Python for Scientific Data Analysis

Data Structures

Section 3: Sequence Functions, Comprehensions, and Lambda Functions

Section 5 of the ***Basic Python*** Module describes the fundamentals of for-loops and while-loops. There, the standard way to construct the loop is with the `range` function, which makes Python similar to looping in other languages like C, Fortran, or IDL (e.g. in IDL, `for i=[min],[max] do begin; endfor`). But unlike these, Python has another set of *sequence_functions* and *comprehensions* to make looping far more powerful and sleeker.

Sequence Functions

enumerate

Motivation

It's common when iterating over a sequence to want to keep track of the index of the current item. A do-it-yourself approach would look like:

```
collection=[11,12,13,14]
i = 0

for value in collection:
 result=value*10
 print(i,result)
 #prints
 #0 110
 #1 120
 #2 130
 #3 140

 i += 1 #advance the index
```

```
0 110
1 120
2 130
3 140
```

Advancing the index manually by `+=1` is cumbersome to many,though. And a common coding mistake is to not update indexes properly at all. E.g.

```
#BAADDDD code
collection=[11,12,13,14]
i = 0

for value in collection:
 result=value*10
 print(i,result)
 #prints
 #0 110
 #0 120
 #0 130
 #0 140
```

```
0 110
0 120
0 130
0 140
```

Now one slightly wonky but totally legit work around is to create the index automatically using a combination of `range` and `len`. E.g.

```
for i in range(len(collection)):
 result=10*collection[i]
 print(i,result)

 #prints
 #0 110
 #1 120
 #2 130
 #3 140
```

```
0 110
1 120
2 130
3 140
```

Implementation

Since this situation is so common, Python has a built-in function, ***enumerate***, which returns a sequence of (i, value) tuples automatically:

```
for i,value in enumerate(collection):
 result=10*value
 print(i,result)
 #prints
 #0 110
 #1 120
 #2 130
 #3 140
```

```
0 110
1 120
2 130
3 140
```

Note, that this also can work for **Numpy** arrays:

```
import numpy as np

collection2=np.array(collection)
for i,value in enumerate(collection2):
 result=10*value
 print(i,result)
 #prints
 #0 110
 #1 120
 #2 130
 #3 140
```

```
0 110
1 120
2 130
3 140
```

When you are indexing data, a helpful pattern that uses enumerate is computing a dict mapping the values of a sequence (which are assumed to be unique) to their locations in the sequence. Here's an example:

```
captainlist = ['Kirk', 'Pike', 'April']
mapping = {}

for i, v in enumerate(captainlist):
 mapping[v]=i

print(mapping)
#{'Kirk': 0, 'Pike': 1, 'April': 2}

##OR
captainlist = ['Kirk', 'Pike', 'April']
mapping = {}

for i, v in enumerate(captainlist):
 mapping[i]=v

print(mapping)
#{0: 'Kirk', 1: 'Pike', 2: 'April'}
```

```
{'Kirk': 0, 'Pike': 1, 'April': 2}
{0: 'Kirk', 1: 'Pike', 2: 'April'}
```

zip

The second major sequence function widely used in scientific Python -- and one I personally like a lot better -- is the ***zip*** function. zip “pairs” up the elements of a number of lists, tuples, or other sequences to create a list of tuples.

E.g.

```
seq1 = ['foo', 'bar', 'baz']
seq2 = ['one', 'two', 'three']
zipped=zip(seq1,seq2)

list(zipped)
#[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

```
[('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

zip can take an arbitrary number of sequences, and the number of elements it produces is determined by the shortest sequence:

```
seq3 = [False, True]
list(zip(seq1, seq2, seq3))
#[('foo', 'one', False), ('bar', 'two', True)]
```

```
[('foo', 'one', False), ('bar', 'two', True)]
```

A very common use of zip is simultaneously iterating over multiple sequences, possibly also combined with ***enumerate***:

```
for i, (a, b) in enumerate(zip(seq1, seq2)):
 print('{0}: {1}, {2}'.format(i, a, b))

#0: foo, one
#1: bar, two
#2: baz, three
```

```
0: foo, one
1: bar, two
2: baz, three
```

