# Review: Week 1-4

Imad Pasha
Chris Agostino

February 26, 2015

## 1 Introduction

It's been a pretty fast four weeks, and we've thrown a lot at you all. Such is the nature of learning programming. There are two main components- logic and syntax. Hopefully for a lot of you, your level of comfort with both areas is steadily improving. If not (and even if you are getting along but wouldn't mind reviewing), we are putting together this review to kind of pull together the major concepts we have covered so far. If I have time, I will also try to make a video with some live examples as a review as we move into the second part of the course.
**Reminder: There's a ton of content in the textbook, so if things from earlier are still confusing you, take a look back through those sections- seeing things a second time really helps.

## 2 The Basics

In programming, we work by setting variables equal to quantities of different data types, and then manipulate them to help us solve calculations or perform analysis.

### 2.1 Lists and Arrays

Lists and arrays are the primary way we store large amounts of information in python. We can manually define a list:

list_1 = [1,2,3,4,5,6]

or array:

array_one = np.array([1,2,3,4,5])

but ultimately we don't spend much of our time doing that. We have large arrays full of information from outside of python that we are attempting to work with.

**Remember: Lists and arrays behave similarly in some cases and vastly differently in others. It is important to keep those situations straight so that you know when to use which (it is almost always safe to use arrays). When you are only appending values and then plotting them, a list is fine. But if you need to do numerical calculations on those numbers, you'd better stick with arrays.

Primarily we pull arrays out of data files. For example, if I have a 3 column data file for time, position, and velocity, I can extract the individual arrays as such:

```
raw_data = np.loadtxt('data.txt')
raw_data = np.transpose(raw_data)
times = raw_data[0]
positions = raw_data[1]
velocities = raw_data[2]
```

Note: Just to reiterate, the reasoning for the transpose is that from a columnized dataset, python by default pulls the data in as sets of 3 (so the first time, position, and velocity would be [time1,pos1,vel1], which could be accessed by indexing raw_data[0]. But we want full arrays for all times, positions, and velocities (ie we want to separate the variables). transposing makes the variable raw_data into one of length 3, with three arrays stored within, [[time1,time2,...],[pos1,pos2,...],[vel1,vel2,...]], which when indexed at 0, 1, and 2 would provide arrays for time, pos, and vel as desired.

In addition to pulling in data arrays, there are some useful types of lists/arrays we can create on the fly, such as range, arange, zeros, ones, linspace, etc. Range and arange are used to quickly make ascending counting order arrays- which are useful as iterators because we can use the values in these arrays as the indices with which we index other arrays.

Speaking of indexing, how do we pull chunks out of arrays/lists/strings? We can index them using the square bracket notation, and pull out individual values or whole slices. If I have an array named john, then john[0] will return the first element in john, john[-1] will return the last. I can also do john[5:15] to return the 6th to 16th elements of john. The colon basically stands for "to" so 5:15 means from 5 to 15. You can index through the end of an array by saying john[5:] or john[:17], which would index from 5 to the end, or the beginning to the 18th element, respectively.

# 3 Logic: For loops and If statements

While it is easy to think we have a hold on all that array slicing and manipulation, we get put to the test as soon as we begin to iterate over arrays using for loops, and checking that our indexing in our conditionals are returning what we want.

**Note: Remember that Solutions to the tutorials are always posted the week after they are due, so you can take a look at the code and annotations to try to understand the examples.

## 3.1 Iteration: Elements vs. Value

So what's up with this whole iterating thing anyway? And what is Imad going on about when he is rambling about iterating over elements vs values?

In short, for loops are pretty simple. They iterate over lists/arrays. If I have a for loop over i in some array "drew", then the enclosed block of code will run over and over, using each successive value in "drew" in place of i, wherever it appears in the block of code. What?

```
drew = range(10)
for i in drew:
    print i
```

So each value in drew will be printed, as it pulls the ith element from drew and plugs it in to where i appears in the block (here, at the print statement).

So lets jump ahead and say I have those time, position, and velocity arrays. For the sake of complexity, lets say the times start at 100 and advance from there in integer units. The question is, when it comes time to iterate over these arrays, what do I choose? The answer is, it depends on whether you care about element number or not.

For example, lets say you wanted to know how many times during the experiment the velocity exceeded 25 m/s. In this case, you don't care at what time this happens, only for what *total duration* of time your object's velocity exceeds the limit. We can iterate directly over velocity:

```
outputs = [ ]
for i in velocity:
    if i > 25:
    outputs.append(i)
duration = len(outputs)
```

Notice that we have no information about at what specific times we were looking at. We only know how many times (at a measuring rate of one measurement per one tick of time) velocity exceeded 25 m/s AND we know what speeds those happened to be.

On the other hand, if we want to know at what times the velocity peaked (hit local maxima), then we somehow need to iterate in a way that hangs on to the index of where these events are happening. We can't use time as our iterator, since time starts at 100, but velocity[100] doesn't return the first value in velocity. But if we define an array containing 0,1,2,3,... up to the length of the velocity array, then that array would be an array of the indices of velocity. By iterating over THIS array, i becomes the index where things are happening or being checked, and to actually know the velocity we use velocity[i]. Our code might look like this:

```
maxima = [ ]
indices = range(len(velocity)-1)
for i in indices:
```

```
if velocity[i] > velocity[i+1]:
    if velocity[i] >velocity[i-1]:
        maxima.append(times[i])
```

Remember, we were interested in the times when the velocity peaked, not the indices within the velocity array where it did. (Sometimes that IS our interest though). But here, in the last step, I use the fact that the times, positions, and velocity arrays all have the same length and the i's produced by my indices array applies equally well to all three. So once i know the index in VELOCITY where the max occurs, I know the time as well; it is time indexed at that same i that produced the max in v. In this example, I set indices to be an array with the proper length (the minus 1 was to prevent attempting to check one past the end of the array with the i+1) in order to illustrate that the for loop is ALWAYS just plucking values out of arrays and plugging them in. Even in the short hand: for i in range(len(arr)): notation, always remember that range(len(arr)) is itself an array, of ascending indices, up to the length of arr.

I'll try to post a video highlighting this distinction, and if it is still confusing, we can try to find a way to discuss it at office hours that makes conceptual sense.