Python Practice 4: Using `for`-loops and lists in Python – *Time series analysis*

In this class, we will practice using Python `for`-loops and lists. As a practical exercise, we will do some processing of time-series data. Along with the iterative numerical techniques we looked at last week, time series analysis has many applications in Science and Engineering.

Time series data are measurements that change with time. An example of a time series is a piece of music, which is stored in an MP3 player as a sequence of numbers representing the sound intensity at evenly spaced times. In Science and Engineering applications, we often want to obtain information from a time series by filtering or other processing steps. In today's lab class, we will write a Python program to recover a signal from a time series that has been corrupted with noise.

1 Python Lists and `for`-loops

Recall from class, that Python lists are ordered collections of items, that can be of mixed types, e.g. as in `["some string", 3.55, "this", 2]` which contains both strings and numbers. The *empty list* (i.e. which contains no items) is `[]`. List items can be accessed by their *position* in the list, with index values from 0 to $(n-1)$ for a list of length n. As the example on the right shows, a list can be changed, with values at existing positions being overwritten, or with a list being made shorter or longer, as in the example (where `append` is used to 'grow' a list).

```
>>> x = ['aa', 'bb', 'cc']
>>> x[1] = 33
>>> x
['aa', 33, 'cc']
>>> x.append('dd')
>>> x
['aa', 33, 'cc', 'dd']
```

Often, `for`-loops are used together with a list in the following pattern: This loop will repeat as many times as there are items in `LIST`, with `VAR` being assigned each member of `LIST` in turn as its value.

```
for VAR in LIST:
    CODE-BLOCK
```

This kind of *iteration* is fine for many purposes, as in the example on the right, where we just want to get hold of each of the values, so that we can print it out.

```
>>> values = [875, 23, 451]
>>> for val in values:
        print('-->', val)
--> 875
--> 23
--> 451
```

However, sometimes we need to access the *positions* in the list by their *index*, for which we must use the `range` function. The `range` function successively generates the index values, as illustrated in the example on the right, where the print statement in the loop prints out both the index and value stored at that position in the list of values.

```
>>> values = [875, 23, 451]
>>> for i in range(len(values)):
        print(i, ':', values[i])
0 : 875
1 : 23
2 : 451
```

An example of when you would need to access the positions in a list by their index is when you want to modify the values stored there. In the example on the right, we need to access the positions by index is so that we can add 1000 to the value there and then store it back into the *same* position.

```
>>> values = [875, 23, 451]
>>> for i in range(len(values)):
        values[i] = values[i] + 1000

>>> values
[1875, 1023, 1451]
```

2 Introductory exercises

Create a file `lab4_sec2.py` for your solutions to this section's exercises, to submit at the end.

2.1 Sum of a list of numbers / Triangular number

Define a function `sum_list`, which takes one argument – a list of numbers – and *which uses a* `for`*-loop* to compute (and return) the sum of these numbers, e.g. so that `sum_list([3,4,5])` should return 12. Define a function *which uses a* `for`*-loop* to compute (and return) *triangular numbers*. For a positive integer n, this is the sum of values from n down to one, i.e. $n + (n-1) + \ldots 2 + 1$.)

2.2 List of squares / triangulars

Define a function `square_list`, which takes a list of numbers as its argument, and which computes (and returns) a list like the first, but with the values squared, e.g. so that `square_list([3,4,5])` returns `[9,16,25]`. NOTE: your definition *should not change the value of the original list*, i.e. if we call `square_list(x)` when `x = [3,4,5]`, then `x` should still be `[3,4,5]` after the call (check this!). (We can copy a list using the `list` function, e.g. `y = list(x)` creates a copy of list `x`.)

Next define a function `triangular_list`, which is similar to `square_list`, except that the numbers in the new list are the triangular numbers of the original values. Your definition should compute the triangular numbers *by calling the function you have defined above* in Sec 2.1.

3 Simple graph plotting using Pylab

`Pylab` is a library of code (a *module*) which provides lots of useful stuff, including *graph plotting*. The basic plotting function takes two arguments: a list of the x-coords of the points to be plotted, and a list of the corresponding y-coords. To plot a graph with (x, y) points (0, 1.2), (1, 2.2) and (2, 1.8), for example, we form a list of the x values `[0,1,2]` and y-values `[1.2, 2.2, 1.8]`.

We could then plot this graph with the code on the right. The `plot` command takes the lists of x and y coord values as stated above, and plots the line on the figure. The final `show` command is needed if your using the python interpreter in a terminal window, to cause the graph to be actually drawn and displayed. If you're using the IPython Console, however, the graph will display even without the `show` command.

```
from pylab import *
Xs = [0, 1, 2]
Ys = [1.2, 2.2, 1.8]
plot(Xs,Ys)
show()
```

By default, we get a continuous line in a random colour, but `plot` takes an *optional* third value, which is a string, that is used to control the line format. In `plot(Xs,Ys,'ro-')`, the format `'ro-'` gives a red (`'r'`) continuous line (`'-'`) with circles (`'o'`) at the data points, but we could instead have blue (`'b'`) or green (`'g'`), asterisks (`'*'`) or crosses (`'x'`), or a line that is dashed (`'--'`), dot-dashed (`'-.'`) or absent. Other functions (`xlabel`,`ylabel`) assign labels to the x/y axes (e.g. `xlabel('time')`), give the figure a title (`title`), or name a file in which the figure is saved (`savefig`) for later use.

