python™ for Scientists and Engineers

Shantnu Tiwari

Python for Scientists and Engineers

Shantnu Tiwari

This book is for sale at http://leanpub.com/pythonforengineers

This version was published on 2021-06-23

Contents

Introduction

Hello, and thanks for reading my book. I hate books with long intros written just to pad the word count, so I will dive right in. We'll start with something that should be simple, but still stumps a lot of people.

Installing Python

You might think installing Python is easy, but you'd be wrong. The problem is, for scientific computing, many libraries like Numpy/Scipy (which I'll discuss below) are written in Fortran or C. That's because they were ported from old libraries written way back.

Most people you will find online, on places like Stack overflow, Reddit etc are web developers. They live in a highly sterilised environment, where every library they use is written in pure Python, and so easy to install. So if you get stuck installing Python and ask them questions, you will not get very helpful answers, and may even get insults.

When writing this book, I looked for many solutions to this problem. The best one I found is the Anaconda Python distribution[1]. Especially if you are on Windows, I recommend using this distribution. It comes with most engineering libraries pre-installed (the notable exception is OpenCv).

If you get stuck with libraries, do ask for help.

Important Python libraries

1. Scipy / Numpy - Scipy stands for Scientific Python. This is a mega project that contains everything under the sun for scientific computing. Scipy added a few things to Python. For example, normal Python only has a integer datatype. However, those of you from a low level or engineering backgrounds may know integers can be 8, 16, 32 or 64 bit. And then you have floating point numbers, usually stored as 64 bit. These additions were very useful in Python, but Scipy was a huge project, so a decision was made to extract some parts of it into a library called Numpy.

Numpy is what we'll mostly use. 90% of the time, Numpy has the functions we need. Now and then, Numpy will not have the functionality required and then we look into Scipy (which has everything you could possibly think of, but may require some Googling).

Keep this in mind. Many people get confused between Scipy and Numpy, not realising they are the same thing.

An important library in Scipy we will use often is Matplotlib (or Pylab). This is a graphing library that was originally based on Matlab (as indeed, most of Scipy). For those who haven't heard of it, Matlab is one the most used packages for scientific computing. It is also super expensive with a complicated licensing scheme which means most companies buy limited copies. The only place you

[1]http://continuum.io/downloads

will find Matlab installed commonly is Universities. But many scientists / teachers are moving to Python (or other open source tools like R), as they want the freedom to play with code in their own time, for their own hobby projects, without having to pay thousands in licensing. The advantage of open source tools is that students can install them on their own laptops.

2 Pandas

Pandas is a fairly recent addition to the Python world. It was originally written to replace the functionality of R, and is used mainly in the financial world for statistical computing.

For a long time, I avoided using Pandas, as I figured (wrongly) that everything I could do in Pandas, I could do with Numpy. Which is technically correct, as Pandas is built on top of Scipy/Numpy.

But once I actually tried it, I loved it. Numpy is okay if you are working with simple text or csv files. But as soon as you start working with Microsoft Excel or other large complicated files, there is too much low level work in Numpy. Pandas takes care of that for you. Nowadays, I prefer to use Pandas, even for reading simple csv, just for all the power it offers us.

If you used the Anaconda distribution I recommend, Scipy and Pandas will be preinstalled. On Liunx Mint, Scipy was installed, but I had to install Pandas myself.

3 OpenCv

These are both libraries for image and video processing. OpenCv is the older one, and has been around forever, though the Python version is comparatively recent (you will find most examples for OpenCv in C/C++, though Python usage is increasing).

4 Other libraries

These are some other libraries you should know about.

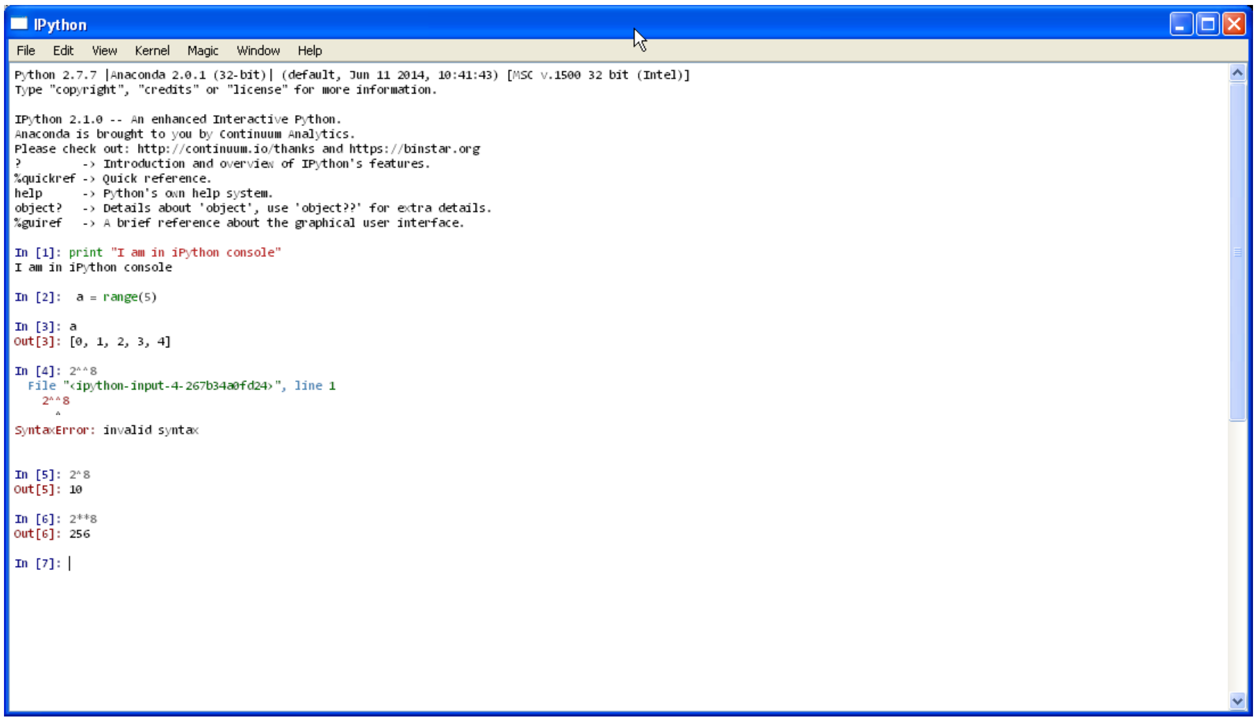
Virtualenv (Virtual environment) is a great library for installing different versions of libraries on the same machine. If you ever had code that only worked with v0.2 of some library, Virtualenv will create a special Python install that comes with that version only. You can have dozens of versions of the same library, with none of them clashing with each other.

iPython

For a long time, I used IDLE, which is the default GUI Python client. Many people prefer GUIs, and IDLE sort of works.

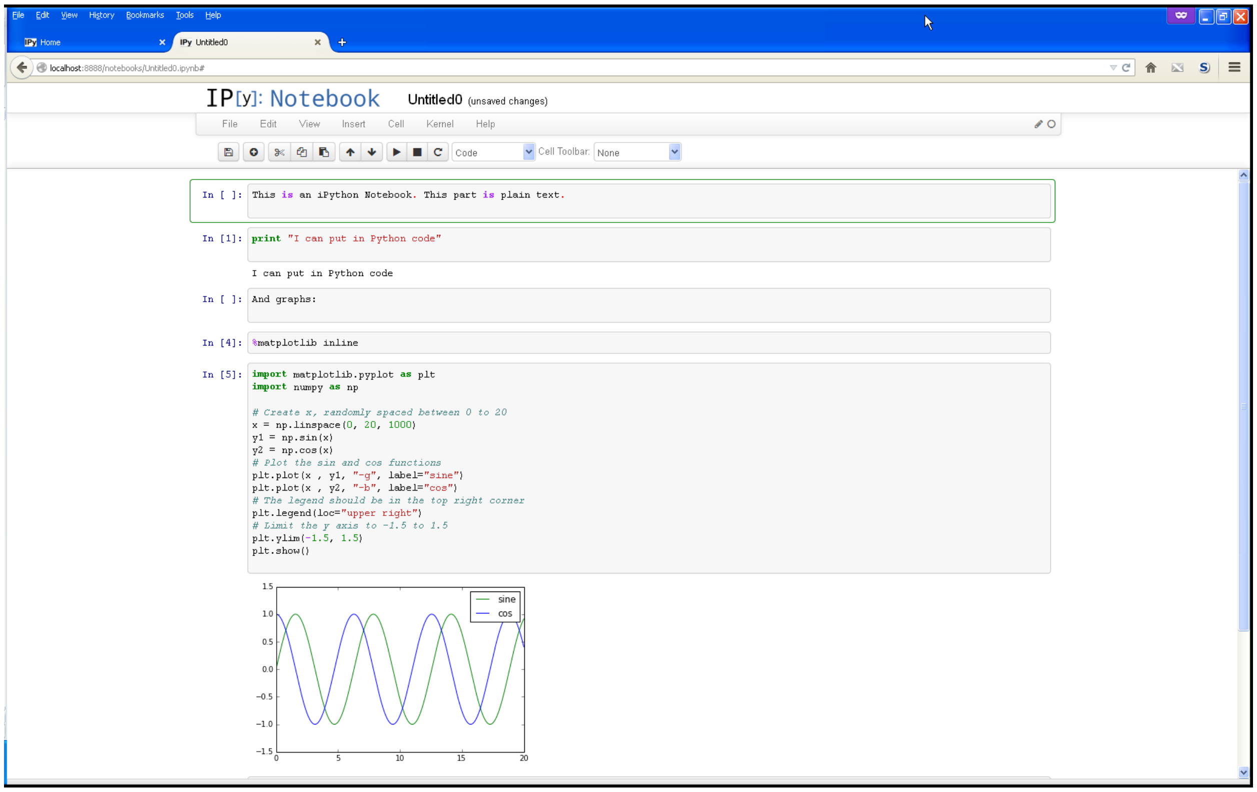
Sort of. It's very clunky and ugly. If you press the up key, it doesn't take you to the last command. Instead, it just moves the cursor up. If you want to run the last command, you have to press up multiple times till the cursor touches the command you want and press enter. And lord help you if you want to change the directory or list the files in the current directory. You have to go trawling through the *os* library commands, and even then you may not get what you want.

Which is why when I first used iPython, I was blown away. It is a lot easier and intuitive to use than IDLE, you can use standard Linux commands to move around (ls, pwd, cd), you can run python scripts online, just to mention a few features. I normally use IPython Qt Console version:



```
Python 2.7.7 |Anaconda 2.0.1 (32-bit)| (default, Jun 11 2014, 10:41:43) [MSC v.1500 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 2.1.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]: print "I am in iPython console"
I am in iPython console

In [2]:  a = range(5)

In [3]: a
Out[3]: [0, 1, 2, 3, 4]

In [4]: 2^^8
  File "<ipython-input-4-267b34a0fd24>", line 1
    2^^8
      ^
SyntaxError: invalid syntax


In [5]: 2^8
Out[5]: 10

In [6]: 2**8
Out[6]: 256

In [7]: |
```

There is another cool thing called IPython notebook, that allows you to put text, graphics and Python code in a web based file that you can share with others. This is great for writing tutorials, as the user doesn't have to copy paste code. They can run the notebook directly, and see the explanation and code running at same time.


File Edit View History Bookmarks Tools Help
Home
Untitled0
localhost:8888/notebooks/Untitled0.ipynb#
IP[y]: Notebook
Untitled0 (unsaved changes)
File Edit View Insert Cell Kernel Help
Code
Cell Toolbar: None
sine
cos

```
In [ ]: This is an iPython Notebook. This part is plain text.

In [1]: print "I can put in Python code"

        I can put in Python code

In [ ]: And graphs:

In [4]: %matplotlib inline

In [5]: import matplotlib.pyplot as plt
        import numpy as np

        # Create x, randomly spaced between 0 to 20
        x = np.linspace(0, 20, 1000)
        y1 = np.sin(x)
        y2 = np.cos(x)
        # Plot the sin and cos functions
        plt.plot(x , y1, "-g", label="sine")
        plt.plot(x , y2, "-b", label="cos")
        # The legend should be in the top right corner
        plt.legend(loc="upper right")
        # Limit the y axis to -1.5 to 1.5
        plt.ylim(-1.5, 1.5)
        plt.show()
```

I won't use the notebook much, but I will use the qt console version. This is what iPython output will look like:

```
In [1]: print "I am in iPython console"
I am in iPython console

In [2]:  a = range(5)

In [3]: a
Out[3]: [0, 1, 2, 3, 4]



In [6]: 2**8
Out[6]: 256
```

In is what I have entered, *Out* is the output.

Seriously, give IPython a try. It's so great I could have written the whole book in iPython. I will save the notebook in the figure above in the source code as first_try.ipynb (under plotting), in case you want to play around.

Source Code

You should get all the source code for the book. That's because I don't want you to waste time typing code.

I recommend you open the code with your favourite code editor along with the book, and run the code as you are reading it. If you really want to understand the code, play with it. Try changing it, try to break it, add more prints to see how each line works. Don't worry about corrupting the code, as you can download the latest version anytime. It's available on my Github account[2]. You don't even need to install Git- if you look at the right side, there is a *Download zip* option– just download the latest code that way.

[2] https://github.com/shantnu/PyEng

Create a Word Counter in Python

This chapter is for those new to Python, but I recommend everyone go through it, just so that we are all on equal footing.

Baby steps: Read and print a file

Okay folks, we are going to start gentle. We will build a simple utility called word counter. Those of you who have used Linux will know this as the wc utility. On Linux, you can type:

```
wc <filename>
```

to get the number of words, lines and characters in a file. The wc utility is quite advanced, of course, since it has been around for a long time. We are going to build a baby version of that. This is more interesting than just printing *Hello World* to the screen.

With that in mind, let's start. The file we are working with is read_file.py, which is in the folder Wordcount.

```
#!/usr/bin/python
```

The first line that starts with a *#!* is used mainly on Linux systems. It tells the shell that this is a Python file, and should be run as such. It also tells Linux which interpreter to use (Python in our case). It doesn't do any harm on Windows (as anything that starts with a # is a comment in Python), so we keep it in.

Let's start looking at the code.

```
f = open("birds.txt", "r")
```

We simply open a file called "birds.txt". It must exist in the current directory (ie, the directory you are running the code from). Later on, we will cover reading from the command line, but for now, the path is hard coded. The *r* means the file will be opened in a read only mode. Other common modes are *w* for write, *a* for append. You can also read/write binary files, however we won't go into that for the moment. Our files are plain text.

```
data = f.read()
f.close()
```

After opening the file, we read its contents into a variable called data, and close the file.

```
print(data)
```

And we print the file. And now, to test our code. If you are on Linux, you can just type:

```
./read_file.py
```

to run it. You might have to make the file executable. On Windows, you'll need to do:

```
python read_file.py
```

Go to the folder called WordCount, and run the file there:

```
WordCount$ ./read_file.py
STRAY BIRDS
BY
RABINDRANATH TAGORE

STRAY birds of summer come to my
window to sing and fly away.

And yellow leaves of autumn, which
have no songs, flutter and fall there
with a sigh.
WordCount$
```

And there you go. Your First Python program.

Count words and lines

Okay, so we can read a file and print it on the screen. Now, to count the number of words. We'll be using the file count_words.py in the WordCount folder.

```
#! /usr/bin/python

f = open("birds.txt", "r")
data = f.read()
f.close()
```

These lines should be familiar by now. We open the file and read it.

```
words = data.split(" ")
```

Python has several in built functions for strings. One is the *split()* function which splits the string on the given parameter. In the example above, we are splitting on a space. The function returns a list (which is what Python calls arrays) of the string split on space.

To see how this works, I'll fire up an IPython console.

```
In [1]: "I am a boy".split(" ")

Out[1]: ['I', 'am', 'a', 'boy']
```

I took a sentence "I am a boy" and split it on a space. Python returned a list with four elements: *['I', 'am', 'a', 'boy']*. We can split on anything. Here, I split on a comma:

```
In [2]: "The birds, they are flying away, he said.".split(",")

Out[2]: ['The birds', ' they are flying away', ' he said.']
```

Coming back to our example:

```
words = data.split(" ")
```

You should know what we are doing now. We are splitting the file we read on spaces. This should give us the number of words, as in English, words are separated by space (as if you didn't know already).

```
print("The words in the text are:")
print(words)
num_words = len(words)
print("The number of words is ", num_words)
```

So we print the words that we found. Next, we call the *len()* function, which returns the length of a list. Remember I said the *split()* function breaks the string into a list? Well, by using the *len()* function, we can find out how many elements the list has, and hence the number of words.

Next, we find the number of lines by using the same method.

```
lines = data.split("\n")
print("The lines in the text are:")
print(lines)
print("The number of lines is", len(lines))
```

We do the same thing, except this we split on the newline character ("\n"). For those who don't know, the newline character is the code that tells the editor to insert a new line, a return. By counting the number of newline characters, we can get the number of lines in the program.

Run the file *count_words.py*, and see the results.

```
$WordCount [master]> python .\count_words.py

The words in the text are:
['STRAY', 'BIRDS', '\nBY', '\nRABINDRANATH', 'TAGORE', '\n\nSTRAY', 'birds', 'of', '\
summer', 'come', 'to', 'my', '\nwindow', 'to', 'sing', 'and', 'fly', 'away.', '\n\nA\
nd', 'yellow', 'leaves', 'of', '
autumn,', 'which', '\nhave', 'no', 'songs,', 'flutter', 'and', 'fall', 'there', '\nw\
ith', 'a', 'sigh.']

The number of words is  34

The lines in the text are:
['STRAY BIRDS ', 'BY ', 'RABINDRANATH TAGORE ', '', 'STRAY birds of summer come to m\
y ', 'window to sing and fly away. ', '', 'And yellow leaves of autumn, which ', 'ha\
ve no songs, flutter and fall th
ere ', 'with a sigh.']

The number of lines is 10
```

Now open the file *birds.txt* and count the number of lines by hand. You'll find the answers are different. That's because there is a bug in our code. It is counting empty lines as well. We need to fix that now.

Count lines fixed

```
#! /usr/bin/python

f = open("birds.txt", "r")
data = f.read()
f.close()

lines = data.split("\n")
print("Wrong: The number of lines is", len(lines))
```

This is the old code, the one we need to correct.

For loop in Python

The syntax of the for loop is:

```
for <counter> in <list / array>:
    <do something>
```

A few key things. There is a colon (:) after the for instruction. And in Python, there are no brackets *{}* or start-end keywords. For example, if you come from a C/C++/Java/C# type world, this is how you would write your for loop:

```
for(i=0; i <10; i++)
{
<do something here>
}
```

The curly braces *{}* tell the compiler that this code is under the for loop. Python doesn't have these braces. Instead, it uses white space/indentation. If you don't use indentation, Python will complain. Example:

```
for i in range(5):
print(i)
  File "<ipython-input-16-c04bfe0468c5>", line 2
    print(i)
        ^
IndentationError: expected an indented block
```

The correct way to do it is:

```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

How much indentation to use? Four spaces is recommended. If you are using a good text editor like Sublime Text, it will do that automatically. Coming back to our code,

```
for l in lines:
    if not l:
        lines.remove(l)
```

Let's go over this line by line.

```
for l in lines:
```

We are looping over our list *lines. l* will contain each line as Python is looping over them.

As a side note, those of you who come from a C/C++ background, you will be surprised by us not using arrays. We don't need to— Python will do that for us automatically. Python will take the list *lines* and automatically loop over it. We don't need to do *lines[0]*, *lines[1]*, *lines[2]* etc like you would do in C/C++. In fact, doing so is an anti-pattern.

So now we have each line. We need to now check if it is empty. There are many ways to do it. One is:

```
if len(l) == 0:
```

This checks if the current line has a length of `0`, which is fine, but there is a more elegant way of doing it.

```
if not l:
    lines.remove(l)
```

The *not* keyword in Python will automatically check for emptiness for us. If the line is empty, we remove it from the list using the *remove()* command.

Again, like the for loop, we need to give four spaces to let Python know that this instruction is under the if condition.

We should now have the correct number of lines. Run *count_lines_fixed.py* to see the results.

```
$WordCount [master]> python .\count_lines_fixed.py

Wrong: The number of lines is 10
Right: The number of lines is 8
```

Bringing it all together

Now we need to tie it all together. *word_count.py* is our final file.

```
#! /usr/bin/python
import sys
```

The only new thing here is the *import sys* command. This is needed to read from the command line.

We will beak our code into functions now. The way to write a function in Python is:

```
def foo(<input>):
    <do something>
    return <value>
```

def defines a function. Notice the colon (:) and the white space? Like loops and *if* conditions, you need to use indentation for code under the for loop.

Our first function counts the number of words:

```
def count_words(data):
    words = data.split(" ")
    num_words = len(words)
    return num_words
```

It takes in the list data, and returns the number of words. Keep in mind this is the exact same code as before, the only difference is now it is in a function.

The function to count lines is similar:

```
def count_lines(data):
    lines = data.split("\n")
    for l in lines:
        if not l:
            lines.remove(l)

    return len(lines)
```

The next part is one of the most Googled lines:

```
if __name__ == "__main__":
```

There are two ways to call Python files:

1. You can call the file directly, *python filename*, which is what we've been doing.
2. You can call the file as an external library.

We haven't covered calling the file as a library yet. If you wanted to use the function *count_words* in another file, you would do this:

```
from word_count import count_words
```

This will take the function count_words and make it available in the new file. You can also do:

```
from word_count import *
```

This will import all functions and variables, but generally, this approach isn't recommended. You should only import what you need.

Now, sometimes you'll have code you only want to run if the file is being called directly, ie, without an import. If so, you can put it under this line:

```
if __name__ == "__main__":
```

This means (in simple English): Only run this code if I am running this file from the command line (or something similar). If we import this file in another, all of this code will be ignored.

By using this syntax, you can ensure your function only runs when someone calls your program directly, and not imports is as a module.

__*name*__ is an an internal variable set to __*main*__ by the Python interpreter when we are running the program standalone.

Now in our examples, we have been calling the files directly, but I thought I'd show you this syntax in case you ever saw it on the web. It is optional in our case, but good practice in case you want to turn your code into a library. Even if you don't want to at the moment, it's a good idea to use the command as it's only one line.

```
    if len(sys.argv) < 2:
        print("Usage: python word_count.py <file>")
        exit(1)
```

Remember we imported the sys library? It contains several system calls, one of which is the sys.argv command, which returns the command line arguments. You already know our old friend *len()*. We check if the number of command line arguments is less than two (the first is always the name of the file), and if so, print a message and exit. This is what happens:

```
WordCount> python .\word_count.py
Usage: python word_count.py <file>
```

The next line:

```
    filename = sys.argv[1]
```

As I said, the first element of sys.argv (or argv[0]) will be the name of the file itself (word_count.py in our case). The second will be the file the user entered. We read that.

```
    f = open(filename, "r")
    data = f.read()
    f.close()
```

We read the data from the file.

