As you can see here, notebooks are structured in such a way that I can intersperse my little notes and commentary about what you're seeing here within the actual code itself, and you can actually run this code within your web browser! So, it's a very handy format for me to give you sort of a little reference that you can use later on in life to go and remind yourself how these algorithms work that we're going to talk about, and actually experiment with them and play with them yourself.

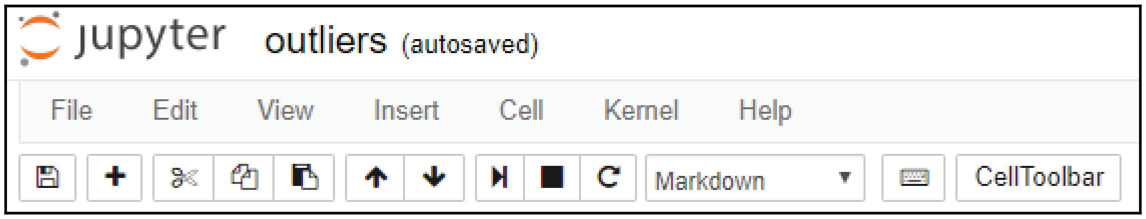
The way that the IPython/Jupyter Notebook files work is that they actually run from within your browser, like a webpage, but they're backed by the Python engine that you installed. So you should be seeing a screen similar to the one shown in the previous screenshot.

You'll notice as you scroll down the notebook in your browser, there are code blocks. They're easy to spot because they contain our actual code. Please find the code box for this code in the Outliers notebook, quite near the top:

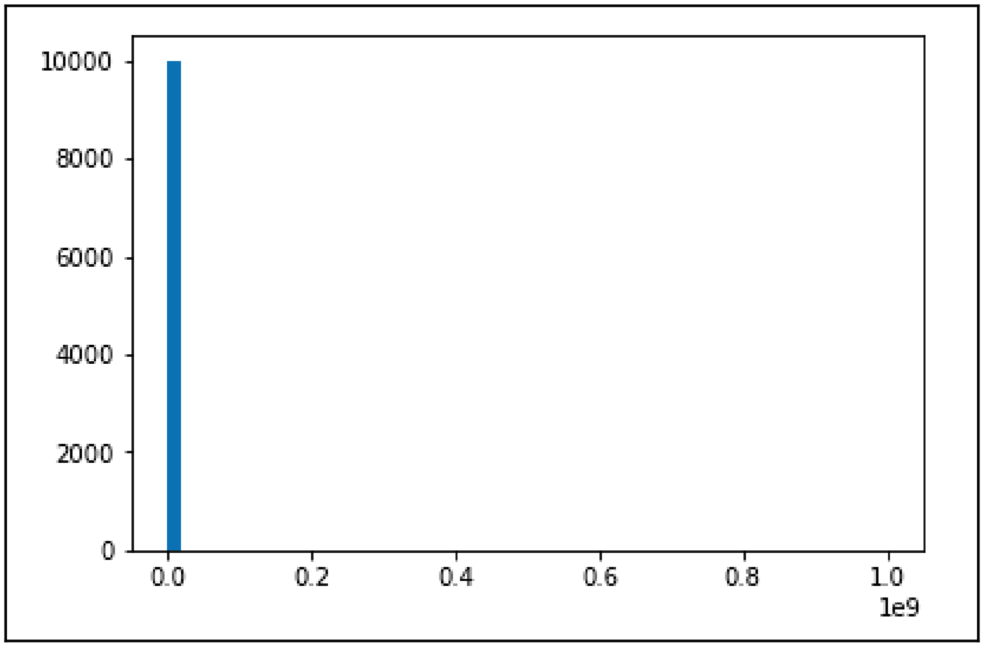
```
%matplotlib inline
import numpy as np

incomes = np.random.normal(27000, 15000, 10000)
incomes = np.append(incomes, [1000000000])

import matplotlib.pyplot as plt
plt.hist(incomes, 50)
plt.show()
```

Let's take a quick look at this code while we're here. We are setting up a little income distribution in this code. We're simulating the distribution of income in a population of people, and to illustrate the effect that an outlier can have on that distribution, we're simulating Donald Trump entering the mix and messing up the mean value of the income distribution. By the way, I'm not making a political statement, this was all done before Trump became a politician. So you know, full disclosure there.

We can select any code block in the notebook by clicking on it. So if you now click in the code block that contains the code we just looked at above, we can then hit the run button at the top to run it. Here's the area at the top of the screen where you'll find the Run button:



Hitting the Run button with the code block selected, will cause this graph to be regenerated:



Similarly, we can click on the next code block a little further down, you'll spot the one which has the following single line of code :

```
incomes.mean()
```

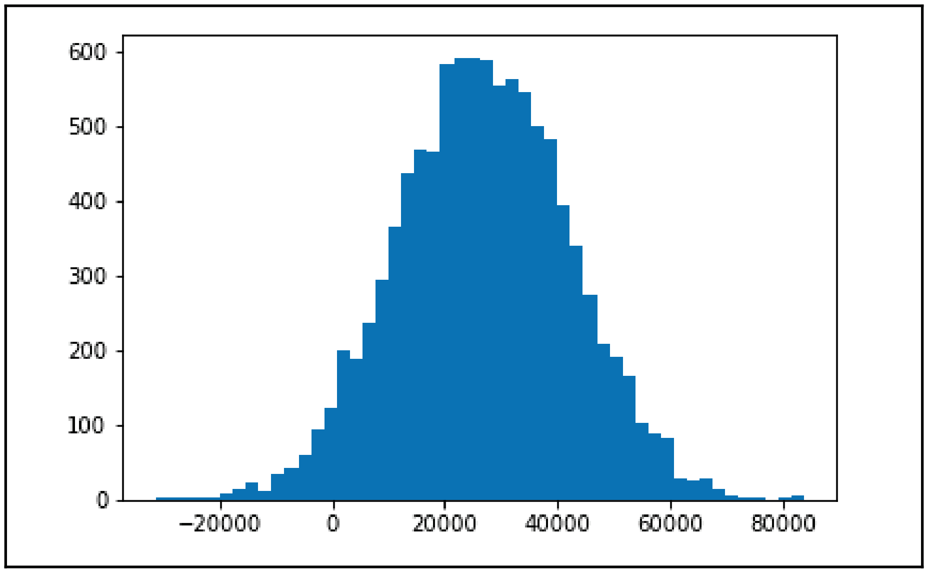
If you select the code block containing this line, and hit the Run button to run the code, you'll see the output below it, which ends up being a very large value because of the effect of that outlier, something like this:

```
127148.50796177129
```

Let's keep going and have some fun. In the next code block down, you'll see the following code, which tries to detect outliers like Donald Trump and remove them from the dataset:

```
def reject_outliers(data):
    u = np.median(data)
    s = np.std(data)
    filtered = [e for e in data if (u - 2 * s < e < u + 2 * s)]
    return filtered

filtered = reject_outliers(incomes)
plt.hist(filtered, 50)
plt.show()
```

So select the corresponding code block in the notebook, and press the run button again. When you do that, you'll see this graph instead:



Now we see a much better histogram that represents the more typical American - now that we've taken out our outlier that was messing things up.

