Python for Scientific Data Analysis

Data Structures

Section 1: The Different Types of Data Structures

As McKinney says "*Python's data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer*."

Our discussion of data structures will generally follow a combination of McKinney's (mostly) and Fuhrer's (to a lesser extent).

Besides the *function*, which we have discussed already in ***Basic_Python***, Python's key built-in data structures include: *tuple*, *list*, and *dictionary*. For scientific Python, the ***Numpy*** package adds the critical, powerful *array* structure (long form name of this is "ndarray"). In this section, we will start with *tuples* and *lists*, then discuss *array*, and finally *dictionary*.

Function

Again, see previous discussion in ***Basic_Python***.

Tuples

Introduction

A *tuple* is a fixed-length, immutable sequence of Python objects. The simplest way to create a tuple is with a comma-separated sequence of values:

```
a=5,6,7,8
b='uno','dos','tres','quattro','sinco'
#printing >>> type(a), type(b) confirms that these are tuples
print(a)
print(b)
```

```
(5, 6, 7, 8)
('uno', 'dos', 'tres', 'quattro', 'sinco')
```

You can make tuples more complicated: e.g. nested tuples (a tuple of tuples)

```
c=(5,6,7),(8,'uno','dos'),('tres',9)
print(c)
```

```
((5, 6, 7), (8, 'uno', 'dos'), ('tres', 9))
```

You can convert a list or numpy array or really any iterator/sequence to a tuple by the `tuple` command:

```
d=[5,6,7] # a list
dtuple=tuple(d) #now a tuple

import numpy as np
e=np.array([5,6,7])
etuple=tuple(e) #equals (5,6,7)

f='thestring'
ftuple = tuple(f) #equals 't,h,e,s,t,r,i,n,g'

print(dtuple)
print(ftuple)
```

```
(5, 6, 7)
('t', 'h', 'e', 's', 't', 'r', 'i', 'n', 'g')
```

As with other things, you can access elements of a tuple using square brackets. E.g. in the above example `etuple[1]` will equal 6. Tuples are not mutable -- i.e. you cannot change elements unlike, say, lists or numpy arrays. I.e. `f=[5,6,7]; f[1]=9; print(f)` yields `[5,9,7]` because we were able to modify the element [1].

But `f=5,6,7; f[1]=9` leads to an error
`Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: 'tuple' object does not support item assignment`
because tuples are not mutable. However, if the specific object inside a tuple is mutable, then you can modify it in-place:

```
f=5,[6,7],8
f[1].append(3)
f # yields (5, [6, 7, 3], 8)
#another example

f=[5,6],['foo','bar',7],9
f[0].append(2)
f[1].append(2)
f #yields ([5, 6, 2], ['foo', 'bar', 7, 2], 9)
f[1].pop()

f #yields ([5, 6, 2], ['foo', 'bar', 7], 9)
f[0].pop(-1)
f # yields ([5, 6], ['foo', 'bar', 7], 9)
```

```
([5, 6], ['foo', 'bar', 7], 9)
```

You can concatenate tuples together to make longer tuples: e.g. `a=5,6,7; b='foo',7,'bar'; c=a+b` will yield the following for c `(5, 6, 7, 'foo', 7, 'bar')`. Also, multiple a tuple by an integer `n` will make `n` copies of the tuple: e.g. `('foo','bar')*4` will equal `('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')`.

Unpacking Tuples

If you try to *assign* to a tuple-like expression of variables, Python will attempt to *unpack* the value on the righthand side of the equals sign. E.g.

```
thetuple=5,6,7
a,b,c=thetuple
b #returned as 6
```

```
6
```

```
#or
thetuple =4,(5,6),7
a,(b,c),d=thetuple
b,c #returns (5,6)
```

```
(5, 6)
```

You can also swap variable names, e.g.:

```
>>>a,b='foo',3
>>>print(a) #returns 'foo'
>>>print(b) #returns 3

>>>b,a=a,b
>>>print(a) #returns 3
>>>print(b) #returns 'foo'
```

Lists

We briefly discussed lists in the "Basic" section of Python, where we described a list as "a container of things that are organized in order from first to last." Now we will give a more detailed description.

