REPETITION

By this time, we've had a quite a thorough introduction to many of the niceties of high-end programming languages that have been careful designed to simplify the programming process. For example, we've learned that Python has quite a bit of built-in functionality that minimizes the amount of typing and maximizes the reusability of our code: e.g., programs and modules.

Thus we would hope that most of you were frustrated by the repetition involved in the simple graphics programming done in the prior chapter: having to create, format, and draw each point individually was most definitely tedious ... and perhaps worrisome. What if you had had to illustrate 10 points? 100 points? 10,000 points? Would you have had to type three lines of code for each point in order to produce it on the screen? That does not seem to be very efficient!

Luckily, Python actually makes it very easy to do repetition so that we don't have to worry so much about getting repetitive strain syndrome from all of that typing!

Doing repetition in Python is very simple. For example, if you wanted to print out the digits from 0 to 9, all you have to do is type the following at the prompt:

```
>>> for i in range(10):
        print i
```

This is a new type of statement in Python, the **`for`-statement**. `for` is a Python reserve word and the `for`-statement one of several types of *loop statements*, or simply, ***loops***. Loops are a means for repeating something a fixed number of times. The basic syntax of a `for`-loop in Python is:

```
for <variable> in <sequence>:
    <do something>
    <do something>
    ...
```

The loop begins with the reserve word `for`, which is followed by a `<variable>` that serves as an ***index*** for the loop. The loop index variable is followed by the reserve word in and a `<sequence>` followed by a colon (`:`). This line sets up the number of times that the repetition will be repeated, or ***iterated***. What follows is a set of statements, indented that are called a ***block*** that forms the ***body*** of the loop (stuff that is repeated).

When the loop is executed, the `<variable>` is assigned successive values in the `<sequence>` and for each of those values, the statements in the body of the loop are executed. A `<sequence>` in Python is a list of values. ***Lists*** are very important in Python and we will discuss them more in the next section. For now, notice that we have used the command `range(10)` to specify the sequence our loop example.

Do this: To see what this command does, start IDLE and enter the following at the prompt:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The result of entering the `range(10)` is a list of ten numbers, ranging from 0 to 9. Notice how the `range` function returns a sequence of values starting from 0 and going all the way up to, but not including 10. Thus the variable `i` in the loop

```
>>> for i in range(10):
        print i
```

will take on the values of 0 through 9 and for each of those values it will execute the `print` statement. Try it the loop in IDLE. What happens when you change the value passed to the range function? Give it a try!

IDLE TIP: You can stop a program at any time by pressing the CTRL-C key; that is, press the CTRL key and C key at the same time.

There are other ways to do repetition in Python and we will explore them at a later time. For now, let's talk a bit more about sequences.

LISTS

We've actually already played with lists quite a bit and they are more familiar to you then you know! Every piece of "data" that we've manipulated so far in Pylab has actually been a sequence, or list!

Do this: Open up and look closely at the `DrugData.py` file in IDLE. You should see the following:

```
Year = [1979,1980,1981,1982,1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,1994,1995,1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007]
AnyDrug = [65.1,65.4,65.6,64.4,62.9,61.6,60.6,57.6,56.6,53.9,50.9,47.9,44.1,40.7,42.9,45.6,48.4,50.8,54.3,54.1,54.7,54.0,53.9,53.0,51.1,51.1,50.4,48.2,46.8]
Marijuana = [60.4,60.3,59.5,58.7,57.0,54.9,54.2,50.9,50.2,47.2,43.7,40.7,36.7,32.6,35.3,38.2,41.7,44.9,49.6,49.1,49.7,48.8,49.0,47.8,46.1,45.7,44.8,42.3,41.8]
Hallucinogens = [14.1,13.3,13.3,12.5,11.9,10.7,10.3,9.7,10.3,8.9,9.4,9.4,9.6,9.2,10.9,11.4,12.7,14.0,15.1,14.1,13.7,13.0,14.7,12.0,10.6,9.7,8.8,8.3,8.4]
Cocaine = [15.4,15.7,16.5,16.0,16.2,16.1,17.3,16.9,15.2,12.1,10.3,9.4,7.8,6.1,6.1,5.9,6.0,7.1,8.7,9.3,9.8,8.6,8.2,7.8,7.7,8.1,8.0,8.5,7.8]
Heroin = [1.1,1.1,1.1,1.2,1.2,1.3,1.2,1.1,1.2,1.1,1.3,1.3,0.9,1.2,1.1,1.2,1.6,1.8,2.1,2.0,2.0,2.4,1.8,1.7,1.5,1.5,1.5,1.4,1.5]
Cigarettes = [74.0,71.0,71.0,70.1,70.6,69.7,68.8,67.6,67.2,66.4,65.7,64.4,63.1,61.8,61.9,62.0,64.2,63.5,65.4,65.3,64.6,62.5,61.0,57.2,53.7,52.8,50.0,47.1,46.2]
Alcohol = [93.0,93.2,92.6,92.8,92.6,92.6,92.2,91.3,92.2,92.0,90.7,89.5,88.0,87.5,80.0,80.4,80.7,79.2,81.7,81.4,80.0,80.3,79.7,78.4,76.6,76.8,75.1,72.7,72.2]
```