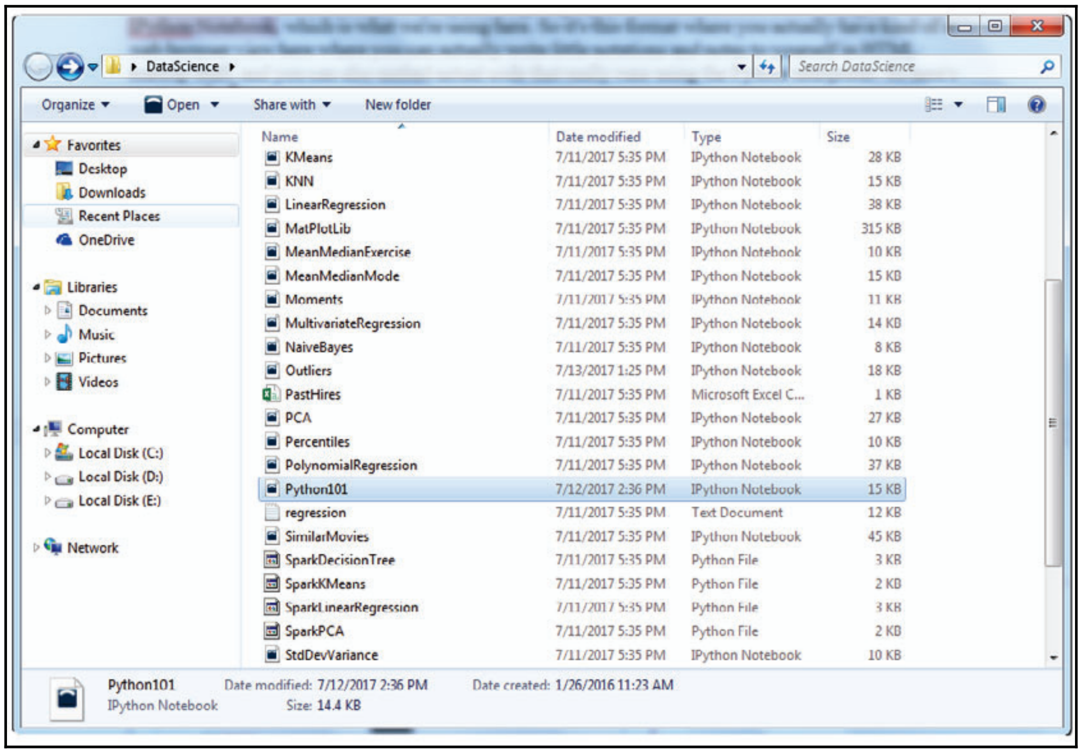
So, at this point, you have everything you need to get started in this course. We have all the data you need, all the scripts, and the development environment for Python and Python notebooks. So, let's rock and roll. Up next we're going to do a little crash course on Python itself, and even if you're familiar with Python, it might be a good little refresher so you might want to watch it regardless. Let's dive in and learn Python.

Python basics - Part 1

If you already know Python, you can probably skip the next two sections. However, if you need a refresher, or if you haven't done Python before, you'll want to go through these. There are a few quirky things about the Python scripting language that you need to know, so let's dive in and just jump into the pool and learn some Python by writing some actual code.

Like I said before, in the requirements for this book, you should have some sort of programming background to be successful in this book. You've coded in some sort of language, even if it's a scripting language, JavaScript, I don't care whether it is C++, Java, or something, but if you're new to Python, I'm going to give you a little bit of a crash course here. I'm just going to dive right in and go right into some examples in this section.

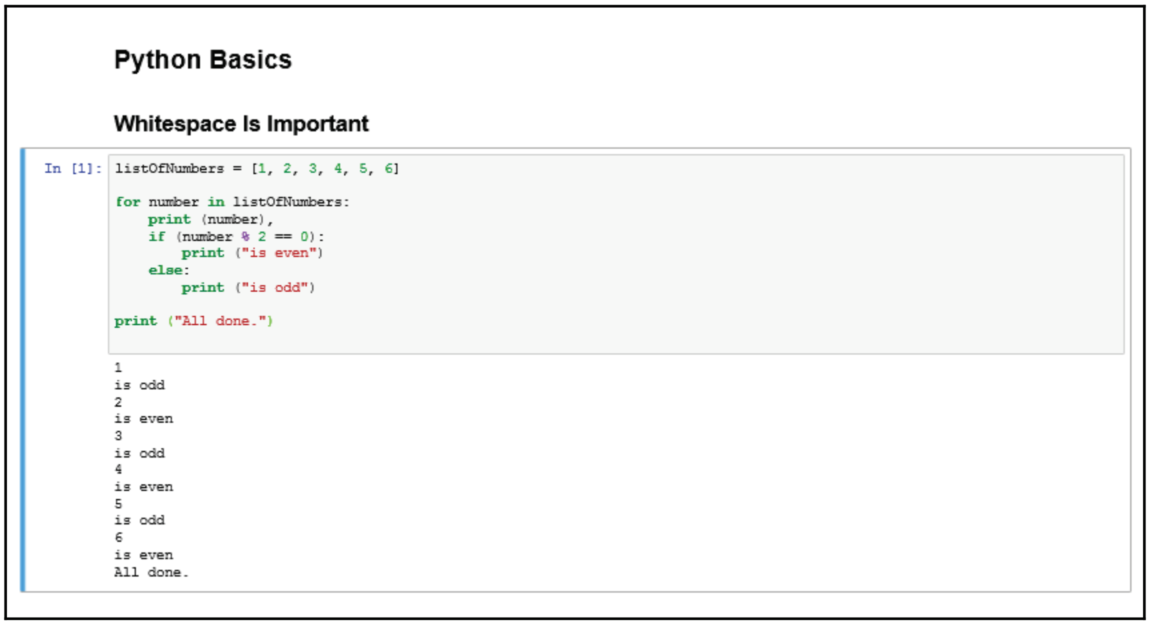
There are a few quirks about Python that are a little bit different than other languages you might have seen; so I just want to walk through what's different about Python from other scripting languages you may have worked with, and the best way to do that is by looking at some real examples. Let's dive right in and look at some Python code:



