt',
 ' Lander',
 ' Lincoln',
 ' Lyon',
 ' Mineral',
 ' Nye',
 ' Pershing',
 ' Storey',
 ' Washoe',
 ' White Pine']
```

As it turns out, this alphabetical order now corresponds with the order in which the population values are sorted in the list from the NVPopulations2017 variable. We will explore ways to combine the values later when we explore list operations.

For now, we know that the sorted names in NVList are county names, but someone outside the state may not now that. In order to communicate this, **run lines 46-50** together.

Line 46 initiates the definite loop, where we iterate through each item encountered in the NVList list, then use the **.strip() method** on line 47 **–** which you have previously seen **–** to remove (strip) either leading or trailing blank spaces that came along with the split command (see screenshot above), and then generates a print statement that combines the value now associated with the x variable with the string constant "County" on line 48.

Line 49, though, brings up a unique point about Nevada: what do we do about Carson City, which is a county equivalent, but not necessarily a county. As in, it's not "Carson City County", it's "Carson City".

What are some ways you can think of to address this issue?

5) Counters and Loops

You will find that counters often are used in advance of loops, as they help with evaluating if conditions are met in advance of a loop, and also a place to store values (that can be overwritten, of course) produced by the loop. There are some conditional statements in here, which we'll cover in more detail in the next module. For now, though, run each block of code and view the outputs.

Lines 54-60 evaluate county values from an ordered set, eventually printing out the largest population value encountered in that set. Make sure you **run lines 54-60 together**. Note that two values are reported for each iteration of the loop. First, we see the print statement on Line 55, but once the loop initiates, we see the largest population value encountered yet first, followed by the value encountered at 'i' position in the list.

Run Lines 63-68 together, which provide a unique count value for each population value encountered in the list at the corresponding position 'i'. Here, we start the count at zero to account for Python's built-in accounting for the first value recorded as position 0.

Run lines 71-76 together, which shows an example of how to report a cumulative sum (and does so for each step through the loop). The county population is reported first (i), followed by the cumulative sum to date, accounting for the newest addition.

Lines 79-85 (run them together) work similarly to Lines 54-60, but this time find the smallest population encountered.

This completes the structured part of Module 5. Feel free to experiment some more with setting variables and values, working with print statements, or anything else you are curious about.