Another more complex (but more useful) example using a list and Numpy arrays

```
starname=['HIP 99770','AF Lep','HR 8799','Vega']
spectype=['A5V','F8V','F0V','A0V']
starmag=np.array([4.9,6.3,5.9,0.0])
dstar=np.array([40.74,26.8, 39.4, 7.7])

for i,j,k,l in zip(starname,spectype,starmag,dstar):
 absmag=k-5*np.log10(l/10.)
 print('The absolute magnitude of star {0:s} with spectral type {1:s} is
{2:.3f}'.format(i,j,absmag))

#The absolute magnitude of star HIP 99770 with spectral type A5V is 1.850
#The absolute magnitude of star AF Lep with spectral type F8V is 4.159
#The absolute magnitude of star HR 8799 with spectral type F0V is 2.923
#The absolute magnitude of star Vega with spectral type A0V is 0.568
```

```
The absolute magnitude of star HIP 99770 with spectral type A5V is 1.850
The absolute magnitude of star AF Lep with spectral type F8V is 4.159
The absolute magnitude of star HR 8799 with spectral type F0V is 2.923
The absolute magnitude of star Vega with spectral type A0V is 0.568
```

It also works for dictionaries:

```
dict_one = {'name': 'William', 'last_name': 'Riker', 'job': 'Stepping Ove
r Chairs'}
dict_two = {'name': 'Wesley', 'last_name': 'Crusher', 'job': 'Shutting Up
'}
dict_three={'name': 'Mr','last_name':'Worf','job':'Getting Denied'}
for (k1, v1), (k2, v2), (k3,v3) in zip(dict_one.items(), dict_two.items()
,dict_three.items()):
   print(k1, '->', v1)
   print(k2, '->', v2)
   print(k3, '->', v3)

#name -> William
#name -> Wesley
#name -> Mr
#last_name -> Riker
#last_name -> Crusher
#last_name -> Worf
#job -> Stepping Over Chairs
#job -> Shutting Up
#job -> Getting Denied
```

```
name -> William
name -> Wesley
name -> Mr
last_name -> Riker
last_name -> Crusher
last_name -> Worf
job -> Stepping Over Chairs
job -> Shutting Up
job -> Getting Denied
```

sorted

The sorted function returns a new sorted list from the elements of any sequence. Here are some examples:

```
#list
a=[7, 1, 2, 6, 0, 3, 2]
sorted(a)
#[0,1,2,2,3,6,7]
```

```
[0, 1, 2, 2, 3, 6, 7]
```

```
#tuple
a=7,1,2,6,0,3,2
sorted(a)
#[0,1,2,2,3,6,7]
```

```
[0, 1, 2, 2, 3, 6, 7]
```

```
#array
a=np.array([7, 1, 2, 6, 0, 3, 2])
sorted(a)
#[0,1,2,2,3,6,7]
```

```
[0, 1, 2, 2, 3, 6, 7]
```

reversed

reversed iterates over the elements of a sequence in reverse order. E.g.

```
list(reversed(range(12)))
#[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

```
[11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

or

```
starship_classes=['Whatever Scott Bakula was On','Constitution','Excelsio
r','Ambassador','Galaxy','Sovereign','Odyssey']
reverse_order=reversed(starship_classes)
list(reverse_order)
#['Odyssey','Sovereign', 'Galaxy', 'Ambassador', 'Excelsior', 'Constituti
on','Whatever Scott Bakula was On']
```

```
['Odyssey',
 'Sovereign',
 'Galaxy',
 'Ambassador',
 'Excelsior',
 'Constitution',
 'Whatever Scott Bakula was On']
```

Comprehensions

Comprehensions are extremely powerful ways of concisely writing code. I LOVE comprehensions (and you should love them too).