Name	Date modified	Type	Size
KMeans	7/11/2017 5:35 PM	IPython Notebook	28 KB
KNN	7/11/2017 5:35 PM	IPython Notebook	15 KB
LinearRegression	7/11/2017 5:35 PM	IPython Notebook	38 KB
MatPlotLib	7/11/2017 5:35 PM	IPython Notebook	315 KB
MeanMedianExercise	7/11/2017 5:35 PM	IPython Notebook	10 KB
MeanMedianMode	7/11/2017 5:35 PM	IPython Notebook	15 KB
Moments	7/11/2017 5:35 PM	IPython Notebook	11 KB
MultivariateRegression	7/11/2017 5:35 PM	IPython Notebook	14 KB
NaiveBayes	7/11/2017 5:35 PM	IPython Notebook	8 KB
Outliers	7/13/2017 1:25 PM	IPython Notebook	18 KB
PastHires	7/11/2017 5:35 PM	Microsoft Excel C...	1 KB
PCA	7/11/2017 5:35 PM	IPython Notebook	27 KB
Percentiles	7/11/2017 5:35 PM	IPython Notebook	10 KB
PolynomialRegression	7/11/2017 5:35 PM	IPython Notebook	37 KB
Python101	7/12/2017 2:36 PM	IPython Notebook	15 KB
regression	7/11/2017 5:35 PM	Text Document	12 KB
SimilarMovies	7/11/2017 5:35 PM	IPython Notebook	45 KB
SparkDecisionTree	7/11/2017 5:35 PM	Python File	3 KB
SparkKMeans	7/11/2017 5:35 PM	Python File	2 KB
SparkLinearRegression	7/11/2017 5:35 PM	Python File	3 KB
SparkPCA	7/11/2017 5:35 PM	Python File	2 KB
StdDevVariance	7/11/2017 5:35 PM	IPython Notebook	10 KB

Python101 Date modified: 7/12/2017 2:36 PM Date created: 1/26/2016 11:23 AM
IPython Notebook Size: 14.4 KB

If you open up the `DataScience` folder for this class, which you downloaded earlier in the earlier section, you should find a `Python101.ipynb` file; go ahead and double-click on that. It should open right up in Canopy if you have everything installed properly, and it should look a little bit something like the following screenshot:



Python Basics

Whitespace Is Important

```
In [1]: listOfNumbers = [1, 2, 3, 4, 5, 6]

        for number in listOfNumbers:
            print (number),
            if (number % 2 == 0):
                print ("is even")
            else:
                print ("is odd")

        print ("All done.")
```

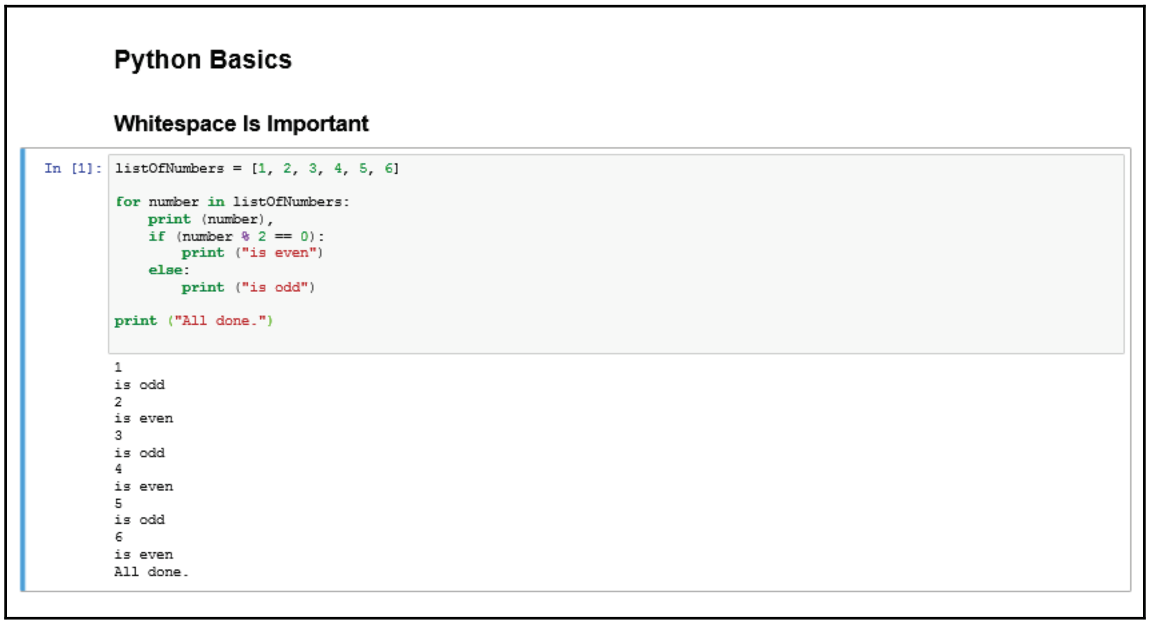
```
1
is odd
2
is even
3
is odd
4
is even
5
is odd
6
is even
All done.
```

New versions of Canopy will open the code in your web browser, not the Canopy editor! This is okay!

One cool thing about Python is that there are several ways to run code with Python. You can run it as a script, like you would with a normal programming language. You can also write in this thing called the *IPython Notebook*, which is what we're using here. So it's this format where you actually have a web browser-like view where you can actually write little notations and notes to yourself in HTML markup stuff, and you can also embed actual code that really runs using the Python interpreter.