```
    num_words = count_words(data)
    num_lines = count_lines(data)

    print("The number of words: ", num_words)
    print("The number of lines: ", num_lines)
```

And now we call our functions to count the number of words and lines, and print the results. Voila! A simple word counter.

```
WordCount> python .\word_count.py .\birds.txt
The number of words:  34
The number of lines:  8
```

The word counter isn't perfect, and if you try it with different files, you will find many bugs. But it's good enough for us to move on to the next chapter.

An introduction to Numpy and Matplotlib

List comprehensions

A very important part of Python is list comprehensions. They will come up again and again in any example you see online, so I thought I'd go over them separately. They are technically not a part of Numpy, but I'll introduce them here, for reasons you will understand.

Move to the *Plotting* folder and open up *list_comp.py*.

```
#! /usr/bin/python
import numpy as np
```

The new thing here is we are importing numpy. We import it as np to save typing numpy each time. This is a common pattern in Python, though it is optional. You could just import numpy as it is.

```
x = [5,10,15,20,25]

# declare y as an empty list
y = []
```

We initiliase *x* with five values, and *y* as an empty list. What we want to do is fill *y* with the values of x divided by 5. The boring way, the way you'll most likely use if you come from a C/C++ background is:

```
for counter in x:
    y.append(counter / 5)
```

We loop over x, and for each value in x, divide it by 5 and add it to *y*.

```
print("\nOld fashioned way: x = {} y = {} \n".format(x, y))
```

We print the values. The only new thing here is the *format()* command. The values to be printed go inside *{}*, and the actual values go inside *format()*. Here are the results:

```
$Plotting [master]>  python .\list_comp.py

Old fashioned way: x = [5, 10, 15, 20, 25] y = [1.0, 2.0, 3.0, 4.0, 5.0]
```

But this method is not how it's done in Python. There is a much easier way called list comprehensions.

List comprehensions are used when we are working with lists. One of the most common tasks is to take a list, do something with it, and create a new list. Because this is so common, an easy way to work with lists was created, rather than having to loop all the time.

The basic format of list comprehension is:

```
[ <do something> for value in list]
```

So for our example, where we want to divide x by 5:

```
z = [n/5 for n in x]
print("List Comprehensions: x = {} z = {} \n".format(x, z))
```

Result:

```
List Comprehensions: x = [5, 10, 15, 20, 25] z = [1.0, 2.0, 3.0, 4.0, 5.0]
```

We use a new variable z. Let's break the list comprehension:

```
[for n in x]
```

This is our normal for loop. We loop over all the values in the list *x*.

```
[n/5 for n in x]
```

For each value *n* in *x*, divide *n* by 5. The square brackets ([]) will convert it to a list. If you compare it with the previous example, you have replaced three lines with one. And this is a very simple example. As a simple rule, any time you want to work with lists to form lists, use list comprehensions (unless a better way is available, as we'll see!).

Why use a list in the 1st place?

But why can't you just divide *x* by 5 directly? Let's try it:

```
try:
    a = x / 5
except:
    print("No, you can't do that with regular Python lists\n")
```

```
No, you can't do that with regular Python lists
```

This is the Python try-except block. It tries to run your code and if it finds an exception, it hits the except block.

In this case, you can’t divide x by 5, so you will hit the exception.

Well, actually you can, as long as you use numpy arrays.

```
a = np.array(x)
b = a / 5

print("With Numpy: a = {} b = {} \n".format(a, b))
```

We convert our *x* to a numpy array with the *np.array* command. And now, we can directly divide *a* by 5.

Try out the code now and see the results. Play with it if you like.

A few final words. I’m not trying to say numpy arrays are better than list comprehensions. They are not, and in most cases, you would stick to list comprehensions. But numpy arrays give us so much power, we don’t need list comprehensions in many cases. I’ve shown you both ways, so you can choose the best one for yourself.

Plotting Sin and Cos waves

Okay. Let’s plot some sine and cosine ways. You all remember the formula’s from school right? No? Then get out of my class.

Just kidding. You don’t need to remember formulas. That’s what Python is for, right?

In this example, we are going to plot a few simple sin and cos graphs, getting an introduction to Python’s plotting library, Matplotlib.

```
#! /usr/bin/python
import numpy as np
import matplotlib.pyplot as plt
```

The new thing is: *import matplotlib.pyplot as plt.* We are importing it as *plt* to save typing. Another thing to note is: Matplotlib is a huge library. We are only importing the *pyplot* part of it. This is useful to save memory and speedup code. Otherwise you’ll be importing gigabytes of libraries everytime you want to print *Hello World* on the screen.

```
# Create x, evenly spaced between 0 to 20
x = np.linspace(0, 20, 1000)
```

Okay, let me remind you what I said earlier. Learning to use Python well means using a lot of libraries and functions. But you don't have to remember them all. I still forget simple things like how to read files. Learning how to Google is one of the most important parts of being a software engineer.

With that in mind, I'm using the numpy *linspace* function. And yes, I misspell it as *linespace* every single time.

Anyway, *linspace* generates evenly spread out values. In the example above, it will generate 1000 values between 0 and 20. Printing 1000 values will take a lot of space here, so let's see what happens when we generate only 10 values:

```
In [2]: np.linspace(0,20,10)
Out[2]:
array([  0.        ,   2.22222222,   4.44444444,   6.66666667,
         8.88888889,  11.11111111,  13.33333333,  15.55555556,
        17.77777778,  20.        ])
```

As you can see, it has generated 10 values evenly spread between 0 to 20.

The numbers are floating point by default, which is good for us. Coming back to our code:

```
y1 = np.sin(x)
y2 = np.cos(x)
```

We take our x and calculate sin and cos values for it. Now to graph it.

```
# Plot the sin and cos functions
plt.plot(x , y1, "-g", label="sine")
plt.plot(x , y2, "-b", label="cos")
```

The first two values are the x and y axis values.

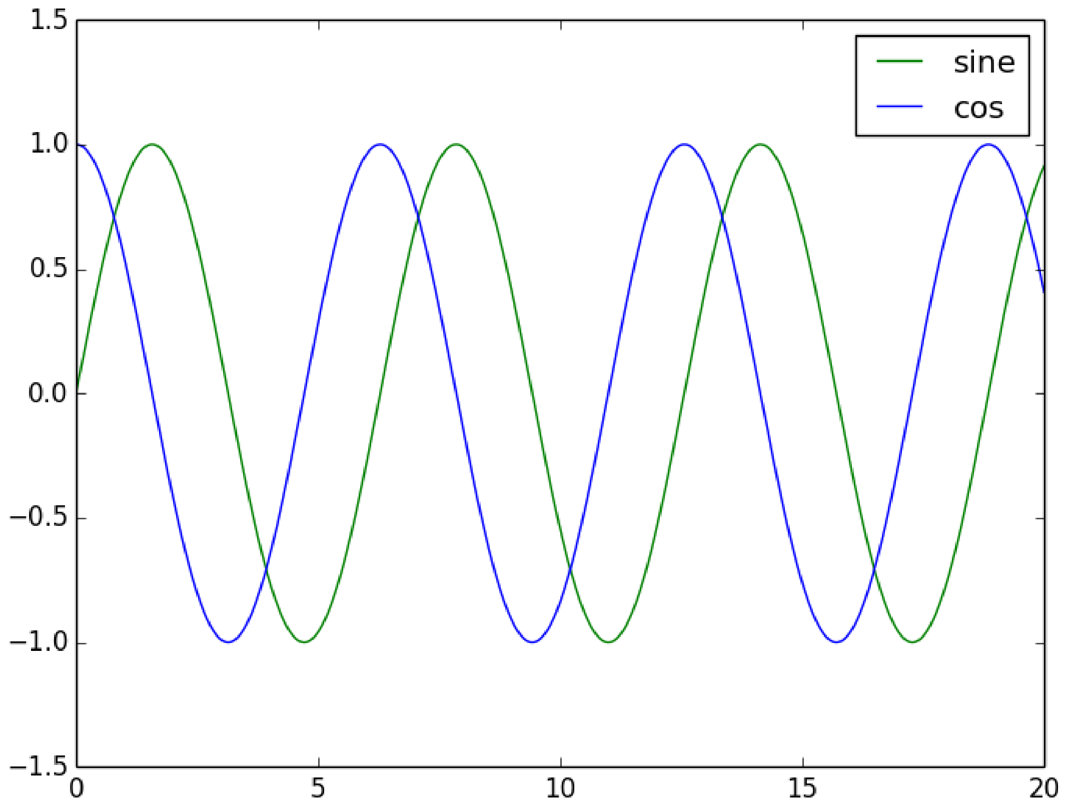
The third value is the color. "-g" for green, "-b" for blue. You can also have "-r" for red, and Google for more. Finally, we have the label, which will show up in the legend. Speaking of legends:

```
# The legend should be in the top right corner
plt.legend(loc="upper right")
```

We want the legend to be on the upper right corner, which in Matplotlib uses the simple English terms "upper right". Can you guess what the instruction would be if you wanted it on lower left?

```
# Limit the y axis to -1.5 to 1.5
plt.ylim(-1.5, 1.5)
plt.show()
```

The first part is optional. The graph looked too squeezed, so I set the limits of the y axis to be between -1.5 and 1.5. Try to comment out the code and see what happens. Finally, we show the graph.



Note the legend on the upper right corner, with the right labels.

Plotting Salary vs Names

In this exercise, we are going to read data from a file and plot it. I have two files: *names.txt* that contains names, and *salaries.txt* that contains, surprise, salaries. Let's look at the files. I'm using the Linux *cat* command which just prints the file contents on the screen.

```
$> cat names.txt

Fluffy, John, Matt, James, Sarah, Jessica, Rupert, Pablo, Mr T, Bill Gates

$> cat salaries.txt

0, 100, 200, 500, 1000, 1200, 1800, 1850, 5000, 10000
```

The data in the two files is linked. So Fluffy has a salary of 0, John of 100 and so on.

I'm using two different files to show different ways of reading data.

```
#! /usr/bin/python
import numpy as np
import matplotlib.pyplot as plt
```

Nothing new here. Let's go ahead.

```
salary = np.fromfile("salaries.txt", dtype=int, sep=",")
```

One of the main powers of numpy arrays, that makes them so much better than normal Python lists, is that it allows different types of number data types. Python lists are normally used for strings. While they can store numbers, they aren't really optimised for numerical processing.

Numpy arrays are. You can store data as 8, 16 or 32 bits. You can choose to use integers or floats. Numpy handles all the conversion and processing internally.

In the line above, I'm setting *dtype=int*. This tells numpy that this is an integer. You could use int8 for 8bit, int16 for 16bit, uint16 for unsigned int 16 bit and so on. All the possible options are available in the documentation online. Just search for "numpy dtypes".

I just use an int, as I don't particularly care for this example. The last line in the instruction is *sep=","*, which tells Numpy that the data in the file is separated by commas. The data is in this format:

```
0, 100, 200, 500, 1000, 1200, 1800, 1850, 5000, 10000
```

It tells Numpy to separate the data in the array by commas. If the data was like this:

```
0 : 100 : 200: 500
```

you would use *sep=":"*. Now, on to the next line.

```
names = np.genfromtxt("names.txt", dtype='str', delimiter=",")
```

Now we are reading names. The *np.fromfile()* function doesn't work so well with text, so I'm using the *genfromtxt* function. It is similar to the previous function.

The *dtype* is string. *delimiter* is the same as *sep* in the previous example.

Now we need to plot the names vs the salaries. A small detail. You can't really plot the names on the x axis, as the x axis has to be a number (actually you can, as you'll see in the later chapter on Pandas. But I'll show this method for now, as it stumps a lot of beginners). However, you can work around that.

The first thing you do is create a variable called x that will contain numbers for each of the names:

```
x = np.arange(len(names))
```

The numpy arange (take care, it's arange not arrange) generates a list of numbers starting from 0.

```
In [3]: np.arange(5)
Out[3]: array([0, 1, 2, 3, 4])
```

In our code, *x* contains a list of numbers from 0 to the number of names.

```
plt.bar(x, salary)
```

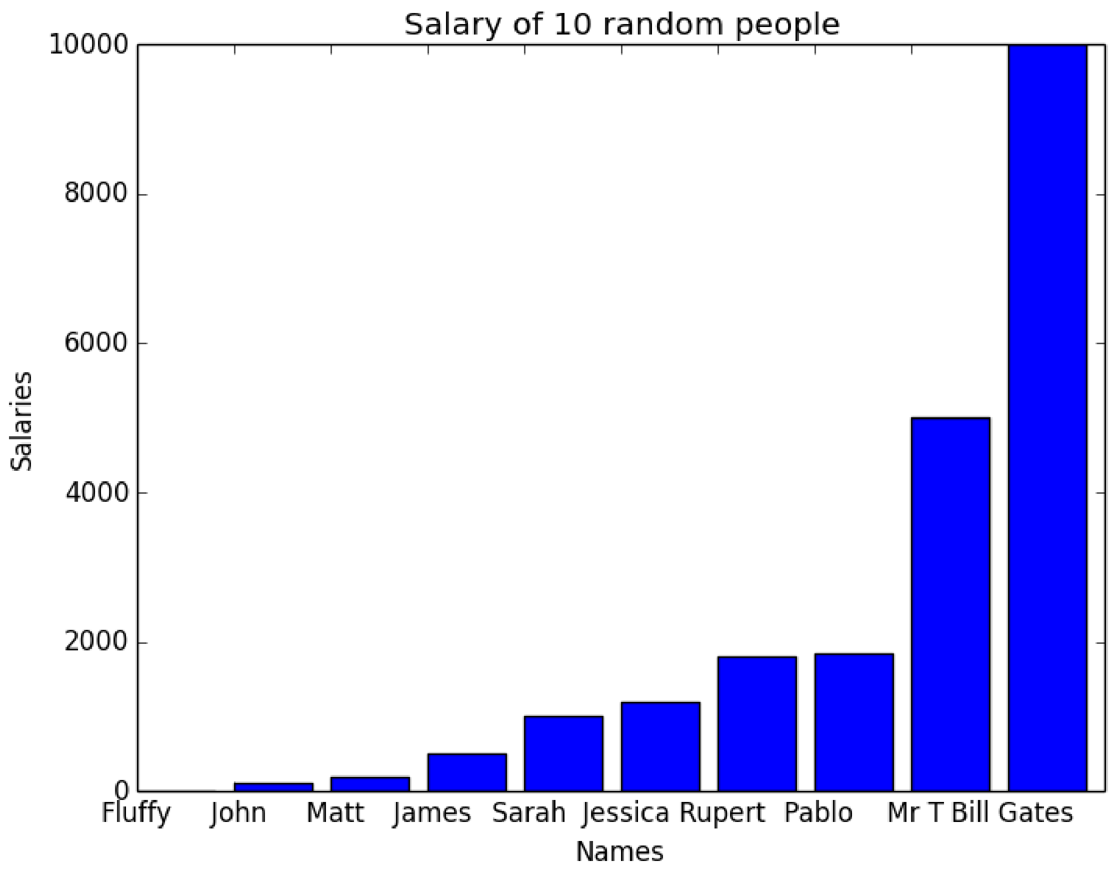
So we plot x vs salary. We are using a bar graph here.

```
plt.xticks(x, names)
```

This line uses something called xticks. All it does it is replace the numbers in x with names. Because we can only plot against numbers, we had to use this round the way approach.

```
plt.ylabel("Salaries")
plt.xlabel("Names")
plt.title("Salary of 10 random people")
plt.show()
```

We set the x and y labels here, as well as add a title. We finally display the graph.



```
print(np.max(salary), np.min(salary), np.average(salary), np.median(salary))
```

```
10000 0 2165.0 1100.0
```

I have added this line just to show that numpy supports many functions like max and min (the maximum and minimum values in the array), average and median. If you are looking for a special mathematical function, it's always worth Googling to find out if it already exists.

If you look at the graph above, you will realise there is a problem. The upper two values are so large, they are suppressing everything else. We can't get clear results from the graph. And this gives me a chance to show you another cool feature of Python. Firing up iPython:

```
a = range(5)

a
Out[2]: [0, 1, 2, 3, 4]
```

We create a list *a* that contains values from 0 to 4. Now, you know you can see any one element of a list by doing something like *a[2]*. But you can also do:

```
a[:2]
Out[3]: [0, 1]
```

This says, starting at the 0th element, show all elements up to the second. Hence we get *0, 1*.

We can also do:

```
a[2:]
Out[4]: [2, 3, 4]
```

Starting at the 2nd element, show all elements after that. Hence we get *2, 3, 4*.

The general format is:

a[start value : end value]

Either *start value* or *end value* can be empty, in which case it will start at the beginning or go to the end, respectively. We can also give both values:

```
a[2:4]
Out[5]: [2, 3]
```

This says show elements between second and fourth position.

A few general rules:

```
a[0:2]  --> Includes the 0th element, but not the 2nd element. Goes from 0 to 1.

a[2:5] --> Includes the 2nd element, but not the 5th. Goes from 2 to 4.
```

The rule is that the start element is included, while the last one is excluded.

Now here is the interesting thing:

```
a[:-1]
Out[6]: [0, 1, 2, 3]
```

What does *-1* mean? Since the list starts at zero, *-1* is the last element. *-2* is the second last element, and so on. In the example above, we are saying starting from 0, show all elements up to the last. We can do it the other way too:

```
a[-1:]
Out[7]: [4]
```

This time we only get one element, as we started at the last one to begin with.

This trick is very neat to get rid of values at the beginning and end. Sometimes, the first few and last values maybe garbage, and you may want to get rid of them. This is how you remove the first and last element:

```
a[1: -1]
Out[10]: [1, 2, 3]
```

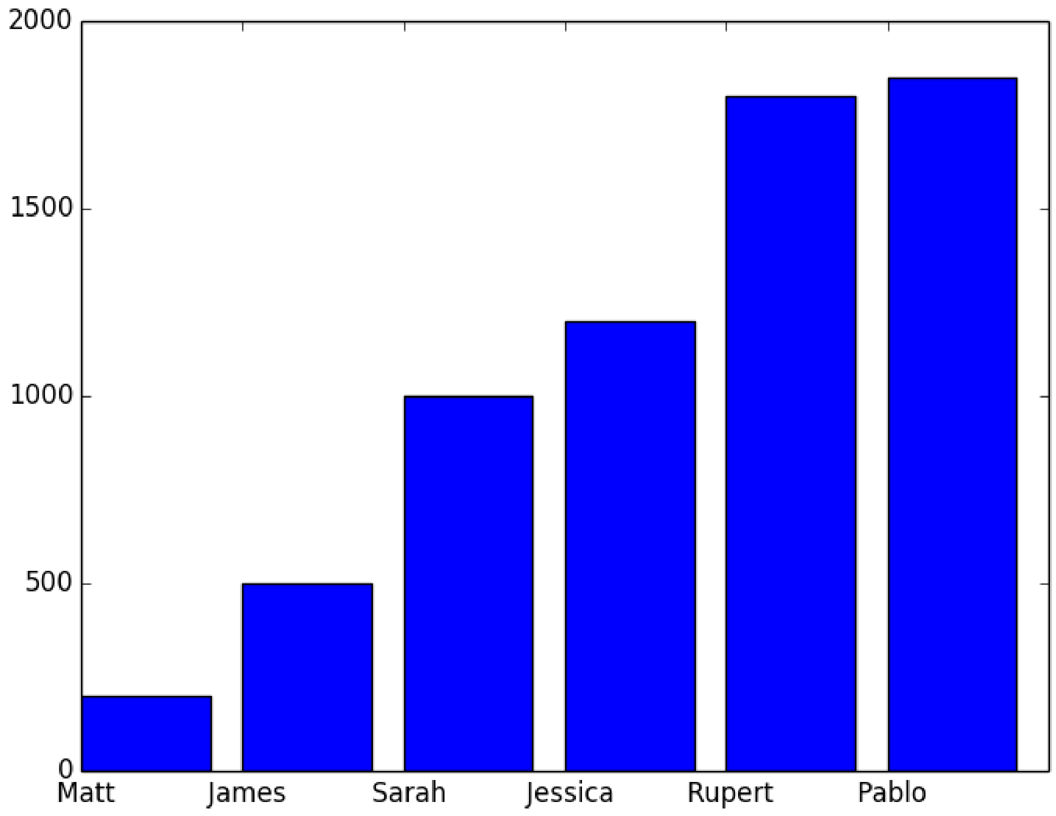
We will use this trick to get rid of the lower two and upper two values in our data. This is a normal technique in statistics to remove extreme values which may be interfering with the results.

```
salaries_new = salary[2:-2]
names_new = names[2:-2]
```

we remove lower two and top two values.

The rest of the code is the same as before:

```
x = range(len(names_new))

plt.plot(x, salaries_new)

plt.xticks(x, names_new)

plt.show()

print(np.max(salaries_new), np.min(salaries_new), np.average(salaries_new))
1850 200 1091.66666667
```



This time, the graph allows for better comparison.

And if you look at the average salary, it has gone down from 2165 to 1091.

Audio processing

Create a sine wave

In this project, we are going to create a sine wave, and save it as a wav file. But before that, some theory you should know.

Frequency: The frequency is the number of times a sine wave repeats a second. I will use a frequency of 1KHz.

Sampling rate: Most real world signals are analog, while computers are digital. So we need a analog to digital converter to convert our analog signal to digital. Details of how the converter work are beyond the scope of this book. The key thing is the sampling rate, which is the number of times a second the converter takes a sample of the analog signal.

Now, the sampling rate doesn't really matter for us, as we are doing everything digitally, but it's needed for our sine wave formula. I will use a value of 48000, which is the value used in professional audio equipment.

Sine Wave formula: If you forgot the formula, don't worry. I had to check Wikipedia as well.

```
y(t) = A * sin(2 * pi * f * t)
```

y(t) is the y axis sample we want to calculate for x axis sample *t*.

A is the amplitude. We'll come to that.

pi is our old friend 3.14159.

f is the frequency.

t is our sample. Since we need to convert it to digital, we will divide it by the sampling rate.

Amplitude

I mentioned the amplitude *A*. In most books, they just choose a random value for A, usually 1. But that won't work for us. The sine wave we generate will be in floating point, and while that will be good enough for drawing a graph, it won't work when we write to a file. The reason being that we are dealing with integers. If you look at wave files, they are written as 16 bit short integers. If we write a floating point number, it will not be represented right.

To get around this, we have to convert our floating point number to fixed point. One of the ways to do so is to multiply it with a fixed constant. How do we calculate this constant? Well, the maximum value of signed 16 bit number is 32767 (2^15 - 1). (Because the left most bit is reserved for the sign, leaving 15 bits. We raise 2 to the power of 15 and then subtract one, as computers count from 0).

So we want full scale audio, we'd multiply it with 32767. But I want an audio signal that is half as loud as full scale, so I will use an amplitude of 16000.

To the code:

```
import numpy as np
import wave
import struct
import matplotlib.pyplot as plt

# frequency is the number of times a wave repeats a second
frequency = 1000

num_samples = 48000

# The sampling rate of the analog to digital convert
sampling_rate = 48000.0

amplitude = 16000
file = "test.wav"
```

I just setup the variables I have declared.

