PHAS1240 Session 3
IPython Notebooks / lists and arrays

October 15, 2015

This session covers two important but unrelated topics - IPython Notebooks (which are a useful document format for combining code and text), and Python lists and arrays.

1. Part A: An introduction to IPython Notebooks

Until now, we've interacted with Python in two ways:

- By using the interactive Python console in Spyder
- By using the code editor in Spyder to write complete codes

There is a third way we are now going to introduce, and which I hope you will find particularly useful in other areas of your studies - the IPython Notebook. (This is also now known as the Jupyter notebook, as it can also be used with other languages like Julia and R, but we will only be using them with Python!)

In fact, this document, and the scripts for the first two sessions, are IPython Notebooks, but so far we've only looked at static versions of them. Now we are going to unleash their full power, and use them interactively.

If you haven't already, watch the screencast on how to open up IPython Notebooks in Anaconda on Desktop@UCL. Remember, they can't be opened by double-clicking, only from the IPython Notebook browser interface.

Then:

- Create a new directory somewhere on your N: drive and name it appropriately
- Download the IPython Notebook version of this document (with file extension .ipynb) and put it in the new directory
- Open up IPython Notebook from the start menu, use the browser interface to navigate to your directory, and open the notebook (you may then want to close the old, static version of the notebook to avoid confusion between them).

An IPython Notebook consists of *cells*. Single click on this paragraph of text and a box will appear around it - everything within this box is in the same cell. There are two basic types of cells:

1.0.1 Code cells

Code cells contain bits of Python code. You can edit these cells by clicking on them. To run code cells, click on the cell so there's a box around it, and press SHIFT+ENTER. Any output the cell generates will be output directly underneath the cell.

```
In [ ]: # This is a comment within a code cell
        print "This is a code cell"
        print "Press SHIFT+ENTER to execute it and produce this text as output"
```

```
In [ ]: # This is a comment within another code cell
        x = 5
        y = 4.2
        # this cell does not generate any output, but still runs when you press shi
```

1.0.2 Text cells

Text cells contain text (and sometimes maths). To edit these cells you double click on them (try double clicking on this cell!). The font will change and you'll be able to edit the text. These cells are actually written in Markdown, which is a markup language (a bit like a very simple version of HTML). We'll come on to how to produce equations in a moment.

1.1 Other stuff you need to know

- Ipython notebooks get saved automatically every so often. To save them manually, click on the disk icon in the top left corner.
- The other icons in that row let you (in order):
- Add a new cell below the current one
- Cut the current cell
- Copy the current cell
- Paste from the clipboard
- Move the current cell up
- Move the current cell down
- Run the current cell (as an alternative to shift+enter)
- Stop/interrupt running the current cell
- Restart the kernel
- Restarting the kernel will clear all the variables from memory, and when you re-run cells they will be numbered starting from [1]. It's a good idea to do this every so often, as you may run into problems if you define a variable, then delete it, but continue to refer to it somewhere else in your code.

- The dropdown menu lets you change the type of the current cell between Markdown (i.e. text) and code, and also format different header styles.

There is some rudimentary text formatting available - just the basics.

For **bold text** enclose the text in double asterisks

For *italic text* enclose the text with underscores, or with single asterisks *like this*

Double click on this cell to see exactly how to do this. If you want to know how to format bullet lists like in the cell above, double click on that one too.

1.2 Writing maths

It's useful to know how to write maths as well. Ipython notebooks use LaTeX markup (pronounced Lay-Tech, not like rubber) which is either a particularly beautiful and elegant markup language for typesetting both maths and everything else (for all right-thinking people) or a form of torture that should be outlawed under the Geneva convention (for the not-yet-enlightened). You can find more information on LaTeX here: http://en.wikipedia.org/wiki/LaTeX

For our purposes, we won't need to do anything particularly complicated and you only need to know the very basics, which are fairly intuitive, so there will be no torture involved.

1.2.1 Basics

Double click on this cell, or on any other cell with maths in it, and you will be able to see exactly how it was generated. Here is a brief summary of the basics:

- to switch to math formatting within a paragraph, like this $x = 2y^3$, enclose the maths in dollar signs.
- to format an equation on a separate line, enclose it in double dollar signs, like this:

$$x = 2y^3$$

- Most greek letters are prefixed with a backslash followed by the name of the letter, eg

  `\alpha, \beta, \gamma, \delta`

  becomes $\alpha, \beta, \gamma, \delta$ when enclosed in dollar signs.
- Subscripts use an underline character, superscripts use ˆ. For example $x_0 = y^2, x_1 = \alpha^2$
- Common functions like sin, cos, exp, are also prefixed by a backslash. For example

$$\exp(5\alpha) = \sin(2\pi x)$$

For most purposes in this course, you'll frequently be able to copy and paste an equation from the session script (itself an Ipython notebook, so don't worry about being expected to know more than this. In case you find it useful, there are also websites like http://www.codecogs.com/latex/eqneditor.php which let you pick maths symbols from a palette and convert them to LaTeX code.