Understanding Python code

The first example that I want to give you of some Python code is right here. The following block of code represents some real Python code that we can actually run right within this view of the entire notebook page, but let's zoom in now and look at that code:



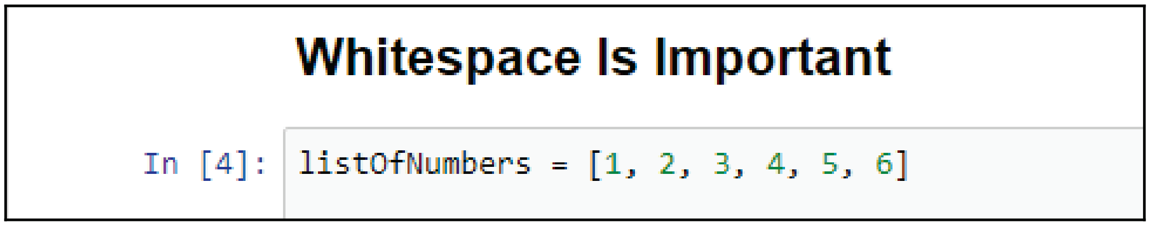
Python Basics

Whitespace Is Important

```
In [1]: listOfNumbers = [1, 2, 3, 4, 5, 6]

        for number in listOfNumbers:
            print (number),
            if (number % 2 == 0):
                print ("is even")
            else:
                print ("is odd")

        print ("All done.")
```

```
1
is odd
2
is even
3
is odd
4
is even
5
is odd
6
is even
All done.
```

Let's take a look at what's going on. We have a list of numbers and a list in Python, kind of like an array in other languages. It is designated by these square brackets:



Whitespace Is Important

```
In [4]: listOfNumbers = [1, 2, 3, 4, 5, 6]
```

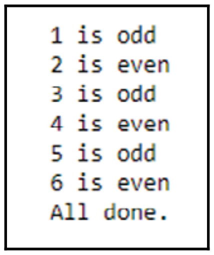
We have this data structure of a list that contains the numbers 1 through 6, and then to iterate through every number in that list, we'll say `for number in listOfNumbers:`, that's the Python syntax for iterating through a list of stuff and a colon.

Tabs and whitespaces have real meaning in Python, so you can't just format things the way you want to. You have to pay attention to them.

The point that I want to make is that in other languages, it's pretty typical to have a bracket or a brace of some sort there to denote that I'm inside a `for` loop, an `if` block, or some sort of block of code, but in Python, that's all designated with whitespaces. Tab is actually important in telling Python what's in which block of code:

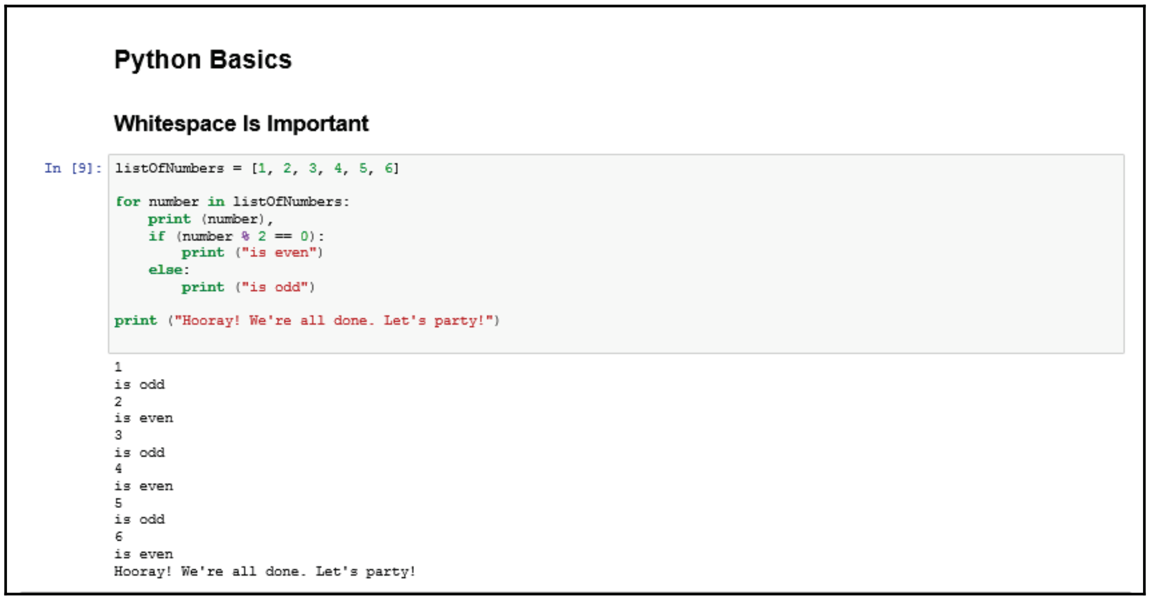
```
for number in listOfNumbers:
    print number,
    if (number % 2 == 0):
        print ("is even")
    else:
        print ("is odd")
print ("Hooray! We're all done.")
```

You'll notice that within this `for` block, we have a tab of one within that entire block, and for every `number in listOfNumbers` we will execute all of this code that's tabbed in by one *Tab* stop. We'll print the number, and the comma just means that we're not going to do a new line afterwards. We'll print something else right after it, and `if (number % 2 = 0)`, we'll say it's `even`. Otherwise, we'll say it's `odd`, and when we're done, we'll print out `All done`:



```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
All done.
```

You can see the output right below the code. I ran the output before as I had actually saved it within my notebook, but if you want to actually run it yourself, you can just click within that block and click on the Play button, and we'll actually execute it and do it again. Just to convince yourself that it's really doing something, let's change the `print` statement to say something else, say, `Hooray! We're all done. Let's party!` If I run this now, you can see, sure enough, my message there has changed:



Python Basics

Whitespace Is Important

```
In [9]: listOfNumbers = [1, 2, 3, 4, 5, 6]

        for number in listOfNumbers:
            print (number),
            if (number % 2 == 0):
                print ("is even")
            else:
                print ("is odd")

        print ("Hooray! We're all done. Let's party!")
```

```
1
is odd
2
is even
3
is odd
4
is even
5
is odd
6
is even
Hooray! We're all done. Let's party!
```

So again, the point I want to make is that whitespace is important. You will designate blocks of code that run together, you know, such as a `for` loop or `if then` statements, using indentation or tabs, so remember that. Also, pay attention to your colons too. You'll notice that a lot of these clauses begin with a colon.

Importing modules

Python itself, like any language, is fairly limited in what it can do. The real power of using Python for machine learning and data mining and data science is the power of all the external libraries that are available for it for that purpose. One of those libraries is called `NumPy`, or numeric Python, and, for example, here we can `import` the `Numpy` package, which is included with Canopy as `np`.