```
sine_wave = [np.sin(2 * np.pi * frequency * x/sampling_rate) for x in range(num_samp\
les)]
```

Half of you are going to quit the book right now. Go on, you want to. That's one killer equation, isn't it?

But if you remembered what I said, list comprehensions are the most powerful features of Python. I could have written the above as a normal for loop, but I wanted to show you the power of list comprehensions. The above code is quite simple if you understand it. Let's break it down, shall we? It will be easier if you have the source code open as well.

The first thing is that the equation is in *[]*, which means the final answer will be converted to a list.

```
[ for x in range(num_samples)]
```

The *range()* function generates a list of numbers from 0 to *num_samples*. So we are saying loop over a variable *x* from 0 to 48000, the number of samples we have.

```
[np.sin(2 * np.pi * frequency * x/sampling_rate)]
```

This says that for each *x* that we generated, run it through the formula for the sine wave,

```
2 * pi * f * t.
```

So if we look at the code again:

```
sine_wave = [np.sin(2 * np.pi * frequency * x/sampling_rate) for x in range(num_samp\
les)]
```

It says generate x in the range of 0 to num_samples, and for each of that *x* value, generate a value that is the sine of that. You can think of this value as the *y* axis values. All these values are then put in a list. Easy peasy.

```
nframes=num_samples
comptype="NONE"
compname="not compressed"
nchannels=1
sampwidth=2
```

Okay, now it's time to write the sine wave to a file. We are going to use Python's inbuilt wave library. Here we set the paramerters. *nframes* is the number of frames or samples.

comptype and *compname* both signal the same thing: The data isn't compressed. *nchannels* is the number of channels, which is 1. *sampwidth* is the sample width in bytes. As I mentioned earlier, wave files are usually 16 bits or 2 bytes per sample.

```
wav_file=wave.open(file, 'w')
wav_file.setparams((nchannels, sampwidth, int(sampling_rate), nframes, comptype, com\
pname))
```

We open the file and set the parameters.

```
for s in sine_wave:
    wav_file.writeframes(struct.pack('h', int(s*amplitude)))
```

This might require some explanation. We are writing the sine_wave sample by sample. *writeframes* is the function that writes a sine wave. All that is simple. This might confuse you:

```
struct.pack('h', int(s*amplitude))
```

So let's break it down into parts.

```
int(s*amplitude)
```

s is the single sample of the sine_wave we are writing. I am multiplying it with the amplitude here (to convert to fixed point). We could have done it earlier, but I'm doing it here, as this is where it matters, when we are writing to a file.

Now,the data we have is just a list of numbers. If we write it to a file, it will not be readable by an audio player.

Struct is a Python library that takes our data and packs it as binary data. The *h* in the code means 16 bit number.

To understand what packing does, let's look at an example in IPython.

```
In [1]: import numpy as np

In [2]: np.sin(0.5)
Out[2]: 0.47942553860420301


In [5]: 0.479*16000
Out[5]: 7664.0
```

I am using *0.5* as an example above.

So we take the sin of 0.5, and convert it to a fixed point number by multiplying it by 16000. Now if we were to write this to file, it would just write 7664 as a string, which would be wrong. Let's look at what struct does:

```
In [6]: struct.pack('h', 7664)
Out[6]: '\xf0\x1d'
```

x means the number is a hexadecimal. As you can see, struct has turned our number 7664 into 2 hex values: 0xf0 and 0x1d.

Why 0xf0 0x1d? Well, if you convert 7664 to hex, you will get 0xf01d.

Why two values? Because we are using 16 bit values and our number can't fit in one. So struct broke it into two numbers.

Since the numbers are now in hex, they can be read by other programs, including our audio players.

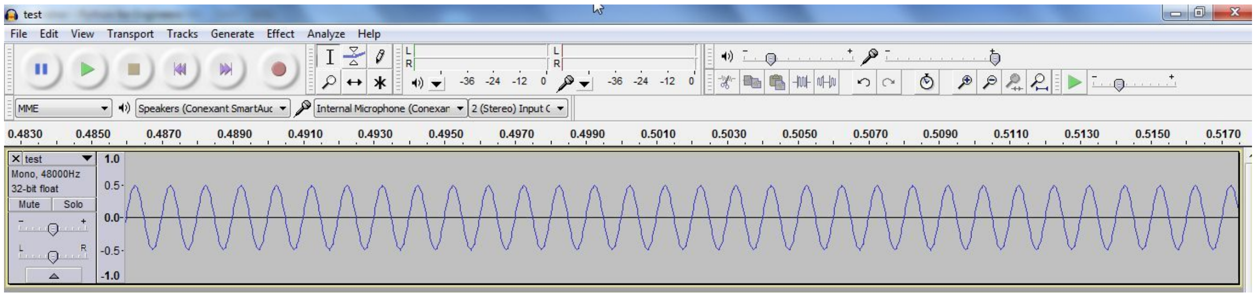
Coming back to our code:

```
for s in sine_wave:
    wav_file.writeframes(struct.pack('h', int(s*amplitude)))
```

This will take our sine wave samples and write it to our file, *test.wav*, packed as 16 bit audio.

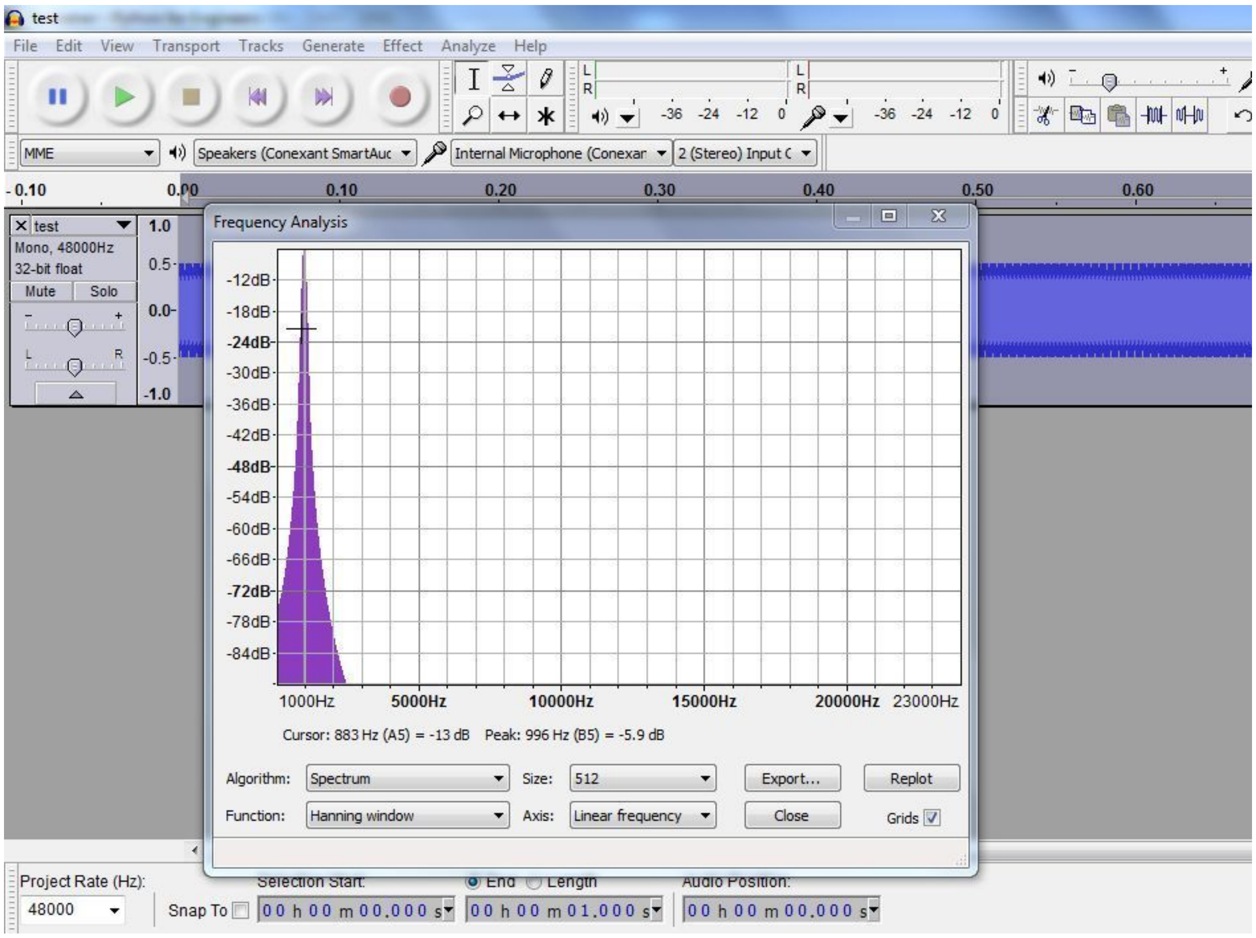
Play the file in any audio player you have- Windows Media player, VLC etc. You should hear a very short tone.

Now, we need to check if the frequency of the tone is correct. I am going to use Audacity, a open source audio player with a ton of features. One of them is that we can find the frequency of audio files. Let's open up Audacity.



So we have a sine wave. Note that the wave goes as high as 0.5, while 1.0 is the maximum value. Remember we multiplied by 16000, which was half of 36767, which was full scale?

Now to find the frequency. Go to Edit-> Select All (or press Ctrl A), then Analyse-> Plot Spectrum.



Cursor: 883 Hz (A5) = -13 dB Peak: 996 Hz (B5) = -5.9 dB

Algorithm: Spectrum
Size: 512
Function: Hanning window
Axis: Linear frequency
Grids ✓

Project Rate (Hz): 48000

You can see that the peak is at around a 1000 Hz, which is how we created our wave file.

Calculate the frequency of a sine wave

I took one course in signal processing in my degree, and didn't understand a thing. We were asked to derive a hundred equations, with no sense or logic. I found the subject boring and pedantic.

Which is why I wasn't happy when I had to study it again for my Masters. But I was in luck.

This time, the teacher was a practising engineer. He ran his own company and taught part time. Unlike the university teachers, he actually knew what the equations were for.

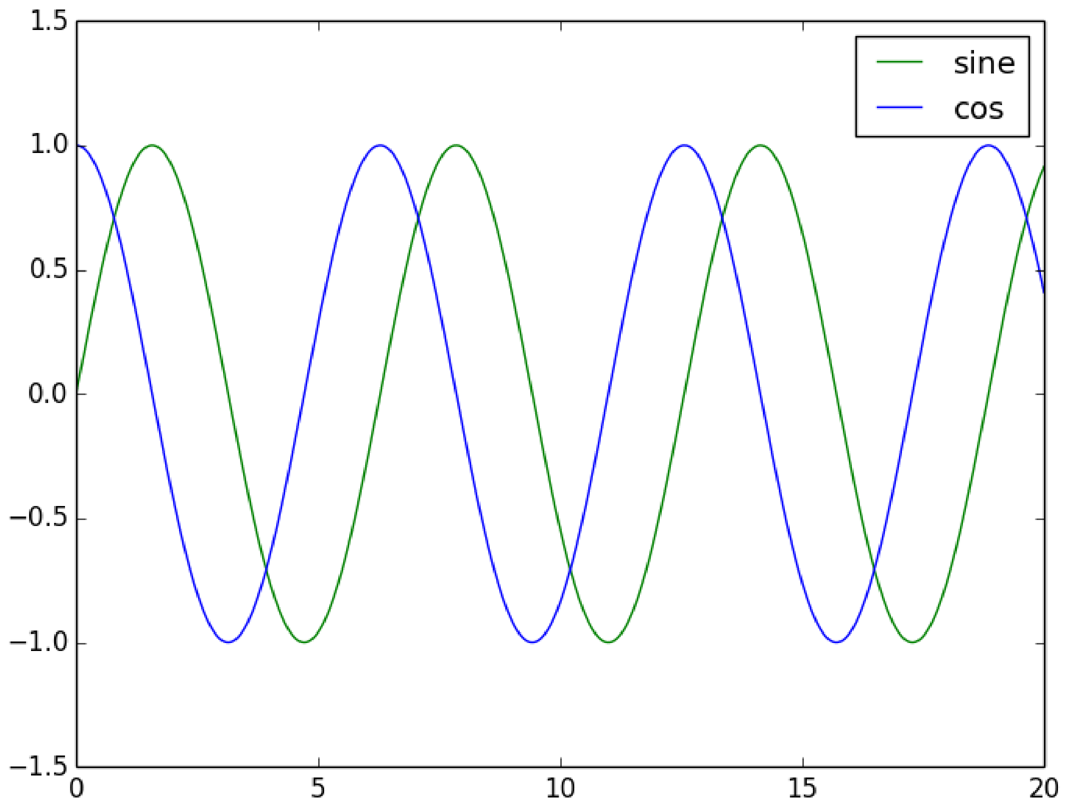
He started us with the Discrete Fourier Transform (DFT). I had heard of the DFT, and had no idea what it did. I could derive the equation, though fat lot of good it did me.

But this teacher (I forgot his name, he was a Danish guy) showed us a noisy signal, and then took the DFT of it. He then showed the results in a graphical window. We clearly saw the original sine wave and the noise frequency, and I understood for the first time what a DFT does.

Use the DFT to get frequencies

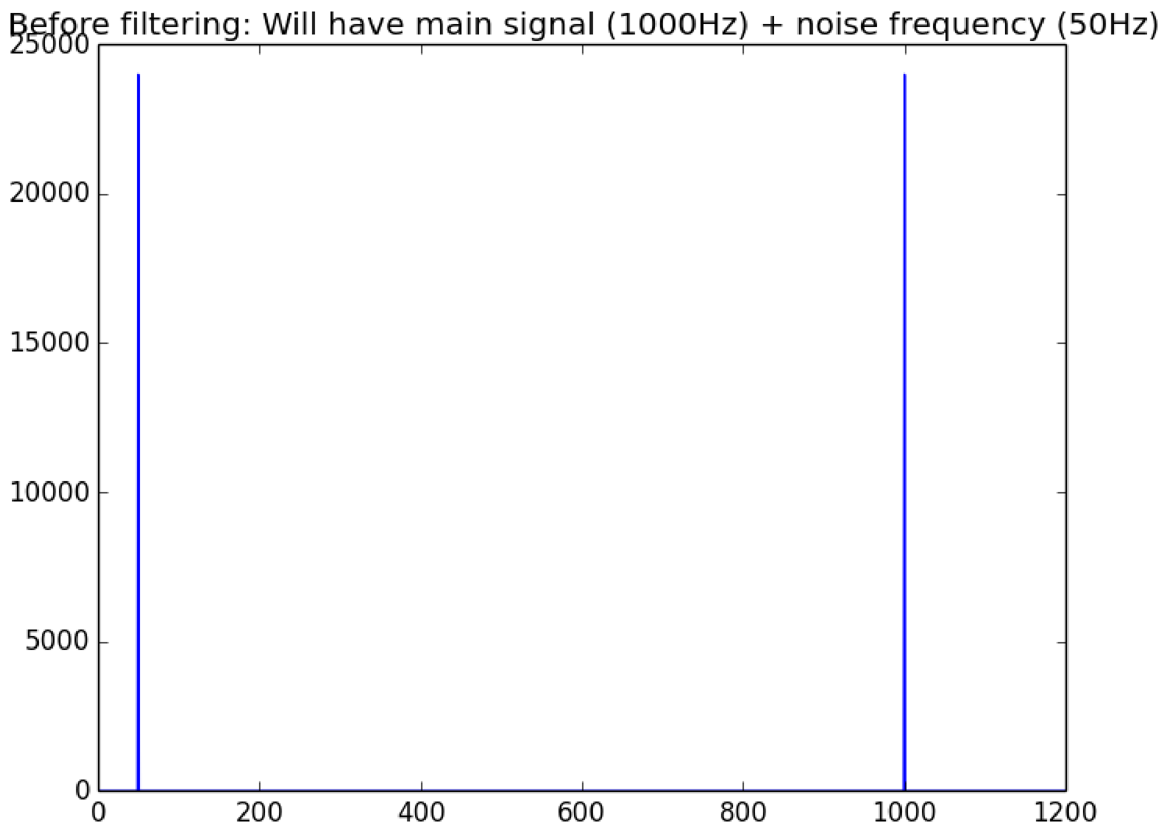
To get the frequency of a sine wave, you need to get its Discrete Fourier Transform(DFT). Contrary to what every book written by Phd types may have told you, you don't need to understand how to derive the transform. You just need to know how to use it.

In its simplest terms, the DFT takes a signal and calculates which frequencies are present in it. In more technical terms, the DFT converts a time domain signal to a frequency domain. What does that mean? Let's look at our sine wave.



The wave is changing with time. If this was an audio file, you could imagine the player moving right as the file plays.

In the frequency domain, you see the frequency part of the signal. This image is taken from later on in the chapter to show you what the frequency domain looks like:



The signal will change if you add or remove frequencies, but will not change in time. For example, if you take a 1000 Hz audio tone and take its frequency, the frequency will remain the same no matter how long you look at it. But if you look at it in the time domain, you will see the signal moving.

The DFT was really slow to run on computers (back in the 70s), so the Fast Fourier Transform (FFT) was invented. The FFT is what is normally used nowadays.

The way it works is, you take a signal and run the FFT on it, and you get the frequency of the signal back.

If you have never used (or even heard of) a FFT, don't worry. I'll teach you how to start using it, and you can read more online if you want. Most tutorials or books won't teach you much anyway. They'll usually blat you with equations, without showing you what to do with them.

On to the code. Open up *get_freq.py*,

```
frame_rate = 48000.0
infile = "test.wav"
num_samples = 48000

wav_file = wave.open(infile, 'r')
data = wav_file.readframes(num_samples)
wav_file.close()
```

We are reading the wave file we generated in the last example. This code should be clear enough. The *wave readframes()* function reads all the audio frames from a wave file.

```
data = struct.unpack('{n}h'.format(n=num_samples), data)
```

Remember we had to pack the data to make it readable in binary format? Well, we do the opposite now. The first parameter to the function is a format string, which is the same thing you use when you do a *print()*. You are telling the unpacker to unpack *num_samples* 16 bit words (remember the *h* means 16 bits).

```
data = np.array(data)
```

We then convert the data to a numpy array.

```
data_fft = np.fft.fft(data)
```

We take the *fft* of the data. This will create an array with all the frequencies present in the signal.

Now, here's the problem. The *fft* returns an array of complex numbers that doesn't tell us anything. If I print out the first 8 values of the fft, I get:

```
In [3]: data_fft[:8]
Out[3]:
array([ 13.00000000 +0.j        ,   8.44107682 -4.55121351j,
         6.24696630-11.98027552j,   4.09513760 -2.63009999j,
        -0.87934285 +9.52378503j,   2.62608334 +3.58733642j,
         4.89671762 -3.36196984j,  -1.26176048 +3.0234555j ])
```

If only there was a way to convert the complex numbers to real values we can use. Let's try to remember our high school formulas for converting complex numbers to real…

Wait. Numpy can do that for us.

```
# This will give us the frequency we want
frequencies = np.abs(data_fft)
```

The *numpy abs()* function will take our complex signal and generate the real part of it.

Side Detour

A bit of a detour to explain how the FFT returns its results.

The FFT returns all possible frequencies in the signal. And the way it returns is that each index contains a frequency element. Say you store the FFT results in an array called *data_fft*. Then:

data_fft[1] will contain frequency part of 1 Hz.

data_fft[2] will contain frequency part of 2 Hz.

…

data_fft[8] will contain frequency part of 8 Hz.

…

data_fft[1000] will contain frequency part of 1000 Hz.

Now what if you have no 1Hz frequency in your signal? You will still get a value at *data_fft[1]*, but it will be minuscule. To give you an example, I will take the real fft of a 1000 Hz wave:

```
data_fft = np.fft.fft(sine_wave)

abs(data_fft[0])
Out[7]: 8.1289678326462086e-13

abs(data_fft[1])
Out[8]: 9.9475299243014428e-12


abs(data_fft[1000])
Out[11]: 24000.0
```

If you look at the absolute values for *data_fft[0]* or *data_fft[1]*, you will see they are tiny. The *e-12* at the end means they are raised to a power of -12, so something like *0.00000000000812* for *data_fft[0]*. But if you look at *data_fft[1000]*, the value is a hue *24000*. This can easily be plotted.

If we want to find the array element with the highest value, we can find it by:

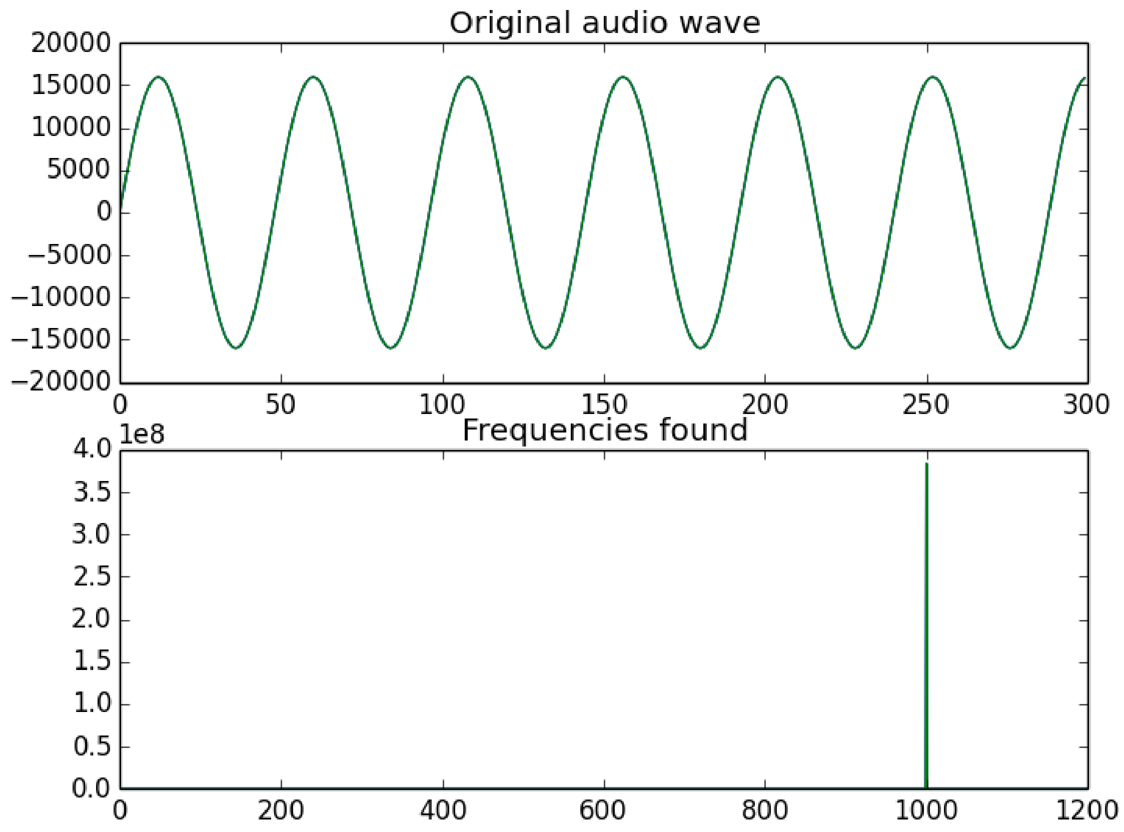
```
print("The frequency is {} Hz".format(np.argmax(frequencies)))
```

np.argmax will return the highest frequency in our signal, which it will then print. As we have seen manually, this is at a 1000Hz (or the value stored at *data_fft[1000]*). And now we can plot the data too.