1.2.2 Using Ipython notebooks elsewhere in your courses

Ipython notebooks are new technology, and are being used more and more as people realise how useful they are - I hope that you too will find them useful beyond PHAS1240, whether just for making notes, or doing quick calculations and solving problems numerically.

We'll now move on to the second part of this session, on Python lists and Numpy arrays. This has no particular relation to Ipython notebooks other than it is a convenient point in the course to introduce both topics. Everything you will see here works both in IPython notebooks and in Spyder (or any other code editor). For this session, however, you need to produce and submit an IPython Notebook.

2. Part B: Lists and NumPy arrays (and tuples)

By the end of this section, you should:

Understand what Python lists and Numpy arrays are, and the differences between them;

Understand how to manipulate Numpy arrays with basic mathematical operations

Understand how to reference different parts of a list or array.

We already have variables as a way of storing single values in Python. But in Physics we often have more than one number associated with an object, for example, a cartesian vector with x, y, and z components:

$$\mathbf{r} = (r_x, r_y, r_z).$$

We could create three variables rx, ry, rz to describe this vector, but this would be quite inefficient. This is even clearer when we consider bigger objects: imagine in a lab experiment we've collected a set of data, like this: V(V) I(A) 1.0 2.0 2.0 4.1 3.0 5.9 4.0 8.0 5.0 10.2 Clearly we don't want to create ten variables v1, i1, v2, i2, v3, etc to describe this data set. And what if we had a data set with hundreds, or even thousands of data points?

Fortunately Python has data structures that can contain lots of elements. There are three basic types:

- Lists
- Tuples
- Arrays

Of these, the most useful for our purposes will be arrays, but to see why, we'll first look at lists (we won't cover tuples at all in this course, but you should be aware that they exist.)

2.1 Lists

In this notebook, you'll need to use `Shift+Enter` in each of the code cells, and make sure you understand any output before proceeding.

A python list is just a "container" for several objects - here is an example:

```
In [ ]: my_list = [1, 5.0, 3.4, "apples", 2.3+0.2j, "bananas"]
```

Note the following characteristics:

- The list is enclosed by square brackets [ ]
- The list items are separated by commas
- The list can contain a mix of different data types (here we have integers, floats, strings and complex all in the same list)

Now consider some lists that are probably more like ones we'd need to use for doing physics - lists of numbers.

```
In [ ]: a_list = [1, 2, 3, 4 ]
        b_list = [5, 6, 7, 8 ]
```

```
In [ ]: a_list
```

```
In [ ]: b_list
```

What happens if we try and add these two lists together?

```
In [ ]: z_list = a_list + b_list
        z_list
```

Is this what you might have expected (or hoped for)?

Python concatenates the two lists, rather than performing arithmetic addition. This is useful for data handling, but less useful for doing maths.

Fortunately, we also have the Numpy array:

2.2 Arrays

To use arrays we need to use the NumPy (numerical python) module. As before, we'll need to import NumPy, and we'll follow best practice (as always!) and import all of numpy and then prefix each numpy function with the abbreviation np.

```
In [ ]: import numpy as np
```

Now let's convert our lists into numpy arrays to see the difference - note how we do this using the `np.array()` function.

```
In [ ]: array_a = np.array(a_list) # convert the list called a_list to an array call
        array_b = np.array(b_list)
        array_z = array_a + array_b
        array_z
```

Now it adds the two arrays element-wise, which is what we wanted!

2.2.1 Other ways to create arrays

There are various ways of creating Numpy arrays. We've already seen the np.array() function above, which converts a list to an array. Others are particularly useful:

- **create an array with a specified number of evenly-spaced points**: Use `np.linspace()`

```
In [ ]: np.linspace(0,10,5)
```

This creates an array which starts at the first number (here 0), finishes at the second number (here 10), using the third number as the number of points - so here we have 5 points starting at 0 and finishing at 10.

- **Create an array with a specified distance between points** Use `np.arange()`. This is almost the same as linspace, the difference is subtle, look at this

```
In [ ]: np.arange(0,10,1.0)
```

The first number gives the starting point, as for `np.linspace()`. The final number gives the step between the points in the array, in this case 1.0. The second number gives the stopping point, but note that this is *not included* in the final array - in this case the array stops at 9.0.