We can have several lines on a graph by having several calls to `plot` before calling `show`. A call to `figure` between `plot` calls causes a new figure to be started (so multiple figures are displayed: see example below left).

```
from pylab import *
Xs = [0, 1, 2]
Ys1 = [1.2, 2.2, 1.8]
Ys2 = [1.5, 2.0, 2.6]
plot(Xs,Ys1)
figure()
plot(Xs,Ys2)
show()
```

3.1 Plotting functions using `for` loops

Next, plot a graph of the function $f(x) = x^2 + 20$, for integer values of x in the range 0–20. We can build the lists of x and y values with a `for` loop, with a call such as "`for i in range(N):`" to produce a suitable number of iterations. Build the list of y values by starting with an empty list (`Ys = []`) and appending values to it (e.g. "`Ys.append(V)`"). Having created the lists of x and y values, plot the graph as shown above. Submit your code for this exercise as file `lab4_sec3.py`. As a further exercise (which you might leave until later), plot the function $g(x) = (x/2)^3 - 100$, for the same range of x values, to appear alongside the first line on the same graph.

4 Task: Recovering signal from noise

Start by downloading the lab class code files, and storing them in a sensible location on your U drive. The files are: `noisy_signal.txt`, `pure_signal.txt`, `mains_noise.txt` and `random_noise.txt`.

Our task is to analyse some data that represents a noisy signal, and to extract the original 'pure signal' from it, to which noise has been added. The data is stored in the file `noisy_signal.txt`.

This noisy signal was created as follows. The initial *pure* signal is a 5 Hz sine wave. To this, two different forms of noise were added. Firstly, some 50 Hz sine wave: the frequency of the mains electrical supply in the UK. 50 Hz "mains hum" interference is always a problem when we try to record small amplitude electrical signals (see `http://en.wikipedia.org/wiki/Mains_hum`). Secondly, some entirely random noise was also added to the signal. (The data files for these three separate components — `pure_signal.txt`, `mains_noise.txt` and `random_noise.txt` — have also been provided, in case you want to display them.)

4.1 Loading the data from file

Our first task is to load the noisy signal data from the file `noisy_signal.txt`. If you inspect the file, you will see that each line has two numbers, of which the first is a time point, and the second is an intensity value. The time points run at *millisecond* intervals across a 3 second period. To read from the file, we first use a command such as:

```
infile = open('noisy_signal.txt')
```

This call to the `open` command creates a *filestream object*, for reading from the named file, which we assign to the variable `infile` (which is just an ordinary variable). We can read the lines of text from the file by using a simple `for` loop to *iterate* over the filestream:

```
for line in infile:
    do_something(line)
```

Here `line` is just the loop variable (its name has no special significance), and `infile` is the 'filestream object' created above. As the loop proceeds, the loop var `line` is assigned each successive line of text from the file in turn. It ends when there are no more lines to read.

For example, the code on the right will read and print the first 10 lines of the file, and then exit due to the `break` command (without which, the loop would print all 3000 lines before finishing).

```
infile = open('noisy_signal.txt')
c = 0
for line in infile:
    print(line)
    c = c + 1
    if c > 9:
        break
```

We can use this approach to read the data into the program, but what it delivers are *lines of text*, i.e. *strings*, rather than the numbers we want. As shown on the right, we can get the values by first using `split` (which splits a string into a list of substrings at positions where spaces occur), and then `float`, which converts a string to a floating point number (provided it *looks like* one):

```
>>> line = '0.33 0.5'
>>> vals = line.split()
>>> vals
['0.33', '0.5']
>>> t = float(vals[0])
>>> v = float(vals[1])
>>> t + v
0.83
```

Using this approach, load the noisy signal data from the file. Store the loaded values into lists suitable for plotting (i.e. one for the time points, and another for the intensities), and then plot the signal. If you have time, you might also load and plot the data for the original pure signal, mains noise and random noise, for comparison.

4.2 Implementing a moving average filter

The simplest way to remove high frequency noise from a digitised signal is to smooth it by calculating the moving average (see `http://en.wikipedia.org/wiki/Moving_average`). In this part of the lab class, we are going to implement a *moving average filter*.

The moving average is calculated by looping through the data, and at each point calculating the average value of the signal within a *window* of specified length. Using 'a window' here means that, for a position i, with a window size W, we include all positions from $i - W/2$ through to $i + W/2$. Python allows a simple means to access such a sequence of values from the list, by using *slicing*. In particular, we can specify *two* values, as in `x[i:j]`, as a way to access a *subsequence* of a list, known as a *slice*, which is itself returned as a list, as the following illustrates:

```
>>> x = [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> x[2:5]
[13, 14, 15]
```

For values that are sensibly in range, the slice `x[i:i+j]` returns the item `x[i]` plus further items to its right, so as to return a total of `j` items. Try this yourself with the interpreter (and also investigate what happens if the first index is less than 0, or the second greater than the list length). The function `sum` computes the sum of a list of numbers, e.g. `sum([5,7,9])` returns 21. Using slicing and `sum`, it is easy to compute the sum of values from a window of the signal.

Create a file `lab4_sec4.py` for your code for this task. Write code to compute a moving average for the signal data you have loaded. Start by creating a list of zero values in to which the moving average values can be written (which must of the same length as the list of time points for plotting to be possible.) Compute the moving average values, using a window size of 20. Plot the filtered signal against time, to see if/how well the original 5 Hz 'pure signal' is recovered.

4.3 To think about

Does your implementation allow the window size to be varied easily? Experiment with different window sizes, to see how much this affects the filtering process. Think about how you might automatically determine the frequency of the signal recovered by filtering?