```
plt.subplot(2,1,1)
plt.plot(data[:300])
plt.title("Original audio wave")
plt.subplot(2,1,2)
plt.plot(frequencies)
plt.title("Frequencies found")

plt.xlim(0,1200)


plt.show()
```

This should be known to you. The only new thing is the *subplot* function, which allows you to draw multiple plots on the same window.

subplot(2,1,1) means that we are plotting a 2x1 grid. The 3rd number is the plot number, and the only one that will change. It will become clearer when you see the graph.



And that's it, folks. We took our audio file and calculated the frequency of it. Next, we will add noise to our plot and then try to clean it.

Cleaning a noisy sine wave

In this example, I'll recreate the same example my teacher showed me. We'll generate a sine wave, add noise to it, and then filter the noise. Let's start with the code.

```
# frequency is the number of times a wave repeats a second
frequency = 1000
noisy_freq = 50
num_samples = 48000

# The sampling rate of the analog to digital convert
sampling_rate = 48000.0
```

The main frequency is a 1000Hz, and we will add a noise of 50Hz to it.

```
#Create the sine wave and noise
sine_wave = [np.sin(2 * np.pi * frequency * x1 / sampling_rate) for x1 in range(num_\
samples)]

sine_noise = [np.sin(2 * np.pi * noisy_freq * x1/  sampling_rate) for x1 in range(nu\
m_samples)]


#Convert them to numpy arrays
sine_wave = np.array(sine_wave)
sine_noise = np.array(sine_noise)
```

I hope the above isn't scary to you anymore, as it's the same code as before. We generate two sine waves, one for the signal and one for the noise, and convert them to numpy arrays.

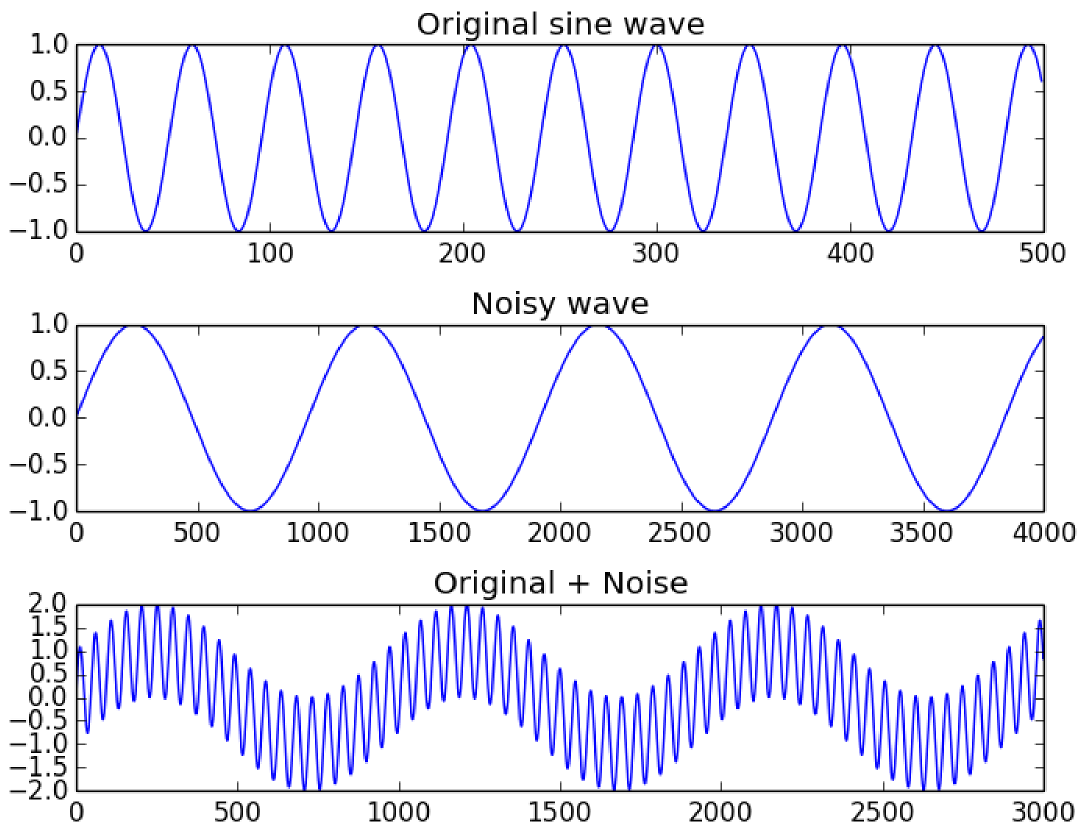
```
# Add them to create a noisy signal
combined_signal = sine_wave + sine_noise
```

I am adding the noise to the signal. As I mentioned earlier, this is possible only with numpy. With normal Python, you'd have to for loop or use list comprehensions. Messy. With numpy, you can add two arrays like they were normal numbers, and numpy takes care of the low level detail for you.

On to some graphing of what we have till now.

```
plt.subplot(3,1,1)
plt.title("Original sine wave")

# Need to add empty space, else everything looks scrunched up!
plt.subplots_adjust(hspace=.5)
plt.plot(sine_wave[:500])

plt.subplot(3,1,2)
plt.title("Noisy wave")
plt.plot(sine_noise[:4000])

plt.subplot(3,1,3)
plt.title("Original + Noise")
plt.plot(combined_signal[:3000])

plt.show()
```

Nothing shocking here.

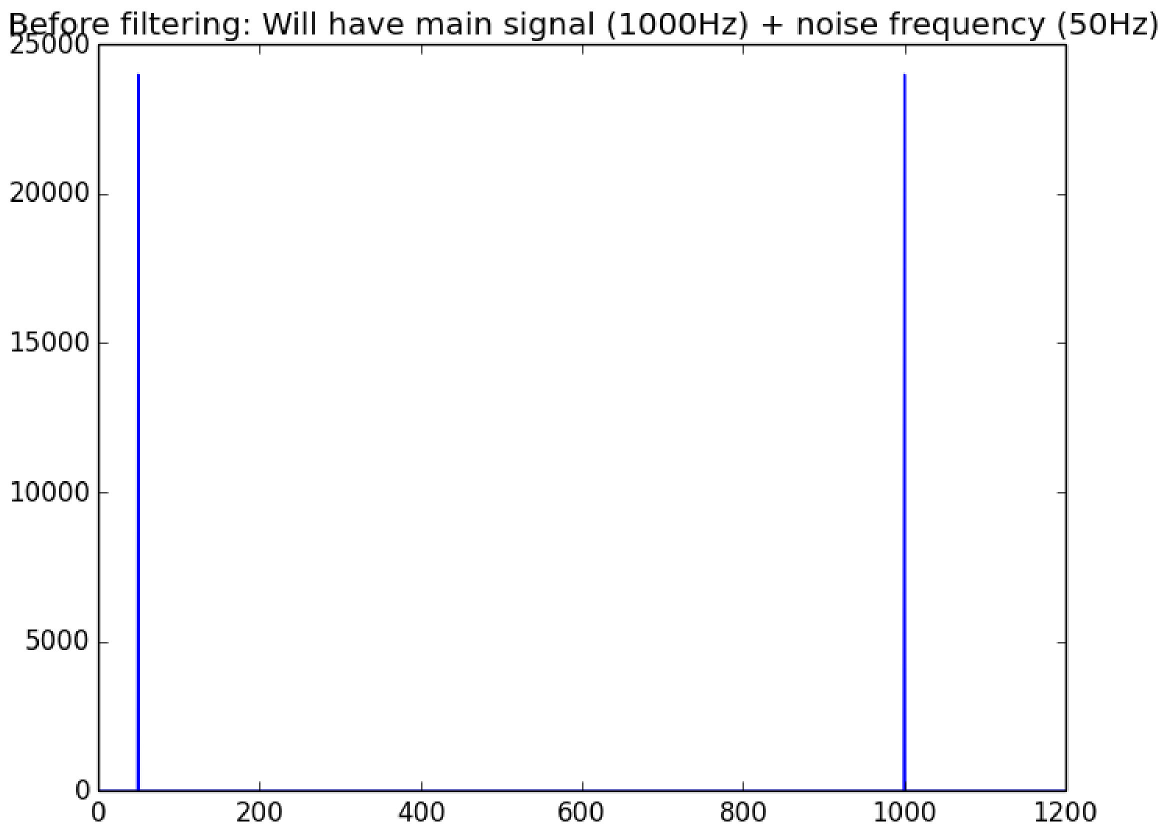


```
data_fft = np.fft.fft(combined_signal)

freq = (np.abs(data_fft[:len(data_fft)]))
```

data_fft contains the fft of the combined noise+signal wave. *freq* contains the absolute of the frequencies found in it.

```
plt.plot(freq)
plt.title("Before filtering: Will have main signal (1000Hz) + noise frequency (50Hz)\
")
plt.xlim(0,1200)
```

We take the fft of the signal, as before, and plot it. This time, we get two signals: Our sine wave at 1000Hz and the noise at 50Hz.



Now, to filter the signal. I won't cover filtering in any detail, as that can take a whole book. Instead, I will create a simple filter just using the fft. The goal is to get you comfortable with Numpy.

First, here is the complete code:

```
filtered_freq = []
index = 0
for f in freq:
    # Filter between lower and upper limits
    # Choosing 950, as closest to 1000. In real world, won't get exact numbers like \
these
    if index > 950 and index < 1050:
        # Has a real value. I'm choosing >1, as many values are like 0.000000001 etc
        if f > 1:
            filtered_freq.append(f)

        else:
            filtered_freq.append(0)
    else:
        filtered_freq.append(0)
    index += 1
```

Now let's go over it line by line:

```
filtered_freq = []
index = 0
for f in freq:
```

We create an empty list called *filtered_freq*. If you remember, *freq* stores the absolute values of the fft, or the frequencies present.

```
     if index > 950 and index < 1050:
```

index is the current array element in the array *freq*. As I said, the *fft* returns all frequencies in the signal. These are stored in the array based on the index, so *freq[1]* will have the frequency of 1Hz, *freq[2]* will have 2Hz and so on.

Since I know my frequency is 1000Hz, I will search around that. In the real world, we will never get the exact frequency, as noise means some data will be lost. So I'm using a lower limit of 950 and upper limit of 1050. I check if the frequency we are looping is within this range.

```
 if f > 1:
    filtered_freq.append(f)
```

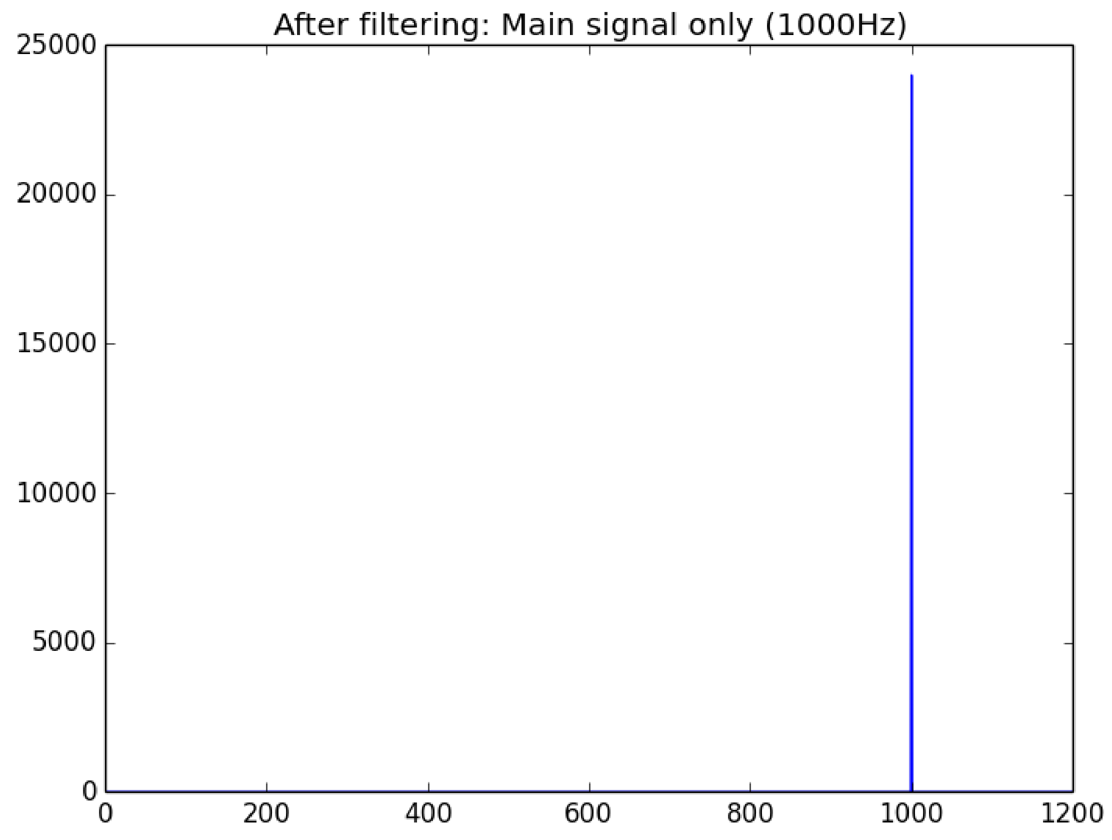
I mentioned this earlier as well: While all frequencies will be present, their absolute values will be minuscule, usually less than 1. So if we find a value greater than 1, we save it to our *filtered_freq* array.

```
        else:
            filtered_freq.append(0)
    else:
        filtered_freq.append(0)
    index += 1
```

If our frequency is not within the range we are looking for, or if the value is too low, we append a zero. This is to remove all frequencies we don't want. And then we increment index.

```
plt.plot(filtered_freq)
plt.title("After filtering: Main signal only (1000Hz)")
plt.xlim(0,1200)
plt.show()
plt.close()
```

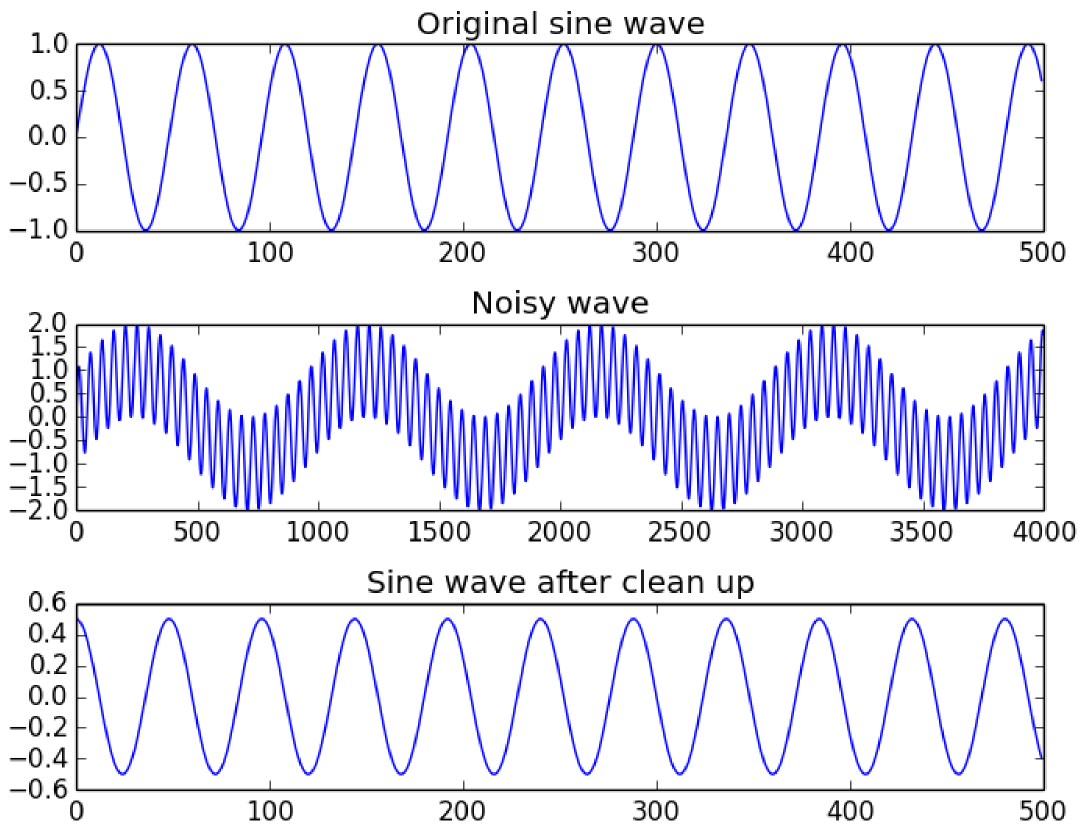
And we plot what we have.



```
recovered_signal = np.fft.ifft(filtered_freq)
```

Now we take the *ifft*, which stands for Inverse FFT. This will take our signal and convert it back to time domain. We can now compare it with our original noisy signal. We do that with graphing:

```
plt.subplot(3,1,1)
plt.title("Original sine wave")
# Need to add empty space, else everything looks scrunched up!
plt.subplots_adjust(hspace=.5)

plt.plot(sine_wave[:500])

plt.subplot(3,1,2)
plt.title("Noisy wave")
plt.plot(combined_signal[:4000])

plt.subplot(3,1,3)
plt.title("Sine wave after clean up")
plt.plot((recovered_signal[:500]))

plt.show()
```



Note that we will receive a warning:

```
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

This is, again, because the *fft* returns an array of complex numbers. Luckily, like the warning says, the imaginary part will be discarded.

And there you go. Using our very simplistic filter, we have cleaned a sine wave. And this brings us to the end of this chapter.

Python Pandas

You must have seen in Chapter on plotting that Python can be used to parse csv files. You might also have noted that it is fairly painful. While there are libraries like *csv_reader()*, they still aren't perfect. You still have to do a lot of stuff manually.

Enter Pandas, which is a great library for data analysis. It is quite high level, so you don't have to muck about with low level details, unless you really want to.

I only just recently discovered it, and was blown away by how powerful it is (especially since I've been doing everything the inefficient way till now). And there is a lot of help on it- if you just Google "How do I do X in Pandas" you will get dozens of results.

If you are dealing with complicated or large datasets, seriously consider Pandas. It is based on numpy/scipy, sort of a superset of it. But it gives you a lot of powerful features, like read directly from a Microsoft Excel file, do data joins (like you would do in SQL) etc, some of which features we will go over.

Just one more thing before we continue. Pandas has two basic data structures: Series and Dataframes.

Series is like numpy's array/dictionary, though it comes with a lot of extra features. Dataframes is a two dimensional data structure that contains both column and row information, like the fields of an Excel file. Remember an Excel file has rows and columns, and an optional header field. All of this data can be represented in a Dataframe. That's what we'll use in our examples below.

Installing pandas

You should get Pandas if you installed Anaconda on Windows. On Linux, run

```
pip install pandas
```

You might need to install some dependencies, which pip will tell you about.

Analyse obesity in England

I'm using the 2014 data from here:

http://data.gov.uk/dataset/statistics_on_obesity_physical_activity_and_diet_england

I'll also put it in the Github directory, so you don't have to download it. Make sure you open the file in Excel (or Openoffice) and view it, so you know what we're talking about. Do that now, and have a look at the different sections. Especially section 7.1 and 7.2, as that's what we'll focus on.

Let's get started then.

```python
import pandas as pd
import matplotlib.pyplot as plt
```

Pandas is imported as *pd* to save typing, in the same way we import numpy as *np*.

```python
data = pd.ExcelFile("Obes-phys-acti-diet-eng-2014-tab.xls")
print(data.sheet_names)
```

We are opening the *xls* file. The great thing about pandas is that you can open Excel files directly. Normally, most libraries can only work with *csv* files. We then print all the sheet names.

```
['Chapter 7', '7.1', '7.2', '7.3', '7.4', '7.5', '7.6', '7.7', '7.8', '7.9', '7.10']
```

For those of you not comfortable with Excel, a sheet is one "page", as it were, of data. Rather than having all the data in one huge unmanageable sheet, users break the data into multiple sheets. Above, we printed all the available sheets.

Section 7.1: Obesity by gender

Let's have a look at sheet 1:

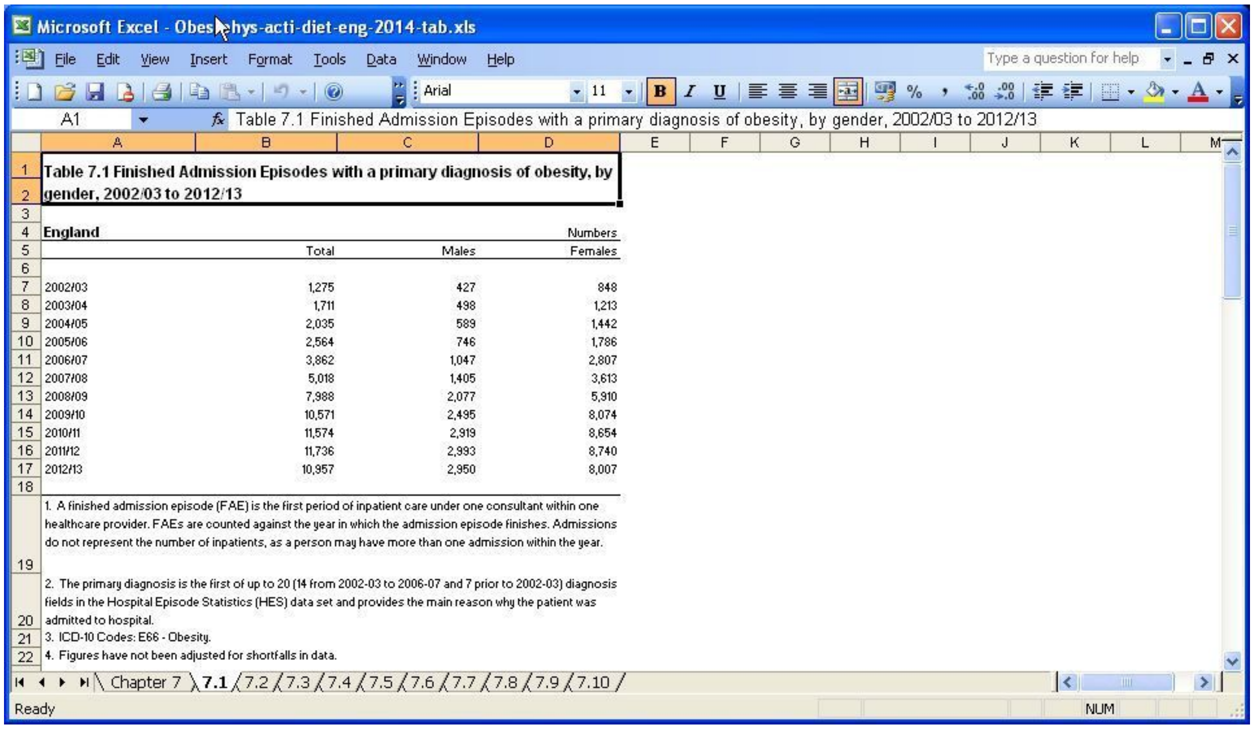

Microsoft Excel - Obes-phys-acti-diet-eng-2014-tab.xls
File Edit View Insert Format Tools Data Window Help
Type a question for help
Arial
11
A1
Chapter 7
7.1
7.2
7.3
7.4
7.5
7.6
7.7
7.8
7.9
7.10
Ready
NUM

Table 7.1 Finished Admission Episodes with a primary diagnosis of obesity, by gender, 2002/03 to 2012/13

England			Numbers
	Total	Males	Females
2002/03	1,275	427	848
2003/04	1,711	498	1,213
2004/05	2,035	589	1,442
2005/06	2,564	746	1,786
2006/07	3,862	1,047	2,807
2007/08	5,018	1,405	3,613
2008/09	7,988	2,077	5,910
2009/10	10,571	2,495	8,074
2010/11	11,574	2,919	8,654
2011/12	11,736	2,993	8,740
2012/13	10,957	2,950	8,007

1. A finished admission episode (FAE) is the first period of inpatient care under one consultant within one healthcare provider. FAEs are counted against the year in which the admission episode finishes. Admissions do not represent the number of inpatients, as a person may have more than one admission within the year.