As you can see, each piece of data that we have plotted in the `DrugData.py` file is in fact a variable to which has been assigned a sequence of values. Each of these sequences is a ***list***. Lists are a very useful way of collecting a bunch of related information so that it can be easily stored and manipulated. Python provides a whole hosts of useful operations for manipulating lists. In Python, *a list is any sequence of "things"*. The "things" can be anything: numbers, letters, strings, Points, Rectangles, figures, etc. The simplest type of list you can have is an ***empty list***:

```
>>> []
[]
```

or

```
>>> L = []
>>> print L
[]
```

An empty list does not contain anything. Here are some lists that do contain information:

```
>>> N = [2, 43, 6, 27]
>>> FamousNumbers = [3.1415, 2.718, 42]
>>> Cities = ['Philadelphia', 'Hicksville', 'Troy', 'Austin']
>>> MyPoints = [Point(100, 42), Point(75, 64), Point(83, 310)]
>>> MyCar = ['Mazda Protégé', 1999, 'red']
```

As you can see, a list can be any collection of things, even those that have different data types. Python provides us with several functions we can use to manipulate these types of sequences. Below are some examples using the lists defined above:

```
>>> len(N)
4

>>> N + FamousNumbers
[2, 43, 6, 27, 3.1315, 2.718, 42]

>>> Cities[0]
'Philadelphia'

>>> N[1:3]
[43, 6]

>>> 'Troy' in Cities
True

>>> 'Cairo' in Cities
False
```

The function `len` takes a list and returns its length, or the number of objects in the list. An empty list has zero objects in it. You can also access individual elements in a list using the ***indexing*** operation (as in `Cities[0]`). The first element in a list has index 0 and the last has element in a list of `n` elements will have an index of `n-1`. You can also use the index operation to get a slice, or ***sublist***, using the index operation (as in `N[1:3]`, which refers to the sublist containing elements 1 through 2 (one less than 3)). You can ***concatenate*** two lists using the `'+'` operator, and check whether or not an object is in the list using the `in` reserve word.

Besides the above operations, Python also provides some functionality for more complicated list manipulations. Two useful ones are **sort** and **reverse**:

```
>>> N = [2, 43, 6, 27]
>>> N.sort() # notice that lists are a type of OBJECT!!!
```

```
[2, 6, 27, 43]

>>> N = [2, 43, 6, 27]
>>> N.reverse()
[27, 6, 43, 2]

>>> N = [2, 43, 6, 27]
>>> N.append(5)
[2, 43, 6, 27, 5]
```

`sort` rearranges elements in the list is ascending order and `reverse` reverses the order of elements in the list. Another useful method, `append`, adds a element to the end of the list.

Remember that all lists are sequences and thus can be used to perform repetitions. For example, the following:

```
>>> for point in MyPoints:
        c = Circle(point, 5)
        c.setFill('red')
        c.draw(win)
```

would draw a red circle centered at each point in the `MyPoints` list in the `GraphWin`, `win`.

Strings

Strings are also sequences. That is, the string:

```
ABC = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

is a sequence of 26 letters. You could write a loop that runs through each individual letter and prints it out:

```
>>> for letter in ABC:
        print letter
```

Any of the methods used to manipulate sequences can be applied to strings; for example, we have already concatenated strings using the `'+'` operator. Python also provides special methods for just manipulating strings. The most useful is a function that converts a string into a list. Say we have a string containing the following sequence:

```
>>> sentence = 'Would you have any Grey Poupon'
```

We could convert the sequence into individual words using the split operation:

```
>>> sentence.split()
['Would', 'you', 'have', 'any', 'Grey', 'Poupon']
```

From now on we will be using lists in many aspects of our Python programming.

THEORY INTO PRACTICE: LABELING IN PYLAB

Now that we know about lists, we can start to use some of the more advanced features of the Pylab interface. We are not going to get too complicated here, but as you will see lists are central to the use of the Pylab interface. As stated earlier, most of the manipulation of data we have done in Pylab has involved the use of lists: every sequence of data imported from our `DrugData.py` file has been a list. Thus, the actual syntax of the `plot` command, for example is:

```
plot(<x-sequence>)
```

or

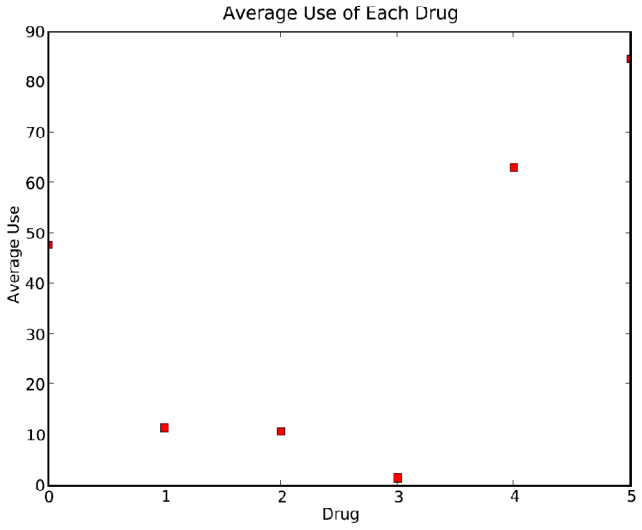
```
plot(<x-sequence>, <y-sequence>)
```

where the x- and y-data are actually lists of `int` or `float` values. Similarly, the actual syntax of the `mean` command is

mean(<sequence>)

and the function calculates the mean of a list of `int` or `float` values.

Besides providing the information to be plotted, lists can also help us improve the quality of our Pylab visualizations. Remember this figure from Exercise 2, #5?



This visualization would be vastly improved if we could accurately label the ticks on the x-axis.

Do This: Modify your code for the exercise so that it includes the following list:

```
Labels = ['Marijuana', 'Hallucinogens', 'Cocaine', 'Heroin',
'Cigarettes', 'Alcohol']
```

Note: the labels should be in the order that you calculated the averages for the ticks to make sense!!!

Then use the command **xticks** to place your labels on the ticks along the x-axis as follows:

```
xticks(arange(6), Labels)
```

The **xticks** command takes two parameters. The first **arange(<num-ticks>)** is a function that tells the **xticks** commands how many ticks are to be labeled. The second is a list of labels.

Other plotting functions can take sequences of strings to label portions of the figure. When you come across the keyword argument **label** as an option that can be manipulated for a figure, chances are a portion of the figure can be labeled using a list of strings.

PYTHON REVIEW

Will be filled in later.

PYLAB REVIEW

Will be filled in later.

EXERCISES

1. Write a function that takes an array of quantitative data and the number of classes into which you would like to subdivide the data and then returns the bin size (or range of data that falls into each class). Hint:

$$Bin\ Size = \frac{Maximum - Minimum}{Number\ of\ Classes}$$

   Hint: how can you find the maximum or minimum values using just your list manipulation functions? Test your function using one of the Candidate lists (**Bush**, **Gore**, **Nader**) in the **BallotData.py** file. Print the IDLE window that shows the output of your function.

2. Write a function that takes an list of frequencies and returns a list of relative frequencies. Hint:

$$Relative\ Frequency = \frac{Frequency}{Total\ Number\ of\ Observations} \times 100$$

   Test using either the **TechnologyFrequency**, **CandidateFrequency**, or **CountyFrequency** data in the **BallotData.py** file. Print the IDLE window that shows the output of your function.

3. Write a function that takes a bin size and a relative frequency value between 0 and 100 and draws a blue rectangle with a width proportional to bin size and a height proportional to the relative frequency. Test your function with the following values: 1) bin size = 5, relative frequency = 15; 2) bin size = 30, relative frequency = 78; 3) bin size = 12, relative frequency = 37. Print the display produced for each test.

4. Choose one of the three candidates in the `BallotData.py` file (`Bush`, `Nader`, or `Gore`). Use the hist Pylab command to generate a histogram of the data. As always, label your axes and title your graph. Save and print the graph. What does this visualization tell you about the number of votes received per county for this candidate? Consider both the shape and pattern of the data. For example, is the histogram symmetrical? Where is the mean? The mode? Are there any outliers? What do such observations tell you about the voting?

   By default, Pylab sets the number of bins to 10. You can change the number of bins (or classes) by using the `bins` keyword argument (e.g., `hist(<data-sequence>, bins = <num-bins>)`). Produce another histogram using the same candidate data but set the number of bins equal to 30. Save and print the figure and then again consider what this visualization tells you about the number of votes received per county for this candidate (see above). How is the information conveyed by this figure different from that conveyed by the one with the large bin-size? Repeat the exercise, but this time set the bin size to 100.

5. Pylab can also be used to produce other types of univariate visualizations. Use the command `pie(<data-sequence>, labels = <label-sequence>)` to produce a pie chart of the Technology data in the `BallotData.py` file. Title the plot appropriately and then save and print your graphic. What insight does this visualization give you into the technology used in the 2000 Florida presidential election?

6. Write a program that generates three figures. In one figure, the program should produce a histogram of the `CountyFrequency` data from the `BallotData.py` file. In another figure, it should use a box and whisker plot to illustrate the same data. For the box and whisker plot, use the `sym` keyword argument so that the outliers are visible (i.e., `boxplot(<data-sequence>, sym = '+')`). In the final figure, the program should produce a pie-chart of the same data. Each figure should be labeled appropriately (axes, ticks, and title where relevant). Save all figures using the `savefig` method. Compare the three visualizations. What insight, if any, do they give into the distribution of votes among counties? Are any of the visualizations more or less effective than others? Why or why not?