- **Create an array full of zeros or ones** These are quite intuitive. By default, the array contains floating point numbers, but you can specify others if you want:

```
In [ ]: np.zeros(6) # create an array of six zeros
```

```
In [ ]: np.ones(8) # create an array of 8 (floating point) ones
```

```
In [ ]: np.ones(8,int) # create an array of 8 (integer) ones
```

```
In [ ]: np.ones(4, complex) # create an array of complex ones.
```

- ** Create an "empty" array** You can do this using the function `np.empty()`. In fact, the array isn't really empty, but will contain whatever value happens to be already in the memory location Python allocates for the array. Again, this array will be floating point by default:

```
In [ ]: np.empty(4)
```

- **Reading in an array from a file** Importing an array from a data file is really useful, especially in the lab. There are several ways to do this, but we'll stick with the easiest and most general way, which is to use `np.loadtxt()` function. Look at this command, which will import the data from the file called "samplefile.txt" and put it in an array called mydata:

```
In [ ]: mydata = np.loadtxt("samplefile.txt")
```

Of course, this will only work if you actually have a file called samplefile.txt in the same directory as your notebook! Otherwise you'll just end up with an error message, like you probably did above. In this case, download the file called "samplefile.txt" from Moodle and retry it.

```
In [ ]: print mydata
```

If that worked, you should see that the array "mydata" contains the same numbers as the file samplefile.txt - open the .txt file in Notepad (or any other program that can read plain text files) to make sure.

2.3 Referring to parts of arrays

We often want to use just one element of an array, or a specified part of it, rather than the whole thing. How can we do this?

First, let's create an array to play around with:

```
In [ ]: a = np.linspace(-3.0, 4.0, 9)
        print a
```

What if we want to do something just to the first element of the array - how do we refer to just the first element? You might think about trying something like this:

```
In [ ]: a[1] # use square brackets to refer to an element of an array.
```

But this isn't the first element of the array - it is the second!

This is because Python is one of the computer languages that start counting at zero, not at one. In order to get the first element of the array, we need to use:

```
In [ ]: a[0]
```

Now we have the first element. This is something else that can easily catch you out if you're not used to it. Similarly the final element of our array is not a[9], it is a[8].

Another way of referencing the last element of an array is using a[-1]. The penultimate element is a[-2], and so on.

2.3.1 Slicing arrays

"Slicing" an array to reference a subset of the data is often useful. Look at the following:

```
In [ ]: a[1:3]
```

This returns all elements from #1 up to but *not including* #3

```
In [ ]: a[3:]
```

This returns all elements from #3 (i.e. the fourth element) to the end of the array

```
In [ ]: a[:2]
```

This returns all elements from the beginning of the array up to but not including #2

```
In [ ]: a[:]
```

This returns all elements of the array. Slicing is even more useful when we're looking at 2-dimensional arrays (which are usually used to represent matrices), but we will not cover this in this course. However, matrix manipulations are really important in physics, especially in quantum mechanics, and we'll look at those in much more depth next year.

3. Your task

Your task for today is to create a new IPython Notebook. Change the title of your notebook to "Yourname-session3".

In the notebook, do the following, and include a commentary explaining what each code cell (or group of cells) does. As a guideline, you should aim to give roughly the same ratio of text to code as you see in this notebook.

3.1 *Part 1:*

1. `np.array([0,5,10])` will create an array of ***integers*** starting at 0, finishing at 10, with step 5. Use a different command to create the same array automatically.
2. Create (automatically, not using np.array!) another array that contains 3 equally-spaced floating point numbers starting at 2.5 and finishing at 3.5.
3. Use the multiplication operator `*` to multiply the two arrays together
4. Use your awesome google skills to find numpy functions that will find the dot product and the cross product of these two arrays. Use a text cell to comment on the differences between this and using the `*` operator.
5. We met the "`while`" loop last week, and mentioned the "`for`" loop. This is an example of a "`for`" loop:

```
In [ ]: r = np.linspace(1, 5, 5)
        print r
        for n in r:
            print "n is ", n
            print "n*3 is ", n*3
        print "Finished"
```

Copy and paste this cell into your Notebook, run it, and use the results to explain how a `for` loop works.

3.2 *Part 2:*

1. Download the file "session3data.txt" from Moodle and save it in the same directory as your Notebook. Import the data from this file into an array using `np.loadtxt()` - choose your own (sensible) variable name for this array.

2. Manipulate this array using the following commands. Use a different code cell for each, and (obviously) replace "yourarray" with the variable name for your array. Explain what each command does.
    1. `np.sin(yourarray)`
    2. `np.size(yourarray)`
    3. `len(yourarray)`
    4. `np.max(yourarray)`
    5. `np.min(yourarray)`
    6. `np.sum(yourarray)`
    7. `np.mean(yourarray)`. For this one, also find another way of calculating the same quantity.
3. Use array slicing to find:
    1. the sum of the first ten elements of the array
    2. the arithmetic mean of the last 5 elements of the array

Make sure you're happy with the balance of text and code in your notebook.

You're also *strongly* advised to restart the kernel of your Notebook, and rerun all the code cells in order to check that everything is working as it should.

Once you're happy with your notebook, upload it via Moodle in the usual way.

```
In [ ]:
```