2. The primary diagnosis is the first of up to 20 (14 from 2002-03 to 2006-07 and 7 prior to 2002-03) diagnosis fields in the Hospital Episode Statistics (HES) data set and provides the main reason why the patient was admitted to hospital.

3. ICD-10 Codes: E66 - Obesity.

4. Figures have not been adjusted for shortfalls in data.

There are four columns: *Year*, *total*, *males* and *females*. The *Year* column doesn't have a header- if you look at line 5, you will see the header for year is empty.

I'll show you two ways to read in data. In the first one, you define the header columns yourself.

```
# Define the columns to be read
columns1 = ['year', 'total', 'males', 'females']
```

So I'm defining a list with four header entries: 'year', 'total', 'males' and 'females'.

```
data_gender = data.parse(u'7.1', skiprows=4, skipfooter=14, names=columns1)
```

We read sheet *7.1*. If you look at the actual sheet, the top 4 and bottom 14 rows contain useless info, so we skip it (*skiprows=4, skipfooter=14,*). Finally, we tell pandas to name the column headers using our list *names=columns1* .

We then print *data_gender*.

```
print(data_gender)
```

```
       year  total  males  females
0       NaN    NaN    NaN      NaN
1   2002/03   1275    427      848
2   2003/04   1711    498     1213
3   2004/05   2035    589     1442
 < snip >
```

If you look at the entry for 0, it is *NaN*. Why's that? Look at the original spreadsheet. There is an empty space on line 6, to make the sheet easier to read. Since it is empty, it is read as Not A Number (NaN) by pandas. How do we get rid of it? Easy.

```
# Remove the N/A from the data
data_gender.dropna(inplace = True)
```

Using the inbuilt *dropna()* function, of course. *inplace = True* means modify the existing Dataframe. If we look at the output again:

```
       year  total  males  females
1   2002/03   1275    427      848
2   2003/04   1711    498     1213
3   2004/05   2035    589     1442

 < snip >
```

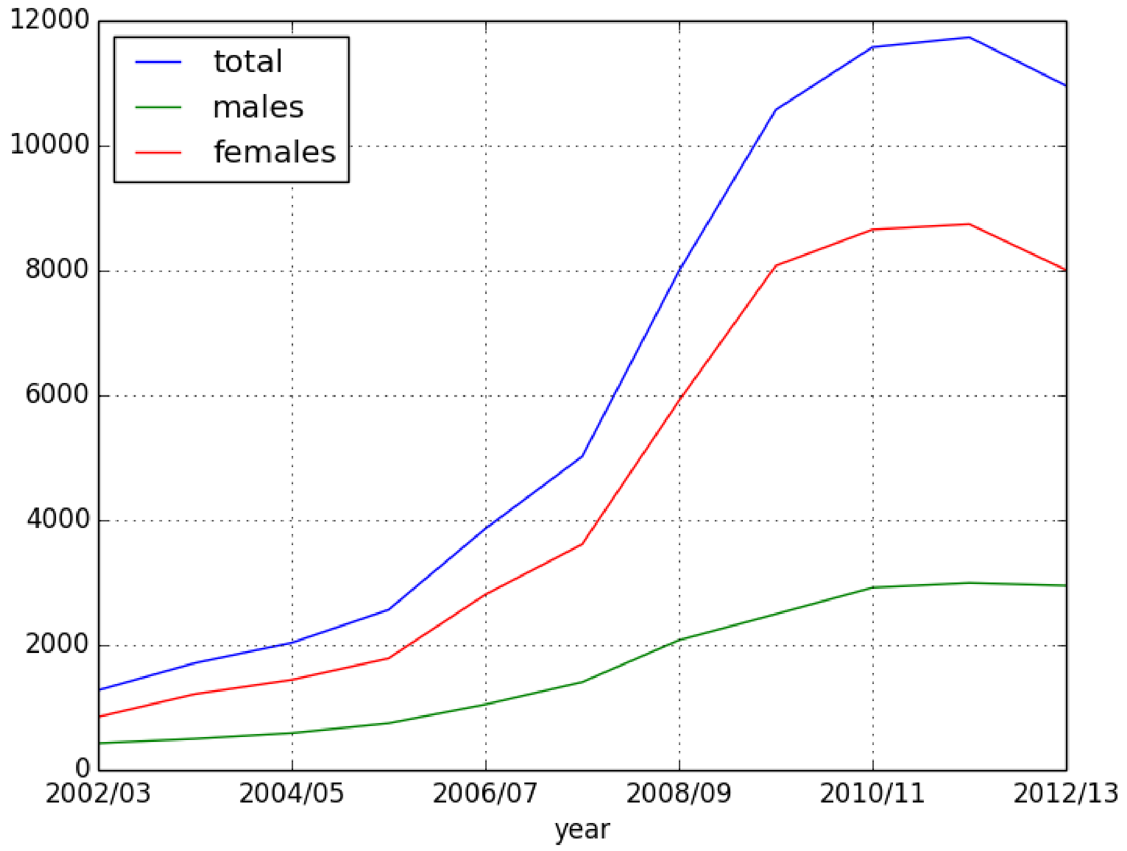
we see the *NaN* values are now gone. There is another problem, though. We have an index at the beginning (the first column, going 1,2,3....). We don't need an index. Instead, we want the year to be the index. Let's fix that next.

```
data_gender.set_index('year', inplace=True)
print(data_gender)
```

```
         total  males  females
year
2002/03   1275    427      848
2003/04   1711    498     1213
2004/05   2035    589     1442
```

That's better.

```
# Plot all
data_gender.plot()
plt.show()
```



We can see that while obesity for men has gone up, obesity for women has gone up more strongly.

Section 7.2: Obesity by age

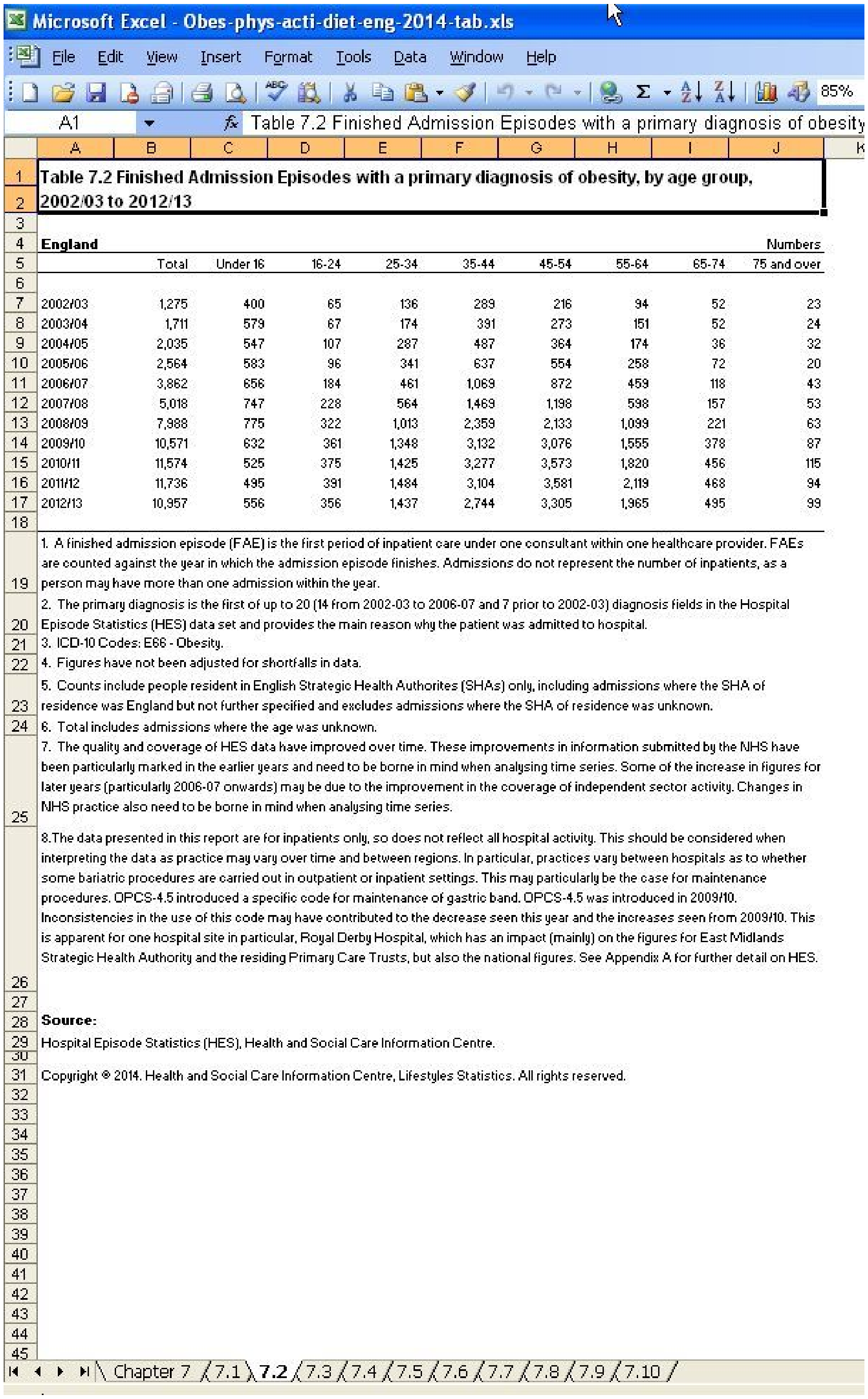

Microsoft Excel - Obes-phys-acti-diet-eng-2014-tab.xls
File Edit View Insert Format Tools Data Window Help
85%
A1
Chapter 7
7.1
7.2
7.3
7.4
7.5
7.6
7.7
7.8
7.9
7.10

Table 7.2 Finished Admission Episodes with a primary diagnosis of obesity

Table 7.2 Finished Admission Episodes with a primary diagnosis of obesity, by age group, 2002/03 to 2012/13

England Numbers

	Total	Under 16	16-24	25-34	35-44	45-54	55-64	65-74	75 and over
2002/03	1,275	400	65	136	289	216	94	52	23
2003/04	1,711	579	67	174	391	273	151	52	24
2004/05	2,035	547	107	287	487	364	174	36	32
2005/06	2,564	583	96	341	637	554	258	72	20
2006/07	3,862	656	184	461	1,069	872	459	118	43
2007/08	5,018	747	228	564	1,469	1,198	598	157	53
2008/09	7,988	775	322	1,013	2,359	2,133	1,099	221	63
2009/10	10,571	632	361	1,348	3,132	3,076	1,555	378	87
2010/11	11,574	525	375	1,425	3,277	3,573	1,820	456	115
2011/12	11,736	495	391	1,484	3,104	3,581	2,119	468	94
2012/13	10,957	556	356	1,437	2,744	3,305	1,965	495	99

1. A finished admission episode (FAE) is the first period of inpatient care under one consultant within one healthcare provider. FAEs are counted against the year in which the admission episode finishes. Admissions do not represent the number of inpatients, as a person may have more than one admission within the year.

2. The primary diagnosis is the first of up to 20 (14 from 2002-03 to 2006-07 and 7 prior to 2002-03) diagnosis fields in the Hospital Episode Statistics (HES) data set and provides the main reason why the patient was admitted to hospital.

3. ICD-10 Codes: E66 - Obesity.

4. Figures have not been adjusted for shortfalls in data.

5. Counts include people resident in English Strategic Health Authorites (SHAs) only, including admissions where the SHA of residence was England but not further specified and excludes admissions where the SHA of residence was unknown.

6. Total includes admissions where the age was unknown.

7. The quality and coverage of HES data have improved over time. These improvements in information submitted by the NHS have been particularly marked in the earlier years and need to be borne in mind when analysing time series. Some of the increase in figures for later years (particularly 2006-07 onwards) may be due to the improvement in the coverage of independent sector activity. Changes in NHS practice also need to be borne in mind when analysing time series.

8.The data presented in this report are for inpatients only, so does not reflect all hospital activity. This should be considered when interpreting the data as practice may vary over time and between regions. In particular, practices vary between hospitals as to whether some bariatric procedures are carried out in outpatient or inpatient settings. This may particularly be the case for maintenance procedures. OPCS-4.5 introduced a specific code for maintenance of gastric band. OPCS-4.5 was introduced in 2009/10. Inconsistencies in the use of this code may have contributed to the decrease seen this year and the increases seen from 2009/10. This is apparent for one hospital site in particular, Royal Derby Hospital, which has an impact (mainly) on the figures for East Midlands Strategic Health Authority and the residing Primary Care Trusts, but also the national figures. See Appendix A for further detail on HES.

Source:
Hospital Episode Statistics (HES), Health and Social Care Information Centre.

Copyright © 2014. Health and Social Care Information Centre, Lifestyles Statistics. All rights reserved.

We will read the data slightly differently this time. Last time, we defined the headers ourself. This time, we'll let pandas pick them up.

```
# Read 2nd section, by age
data_age  = data.parse('7.2', skiprows=4, skipfooter=14)
print(data_age)
```

```
   Unnamed: 0  Total  Under 16  16-24  25-34  35-44  45-54  55-64  65-74
0         NaN    NaN       NaN    NaN    NaN    NaN    NaN    NaN    NaN
1     2002/03   1275       400     65    136    289    216     94     52
```

If you remember, the year column didn't have a header, which is why pandas names it *Unnamed*. Let's rename it:

```
# Rename unames to year
data_age.rename(columns={'Unnamed: 0': 'Year'}, inplace=True)
print(data_age)
```

```
       Year  Total  Under 16  16-24  25-34  35-44  45-54  55-64  65-74
0       NaN    NaN       NaN    NaN    NaN    NaN    NaN    NaN    NaN
1   2002/03   1275       400     65    136    289    216     94     52
```

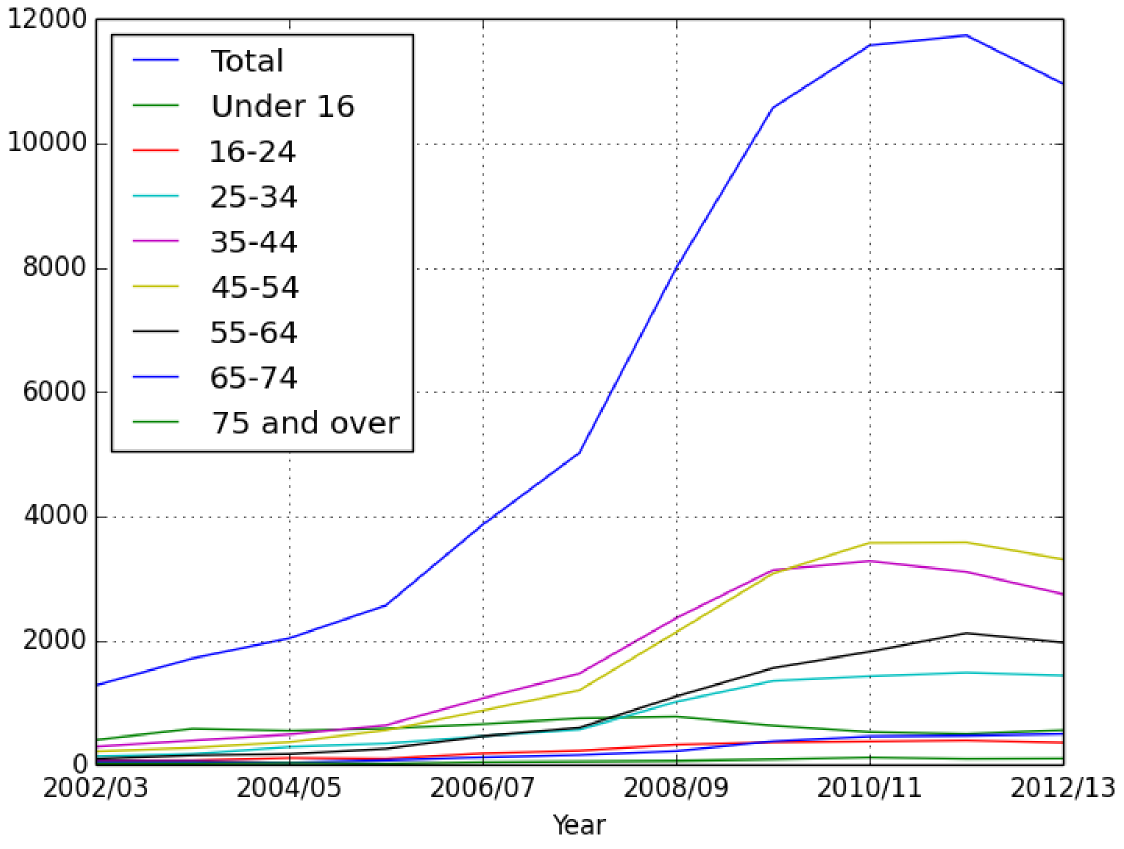
That's better. Let's drop the Nan and set the index to year.

```
# Drop empties and reset index
data_age.dropna(inplace=True)
data_age.set_index('Year', inplace=True)
print(data_age)
```

```
         Total  Under 16  16-24  25-34  35-44  45-54  55-64  65-74
Year
2002/03   1275       400     65    136    289    216     94     52
2003/04   1711       579     67    174    391    273    151     52
```

Let's plot all the ages first.

```
data_age.plot()
plt.show()
```

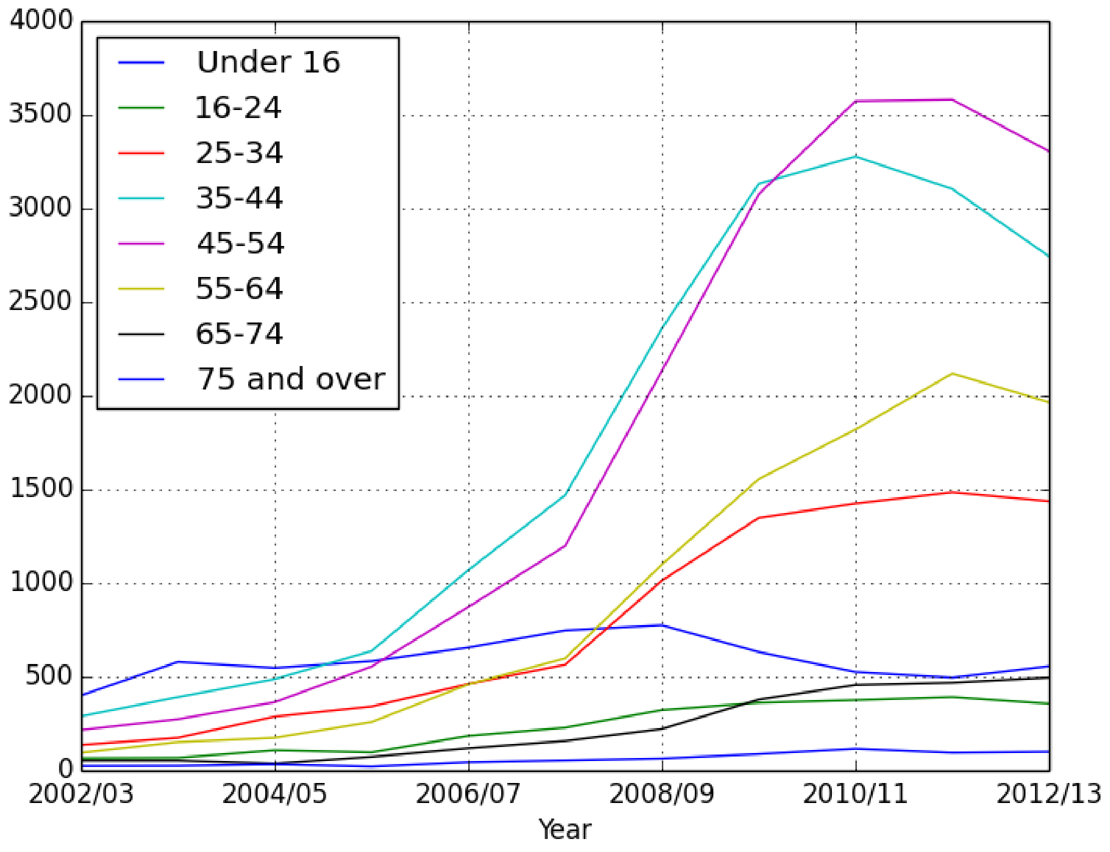


You see a big problem: The *Total* column is huge and suppresses the other graphs. So we need to get rid of it to make the graph clearer.

```
# Plotting everything cause total to override everything. So drop it.

# Drop the total column and plot
data_age_minus_total = data_age.drop('Total', axis = 1)
```

The *drop()* function can drop entries from a table. Here, we are dropping the *Total* column.

```
data_age_minus_total.plot()
plt.show()
```



We can quickly see that the age groups 35-44 and 45-54 have had the highest obesity growth.

What if we want to compare age groups? How do children compare to adults?

