Math 246 Unit 2, part A: Python variables, including lists, arrays and tuples ...

(and the packages Numpy and Pylab)

Professor Brenton LeMesurier

Revised on September 22, 2015, adding notes on negative indices in lists

Note: although you could try the Python commands in this file by using the notebook version, I encourage you to use Spyder to create a file containing these commands, and then run them. This part A is preparation for part B, where we start creating and running Python files.

Python numerical variables

The first step beyond using Python merely as a calculator is storing value in variables, for reuse later in more elaborate calculations. For example, to find both roots of a quadratic equation

$$ax^2 + bx + c = 0$$

we want the values of each coefficient and are going to use each of them twice, which we might want to do without typing in each coefficient twice over.

Example

We can solve the specific equation

$$2x^2 - 8x + 6 = 0$$

using the quadratic formula. But first we need to get the square root function:

In [99]:

```
from math import sqrt
```

Then the rest looks almost like normal mathematical notation:

In [100]:

```
a = 2
b = -10
c = 8
root0 = (-b - sqrt(b**2 - 4 * a * c))/(2 * a)
root1 = (-b + sqrt(b**2 - 4 * a * c))/(2 * a)
```

(Aside: why did I number the roots 0 and 1 instead of 1 and 2? The answer is coming up soon.)

Where are the results? They have been stored in variables rather than printed out, so to see them, use the `print` function:

In [101]:

```
print('The smaller root is', root0, 'and the larger root is', root1)
```

```
The smaller root is 1.0 and the larger root is 4.0
```

Lists

Python has several ways of grouping together information into one variable. We first look at *lists*, which can collect all kinds of information together:

In [102]:

```
coefficients = [2, -10, 8]
name = ["LeMesurier", "Brenton"]
phone = [9535917]
```

Lists can be combined by "addition":

In [103]:

```
name + phone
```

Out[103]:

```
['LeMesurier', 'Brenton', 9535917]
```

Individual entries ("elements") can be extracted from lists; note that **Python always counts from 0**:

In [104]:

```
coefficients[0], coefficients[1], coefficients[2]
```

Out[104]:

```
(2, -10, 8)
```

In [105]:

```
firstname = name[1]
print(firstname)
```

```
Brenton
```

and we can modify list elements this way too:

In [106]:

```
coefficients[2] = 12
print('Now the coefficients are', coefficients)
```

```
Now the coefficients are [2, -10, 12]
```

We can use the list of coefficients to specify the quadratic, and store both roots in a new list.

But let's shorten the name first, by making "q" a synonym for "coefficients":

In [107]:

```
q = coefficients
print('The list of coefficients is', q)
roots = [(-q[1] - sqrt(q[1]**2 - 4 * q[0] * q[2]))/(2 * q[0]),
         (-q[1] + sqrt(q[1]**2 - 4 * q[0] * q[2]))/(2 * q[0])]
print('The list of roots is', roots)
print('The individual roots are', roots[0], 'and', roots[1])
```

```
The list of coefficients is [2, -10, 12]
The list of roots is [2.0, 3.0]
The individual roots are 2.0 and 3.0
```

See now why I enumerated the roots from 0 previously?

For readability, you might want to "unpack" the coefficients and then use the more familiar formulas above:

In [108]:

```
a = q[0]
b = q[1]
c = q[2]
roots = [(-b - sqrt(b**2 - 4 * a * c))/(2 * a),
         (-b + sqrt(b**2 - 4 * a * c))/(2 * a)]
print('The list of roots is again', roots)
```

```
The list of roots is again [2.0, 3.0]
```

Looking at the end of a list, with negative indices

Python allows you to count backwards from the end of a list, by using negative indices:

- index -1 refers to the last element
- index -k refers to the element k from the end.

For example

In [109]:

```
digits = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print('The last digit is', digits[-1])
print('The third to last digit is', digits[-3])
```

```
The last digit is 9
The third to last digit is 7
```

This also works with the arrays and tuples introduced below.

Numpy and arrays: for vectors, matrices, and beyond

Many mathematical calculations involve vectors, matrices and other arays of numbers. At first glance, Python lists look like vectors, but as seen above, "addition" of lists does not do what you want with vectors.

Thus we need numerical **arrays**. These are not a part of the core Python language, but there is a method to add features to Python from **modules** and **packages** (which are colections of modules). We can get arrays along with many other useful mathematical tools from the module `numpy`, by using the command:

In [110]:

```
from numpy import *
```

Notes

1. Actually Python does have something called an *array*, but we never use that version, and I advise you to avoid it: Numpy arrays are far better for what we want to do!
2. There is another package `pylab`, which contains most of `numpy` plus some stuff for graphics. It is intended to reproduce a Matlab-like environment, especially when used in Spyder, which is deliberately Matlab-like. So you could instead use "`from pylab import *`", and that will sometimes be more convenient. However, when you search for documentation, you will find it by searching for `numpy`, not for `pylab`.