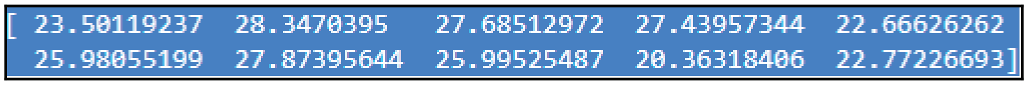
This means that I'll refer to the `NumPy` package as `np`, and I could call that anything I want. I could call it `Fred` or `Tim`, but it's best to stick with something that actually makes sense; now that I'm calling that `NumPy` package `np`, I can refer to it using `np`:

```
import numpy as np
```

In this example, I'll call the `random` function that's provided as part of the `NumPy` package and call its normal function to actually generate a normal distribution of random numbers using these parameters and print them out. Since it is random, I should get different results every time:

```
import numpy as np
A = np.random.normal(25.0, 5.0, 10)
print (A)
```

The output should look like this:



```
[ 23.50119237  28.3470395   27.68512972  27.43957344  22.66626262
  25.98055199  27.87395644  25.99525487  20.36318406  22.77226693]
```

Sure enough, I get different results. That's pretty cool.

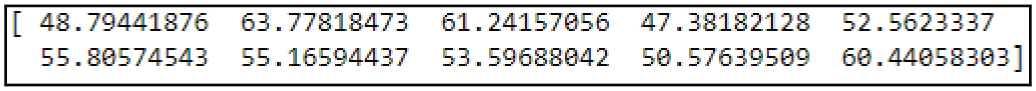
Data structures

Let's move on to data structures. If you need to pause and let things sink in a little bit, or you want to play around with these a little bit more, feel free to do so. The best way to learn this stuff is to dive in and actually experiment, so I definitely encourage doing that, and that's why I'm giving you working IPython/Jupyter Notebooks, so you can actually go in, mess with the code, do different stuff with it.

For example, here we have a distribution around `25.0`, but let's make it around `55.0`:

```
import numpy as np
A = np.random.normal(55.0, 5.0, 10)
print (A)
```

Hey, all my numbers changed, they're closer to 55 now, how about that?



```
[ 48.79441876  63.77818473  61.24157056  47.38182128  52.5623337
  55.80574543  55.16594437  53.59688042  50.57639509  60.44058303]
```

Alright, let's talk about data structures a little bit here. As we saw in our first example, you can have a list, and the syntax looks like this.

Experimenting with lists

```
x = [1, 2, 3, 4, 5, 6]
print (len(x))
```

You can say, call a list `x`, for example, and assign it to the numbers `1` through `6`, and these square brackets indicate that we are using a Python list, and those are immutable objects that I can actually add things to and rearrange as much as I want to. There's a built-in function for determining the length of the list called `len`, and if I type in `len(x)`, that will give me back the number `6` because there are 6 numbers in my list.

Just to make sure, and again to drive home the point that this is actually running real code here, let's add another number in there, such as `4545`. If you run this, you'll get `7` because now there are 7 numbers in that list:

```
x = [1, 2, 3, 4, 5, 6, 4545]
print (len(x))
```

The output of the previous code example is as follows:

```
7
```

Go back to the original example there. Now you can also slice lists. If you want to take a subset of a list, there's a very simple syntax for doing so:

```
x[3:]
```

The output of the above code example is as follows:

```
[1, 2, 3]
```

Pre colon

If, for example, you want to take the first three elements of a list, everything before element number 3, we can say `:3` to get the first three elements, `1`, `2`, and `3`, and if you think about what's going on there, as far as indices go, like in most languages, we start counting from 0. So element 0 is `1`, element 1 is `2`, and element 2 is `3`. Since we're saying we want everything before element 3, that's what we're getting.

So, you know, never forget that in most languages, you start counting at 0 and not 1.

Now this can confuse matters, but in this case, it does make intuitive sense. You can think of that colon as meaning I want everything, I want the first three elements, and I could change that to four just again to make the point that we're actually doing something real here:

```
x[:4]
```

The output of the above code example is as follows:

```
[1, 2, 3, 4]
```

Post colon

Now if I put the colon on the other side of the `3`, that says I want everything after `3`, so `3` and after. If I say `x[3:]`, that's giving me the third element, 0, 1, 2, 3, and everything after it. So that's going to return 4, 5, and 6 in that example, OK?

```
x[3:]
```

The output is as follows:

```
[4, 5, 6]
```

You might want to keep this IPython/Jupyter Notebook file around. It's a good reference, because sometimes it can get confusing as to whether the slicing operator includes that element or if it's up to or including it or not. So the best way is to just play around with it here and remind yourself.

Negative syntax

One more thing you can do is have this negative syntax:

```
x[-2:]
```

The output is as follows:

```
[5, 6]
```

By saying `x[-2:]`, this means that I want the last two elements in the list. This means that go backwards two from the end, and that will give me `5` and `6`, because those are the last two things on my list.

Adding list to list

You can also change lists around. Let's say I want to add a list to the list. I can use the `extend` function for that, as shown in the following code block:

```
x.extend([7,8])
x
```

The output of the above code is as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

I have my list of `1`, `2`, `3`, `4`, `5`, `6`. If I want to extend it, I can say I have a new list here, `[7, 8]`, and that bracket indicates this is a new list of itself. This could be a list implicit, you know, that's inline there, it could be referred to by another variable. You can see that once I do that, the new list I get actually has that list of `7`, `8` appended on to the end of it. So I have a new list by extending that list with another list.

The append function

If you want to just add one more thing to that list, you can use the `append` function. So I just want to stick the number `9` at the end, there we go:

```
x.append(9)
x
```

The output of the above code is as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Complex data structures

You can also have complex data structures with lists. So you don't have to just put numbers in it; you can actually put strings in it. You can put numbers in it. You can put other lists in it. It doesn't matter. Python is a weakly-typed language, so you can pretty much put whatever kind of data you want, wherever you want, and it will generally be an OK thing to do:

```
y = [10, 11, 12]
listOfLists = [x, y]
listOfLists
```

In the preceding example, I have a second list that contains `10`, `11`, `12`, that I'm calling `y`. I'll create a new list that contains two lists. How's that for mind blowing? Our `listofLists` list will contain the `x` list and the `y` list, and that's a perfectly valid thing to do. You can see here that we have a bracket indicating the `listofLists` list, and within that, we have another set of brackets indicating each individual list that is in that list:

```
[[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ], [10, 11, 12]]
```

So, sometimes things like these will come in handy.

Dereferencing a single element

If you want to dereference a single element of the list you can just use the bracket like that:

```
y[1]
```

The output of the above code is as follows:

```
11
```

So `y[1]` will return element `1`. Remember that `y` had `10`, `11`, `12` in it - observe the previous example, and we start counting from 0, so element 1 will actually be the second element in the list, or the number `11` in this case, alright?