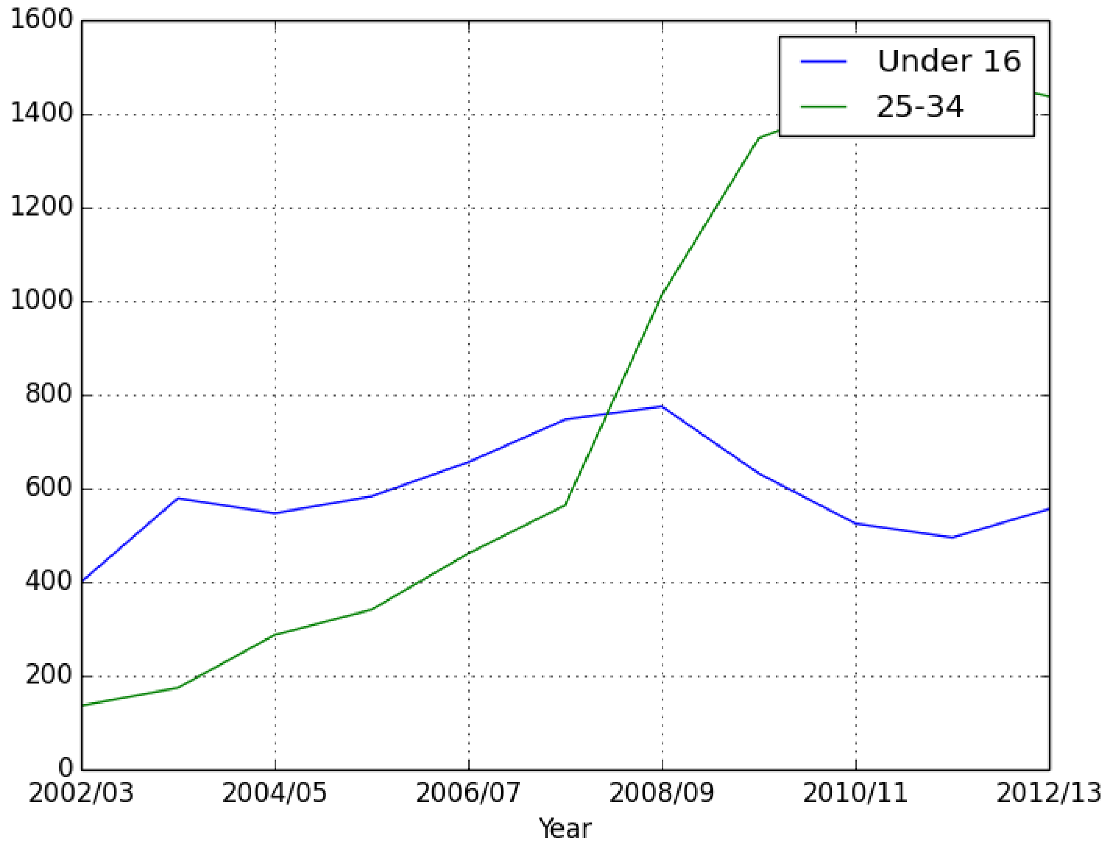
In Pandas, you can view any column by doing *data_age[column_name].*

```
print(data_age['Under 16'])
```

```
Year
2002/03     400
2003/04     579
2004/05     547
2005/06     583
```

This gives the data for just the under 16s. We can plot any age group this way.

```
#Plot children vs adults
data_age['Under 16'].plot(label = "Under 16")
data_age['25-34'].plot(label = "25-34")
plt.legend(loc="upper right")
plt.show()
```



We see that while children's obesity has gone down, that for adults has ballooned. Maybe adults should start following the advice they give to children?

Movie Lens

The movie lens database is a collection of data based on users reviews of movies. It is one of the most popular open datasets out there. Download it from http://grouplens.org/datasets/movielens/, downloading the 100K version.

It is also different that the data isn't in an Excel format, but plain text. This makes it easy to parse. Now, we have seen that Python pandas makes parsing Excel files easy as well, but many programming languages don't have this feature.

Download the data and have a look at it. There is a large amount of data, and we will only work with a small subset. If you want to know the details of the files, they are in the Readme.

In a text editor, open the first file we will work with, *u.item*:

```
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%20(1995\
)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0
2|GoldenEye (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?GoldenEye%20(1995)|\
0|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
3|Four Rooms (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Four%20Rooms%20(19\
95)|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
```

It contains the data movie id, movie name, release date, imdb.com link, and a series of flags for genre. These are separated by a pipe (|). I will only use the first two of the entries (movie id and name). Open *u.user*:

```
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
```

The details are user id, age, gender, occupation and post code. By the way, these details are in the Readme, if you forget. Again, I will only use the first three enteries (id, age and gender).

Finally, we have *u.data*

```
196         242         3         881250949
186         302         3         891717742
22         377         1         878887116
```

The entries are user id, movie id, rating and timestamp (which I'll ignore in this example).

If you look at the other files, you'll see there is a wealth of data. You can draw many conclusions from the data, which is why the dataset is so popular.

We'll focus on a small set of the data, mainly on how users of different ages and genders rated movies.

Reading the data

Since the data is plain text separated by |, it will not have column headers like Excel files. Which means, we'll have to provide our own, like I did in the first part of the last section.

Pandas has a function to read plain text files, which is a lot similar to numpy's read file functions we covered in chapter on plotting (not surprising, since pandas is based in numpy).

On to the code.

```
user_columns = ['user_id', 'age', 'sex']
```

We declare *user_columns* for the three entries we want to read from *u.user*. Note that *u.user* has more than three entries per row, but we will ignore the others.

```
users = pd.read_csv('u.user', sep='|', names=user_columns, usecols=range(3))
```

We will use the *read_csv* function to read the file, even though it isn't technically a csv(comma separated values) file. The first argument is the file name. The second says that our values are separated by *|*. The third argument merely passes in our *user_columns* as the column names. The final argument says that we need to only read three values per row. Without this, pandas will try to read the whole line and cause a mess.

In exactly the same way, we load the other values as well:

```
rating_columns = ['user_id', 'movie_id', 'rating']
ratings = pd.read_csv('u.data', sep='\t', names=rating_columns,  usecols=range(3))

movie_columns = ['movie_id', 'title']
movies = pd.read_csv('u.item', sep='|', names=movie_columns, usecols=range(2))
```

Now comes the important part. The data in the movielens dataset is spread over multiple files. But that is no good to us. We need to merge it together, so we can analyse it in one go. If you have used Sql, you will know it has a JOIN function to join tables. Pandas has something similar.

```
# create one merged DataFrame
movie_ratings = pd.merge(movies, ratings)
movie_data = pd.merge(movie_ratings, users)
```

The *merge()* function will do the same thing. It will take two different DataFrames and merge them together. You can specify an index to merge them on, but pandas is smart enough to find the common index and merge on it.

So for the first merge:

```
movie_ratings = pd.merge(movies, ratings)
```

pandas will see that *movie_id* is common between *movies* and *ratings*, so will merge on that.

We then merge this newly created *movie_ratings* with *users*:

```
movie_data = pd.merge(movie_ratings, users)
```

Again, pandas will figure out *user_id* is common and merge around that.

The final result is a dataset that contains all the info we need, so we can start working on it.

```
# Top rated movies
print("Top rated movies (overall):\n", movie_data.groupby('title').size().sort_value\
s(ascending=False)[:20]
```

The first thing we are going to do is find movies which have the most ratings. The code above may look big and confusing, but this is actually a feature of Python: You can chain multiple commands. Let's break it down.

```
 movie_data.groupby('title')
```

The *groupby()* function allows you to group the data by a chosen column (remember, the data in normally printed by index. Here, we are saying arrange the data by title, not index).

```
movie_data.groupby('title').size()
```

After we have the movies by title, we call the *size()* function to arrange them by size. Normall, this is in ascending order, so:

```
movie_data.groupby('title').size().sort_values(ascending=False)[:20]
```

we call the *sort_values(ascending=False)* function, which arranged the data in descending order (so movies with more ratings appear at top). Finally, we use *[:20]* to only show the top 20 movies.

```
Top rated movies (overall):
title
Star Wars (1977)                    583
Contact (1997)                      509
Fargo (1996)                        508
Return of the Jedi (1983)           507
Liar Liar (1997)                    485
English Patient, The (1996)         481
Scream (1996)                       478
Toy Story (1995)                    452
Air Force One (1997)                431
Independence Day (ID4) (1996)       429
Raiders of the Lost Ark (1981)      420
```

```
Godfather, The (1972)               413
Pulp Fiction (1994)                 394
Twelve Monkeys (1995)               392
Silence of the Lambs, The (1991)    390
Jerry Maguire (1996)                384
Chasing Amy (1997)                  379
Rock, The (1996)                    378
Empire Strikes Back, The (1980)     367
Star Trek: First Contact (1996)     365
dtype: int64
```

The second number is the number of votes. So Star Wars has 583 votes, making it the highest rated movie.

To no one's surprise, all the Star War movies are there (only the real ones, none of that Jar Jar Binks garbage). Scream and Liar Liar were a bit surprising, as they aren't really classics. But as we'll see later, some people may consider them classics.

Let's compare how the tastes of teenagers(13-19) vs the oldies (60+) differ.

If you want to see all movies rated by people greater than 60 years, the easiest way is:

```
movie_data[movie_data.age > 60]
```

You just pass *movie_data.age > 60* to itself, and pandas will return a DataFrame which meets this condition. Let's do that now:

```
oldies = movie_data[(movie_data.age > 60)]
```

This will return the movies sorted by index. We want the movies sorted by number of votes (as above). Let's do that now:

```
# Get the top rated ones
oldies = oldies.groupby('title').size().sort_values(ascending=False)
```

This is the same code as above, except this time we will see the top rated movies by old people only.

Let's do the same for teenagers.

```
# Extract movies for teens
teens = movie_data[(movie_data.age > 12) & (movie_data.age < 20)]
# Get the top rated ones
teens = teens.groupby('title').size().sort_values(ascending=False)
```

I'm doing the same as above, except this time I have two checks : *(movie_data.age > 12) & (movie_-data.age < 20)*. We can have as many checks as we want. We could have added a *movie_data.gender == 'F'* if we wanted teenage girls.

In the next line, again we sort by number of votes.

And now, let's print the results, only printing the top 10 results:

```
print("Top ten movies for oldies: \n", oldies[:10])
```

```
Top ten movies for oldies:
title
English Patient, The (1996)    17
Fargo (1996)                   12
Full Monty, The (1997)         10
Titanic (1997)                 10
Godfather, The (1972)           9
Seven Years in Tibet (1997)     9
Air Force One (1997)            9
Evita (1996)                    9
L.A. Confidential (1997)        8
Volcano (1997)                  8
dtype: int64
```

```
print("Top ten movies for teens: \n", teens[:10])
```

```
Top ten movies for teens:
Scream (1996)                      65
Contact (1997)                     49
Liar Liar (1997)                   47
Star Wars (1977)                   45
Return of the Jedi (1983)          43
Independence Day (ID4) (1996)      41
Titanic (1997)                     36
Toy Story (1995)                   35
Chasing Amy (1997)                 35
Twelve Monkeys (1995)              35
```

You can see that Scream is only liked by teenagers. The English Patient is the older people's favourite. But note that it only has 17 votes, which maybe just the fact that there are fewer older people compared to teenagers.

Let's now compare the the ratings by gender.

```
ratings_by_title = movie_data.groupby('title').size()
```

We get the ratings by size.

```
popular_movies = ratings_by_title.index[ratings_by_title >= 250]
```

We want to avoid movies that have one review. So we are only selecting those movies with at least 250 reviews. This is an arbitrary number. This will help us filter out popular movies only.

The next feature is very powerful. Say you want to reshape a dataset. You could perform multiple select and join operations. But the *pivot_table()* function allows us to do this more simply. Here, I want to create a new table which has the ratings of the movies selected by gender.

```
ratings_by_gender = movie_data.pivot_table('rating', index='title',columns='gender')
```

The *pivot_table()* takes the *movie_data* and reshapes it. The first argument says we want to arrange it by rating. The second argument says that the index should be the title, and the third says the columns should be based on gender. Let's see what we get:

```
print("Top rated movies by women \n", top_movies_women.head())
```

```
gender                                        F         M
title
'Til There Was You (1997)              2.200000  2.500000
1-900 (1994)                           1.000000  3.000000
101 Dalmatians (1996)                  3.116279  2.772727
12 Angry Men (1957)                    4.269231  4.363636
```

You can see we have a new table that shows the ratings by gender. But it has a lot of movies that may only have one or two ratings.

```
ratings_by_gender = ratings_by_gender.loc[popular_movies]
```

The *.loc* is another way to select data. We are selecting the data that is in *popular_movies* only, which means it has at least 250 ratings. If I print *ratings_by_gender* now:

```
gender                                F          M
title
2001: A Space Odyssey (1968)   3.491228  4.103960
Air Force One (1997)           3.690476  3.606557
Alien (1979)                   3.660714  4.123404
Aliens (1986)                  3.672727  4.013100
Amadeus (1984)                 4.038961  4.211055
Apollo 13 (1995)               4.000000  3.909953
```

That's better. We are getting movies we've heard of.

Let's see which movies were top rated by women:

```
top_movies_women = ratings_by_gender.sort_values(by='F', ascending=False)
```

You should understand what's going on by now. We are sorting *ratings_by_gender* by the 'F' column, which remember, stands for female. Let's print that:

```
print("Top rated movies by women \n", top_movies_women.head())
```

```
Top rated movies by women

gender                                          F          M
title
Schindler's List (1993)                  4.632911   4.406393
Shawshank Redemption, The (1994)         4.562500   4.410959
Usual Suspects, The (1995)               4.333333   4.399061
Silence of the Lambs, The (1991)         4.320000   4.279310
Titanic (1997)                           4.278846   4.231707
Sense and Sensibility (1995)             4.252632   3.878613
```

As you can see, most top rated movies by women also got high ratings by men. Except for the last one, *Sense and Sensibility (1995)*. Are there other movies men and women disagree on?

We are going to add a new column to our table, which will calculate the difference between the men and women's ratings:

```
ratings_by_gender['diff'] = ratings_by_gender['M'] - ratings_by_gender['F']
```

Here, *diff* is a new column. This is what it will look like:

```
print("Difference by gender \n", ratings_by_gender.head())
```

```
Difference by gender

gender                              F         M      diff
title
2001: A Space Odyssey (1968)  3.491228  4.103960  0.612732
Air Force One (1997)          3.690476  3.606557 -0.083919
Alien (1979)                  3.660714  4.123404  0.462690
Aliens (1986)                 3.672727  4.013100  0.340373
```

Now, we want to see which movies had the greatest difference in votes. After I wrote this code, I saw Wes McKinney's book on Pandas (Wes wrote the Pandas library), and he too used this example. He used standard deviation, which I guess will give a more accurate answer. I use the simple absolute value (the absolute value of the *diff* parameter we created).

```
gender_diff = ratings_by_gender['diff']
```

I store just the differences in a new DataFrame.

```
# Only get absolute values
gender_diff = abs(gender_diff)
```

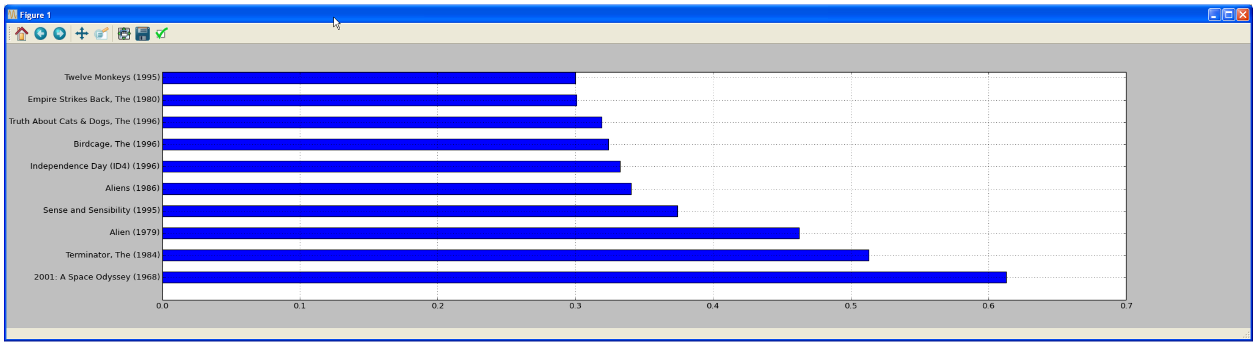
And get the absolute values, to remove the negative sign.

```
#Sort by size
gender_diff.sort_values(inplace=True, ascending = False)
```

Sort it by descending size, so the biggest differences will show up on top. And finally, we graph it:

```
# Show top 10 differences

gender_diff[:10].plot(kind='barh')
plt.show()
```

The barh is a type of bar graph, except it is drawn in the horizontal direction:



Conclusion

That was a quick intro to Pandas. My advice is, search for some open data (just Google open data. Many countries, like the UK and USA, put a lot of their government data online for free) and find something you like. Pandas is quite easy to use, and there is a lot of help online. Any time you get stuck, just Google it.

Image and Video Processing

A bit about the RGB model

Computer graphics often use the *RBG* model, which stands for Red, Green and Blue. These are the three primary colors that can be used to create other colors. If you want an overview, Wikipedia[3] has a good one.

The main thing you need to know is that you can create different colors by combining these primary colors. RGB colors usually have values of 0-255, where 0 means the color isn't present at all, and 255 means it's present with full strength. So the Rgb values for the color red are:

```
255, 0 , 0
```

So the first part is 255, which is Red. The other two are zero. This will create pure red.

You can create other colors by mixing these three. For example, pink is:

```
255, 51, 255
```

Red part is 255, green is 51 and blue is 255 again.

I found these values by Googling *rgb codes*, and opening one the dozens of results that come up.

The only other thing you need to know is OpenCv inverts this. So instead of RGB, you have BGR, or Blue, green, red.

Display an image

So we are going to start really simple. How to display an image on the screen.

You might be surprised at how hard even this simple thing is. Try to search for how to display an image with Python, and you won't find many results. I had to find a complicated example and extract the code from that.

Fire up a Python prompt and type:

```
import cv2
```

If you see no problems, you're good. Open the file *display.py*

To our code:

[3] http://en.wikipedia.org/wiki/RGB_color_model

```
import cv2
import sys
```

We import OpenCv and sys. sys will be used for reading from the command line.

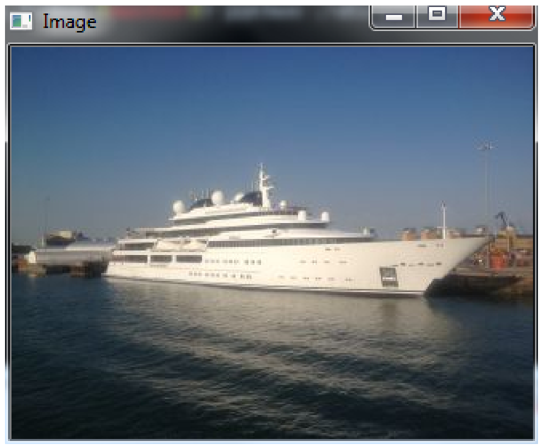
```
# Read the image. The first command line argument is the image
image = cv2.imread(sys.argv[1])
```

The function to read from an image into OpenCv is *imread()*. We give it the arugment of *sys.argv[1]*, which is just the first commandline argument. The image is read in a variable called *image*.

```
cv2.imshow("Image", image)
cv2.waitKey(0)
```

imshow() is the function that displays the image on the screen. The first value is the title of the window, the second is the image file we have previously read. *cv2.waitKey(0)* is required so that the image doesn't close immediately. It will wait for a key press before closing the image.

```
python display.py ship.jpg
```



And you should see the image.

Blur and grayscale

Two important functions in image processing are blurring and grayscale. Many image processing operations take place on grayscale (or black and white) images, as they are simpler to process (having just two colors).

Similarly, blurring is also useful in edge detection, as we will see in later examples. Open the file *blur.py*.

```
import cv2
import sys

# The first argument is the image
image = cv2.imread(sys.argv[1])
```

This is the same as before.

```
#conver to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

First, we convert the image to gray. The function that does that is *cvtColor()*. The first argument is the image to be converted, the second is the color mode. *COLOR_BGR2GRAY* stands for *Blue Green Red to Gray*.

You must have heard of the RGB color scheme. OpenCv does it the other way round- so blue is first, then green, then red.

```
#blur it
blurred_image = cv2.GaussianBlur(image, (7,7), 0)
```

If you have ever used Photoshop (or its ugly cousin Gimp), you may have heard of the Gaussian blur. It is the most popular function to blur images, as it offers good blurring at fairly fast speed. That's what we'll use.

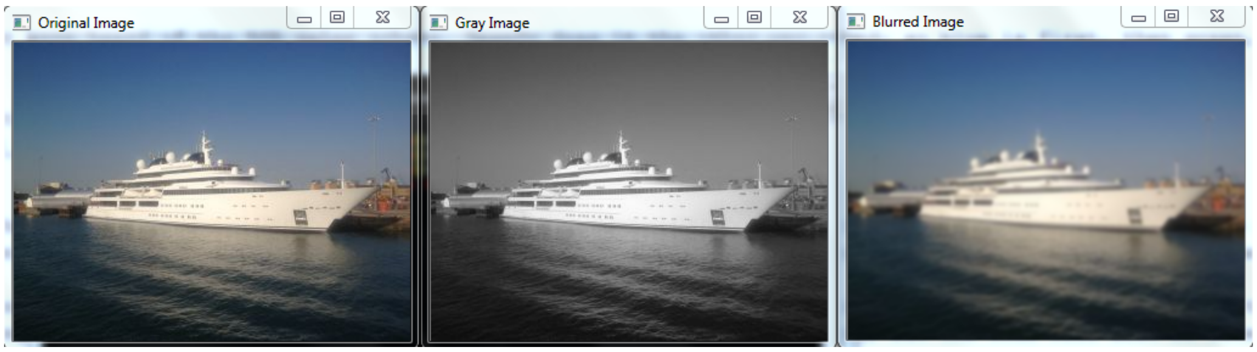
The first argument is the image itself.

The second argument is the window size. Gaussian Blur works over a small window, and blurs all the pixels in that window (by averaging their values). The larger the window, the more blurring will be done, but the code will also be slower. I'm choosing a window of *(7,7)* pixels, which is a box 7 pixels long and 7 pixels wide. The last value is not important, so I'm setting it to the default*(0)*.

```
# Show all 3 images
cv2.imshow("Original Image", image)
cv2.imshow("Gray Image", gray_image)
cv2.imshow("Blurred Image", blurred_image)

cv2.waitKey(0)
```

And now we show all images.

```
python blur.py ship.jpg
```



Edge detection

Edge detection is a very useful function in image processing. Edge detection means detecting where the edges of an object in an image are. The algorithm looks for things like change in color, brightness etc to find the edges.

The most pioneering work in this domain was done by John Canny, and his algorithm is still the most popular. You don't need to understand how the algorithms work under the hood to use them, but if you are interested in learning more, Wikipedia has good summaries:

http://en.wikipedia.org/wiki/Edge_detection

http://en.wikipedia.org/wiki/Canny_edge_detector

We will jump straight into the code. Open edge_detect.py.

```
import cv2
import sys

# The first argument is the image
image = cv2.imread(sys.argv[1])

#conver to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

#blur it
blurred_image = cv2.GaussianBlur(gray_image, (7,7), 0)

cv2.imshow("Orignal Image", image)
```

All this should be familiar, as it is similar to the last section.