Creating arrays (from lists and otherwise)

Arrays are so similar to lists that one way to create an array is to convert a list:

In [111]:

```
list0 = [1, 2, 3]
list1 = [2, 4, 8]
array0 = array(list0)
array1 = array(list1)
```

We can skip the intermediate step of creating lists and instead create arrays directly:

In [112]:

```
array0 = array([1, 2, 3])
array1 = array([2, 4, 8])
```

Printing makes these seem very similar ...

In [113]:

```
print('list0 =', list0)
print('array0 =', array0)
```

```
list0 = [1, 2, 3]
array0 = [1 2 3]
```

... and we can extract elements in the same way.

In [114]:

```
print('The first element of list0 is', list0[0])
print('The last element of array1 is', array1[2])
```

```
The first element of list0 is 1
The last element of array1 is 8
```

However, displaying by simply typing the names describes them more carefully, with a description that could be used to create them:

In [115]:

```
list0
```

Out[115]:

```
[1, 2, 3]
```

In [116]:

```
array0
```

Out[116]:

```
array([1, 2, 3])
```

Addition and other arithmetic also reveal some important differences:

In [117]:

```
print(list0 + list1)
print(array0 + array1)
print(2 * list0)
print(2 * array0)
```

```
[1, 2, 3, 2, 4, 8]
[ 3  6 11]
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

(Note what multiplication does to lists!)

Matrices; sometimes considered as "arrays of arrays of numbers"

A list can have other lists as its elements, and likewise an array can be described as having other arrays as its elements, so that a matrix can be described as a succession of rows. First, a list of lists can be created:

In [118]:

```
listoflists = [list0, list1]
print(listoflists)
```

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

Then this can be converted to a two dimensional array:

In [119]:

```
matrix = array(listoflists)
print(matrix)
```

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

We can also combine arrays into new arrays directly:

In [120]:

```
anothermatrix = array([array1, array0])
anothermatrix
```

Out[120]:

```
array([[2, 4, 8],
       [1, 2, 3]])
```

Nte that we must use the notation `array([...])` to do this; without the function `array()` we would get a *list of arrays*, which is a different animal, and much less fun for doing mathematics with:

In [121]:

```
listofarrays = [array1, array0]
listofarrays
```

Out[121]:

```
[array([2, 4, 8]), array([1, 2, 3])]
```

Referring to array elements with double indices, or with successive single indices

The elements of a multi-dimensional array can be referred to with double indices:

In [122]:

```
matrix[1,2]
```

Out[122]:

```
8
```

but you can also use a single index to extract an "element" that is a row:

In [123]:

```
matrix[1]
```

Out[123]:

```
array([2, 4, 8])
```

and you can use indices successively, to specify first a row and then an element of that row:

In [124]:

```
matrix[1][2]
```

Out[124]:

```
8
```

This ability to manipulate rows of a matrix can be useful for linear algebra. For example, in a row reduction we might want to subtract twice the first row from the second row, and this is done with:

In [125]:

```
print('Before the row operation, the matrix is:')
print(matrix)
matrix[1] = matrix[1] - 2. * matrix[0]
print('After the row operation, it is:')
print(matrix)
```

```
Before the row operation, the matrix is:
[[1 2 3]
 [2 4 8]]
After the row operation, it is:
[[1 2 3]
 [0 0 2]]
```

Note well the effect of Python indexing starting at zero: the indices used with a vector or matrix are all one less than you might expect.

Higher dimensional arrays

Arrays with three or more indices are possible, though we will not see much of them in this course:

In [126]:

```
arrays_now_in_3D = array([matrix, anothermatrix])
arrays_now_in_3D
```

Out[126]:

```
array([[[1, 2, 3],
        [0, 0, 2]],

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

Python tuples

One other useful kind of Python collection is a **tuple**, which is a lot like a list except that it is **immutable**: you cannot change individual elements. Tuples are denoted by surrounding the elements with parentheses "(...)" in place of the brackets "[...]" used with lists:

In [127]:

```
qtuple = (2, -10, 8)
```

In [128]:

```
qtuple
```

Out[128]:

```
(2, -10, 8)
```

In [129]:

```
qtuple[2]
```

Out[129]:

```
8
```

So far, so good, and as with lists above, but this fails:

In [130]:

```
qtuple[2] = 12
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-130-0933764609d7> in <module>()
----> 1 qtuple[2] = 12

TypeError: 'tuple' object does not support item assignment
```

Actually, we have seen tuples before without the name being mentioned: when a list of expressions is put on one line separated by commas, the result is a tuple. This is because when creating a tuple, the surrounding parentheses can usually be omitted:

In [131]:

```
nameandphoneandoffice = "LeMesurier", "Brenton", "953-5917", 200
nameandphoneandoffice
```

Out[131]:

```
('LeMesurier', 'Brenton', '953-5917', 200)
```

For now, we have far more use for arrays than for either lists or tuples, but tuples will come up when we start defining our own Python functions (a.k.a. "procedures").