In contrast to tuples, lists are variable-length and their contents can be modified in place.

Creating Lists

Lists are typically defined by enclosing some things by square brackets `[ ]`. You can also define a list by the `list` type function: e.g. to convert a tuple to a list use the `list` keyword.

Below are examples of lists -- a*list and b*list -- with some operations.

```
>>>a_list = [2, 3, 7, None]
>>>tup = ('foo', 'bar', 'baz')
>>>b_list = list(tup)
>>>b_list #prints ['foo','bar','baz']
>>>b_list[1] = 'peekaboo'
>>>b_list # prints ['foo','peekaboo','baz']
```

Modifying Lists

You can add and remove elements with the property `append` and `insert`. E.g. `b_list.append('dwarf')` gives you `b_list=['foo','peekaboo','baz','dwarf']`. You can insert an element at a specific location in a list as follows:

```
>>>b_list.insert(1,'red')
>>>b_list
>>> ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
>>>b_list.insert(0,'blue')
>>>['blue', 'foo', 'red', 'peekaboo', 'baz', 'dwarf']
```

The inverse of `insert` is `pop`, whcih removes and returns an element at an index. E.g. `b_list.pop(2)` = `` `peekaboo' `` and then `b_list=['blue', 'foo', 'peekaboo', 'baz', 'dwarf']`. Called without an index, `pop` removes the last element. E.g. `b_list.pop()` removes `` `dwarf` ``.

Elements can also be removed using `remove`, which locates the first matching value and removes it:

```
>>> c_list=['foo','bar','foo','foo']
>>> c_list.remove('foo')
>>> c_list
['bar', 'foo', 'foo']
```

You can also add multiple elements to a list at a time using the `extend` property:

```
#simple example
>>> d_list=['foo','bar']
>>> d_list.extend([5,6,'red','dwarf'])
>>> d_list
    ['foo', 'bar', 5, 6, 'red', 'dwarf']
#slightly more complex
>>> e_list=['foo','bar']
>>> e_list.extend([(7,8),(9,10)])
>>> e_list
    ['foo', 'bar', (7, 8), (9, 10)]
```

Just like tuples, you can concatenate lists. E.g. `b_list=[5,6,7,8]; b2_list=[0.55,66,7e-6,88]`, so `b_list+b2_list = [5, 6, 7, 8, 0.55, 66, 7e-6, 88]`.

Note that `extend` is faster than concatenation.

You can sort a list in place (without creating a new object) by calling its `sort` function:

```
>>>anotherlist = [7,2,5,1,3]
>>>anotherlist.sort()
   #[1,2,3,5,7]
```

You can also unpack variables by iterating over sequences of tuples or lists:

```
seq=[(1,2,3),('foo','bar','yo'),(4,5,6)]
for a,b,c in seq:
  print('a={0},b={1},c={2}'.format(a,b,c))
```

Arrays

While native Python can do arrays, those in ***Numpy*** are far superior. Arrays are container structures for manipulating vectors, matrices, or even higher-order tensors. We will reserve a more detailed discussion of arrays for the ***Numpy*** module later. For now we discuss core concepts of arrays.

To instantiate arrays, we first need to import the *numpy* package. The canonical way to do this is as follows:

```
import numpy as np
```

Arrays can then be constructed from lists by the numpy function *array*.

E.g.:

```
v=np.array([1,2,3.])
#A=np.array([1,2,3],[4,5,6])
A=np.array([[1,2,3.],[4,5,6]])
print(v)
print(A)
```

```
[1. 2. 3.]
[[1. 2. 3.]
 [4. 5. 6.]]
```

These numpy arrays hold different vectors, matrices, etc. To access an element of a vector we need to define the *index*; for a matrix, we need to define two indexes. E.g.

```
v[2] #returns 3.0
```

```
3.0
```

```
A[1,2] #returns 6
```

```
6.0
```

Superficially, arrays and lists look like the same thing. However, there are some differences. Arrays only store elements of the same numeric type. Unlike lists, there is no `append` method for arrays. However, you can construct arrays by stacking smaller size arrays together.

