Python Practice 5: Loops and Lists continued — further data analysis

In this class, we will tackle a different data analysis task, with a view to firming up the programming skills we've acquired so far. The first task involves analysing some data so that this can be usefully plotted, as we have done before. This time, however, our aim is to plot a histogram of the data, which requires that we group values together into *bins*.

Along the way, we'll gain further practice at using Python loops and lists, and at doing *incremental code development*. Remember that a useful technique in incremental code development is the use of *print statements*, as a basis for checking at any point that the code is actually doing what we think it is doing. As discussed previously, we can use print statements to check many things about how a piece of code runs, e.g. 'how far' a program gets before crashing (i.e. generating a runtime error and exiting), whether variable values are as we'd expect them to be at a given stage, whether an `if-else` statement makes the 'choice' that we expect of it, *etc*, *etc*.

Start by downloading the lab class code file `pulse_data.txt`, and storing it in a sensible location on your U drive.

1 Finding Patterns in Data

Pulses of light are observed from a distant source in the galaxy. The pulses vary in intensity in a seemingly random fashion. Our aim is to find out if there is something more systematic to these variations. The file `pulse_data.txt` records relative intensity values from a large number ($\sim$100K) of observed pulses. Create a file for your solution to this week's lab task.

1.1 Initial data explorations

Begin by writing code to load the data, storing the values in a list, in the same order that they are found in the data file. If you have forgotten how to do this, refer back to Sec. 4.1 of last week's lab sheet. Be sure that you are storing the actual numeric values, *not* just strings that *look like* those values.

Next, just plot the values in order, to see if any simple pattern emerges. Refer to Sec. 3 of last week's lab sheet for a reminder of Pylab's graph plotting functionality. For this plot, we will use the list of data values as the y-dimension values, and for the corresponding x-dimension, we could simply use a suitably-sized list of ascending integers (as might be produced using the `range` function). In fact, if you give the `plot` function just a *single* list of values, it will plot them in exactly this manner, i.e. as being y-dimension values that are plotted against their own position in the list. Inspect the resulting plot — I think you'll agree there's not much to see here but randomness.

Now try putting the values into sorted order. Python will do the sorting for you, using the list `sort` method, e.g. for a list `data`, if we call `data.sort()` the list is sorted *in-place*, i.e. the variable `data` now stores the sorted variant of its initial list. Plot the sorted values and inspect the resulting plot. This plot is somewhat more informative than the last. We can see that the more extreme high and low values occupy quite narrow zones at the periphery of the plot, and *width* on this plot corresponds to the number of values plotted. This suggests that more extreme values are fairly rare, and more 'middling' values predominate.

1.2 Data *binning* for histogram plotting

We next want to plot a *histogram* of the data. Histograms of real-valued numeric data usually work by sub-dividing the range across which the values lie into a number of *sub-regions*, which are identified by bounding values. For example, for values in the range from 0.0 to 9.0, we might set boundaries 3.0 and 6.0, and say that any value $v < 3.0$ belongs to region 0, any value v such that $3.0 \leq v < 6.0$ to region 1, and any value $v \geq 6.0$ to region 2. These sub-regions are usually called *bins*, i.e. with values in a given sub-range being assigned to that bin, and with the histogram showing the count of items in the bins. Thus, value 1.0 belongs to Bin 0, value 4.0 to Bin 1.

Let's look at another example in more detail, and consider how the *binning* process might be performed automatically. Assume we have the following data (here shown as a Python list):

```
data = [4, 3, 5, 7.5, 3.8, 1.5]
```

We need first to determine the *range* across which the values lie. To do this, we must find the *minimum* and *maximum* values in the data, which here are 1.5 and 7.5. Thus, this range has a *span* of 6 units, i.e. 7.5−1.5. The value that lies *halfway* across this range is 4.5, since it is exactly 3 units (half of 6) past the minimum value 1.5, or equivalently we can say:

$$(4.5 - 1.5)/6 = 0.5$$

To restate this more generally, and in Python, we could first compute the minimum and maximum values from the data. Python provides build-in functions `min` and `max` to get the min/max values from a list of numerics. Thus:

```
minval = min(data)
maxval = max(data)
```

All values therefore range between `minval` and `maxval`, across a span given by:

```
vspan = maxval - minval
```

For any single value `value` from the data, its position across the value range, in *proportionate* terms is therefore:

```
(value - minval) / vspan
```

We can assign different sub-stretches of the value range to different bins (or different *bin ids*) as follows. First let `BINS` be an integer that we set to determine the number of bins. The bin id for a given value `value` is computed by determining how far across the value range it lies, as above, and then using this to select the appropriate bin id from the range 0 . . . `BINS`, which we can do by multiplying the two value together, and then rounding down to the nearest integer (which can be done with the Python `int` functions). Thus we get:

```
binid = int(BINS * (value - minval) / vspan)
```

For example, for a value that lay one third (0.333) of the way across the value range, and with `BINS` set to 100, this would compute the bin id as 33. With `BINS = 50`, however, we would instead get bin id 16.

The above method works quite well, but has the unfortunate characteristic that the only values assigned to the *final* bin (i.e. bin 50, when `BINS = 50`) must be the same as the maximum value, which is might be just a single value. This is typically *not* the behaviour we want, so we might *override* it by adding a step to spot when the assigned bin id is equal to `BINS`, in which case the bin id is reset to `BINS-1` (i.e. reassigning the maximum value in the data to this bin). This has the effect that we get only bin ids in the range 0 . . . `BINS-1`, and therefore that there are precisely

`BINS` distinct bins allowed (as we originally intended).

The fact that the bin ids here are integers in the range 0 to `BINS-1` allows us to use a simple trick for storing the counts for each bin. This is to store the counts in a *list* of length `BINS`, and then to use a given bin identifier i as an *index* for the position in the list where that bin's counts are stored. The list must be initialised with zeros, one for each bin. Then, when we encounter a data value d, we can just compute its bin identifier and *increment* the count at that index in the list.

For example, if we had only three bins, we could start with a list of zeros `[0,0,0]`. Seeing a value that belongs to bin 1, we could add one to this counter, giving `[0,1,0]`. If the next two values belonged to bin 0, we update the counts firstly to `[1,1,0]`, and then to `[2,1,0]`, and so on.

Implement the above binning method and apply it to the data, with $B = 50$ (or any other number you wish to try). Plot the counts produced (perhaps plotting as points rather than as a continuous line). Does this presentation of the data reveal anything about the underlying pattern of the variations in the data? Does this lead you to a particular hypothesis about the astronomical origins of the observed pulses?

1.3 Other plotting options

You might prefer to plot your results as a bar chart proper, for which `pylab` provides the function `bar`, as in e.g.:

```
bar(xvals,yvals)
```

This function requires both x and y value lists to be provided. Here, `yvals` would be your bin counts, whilst `xvals` might be a list of the bin id integers (although it could instead be a list of boundary values).

Although today's lab task is a useful exercise, we note that Pylab will also produce histograms for you, using the `hist` function rather than `plot`. For example, if `data` is a list storing your data values, then we can ask Python to bin the data into 50 bins, and plot the resulting histogram, with just following single command:

```
hist(data,50)
```