Python for Scientific Data Analysis

Data Structures

Section 2: Slicing

Tuples, Lists, Arrays, and Dictionaries can have any number of elements. What if you only want some subset of these elements (e.g. to perform some mathematical operation, some conditional print, ... anything)? Then you need to *slice* the data structure.

Basic Operation

You can select sections of most sequence types by using slice notation, which in its basic form consists of start:stop passed to the indexing operator []. If you are familiar with IDL, the syntax is similar (but not exactly the same!).

For slicing a tuple/list/array called L, with a `start` of *i* index and `stop` of *j* index -- i.e. L[i,j]_ -- means to take all the elements from L starting at L[i] until L[j-1].

Here's an example for slicing a ...

list:

```
a=[3,4,5,6]
b=a[1:2]
print(b)
#[4]
```

tuple:

```
a=3,4,5,6
b=a[1:3]
print(b)
#(4, 5)
```

Numpy ***array***:

```
a=np.array([3,4,5,6])
b=a[2:4]
print(b)
#[5 6]
```

Now, technically you can do slicing for dictionaries as well but it gets tricky: you have to use the `.items()` attribute and type conversions. Here's slicing of our previous dictionary example

```
stuff = {'name': 'Beavis', 'age': 15, 'weight': 145}
result=dict(list(stuff.items())[1:3])
#this also works: dict(tuple(stuff.items())[1:3])
print(result)
#{'age': 15, 'weight': 145}
```

Rules with Slicing Ranges

- 1. **Caution**!!!! - you may have noticed that the `stop` in this `start:stop` range seems to be missing a value. E.g. in the example above, a[i:j] where i = 0 and j= 1 only prints out the 0th element, not the 0th and 1st (i.e. it prints out [3] for a list). This is unlike IDL which would take values i through j *inclusive*.
- 2. You may omit the first or last bound of the slicing. i.e. L[0:2] and L[:2] mean the same thing.
- 3. You can use negative indexes for counting from the right. E.g. L[-1] is the last element of the list L.
- 4. Just using the semicolon -- i.e. L[:] -- means to select all values

Here are some examples from our list, a = [3,4,5,6]

```
a=[3,4,5,6]

a[0:1]
#[3]
a[:1]
#3
a[0:3]
#[3,4,5]

a[:3]
#[3,4,5]
a[2:4]
#[5,6]
a[2:]
#[5,6]

a[-3:]
#[4,5,6]
a[-1:]
#[6]
a[-3:3]
#[4,5]

a[:]
#[3,4,5,6]
```

The results are similar if a were a tuple or a Numpy array.

Steps and Reversing a Sequence

A step can also be used after a second colon to, say, take every other element of a list, tuple, or Numpy array:

```
a=[3,4,5,6]
a[::2]
#[3, 5]
a[1:4:2]
#[4,6]
```

A clever use of this is to pass a negative number -- e.g. -1 -- which has the useful effect of reversing a list, tuple, or Numpy array:

```
a=[3,4,5,6]
a[::-1]
#[6, 5, 4, 3]
a[::-2]
#[6, 4]
```

Conditional/Boolean Slicing

It's a little bit cart-before-the-horse to discuss the details of indexing and slicing Numpy arrays. That will be a focus of the Numpy discussion. But this is so fundamental to how to do slicing with a common data structure (i.e. a Numpy array), that it's worth a brief mention here before we discuss in full later.

Specifically, you can slice a Numpy array based on whether or not some statement is true/false etc for elements of that Numpy array using the `np.where` command. This is called "taking a Boolean array as a condition".

One quick example before we reserve this topic for later. See if you can figure out how to make this work in general:

```
a=np.array([3,4,5,6])
larger_than_three=np.where(a > 3)
between_three_and_six=np.where( (a > 3) & (a < 6) )
extrema=np.where( (a < 4) | (a > 5))

print(a[larger_than_three])
#[4 5 6]
print(a[between_three_and_six])
#[4 5]
print(a[extrema])
#[3 6]
```

Strides

One more trick Python allows with slicing is *strides*, whjich is the step size from one index to the other. While the default is one, you can set the stride to be an arbitrary number. For example:

```
a=[1,2,3,4,5,6,7,8,9,10]
a[1:7:2] #element 1 to <3, step size of 2
#[2,4,6]
```