The sort function

Finally, let's have a built-in sort function that you can use:

```
z = [3, 2, 1]
z.sort()
z
```

So if I start with list `z`, which is `3,2`, and `1`, I can call sort on that list, and `z` will now be sorted in order. The output of the above code is as follows:

```
[1, 2, 3]
```

Reverse sort

```
z.sort(reverse=True)
z
```

The output of the above code is as follows:

```
[3, 2, 1]
```

If you need to do a reverse sort, you can just say `reverse=True` as an attribute, as a parameter in that `sort` function, and that will put it back to `3`, `2`, `1`.

If you need to let that sink in a little bit, feel free to go back and read it a little bit more.

Tuples

Tuples are just like lists, except they're immutable, so you can't actually extend, append, or sort them. They are what they are, and they behave just like lists, apart from the fact that you can't change them, and you indicate that they are immutable and are tuple, as opposed to a list, using parentheses instead of a square bracket. So you can see they work pretty much the same way otherwise:

```
#Tuples are just immutable lists. Use () instead of []
x = (1, 2, 3)
len(x)
```

The output of the previous code is as follows:

```
3
```

We can say `x= (1, 2, 3)`. I can still use `length - len` on that to say that there are three elements in that tuple, and even though, if you're not familiar with the term `tuple`, a `tuple` can actually contain as many elements as you want. Even though it sounds like it's Latin based on the number three, it doesn't mean you have three things in it. Usually, it only has two things in it. They can have as many as you want, really.

Dereferencing an element

We can also dereference the elements of a tuple, so element number 2 again would be the third element, because we start counting from 0, and that will give me back the number `6` in the following screenshot:

```
y = (4, 5, 6)
y[2]
```

The output to the above code is as follows:

```
6
```

List of tuples

We can also, like we could with lists, use tuples as elements of a list.

```
listOfTuples = [x, y]
listOfTuples
```

The output to the above code is as follows:

```
[(1, 2, 3), (4, 5, 6)]
```

We can create a new list that contains two tuples. So in the preceding example, we have our `x` tuple of `(1, 2, 3)` and our `y` tuple of `(4, 5, 6)`; then we make a list of those two tuples and we get back this structure, where we have square brackets indicating a list that contains two tuples indicated by parentheses, and one thing that tuples are commonly used for when we're doing data science or any sort of managing or processing of data really is to use it to assign variables to input data as it's read in. I want to walk you through a little bit on what's going on in the following example:

```
 (age, income) = "32,120000".split(',')
print (age)
print (income)
```

The output to the above code is as follows:

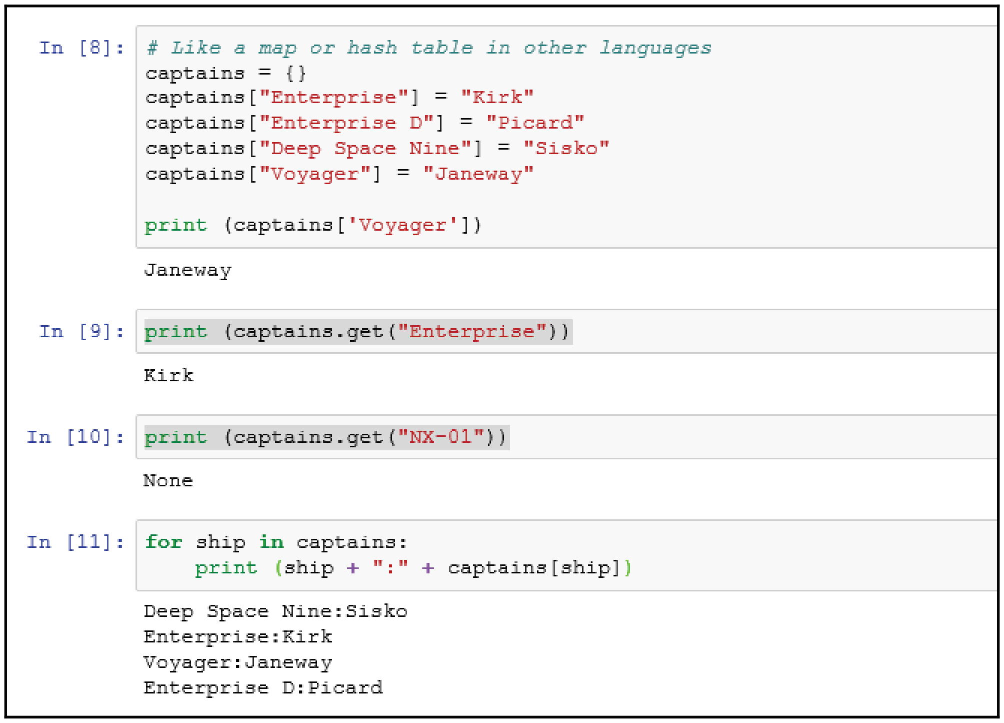
```
32
120000
```

Let's say we have a line of input data coming in and it's a comma-separated value file, which contains ages, say `32`, comma-delimited by an income, say `120000` for that age, just to make something up. What I can do is as each line comes in, I can call the `split` function on it to actually separate that into a pair of values that are delimited by commas, and take that resulting tuple that comes out of split and assign it to two variables-`age` and `income`-all at once by defining a tuple of age, income and saying that I want to set that equal to the tuple that comes out of the `split` function.

So this is basically a common shorthand you'll see for assigning multiple fields to multiple variables at once. If I run that, you can see that the `age` variable actually ends up assigned to `32` and `income` to `120,000` because of that little trick there. You do need to be careful when you're doing this sort of thing, because if you don't have the expected number of fields or the expected number of elements in the resulting tuple, you will get an exception if you try to assign more stuff or less stuff than you expect to see here.

Dictionaries

Finally, the last data structure that we'll see a lot in Python is a dictionary, and you can think of that as a map or a hash table in other languages. It's a way to basically have a sort of mini-database, sort of a key/value data store that's built into Python. So let's say, I want to build up a little dictionary of Star Trek ships and their captains:



```
In [8]: # Like a map or hash table in other languages
        captains = {}
        captains["Enterprise"] = "Kirk"
        captains["Enterprise D"] = "Picard"
        captains["Deep Space Nine"] = "Sisko"
        captains["Voyager"] = "Janeway"

        print (captains['Voyager'])

        Janeway

In [9]: print (captains.get("Enterprise"))

        Kirk

In [10]: print (captains.get("NX-01"))

        None

In [11]: for ship in captains:
             print (ship + ":" + captains[ship])

        Deep Space Nine:Sisko
        Enterprise:Kirk
        Voyager:Janeway
        Enterprise D:Picard
```

I can set up a `captains = {}`, where curly brackets indicates an empty dictionary. Now I can use this sort of a syntax to assign entries in my dictionary, so I can say `captains` for `Enterprise` is `Kirk`, for `Enterprise D` it is `Picard`, for `Deep Space Nine` it is `Sisko`, and for `Voyager` it is `Janeway`. Now I have, basically, this lookup table that will associate ship names with their captain, and I can say, for example, `print captains["Voyager"]`, and I get back `Janeway`.