```
canny = cv2.Canny(blurred_image, 10, 30)
cv2.imshow("Canny with low thresholds", canny)
```

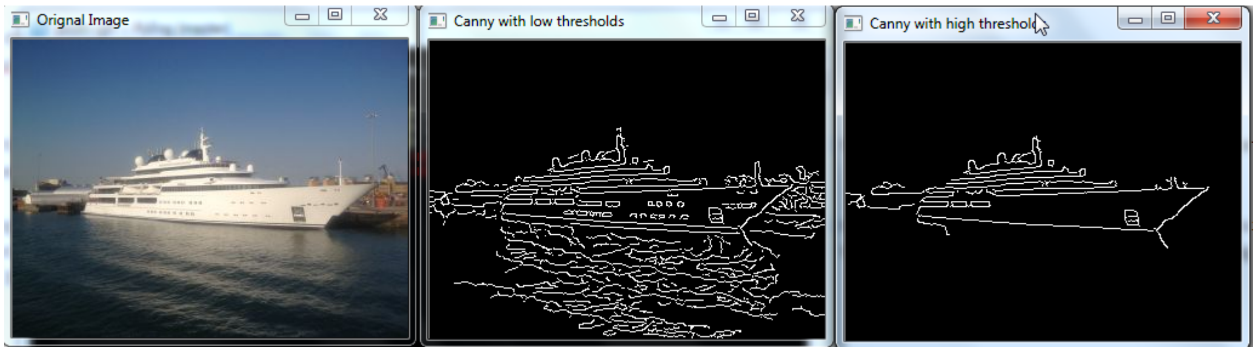
The function for Canny edge detection is, unsurprisingly, called *Canny()*. It takes three arguments. The first is the image. The second and third are the lower and upper thresholds respectively.

The Canny edge detector detects edges by looking in the difference of pixel intensities. Now, I could spend hours explaining what that means, or I could just show you. So bear with me for a moment. For the first example above, I'm using low thresholds of *10, 30*, which means a lot of thresholds will be detected.

```
canny2 = cv2.Canny(blurred_image, 50, 150)
cv2.imshow("Canny with high thresholds", canny2)
```

In this second example, we will use higher thresholds. Let's see what that means.

```
python edge_detect.py ship.jpg
```



The leftmost is the original image. The middle is the one with low thresholds. You can see it detected a lot of edges. Look inside the ship. The algorithm detected the windows of the ship, as well as a small hatch near the front. But it also detected a lot of unnecessary details in the sea. The rightmost image has the high thresholds. It didn't detect the unneeded info in the sea, but it also failed to detect the windows in the ship.

So how will you choose the thresholds? One thing I will say repeatedly in this chapter- there are no fixed answers. Try different values till you find ones you like. This is because the values will depend on your application and the type of images you are working with.

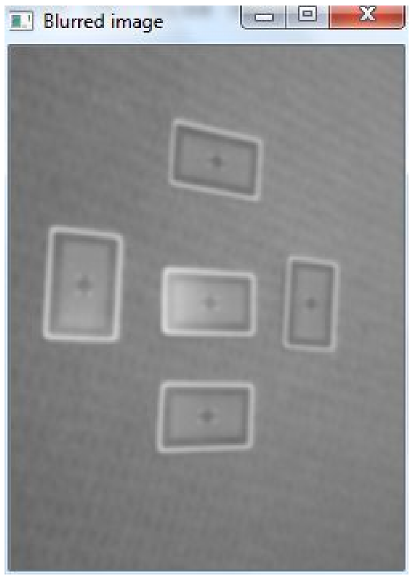
Before I close this section, a bit of info about the image. I took the photo in Southampton when on a river cruise. The ship is at the exact place where the Titanic sailed from. That parking spot costs £1000 a day (around $1500). Still cheap for a £30 million ship.

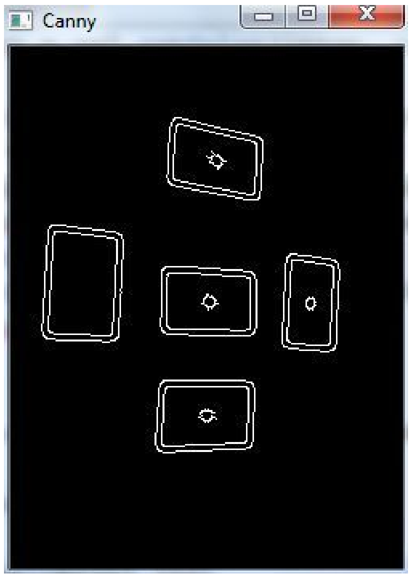
Count objects

Okay, now that we can detect the edges of an object, we can do useful stuff with it. Like detect objects. Let’s start.

```
import cv2
import sys
# Read the image
image = cv2.imread(sys.argv[1])
#convert to grayscale
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
#blur it
blurred_image = cv2.GaussianBlur(gray_image, (7,7), 0)
# Show both our images
cv2.imshow("Original image", image)
cv2.imshow("Blurred image", blurred_image)
# Run the Canny edge detector
canny = cv2.Canny(blurred_image, 30, 100)
cv2.imshow("Canny", canny)
```

This code is the same as before. We have detected the edges in the image and the blurred image.





Now, we are going to find the contours (which is just a fancy word for edges) in the image. If you are wondering why we ned to do that, since we can clearly see the edges in the image above, it's because the code isn't aware of it. The code below finds the edges programatically:

```
contours, hierarchy= cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIM\
PLE)
```

The *findContours()* finds the contours in the given image. The first option is the output of the canny edge detector. *RETR_EXTERNAL* tells OpenCv to only find the outermost edges (as you can find contours within contours). The second arguments tells OpenCv to use the simple approximation.

The function returns two values: A list of contours found, and the hierarchy (which we'll ignore. It is used if you have many contours embedded within others).

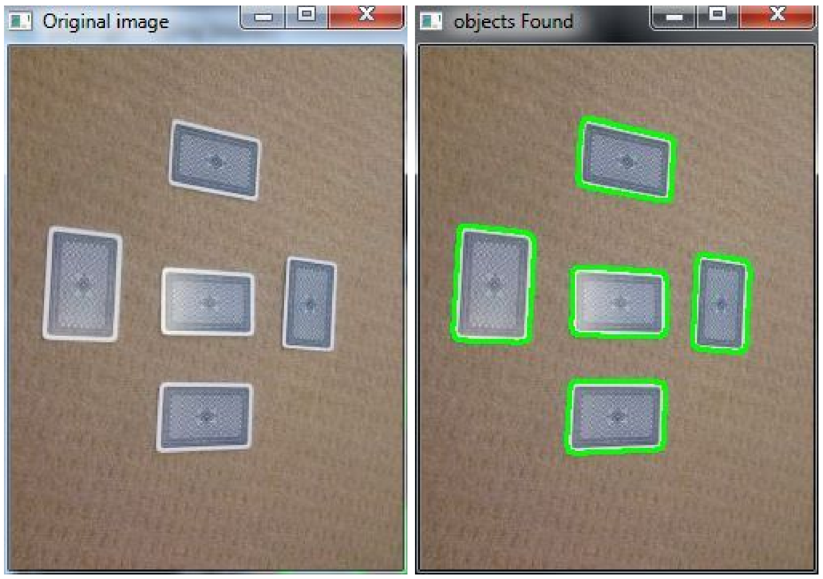
The *contours* return value is a simple list that contains the number of contours found. Taking the length of it will give us number of objects found.

```
print("Number of objects found = ", len(contours))
```

```
cv2.drawContours(image, contours, -1, (0,255,0), 2)
cv2.imshow("objects Found", objects)
cv2.waitKey(0)
```

Finally, we use the *drawContours()* function. The first argument is the image we want to draw on. The second is the contours we found in the last function. The 3rd is -1, to say that we want all contours to be drawn (we can choose to only draw certain contours). The fourth is the color, green in this case, and the last is the thickness.

Finally, we show the image.



If you want to use your own images, make sure they are not too high quality. In the first attempt, I was using Hd quality images, and opencv was detecting carpet swirls as objects. It also detected shadows as objects (including my own). Though blurring is supposed to get rid of this, if the photo is of very high quality, you will need to do a lot of blurring.

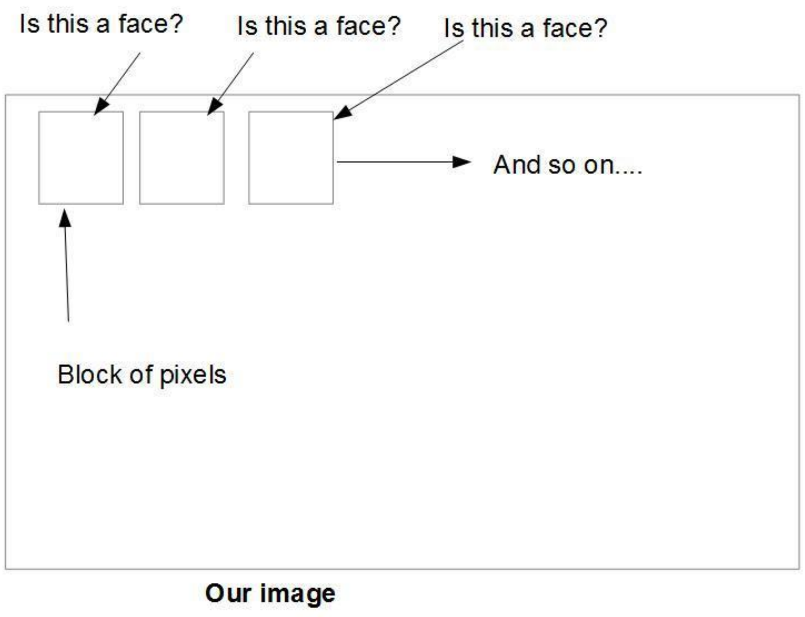
Also, make sure you have a plain background. If you have fancy tiles or something in the background, that will be detected too.

Face Detection

Face detection with OpenCV

OpenCV uses machine learning algorithms to search for faces within a picture. For something as complicated as a face, there isnt one simple test that will tell you if it found a face or not. Instead, there are thousands of small patterns/features that must be matched. The algorithms break the task of identifying the face into thousands of smaller, bite-sized tasks, each of which is easy to solve. These tasks are also called classifiers.

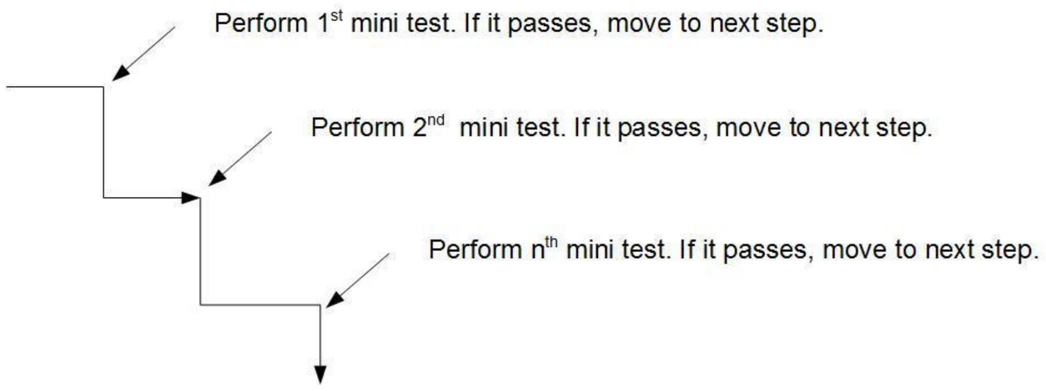
For something like a face, you might have 6,000 or more classifiers, all of which must match for a face to be detected (within error limits, of course). But therein lies the problem: For face detection, the algorithm starts at the top left of a picture and moves down across small blocks of data, looking at each block, constantly asking, Is this a face? Is this a face? Is this a face? Since there are 6,000 or more tests per block, you might have millions of calculations to do, which will grind your computer to a halt.



The image above is a rough example of how face detection works. The algorithm breaks the image into small blocks of pixels, and does the face detection on each.

To get around this, OpenCV uses cascades. Whats a cascade? The best answer can be found from the dictionary: A waterfall or series of waterfalls

Like a series of waterfalls, the OpenCV cascade breaks the problem of detecting faces into multiple stages. For each block, it does a very rough and quick test. If that passes, it does a slightly more detailed test, and so on.


Perform 1st mini test. If it passes, move to next step.
Perform 2nd mini test. If it passes, move to next step.
Perform nth mini test. If it passes, move to next step.

The algorithm may have 30-50 of these stages or cascades, and it will only detect a face if all stages pass. The advantage is that the majority of the pictures will return negative during the first few stages, which means the algorithm wont waste time testing all 6,000 features on it. Instead of taking hours, face detection can now be done in real time.

Cascades in practice

Though the theory may sound complicated, in practice it is quite easy. The cascades themselves are just a bunch of XML files that contain OpenCV data used to detect objects. You initialize your code with the cascade you want, and then it does the work for you.

Since face detection is such a common case, OpenCV comes with a number of built-in cascades for detecting everything from faces to eyes to hands and legs. There are even cascades for non-human things. For example, if you run a banana shop and want to track people stealing bananas, this guy[4] has built one for that!

Understanding the code

Lets break down the actual code, face_detect.py.

[4]http://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html

```
# Get user supplied values
imagePath = sys.argv[1]
cascPath = "haarcascade_frontalface_default.xml"
```

You first pass in the image and cascade names as command-line arguments. Well use the Abba image as well as the default cascade for detecting faces provided by OpenCV.

```
# Create the haar cascade
faceCascade = cv2.CascadeClassifier(cascPath)
```

Now we create the cascade and initialize it with our face cascade. This loads the face cascade into memory so its ready for use. Remember, the cascade is just an XML file that contains the data to detect faces.

```
# Read the image
image = cv2.imread(imagePath)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Here we read the image and convert it to grayscale. Many operations in OpenCv are done in grayscale.

```
# Detect faces in the image
faces = faceCascade.detectMultiScale(
    gray,
    scaleFactor=1.1,
    minNeighbors=5,
    minSize=(30, 30),
    flags = cv2.cv.CV_HAAR_SCALE_IMAGE
)
```

This function detects the actual face and is the key part of our code, so lets go over the options.

The detectMultiScale function is a general function that detects objects. Since we are calling it on the face cascade, thats what it detects. The first option is the grayscale image.

The second is the scaleFactor. Since some faces may be closer to the camera, they would appear bigger than those faces in the back. The scale factor compensates for this.

The detection algorithm uses a moving window to detect objects. minNeighbors defines how many objects are detected near the current one before it declares the face found. minSize, meanwhile, gives the size of each window.

I took commonly used values for these fields. In real life, you would experiment with different values for the window size, scale factor, etc., until you find one that best works for you.

The function returns a list of rectangles where it believes it found a face. Next, we will loop over where it thinks it found something.

```
print "Found {0} faces!".format(len(faces))

# Draw a rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

This function returns 4 values: the x and y location of the rectangle, and the rectangles width and height (w , h).

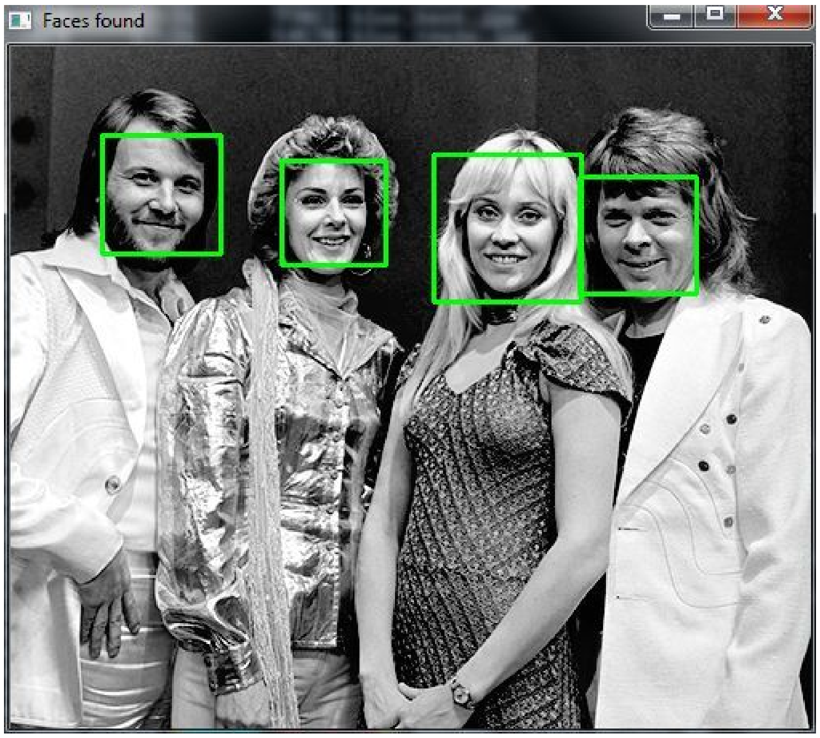
We use these values to draw a rectangle using the built-in rectangle() function.

```
cv2.imshow("Faces found" ,image)
cv2.waitKey(0)
```

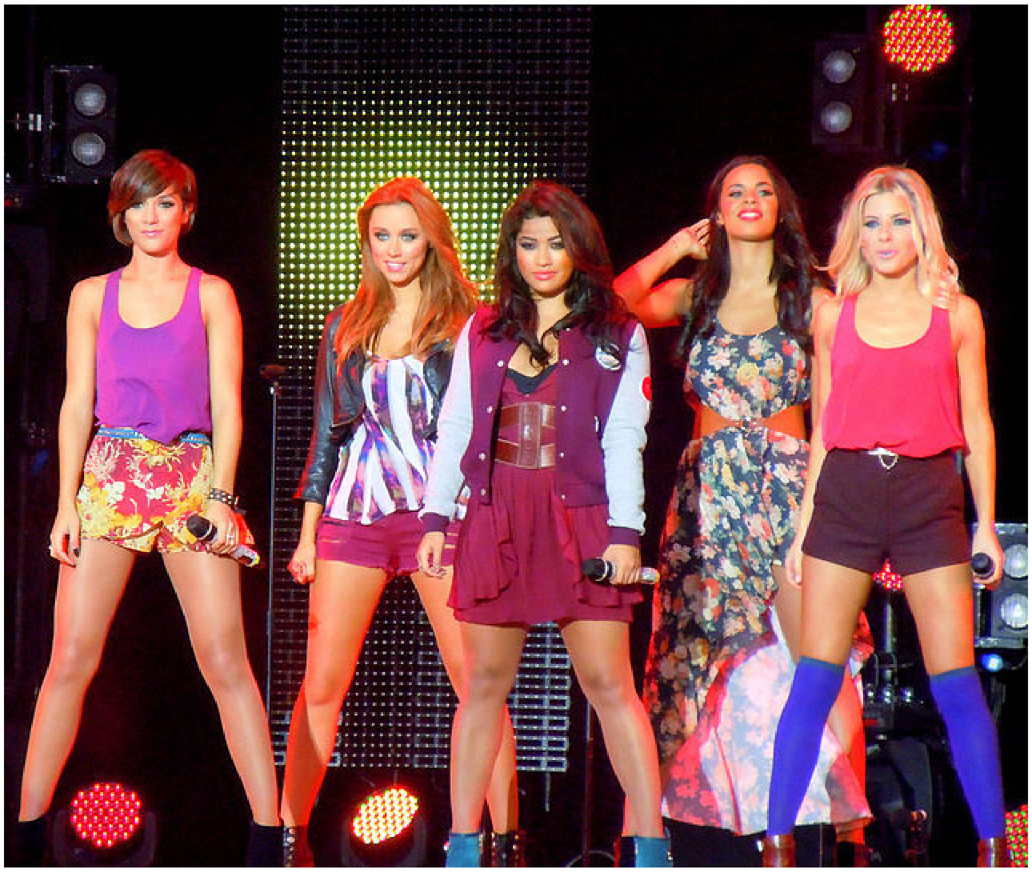
In the end, we display the image, and wait for the user to press a key. Checking the results

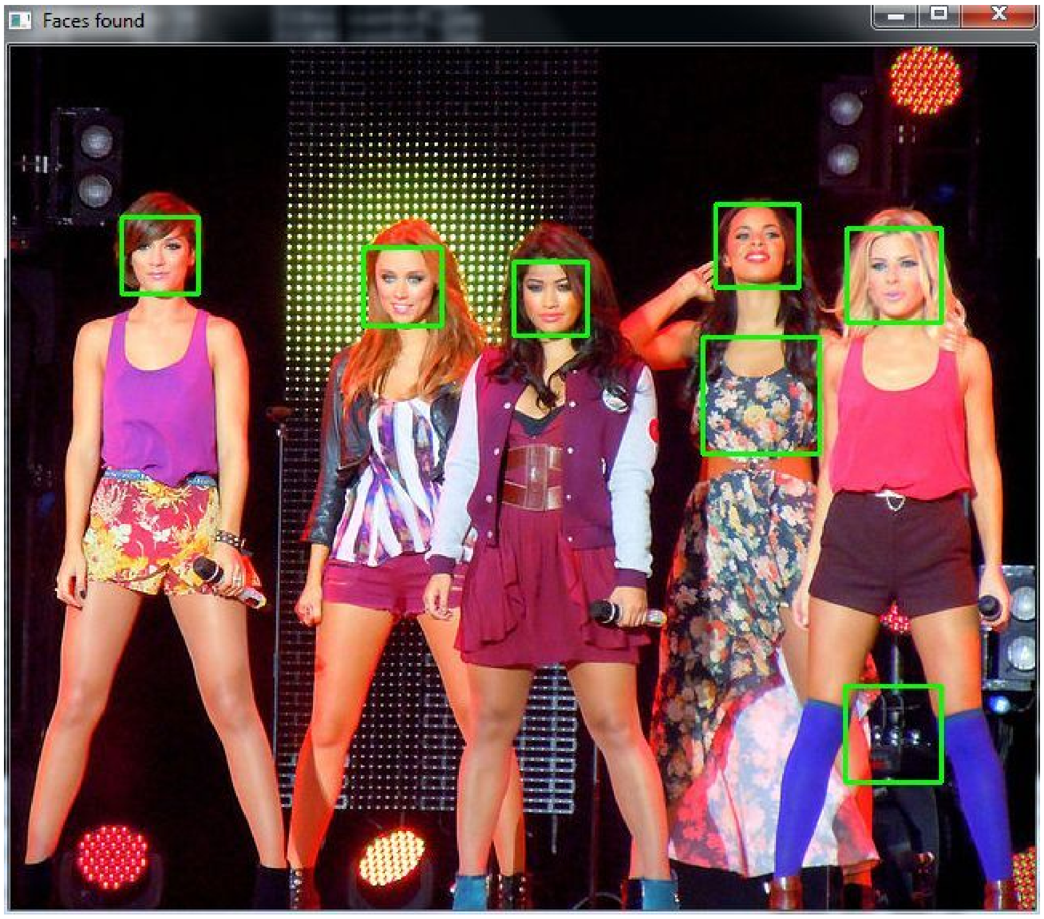
Lets test against the Abba photo:

```
$ python face_detect.py abba.png
```

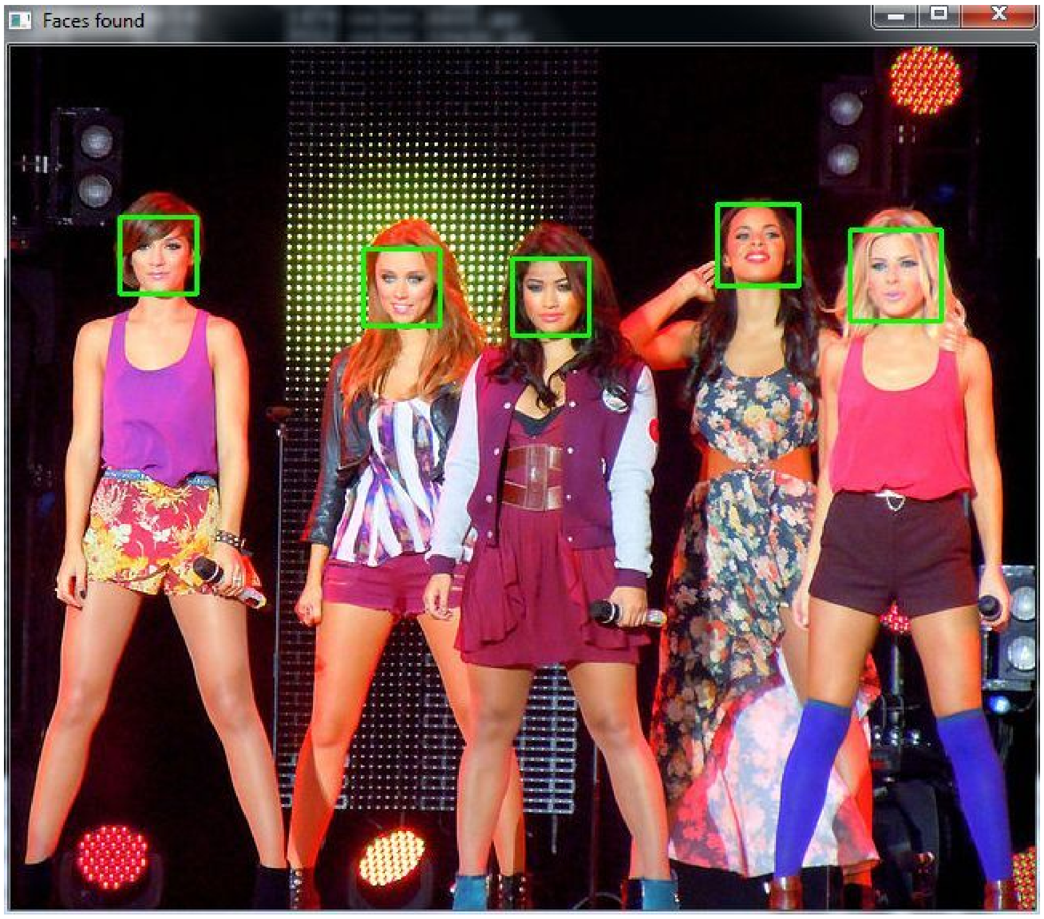


That worked. How about another photo:





That is not a face. Lets try again. I changed the parameters and found that setting the scaleFactor to 1.2 got rid of the wrong face.



What happened? Well, the first photo was taken fairly close up with a high quality camera. The second one seems to have been taken from afar and possibly from a mobile phone. This is why the scaleFactor had to be modified. As I said, youll have to tweak the algorithm on a case by case basis to avoid false positives.

Be warned though that since this is based on machine learning, the results will never be 100% accurate. You will get good enough results in most cases, but occasionally the algorithm will identify incorrect objects as faces.

Extending to a webcam

So what if you want to use a webcam? OpenCV grabs each frame from the webcam and you can then detect faces by processing each frame. You do the same processing as you do with a single image, except this time you do it frame by frame. This will require a lot of processing, though. I saw close to 90% CPU usage on my laptop.