List Comprehensions

List comprehensions allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression. They take the basic form:

`[expr for val in collection if condition]`

This is equivalent to the following for loop:

```
result = []
for val in collection:
 if condition:
  result.append(expr)
```

The filter condition can be omitted, leaving only the expression. For example, given a list of strings, we could filter out strings with length 2 or less and also convert them to uppercase like this:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
result=[x.upper() for x in strings if len(x) > 2]
#['BAT', 'CAR', 'DOVE', 'PYTHON']

print(result)
```

```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Dictionary Comprehensions

A dictionary comprehensions is a natural extension, producing dicts in an idiomatically similar way instead of lists. A dict comprehension looks like this:

```
dict_comp = {key-expr : value-expr for value in collection
             if condition}
```

Here is one example:

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
loc_mapping = {val : index for index, val in enumerate(strings)}
print(loc_mapping)
#{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}

loc_mapping2 = {index: val for index, val in enumerate(strings)}
print(loc_mapping2)
```

```
{'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
{0: 'a', 1: 'as', 2: 'bat', 3: 'car', 4: 'dove', 5: 'python'}
```

with Arrays

You can also use arrays with comprehensions if you are careful. A common situation is where you have a list of numbers (say, something you want to read out of a file) but what you *actually* want is a different number which does element-wise math, etc etc on that list. See example below

```
alistofnums=[2,5,6,8,10,12,19,23.5e-2]

#note: don't worry about the np.[stuff] calls yet
result=[np.exp(np.float64(j)) for j in alistofnums]  #takes "e to the [al
istofnums] power"

print(result)
```

```
[7.38905609893065, 148.4131591025766, 403.4287934927351, 2980.95798704172
83, 22026.465794806718, 162754.79141900392, 178482300.96318725, 1.2649087
687328917]
```

Lambda Functions

Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than “we are declaring an anonymous function”.

E.g. you could write a separate function:

```
def short_function(x):
 return x * 2
```

Or, declare an equivalent lambda function:

```
equiv_anon = lambda x: x * 2
```

Here's another example:

```

def apply_to_list(some_list, f):
  return [f(x) for x in some_list]
```

```
#now give a value for "some list" (or numpy array) and apply a lambda fun
ction
numbers=np.array([np.pi,np.e,40.74,21.8])
apply_to_list(numbers,lambda x: x**2/3.)
#[3.289868133696453, 2.4630186996435497, 553.2492000000001, 158.413333333
33333]
```

```
[3.289868133696453, 2.4630186996435497, 553.2492000000001, 158.4133333333
3333]
```

Now, lambda functions may seem a Rube Goldberg-machine-ish. E.g. in the example above we could have just done this:

```
#``result=list(numbers**2/3.)``
result=list(numbers**2/3.)
print(result)
```

```
[3.289868133696453, 2.4630186996435497, 553.2492000000001, 158.4133333333
3333]
```

and get the same answer and same data type. Lambda functions become somewhat more useful if you use them as an anonymous function within *another* function. Here's an example:

```
def rangesep(x):
 return lambda a: x/a

jupiterphyssep=rangesep(5.2) #5.2 = x in this case, jupiterphyssep is sav
ed as a function
distances=np.array([18.6,39.4,40.74])

jupiterangsep=jupiterphyssep(distances)
#now we tell it what 'a' is: it is the numpy array 'distances'-->answer
#array([0.27956989, 0.1319797 , 0.12763868])
print(jupiterangsep)
```

```
[0.27956989 0.1319797  0.12763868]
```

Note how Python treats each one of these variables. E.g. if you do `type(jupiterphyssep)` you get `<class 'function'>`; `type(jupiterangsep)` is `<class 'numpy.ndarray'>`.

```
print(type(jupiterphyssep))
print(type(jupiterangsep))
```

```
<class 'function'>
<class 'numpy.ndarray'>
```

Now, in this one instance again lambda functions still don't seem that particularly crucial (e.g. why not `jupiterangsep=5.2/distances`?). But if you have to perform this mathematical calculation repeatedly for different `a` values (in this case, different distances arrays) it can become more useful. Still not convinced? We will later encounter another example of a lambda function that makes ***Matplotlib*** plotting work well.

```

```