One key difference is that the operations `+,-,*, and /` are all element-wise. E.g.

```
a=np.array([1,2,3,4]) #an array
b=[1,2,3,4] #a list
print(a*2)
#returns array([2, 4, 6, 8])
print(b*2)
#returns [1, 2, 3, 4, 1, 2, 3, 4]
```

```
[2 4 6 8]
[1, 2, 3, 4, 1, 2, 3, 4]
```

The function `dot` and `@` are used for scalar product and corresponding matrix operations. Vector slices can be used to modify the existing array.

The number of elements in a vector or the number of rows in a matrix can be obtained by the function `len`. This is useful for both lists and arrays. However, a numpy array has a special `shape` attribute that gives the full dimensionality of the array.

E.g.

```
c=np.array([[1,2,3],[4,5,6]])
len(c) #returns 2
```

```
2
```

```
c.shape #returns (2,3)
```

```
(2, 3)
```

We will discuss arrays in far more detail in the later ***Numpy*** module.

Dictionaries

For lists (and arrays), Python uses numbers to define indices. E.g. things[1] means "the 2nd element of the list `things"'. A *dictionary* goes a step further, you can use *anything* to index a list: names, numbers, whatever.

Here's an example of how it works:

```
stuff = {'name': 'Beavis', 'age': 15, 'weight': 145}
```

stuff is a dictionary. What is the 'name' in the dictionary? *Beavis*. Age? *15*. Weight? *145*.

You can pull up this information as follows:

```
stuff = {'name': 'Beavis', 'age': 15, 'weight': 145}
print(stuff['name'])
#Beavis
print(stuff['age'])
#15
print(stuff['weight'])
#145
```

```
Beavis
15
145
```

Like with lists indexed by numbers, to grab the *dictionary* value for a variable, you use the brackets '[ ]'.

You can also add elements to a *dictionary*:

```
stuff['city']="Highland, Texas"
print(stuff['city'])
#Highland, Texas
```

```
Highland, Texas
```

You can also put things into the dictionary with strings.

```
stuff[1]="Settle Down Beavis"
stuff[2]="I dreamed I was at school last night"
stuff
```

```
{'name': 'Beavis',
 'age': 15,
 'weight': 145,
 'city': 'Highland, Texas',
 1: 'Settle Down Beavis',
 2: 'I dreamed I was at school last night'}
```

```
#Now ...


print(stuff)
{'name': 'Beavis', 'age': 15, 'weight': 145, 'city': 'Highland, Texas', 1: 'Settle Down Beavis', 2: 'I dreamed I was at school
last night'}
```

```
{'name': 'Beavis', 'age': 15, 'weight': 145, 'city': 'Highland, Texas', 1: 'Settle Down Beavis', 2: 'I dreamed I was at school
last night'}
```

```
{'name': 'Beavis',
 'age': 15,
 'weight': 145,
 'city': 'Highland, Texas',
 1: 'Settle Down Beavis',
 2: 'I dreamed I was at school last night'}
```

You can also delete elements of a dictionary:

```

del stuff['weight']
del stuff[2]
print(stuff)
{'name': 'Beavis', 'age': 15, 'city': 'Highland, Texas', 1: 'Settle Down Beavis'}
```

```
{'name': 'Beavis', 'age': 15, 'city': 'Highland, Texas', 1: 'Settle Down Beavis'}
```

```
{'name': 'Beavis',
 'age': 15,
 'city': 'Highland, Texas',
 1: 'Settle Down Beavis'}
```

