Python for Scientific Data Analysis

Basic Python

Section 5: Looping

For-Loops

Introduction

Programs need to do repetitive things very quickly. If you only want to do an operation, say, twice then you could just copy and paste the same operation text on subsequent lines. What about 3 times? A bit more annoying -- especially if you want to change subtle things about the operation each time -- but possible. 10,000 times? No way. Hence, we need a way of *loop*-ing an operation. And that's where *Loops* come into play.

The most straightforward version of a *loop*-ing operation is a `for-loop`. A `for-loop` means literally that: "for [some number of iterations], do the following block of code". Like if-else statements, the block of code following the loop designation is indented.

Writing For-Loops

So how do you decide how to do a number of iterations?

The `range` function provides a simple and pretty bulletproof way of defining the number of loop iterations.

```
Here's a simple example:
```

```
for i in range(0,5):
  print(i*5+5,i)
```

Which prints out

```
5 0
10 1
15 2
20 3
25 4
```

The "0" and "5" enclosed within the parentheses for range define the "start" and "stop" values for the loop. Note here that `range(0,5)` and `range(5)` are equivalent statements. However, `range(1,5)` is different:

```
for i in range (1,5):
  print(i*5+5,i)
```

Which prints out

```
10 1
15 2
20 3
25 4
```

I.e. we are missing the 0th iteration because our "start" point is the 1st iteration.

Loops and Lists

In addition to looping over numbers -- e.g. over a `range` -- you can loop over elements of a `list`. Lists are exactly what their name says: a container of things that are organized in order from first to last. It's not complicated; you just have to learn a new syntax. We will learn about these a little bit more later. But for now, let's just work with the definition I gave above: `lists` = a container of things that are organized in order from first to last.

First, there's how you make lists:

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green']
weights = [1, 2, 3, 4]
```

You start the `list` with the [(left bracket) which "opens" the list. Then you put each item you want in the list separated by commas, similar to function arguments. Lastly, end the list with a] (right bracket) to indicate that it's over. Python then takes this list and all its contents and assigns them to the variable.

Now we are going to loop through the elements of one of these lists and you can see what happens.

```
hairs = ['brown', 'blond', 'red']

for i in hairs:
  print("these are the hairs %s " i)
```

Which should print out:

```
the hairs are brown
the hairs are blond
the hairs are red
```

Now if you do `for i in hairs:` and `print("the hairs are %s" % hairs)` in the next line then you get the wrong answer, since you are literally saying "print out the entire list called `hairs'":

```
the hairs are ['brown', 'blond', 'red']
the hairs are ['brown', 'blond', 'red']
the hairs are ['brown', 'blond', 'red']
```

Here's an alternate way of iterating that is a bit more complex but perhaps more intuitive if you are used to other languages (e.g. IDL, C, Fortran):

```
for i in range(len(hairs)):
  print("the hairs are %s" % hairs[i])
```

Another cool trick as noted before: you can control which elements of a list are printed out. E.g. you get the same answer as above if you do this:

```
for i in range(0,len(hairs)):
  print("the hairs are %s" % hairs[i])
```

But if you do this:

```
for i in range(1,len(hairs)):
 print("the hairs are %s " % hairs[i])
```

then only the last two elements of *hairs* are printed because the first list element (indexed as 0) is not included.

you can also add new elements to a list with the attribute *append* , e.g.

```
elements = []
for i in range(0,3):
 print("Adding element %d to the list." % i)
 elements.append(i)
print(elements)
```

You get the following:

```
adding element 0 to the list
adding element 1 to the list
adding element 2 to the list

[0, 1, 2]
```

Now, try to import and then execute ex32.py

```
def runme():
 the_count=[1,2,3,4,5]
 fruits=['apples','oranges','pears','apricots']
 change=[1,'pennies',2,'dimes',3,'quarters']


 for number in the_count:
        print("This is count %d" % number)

 for fruit in fruits:
        print("A fruit of type: %s" % fruit)

 for i in change:
        print("I got %r" % i)

 elements = []

 for i in range(0,6):
        print("Adding %d to the list." % i)
        elements.append(i)

 for i in elements:
        print("Element was: %d" % i)
```

You get the following:

```
>>> from ex32 import runme
>>> runme()
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got 'pennies'
I got 2
I got 'dimes'
I got 3
I got 'quarters'
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
```

While-Loops

You can also execute a *while-loop* . A *while-loop* will keep executing code under some block, so long as the boolean expression under the *while* loop is *True*. I.e. "while [some condition], do [ ]". If [some condition] is always true, then the code never stops (!!!). Because of that, *while-loops* can be a bit tricky, even dangerous. Use them selectively.

An example:

```
i = 0
numbers = []

while i < 6:
    print("At the top i is %d" % i)
    numbers.append(i)

    i = i + 1
    print("Numbers now: ", numbers)
    print("At the bottom i is %d" % i)


print("The numbers: ")

for num in numbers:
    print(num)
```

Whose result looks like:

```
>>> import ex33
>>> ex33.runme()
At the top i is 0
Numbers now:  [0]
At the bottom i is 1
At the top i is 1
Numbers now:  [0, 1]
At the bottom i is 2
At the top i is 2
Numbers now:  [0, 1, 2]
At the bottom i is 3
At the top i is 3
Numbers now:  [0, 1, 2, 3]
At the bottom i is 4
At the top i is 4
Numbers now:  [0, 1, 2, 3, 4]
At the bottom i is 5
At the top i is 5
Numbers now:  [0, 1, 2, 3, 4, 5]
At the bottom i is 6
The numbers:
0
1
2
3
4
5
```

Accessing Loop Elements

Say you create some array through a *for-loop* or a *while-loop* or really any other means. It's an array with some number of elements. How do you access those elements?

Easy. For example, if your array is test=[1,3,5,6], then test[0] is 1, test[3]=6 and so on.

Or how about this one: `animals = ['bear', 'tiger', 'penguin', 'zebra']` .

`animals[1] = 'tiger'` and so on.

Python array and list elements are indexed from 0, not 1.

Manipulating Lists

Now we are going to manipulate list entries. There are many ways to manipulate, including using *pop*, *append*, etc. Here's an example.

```
ten_things = "Apples Oranges Crows Telephone Light Sugar"

print("Wait there are not 10 things in that list. Let's fix that.")

stuff = ten_things.split(' ')
more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana", "Girl"
, "Boy"]

while len(stuff) != 10:
    next_one = more_stuff.pop()
    print("Adding: ", next_one)
    stuff.append(next_one)
    print("There are %d items now." % len(stuff))

print("There we go: ", stuff)

print("Let's do some things with stuff.")

print(stuff[1])
print(stuff[-1]) # whoa! fancy
print(stuff.pop())
print(' '.join(stuff)) # what? cool!
print('#'.join(stuff[3:5])) # super stellar!
```

You should see the following:

```
>python ex38.py

Wait there are not 10 things in that list. Let's fix that.
Adding:  Boy
There are 7 items now.
Adding:  Girl
There are 8 items now.
Adding:  Banana
There are 9 items now.
Adding:  Corn
There are 10 items now.
There we go:  ['Apples', 'Oranges', 'Crows', 'Telephone', 'Light', 'Sugar
', 'Boy', 'Girl', 'Banana', 'Corn']
Let's do some things with stuff.
Oranges
Corn
Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana
Telephone#Light
```

What happened here? So first, you have a string called *ten_things* and this string is *split* at every empty space and saved to a variable *stuff* . Now, it becomes obvious that there aren't actually 10 entries (in fact, you can check by defining a conditional by asking the length of the list -- i.e. len(stuff)).

So in a *while-loop*, the program then keeps *append*-ing things from the end of *more_stuff* ('popped' off the end) to *stuff* until there are 10 items. Then the print statements ... the first one -- *stuff[1]* -- will print out the 2nd element of *stuff*. The line after that? The last element of *stuff*. The line *stuff.pop()* prints out and removes the last array element of *stuff*.

Then *stuff[3:5]* prints out array elements "from the 4th and prior to the 6th element" (i.e. elements 4 and 5). What would *stuff[3:4]* print? Just "Telephone". What about *stuff[3:len(stuff)]* ? It would print:
`['Telephone', 'Light', 'Sugar', 'Boy', 'Girl', 'Banana', 'Corn'].`

Or *stuff[3:len(stuff)-1]* ? You guessed it:
`['Telephone', 'Light', 'Sugar', 'Boy', 'Girl', 'Banana'].`

Epilogue

That is a short introduction to *loop*-ing. Loops are extremely powerful. In the `Data Structures` section we will introduce ***Comprehensions*** for key data structures: lists,

tuples, arrays, and dictionaries. You don't need to worry about the details of these right now. Just know that we can do far more complex loop-like operations dealing with multiple variables in Python.