A very useful tool for basically doing lookups of some sort. Let's say you have some sort of an identifier in a dataset that maps to some human-readable name. You'll probably be using a dictionary to actually do that look up when you're printing it out.

We can also see what happens if you try to look up something that doesn't exist. Well, we can use the `get` function on a dictionary to safely return an entry. So in this case, `Enterprise` does have an entry in my dictionary, it just gives me back `Kirk`, but if I call the `NX-01` ship on the dictionary, I never defined the captain of that, so it comes back with a `None` value in this example, which is better than throwing an exception, but you do need to be aware that this is a possibility:

```
print (captains.get("NX-01"))
```

The output of the above code is as follows:

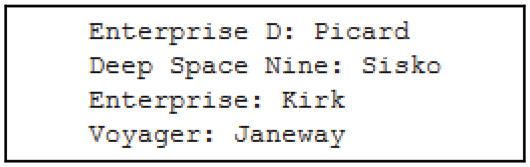
```
None
```

The captain is Jonathan Archer, but you know, I'm get a little bit too geeky here now.

Iterating through entries

```
for ship in captains:
     print (ship + ": " + captains[ship])
```

The output of the above code is as follows:



```
Enterprise D: Picard
Deep Space Nine: Sisko
Enterprise: Kirk
Voyager: Janeway
```

Let's look at a little example of iterating through the entries in a dictionary. If I want to iterate through every ship that I have in my dictionary and print out `captains`, I can type for `ship` in `captains`, and this will iterate through every single key in my dictionary. Then I can print out the lookup value of each ship's captain, and that's the output that I get there.

There you have it. This is basically the main data structures that you'll encounter in Python. There are some others, such as sets, but we'll not really use them in this book, so I think that's enough to get you started. Let's dive into some more Python nuances in our next section.

Python basics - Part 2

In addition to *Python Basics - Part 1*, let us now try to grasp more Python concepts in detail.

Functions in Python

Let's talk about functions in Python. Like with other languages, you can have functions that let you repeat a set of operations over and over again with different parameters. In Python, the syntax for doing that looks like this:

```
def SquareIt(x):
    return x * x
print (SquareIt(2))
```

The output of the above code is as follows:

```
4
```

You declare a function using the `def` keyword. It just says this is a function, and we'll call this function `SquareIt`, and the parameter list is then followed inside parentheses. This particular function only takes one parameter that we'll call `x`. Again, remember that whitespace is important in Python. There's not going to be any curly brackets or anything enclosing this function. It's strictly defined by whitespace. So we have a colon that says that this function declaration line is over, but then it's the fact that it's tabbed by one or more tabs that tells the interpreter that we are in fact within the `SquareIt` function.

So `def SquareIt(x):` tab returns `x * x`, and that will return the square of `x` in this function. We can go ahead and give that a try. `print squareIt(2)` is how we call that function. It looks just like it would be in any other language, really. This should return the number `4`; we run the code, and in fact it does. Awesome! That's pretty simple, that's all there is to functions. Obviously, I could have more than one parameter if I wanted to, even as many parameters as I need.

Now there are some weird things you can do with functions in Python, that are kind of cool. One thing you can do is to pass functions around as though they were parameters. Let's take a closer look at this example:

```
#You can pass functions around as parameters
def DoSomething(f, x):
    return f(x)
print (DoSomething(SquareIt, 3))
```

The output of the preceding code is as follows:

```
9
```

Now I have a function called `DoSomething`, `def DoSomething`, and it will take two parameters, one that I'll call `f` and the other I'll call `x`, and if I happen, I can actually pass in a function for one of these parameters. So, think about that for a minute. Look at this example with a bit more sense. Here, `DoSomething(f,x):` will return `f` of `x`; it will basically call the f function with x as a parameter, and there's no strong typing in Python, so we have to just kind of make sure that what we are passing in for that first parameter is in fact a function for this to work properly.

For example, we'll say print `DoSomething`, and for the first parameter, we'll pass in `SquareIt`, which is actually another function, and the number `3`. What this should do is to say do something with the `SquareIt` function and the `3` parameter, and that will return `(SquareIt, 3)`, and `3` squared last time I checked was `9`, and sure enough, that does in fact work.

This might be a little bit of a new concept to you, passing functions around as parameters, so if you need to stop for a minute there, wait and let that sink in, play around with it, please feel free to do so. Again, I encourage you to stop and take this at your own pace.

Lambda functions - functional programming

One more thing that's kind of a Python-ish sort of a thing to do, which you might not see in other languages is the concept of lambda functions, and it's kind of called **functional programming**. The idea is that you can include a simple function into a function. This makes the most sense with an example:

```
#Lambda functions let you inline simple functions
print (DoSomething(lambda x: x * x * x, 3))
```

The output of the above code is as follows:

```
27
```

We'll print `DoSomething`, and remember that our first parameter is a function, so instead of passing in a named function, I can declare this function inline using the `lambda` keyword. Lambda basically means that I'm defining an unnamed function that just exists for now. It's transitory, and it takes a parameter `x`. In the syntax here, `lambda` means I'm defining an inline function of some sort, followed by its parameter list. It has a single parameter, `x`, and the colon, followed by what that function actually does. I'll take the `x` parameter and multiply it by itself three times to basically get the cube of a parameter.

In this example, `DoSomething` will pass in this lambda function as the first parameter, which computes the cube of `x` and the `3` parameter. So what's this really doing under the hood? This `lambda` function is a function of itself that gets passed into the `f` in `DoSomething` in the previous example, and `x` here is going to be `3`. This will return `f` of `x`, which will end up executing our lambda function on the value `3`. So that `3` goes into our `x` parameter, and our lambda function transforms that into `3` times `3` times `3`, which is, of course, `27`.

Now this comes up a lot when we start doing MapReduce and Spark and things like that. So if we'll be dealing with Hadoop sorts of technologies later on, this is a very important concept to understand. Again, I encourage you to take a moment to let that sink in and understand what's going on there if you need to.