Here's a more detailed example:

```
# create a mapping of state to abbreviation
states = {
    'Oregon': 'OR',
    'Florida': 'FL',
    'California': 'CA',
    'New York': 'NY',
    'Michigan': 'MI'
}

# create a basic set of states and some cities in them
cities = {
    'CA': 'San Francisco',
    'MI': 'Detroit',
    'FL': 'Jacksonville'
}

# add some more cities
cities['NY'] = 'New York'
cities['OR'] = 'Portland'

# print out some cities
print('-' * 10)
print("NY State has: ", cities['NY'])
print("OR State has: ", cities['OR'])

# print some states
print('-' * 10)
print("Michigan's abbreviation is: ", states['Michigan'])
print("Florida's abbreviation is: ", states['Florida'])

# do it by using the state then cities dict
print('-' * 10)
print("Michigan has: ", cities[states['Michigan']])
print("Florida has: ", cities[states['Florida']])

# print every state abbreviation
print('-' * 10)
for state, abbrev in states.items():
    print("%s is abbreviated %s" % (state, abbrev))

# print every city in state
print('-' * 10)
for abbrev, city in cities.items():
    print("%s has the city %s" % (abbrev, city))

# now do both at the same time
print('-' * 10)
for state, abbrev in states.items():
    print("%s state is abbreviated %s and has city %s" % (
        state, abbrev, cities[abbrev]))

print('-' * 10)
# safely get a abbreviation by state that might not be there
state = states.get('Texas')
#print("texas?",state)
if not state:
    print("Sorry, no Texas.  We have messed with Texas.")

# get a city with a default value
city = cities.get('TX', 'Does Not Exist')
print("The city for the state 'TX' is: %s" % city)
```

```
----------
NY State has:  New York
OR State has:  Portland
----------
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
----------
Michigan has:  Detroit
Florida has:  Jacksonville
----------
Oregon is abbreviated OR
Florida is abbreviated FL
California is abbreviated CA
New York is abbreviated NY
Michigan is abbreviated MI
----------
CA has the city San Francisco
MI has the city Detroit
FL has the city Jacksonville
NY has the city New York
OR has the city Portland
----------
Oregon state is abbreviated OR and has city Portland
Florida state is abbreviated FL and has city Jacksonville
California state is abbreviated CA and has city San Francisco
New York state is abbreviated NY and has city New York
Michigan state is abbreviated MI and has city Detroit
----------
Sorry, no Texas.  We have messed with Texas.
The city for the state 'TX' is: Does Not Exist
```

This results in the following the following:

```
----------
NY State has:  New York
OR State has:  Portland
----------
Michigan's abbreviation is:  MI
Florida's abbreviation is:  FL
----------
Michigan has:  Detroit
Florida has:  Jacksonville
----------
Oregon is abbreviated OR
Florida is abbreviated FL
California is abbreviated CA
New York is abbreviated NY
Michigan is abbreviated MI
----------
CA has the city San Francisco
MI has the city Detroit
FL has the city Jacksonville
NY has the city New York
OR has the city Portland
----------
Oregon state is abbreviated OR and has city Portland
Florida state is abbreviated FL and has city Jacksonville
California state is abbreviated CA and has city San Francisco
New York state is abbreviated NY and has city New York
Michigan state is abbreviated MI and has city Detroit
----------
Sorry, no Texas.  We have messed with Texas.
The city for the state 'TX' is: Does Not Exist
```

Checking Data Structure Types and Type Conversions

Say you have some variable in your Python code (or, more likely, someone else's) and you want to figure out what it is. Is it a function? A tuple? A Numpy Array? How do you check? Simple: `type([thing you are wanting to check])`.

E.g.

```
a=np.array([1,2,3,4,5])
b=[1,2,3,4,5]
c=1,2,3,4,5
d={'0':'Enterprise','1': 'Defiant','2':'Excelsior'}
```

```
type(a) #<class 'numpy.ndarray'>
```

```
numpy.ndarray
```

```
type(b) #<class 'list'>
```

```
list
```

```
type(c) #<class 'tuple'>
```

```
tuple
```

```
type(d) #<class 'dict'>
```

```
dict
```

What if you are trying to figure out the dimensions of the the variable? We mentioned it before, but again the `len` function is your first stop. E.g. `len(a), len(b), len(c)` return `(5, 5, 5)` (i.e. each of them have length 5). `len(d)` returns `3`.

Also to reiterate, there are additional capabilties (or complications for the dimensionality of Numpy arrays, as they can be n-dimensional. Specifically you can use the `shape` attribute to check dimensionality. We will talk more about arrays in the ***Numpy*** section.

```

```