Understanding boolean expressions

Boolean expression syntax is a little bit weird or unusual, at least in Python:

```
print (1 == 3)
```

The output of the above code is as follows:

```
False
```

As usual, we have the double equal symbol that can test for equality between two values. So does `1` equal `3`, no it doesn't, therefore `False`. The value `False` is a special value designated by F. Remember that when you're trying to test, when you're doing Boolean stuff, the relevant keywords are `True` with a T and `False` with an F. That's a little bit different from other languages that I've worked with, so keep that in mind.

```
print (True or False)
```

The output of the above code is as follows:

```
True
```

Well, `True` or `False` is `True`, because one of them is `True`, you run it and it comes back `True`.

The if statement

```
print (1 is 3)
```

The output of the previous code is as follows:

```
False
```

The other thing we can do is use `is`, which is sort of the same thing as equal. It's a more Python-ish representation of equality, so `1 == 3` is the same thing as `1 is 3`, but this is considered the more Pythonic way of doing it. So `1 is 3` comes back as `False` because `1` is not `3`.

The if-else loop

```
if 1 is 3:
    print "How did that happen?"
elif 1 > 3:
    print ("Yikes")
else:
    print ("All is well with the world")
```

The output of the above code is as follows:

```
All is well with the world
```

We can also do `if-else` and `else-if` blocks here too. Let's do something a little bit more complicated here. If `1 is 3`, I would print `How did that happen?` But of course `1` is not `3`, so we will fall back down to the `else-if` block, otherwise, if `1` is not `3`, we'll test if `1 > 3`. Well that's not true either, but if it did, we print `Yikes`, and we will finally fall into this catch-all `else` clause that will print `All is well with the world`.

In fact, `1` is not `3`, nor is `1` greater than `3`, and sure enough, `All is well with the world`. So, you know, other languages have very similar syntax, but these are the peculiarities of Python and how to do an `if-else` or `else-if` block. So again, feel free to keep this notebook around. It might be a good reference later on.

Looping

The last concept I want to cover in our Python basics is looping, and we saw a couple of examples of this already, but let's just do another one:

```
for x in range(10):
 print (x),
```

The output of the previous code is as follows:

```
0 1 2 3 4 5 6 7 8 9
```

For example, we can use this range operator to automatically define a list of numbers in the range. So if we say `for x` in `range(10)`, `range 10` will produce a list of `0` through `9`, and by saying for `x` in that list, we will iterate through every individual entry in that list and print it out. Again, the comma after the `print` statement says don't give me a new line, just keep on going. So the output of this ends up being all the elements of that list printed next to each other.

To do something a little bit more complicated, we'll do something similar, but this time we'll show how `continue` and `break` work. As in other languages, you can actually choose to skip the rest of the processing for a loop iteration, or actually stop the iteration of the loop prematurely:

```
for x in range(10):
    if (x is 1):
 continue
 if (x > 5):
    break
 print (x),
```

The output of the above code is as follows:

```
0 2 3 4 5
```

In this example, we'll go through the values 0 through 9, and if we hit on the number 1, we will continue before we print it out. We'll skip the number 1, basically, and if the number is greater than `5`, we'll break the loop and stop the processing entirely. The output that we expect is that we will print out the numbers `0` through `5`, unless it's `1`, in which case, we'll skip number `1`, and sure enough, that's what it does.

The while loop

Another syntax is the while loop. This is kind of a standard looping syntax that you see in most languages:

```
x = 0
while (x < 10):
    print (x),
    x += 1
```

The output of the previous code is as follows:

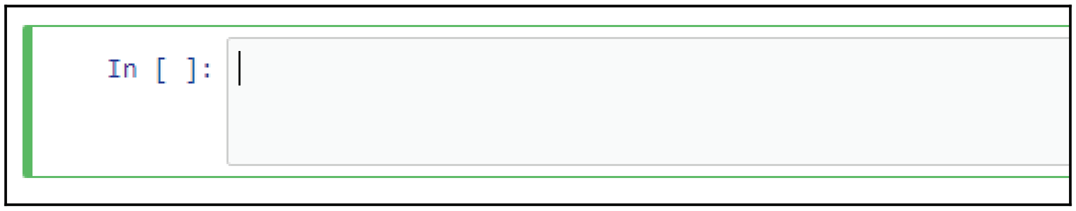
```
0 1 2 3 4 5 6 7 8 9
```

We can also say, start with `x = 0`, and `while (x < 10):`, print it out and then increment `x` by `1`. This will go through over and over again, incrementing x until it's less than 10, at which point we break out of the `while` loop and we're done. So it does the same thing as this first example here, but just in a different style. It prints out the numbers `0` through `9` using a `while` loop. Just some examples there, nothing too complicated. Again, if you've done any sort of programming or scripting before, this should be pretty simple.

Now to really let this sink in, I've been saying throughout this entire chapter, get in there, get your hands dirty, and play with it. So I'm going to make you do that.

Exploring activity

Here's an activity, a little bit of a challenge for you:



Here's a nice little code block where you can start writing your own Python code, run it, and play around with it, so please do so. Your challenge is to write some code that creates a list of integers, loops through each element of that list, pretty easy so far, and only prints out even numbers.

Now this shouldn't be too hard. There are examples in this notebook of doing all that stuff; all you have to do is put it together and get it to run. So, the point is not to give you something that's hard. I just want you to actually get some confidence in writing your own Python code and actually running it and seeing it operate, so please do so. I definitely encourage you to be interactive here. So have at it, good luck, and welcome to Python.

So that's your Python crash course, obviously, just some very basic stuff there. As we go through more and more examples throughout the book, it'll make more and more sense since you have more examples to look at, but if you do feel a little bit intimidated at this point, maybe you're a little bit too new to programming or scripting, and it might be a good idea to go and take a Python revision before moving forward, but if you feel pretty good about what you've seen so far, let's move ahead and we'll keep on going.

Running Python scripts

Throughout this book, we'll be using the IPython/Jupyter Notebook format (which are `.ipynb` files) that we've been looking at so far, and it's a great format for a book like this because it lets me put little blocks of code in there and put a little text and things around it explaining what it's doing, and you can experiment with things live.

Of course, it's great from that standpoint, but in the real world, you're probably not going to be using IPython/Jupyter Notebooks to actually run your Python scripts in production, so let me just really briefly go through the other ways you can run Python code, and other interactive ways of running Python code as well. So it's a pretty flexible system. Let's take a look.