```
import cv2
import sys

cascPath = "haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascPath)
```

This should be familiar to you. We are creating a face cascade, as we did in the image example.

```
if len(sys.argv) < 2:
    video_capture = cv2.VideoCapture(0)
else:
    video_capture = cv2.VideoCapture(sys.argv[2])
```

This line sets the video source to the default webcam (*VideoCapture(0)* always points to the default webcam) if no video file is specified. Else, it loads the file.

```
while True:
    # Capture frame-by-frame
    ret, frame = video_capture.read()
```

Here, we capture the video. The read() function reads one frame from the video source, which in this example is the webcam. This returns:

- The actual video frame read (one frame on each loop)
- A return code

The return code tells us if we have run out of frames, which will happen if we are reading from a file. This doesnt matter when reading from the webcam, since we can read forever.

```
# Capture frame-by-frame
ret, frame = video_capture.read()

gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

faces = faceCascade.detectMultiScale(
    gray,
    scaleFactor=1.1,
    minNeighbors=5,
    minSize=(30, 30),
    flags=cv2.cv.CV_HAAR_SCALE_IMAGE
)
```

```
# Draw a rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)

# Display the resulting frame
cv2.imshow('Video', frame)
```

Again, this code should be familiar as it's the same as before. We are merely searching for the face in our captured frame.

```
if cv2.waitKey(1) & 0xFF == ord('q'):
    break
```

We wait for the q key to be pressed. If it is, we exit the script.

```
# When everything is done, release the capture
video_capture.release()
cv2.destroyAllWindows()
```

Here, we are just cleaning up.

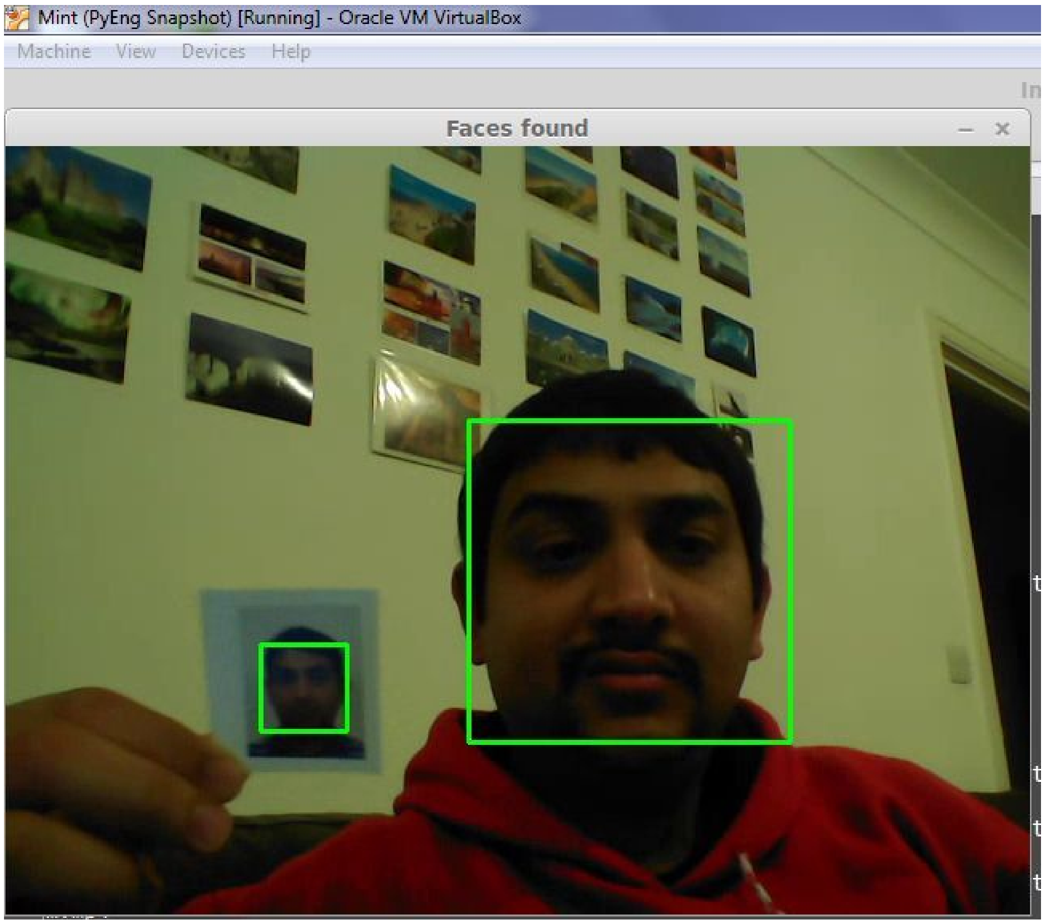
If you want to use the webcam:

```
python webcam_face_detect.py
```

If you are using the VM:

```
python webcam_face_detect.py webcam.mp4
```

This will play the file I provided. It may not work on Windows unless you have a video decoder like Ffmpeg installed.


Mint (PyEng Snapshot) [Running] - Oracle VM VirtualBox
Machine View Devices Help
Faces found

So, thats me with a passport sized photo in my hand. And you can see that the algorithm tracks both the real me and the photo me.

Like I said in the last section, machine learning based algorithms are rarely 100% accurate. We arent at the stage where Robocop driving his motorcycle at 100 mph can track criminals using low quality CCTV cameras yet.

The code searches for the face frame by frame, so it will take a fair amount of processing power. For example, on my five year old laptop, it took almost 90% of the CPU.

Motion detection

We'll use our webcam example, and extend it so it can detect motion.

How do you detect motion? You take two consecutive frames, and find the difference between them. If the difference is minor, that means no motion occurred. If you find a large difference between frames, then motion must have occurred.

```
#!/usr/bin/python

import cv2
import sys
import numpy as np

if len(sys.argv) < 2:
    video_capture = cv2.VideoCapture(0)
else:
    video_capture = cv2.VideoCapture(sys.argv[2])
```

This code is the same as before. We are creating a video capture instance.

```
# Read two frames, last and current, and convert them to gray.
# Since this is the first time, read last_frame as well.
ret, last_frame = video_capture.read()
last_frame = cv2.cvtColor(last_frame, cv2.COLOR_BGR2GRAY)

ret, current_frame = video_capture.read()
current_frame = cv2.cvtColor(current_frame, cv2.COLOR_BGR2GRAY)
```

Here we read two frames and convert them to gray. You may notice we are doing this outside the while loop. This is because we want two consecutive frames captured before the main loop starts. We call them *last_frame* and *current_frame*. Why do we need two? So that we can see the difference between them.

```
i = 0
while(True):
    # We want two frames- last and current, so that we can calculate the different b\
etween them.
    # Store the current frame as last_frame, and then read a new one
    last_frame = current_frame
```

We enter our while loop now, and the first thing we do is store the *current_frame* as the last frame. That's because we are going to read a new frame, and each loop iteration, the *current_frame* from last iteration will become the *last_frame* of this iteration.

```
     ret, current_frame = video_capture.read()
     current_frame = cv2.cvtColor(current_frame, cv2.COLOR_BGR2GRAY)
```

We read a new frame and convert it to grayscale.

```
    # Find the absolute difference between frames
    diff = cv2.absdiff(last_frame, current_frame)
```

We use the inbuilt *absdiff()* to find the absolute difference between consecutive frames.

Now, I have some code that will show us what the difference is. Before that, you must understand that OpenCv video and image frames are just numpy arrays that contain the values of all the pixels in the image or video. If you want, you can do something like to print the whole array.

```
    print(current_frame)

    [[[10 35  5]
  [ 9 34  4]
  [13 32  8]
  ...,
  [87 66 68]
  [87 70 62]
  [86 69 61]]

 [[12 34  9]
  [12 34  9]
  [19 31  8]
  ...,
  [87 66 68]
  [86 69 61]
  [86 69 61]]

 [[14 32 10]
  [14 32 10]
  [21 30  9]
  ...,
  [85 68 65]
  [86 69 61]
  [86 69 61]]
```

The above is just a snippet-you can see the array is huge.

What I will do is just print the average of the array.

I have some code that I've commented out. It prints the values of the average of the current_frame and the difference. I only print once every ten times, to avoid too much data on the screen.

```
# Uncomment the below to see the difference values
'''
i += 1
if i % 10 == 0:
    i = 0
    print np.mean(current_frame)
    print np.mean(diff)
'''
```

Here is some sample output:

```
Current frame =  83.9231391059
Diff =  5.06139973958
Current frame =  84.3319932726
Diff =  4.31867404514
Current frame =  88.6718229167
Diff =  0.461206597222
Current frame =  88.5050021701
Diff =  0.156022135417
Current frame =  89.7750976562
Diff =  2.8298578559
```

The above is without motion. With motion:

```
Current frame =  82.5932790799
Diff =  7.46467230903
```

As you can see, the average of the difference frame is very little when you aren't moving. Try it yourself (if you have a webcam). Sit silently for a few seconds, and you will see the difference is 1.0 or less. Start moving around, and it will jump to 10 or even more.

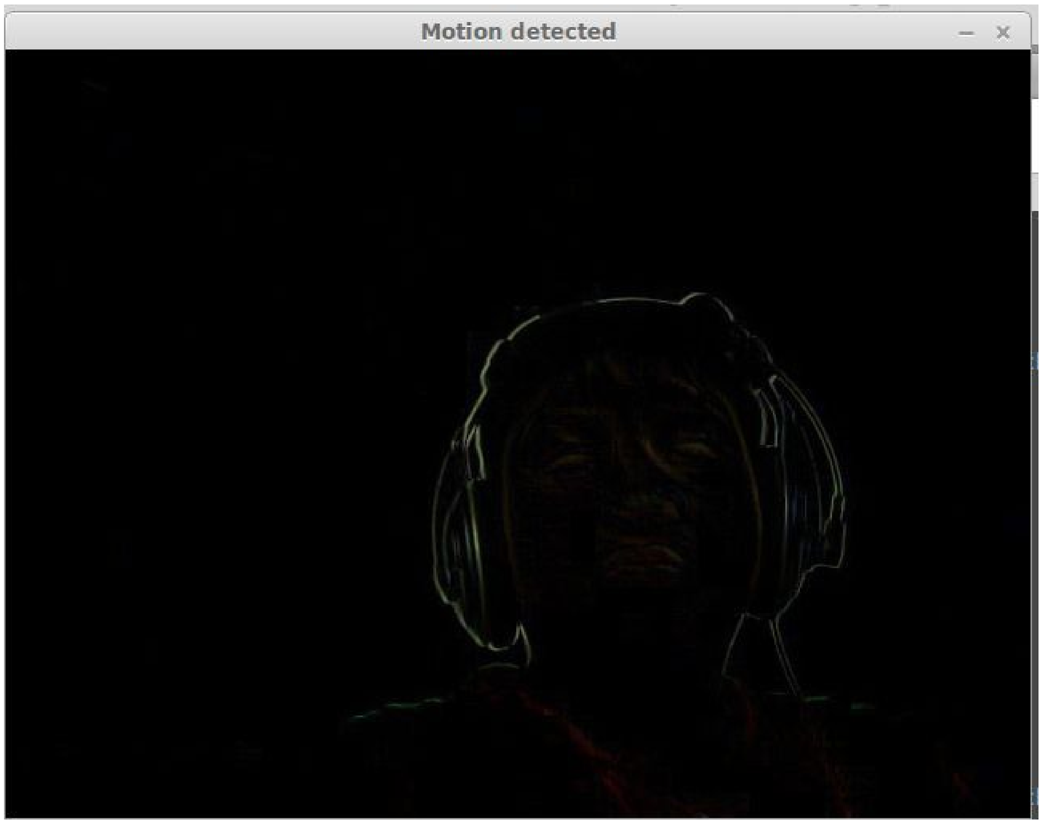
That's how I got the values I'm going to use- by experimentation:

```
    # If difference is greater than a threshold, that means motion detected.
    if np.mean(diff) > 10:
        print("Achtung! Motion detected.")
```

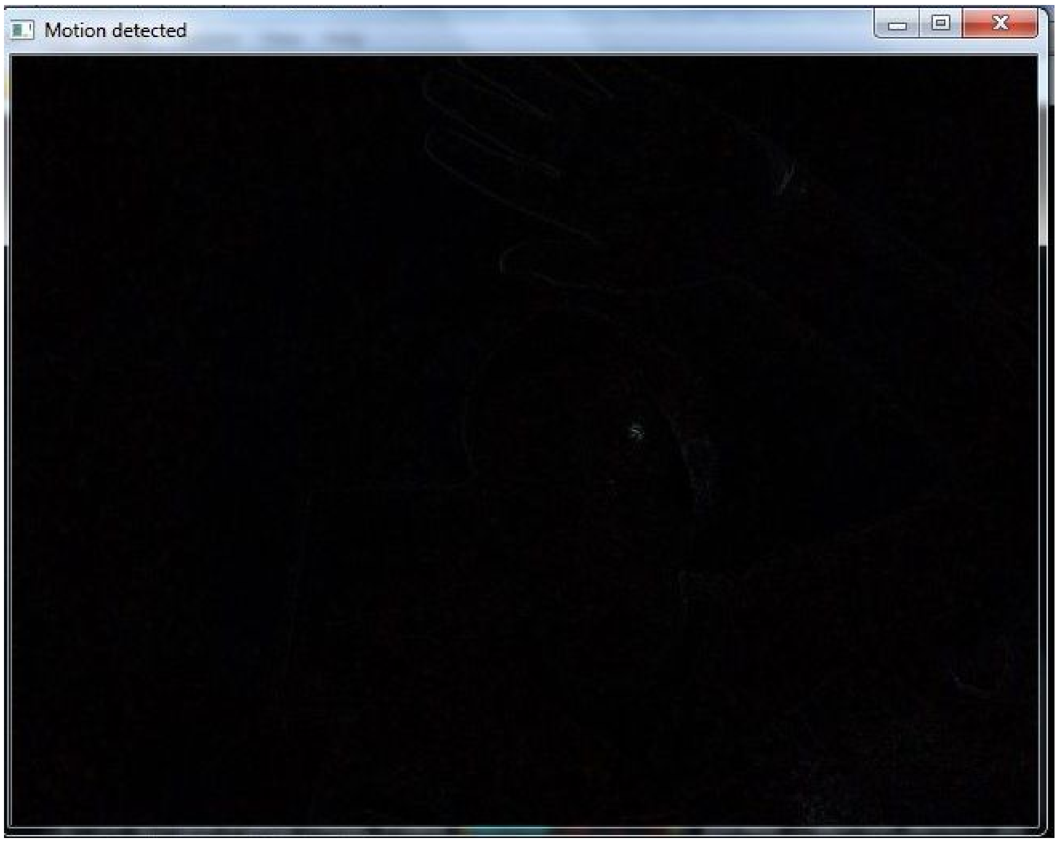
If the average difference is greater than 10 (a value I got by experiment), I take it to mean motion has been detected, and print the warning.

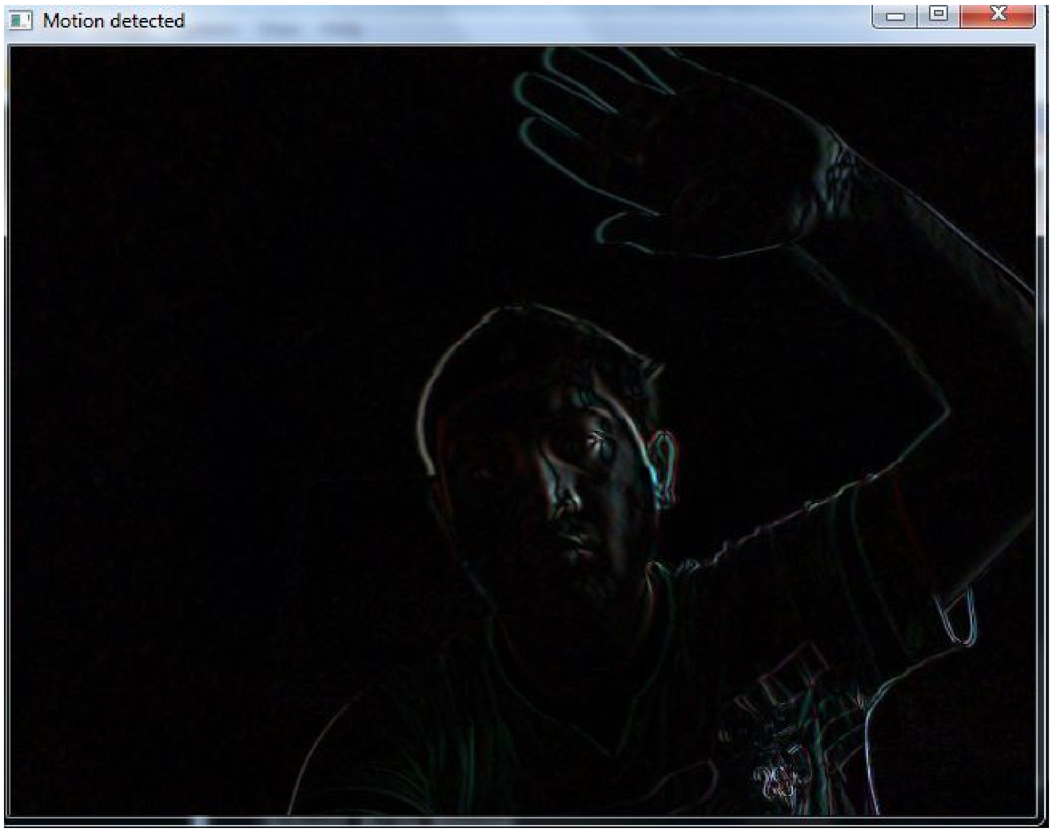
```
    # Display the resulting frame
    cv2.imshow('Video',diff)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
# When everything done, release the capture
video_capture.release()
cv2.destroyAllWindows()
```

Finally, we display our difference image and exit.



Here is another example taken from my webcam. In the first image, I'm just sitting there. In the second one, I move a little:


Motion detected


Motion detected

MultiThreading vs Multiprocessing in Python

Threads vs Processes on Linux

I will stick to Linux for this chapter, as Python is closely tied with Linux development. The theory is the same for Windows as well, although the underlying implementation differs.

Processes

Any program is run in Linux as a process. Each process has it's own memory, stack and access to shared peripherals. As far as each process is concerned, it has the whole computer to itself (a feeling many Windows programs take to an extreme). Additionally, if you have done any Windows programming (not programming *on* Windows, but programming the Microsoft Windows API), you will find that there is some difference in terminology. Microsoft has a few special terms it uses for its processes. I won't go into the differences here.

So how do multiple processes run at the same time? That's the job of the operating system. It ensures that each process can only access its own memory, gets a chance to run on the CPU, can access external peripherals like network and so on.

Processes are considered heavy duty as each starts with a complete execution environment. To get around this, sometimes programmers uses threads.

Threads

A thread is a lightweight process. Basically, they are smaller units of a running process. Which means they share the memory and other resources of the process that started them.

You can immediately see the benefit of this. If you have a complicated algorithm, break it into threads and run it in parallel. Since all the threads can see the common memory, they can all work together without a complicated messaging mechanism. Everyone's happy!

Not. Threads are messy and dangerous, and can lead to horrible bugs than occur once in six months. This is due to things like timing issues (does Thread A run before Thread B?). It can also lead to a very dangerous thing called race conditions, where two threads "race" to update the same code/memory. This leads to unpredictable behaviour.

Obviously, there are ways around this. Locks, semaphores etc were invented to get around these issues. I won't go into that since it's beyond the scope of this chapter.

Python doesn't have thread support builtin. Instead, it uses the thread of the operating system (Pthreads on Linux, Windows threads on Windows).

But let's say you find a dream problem which is truly independent. There is no data sharing, so no risk of race conditions or timing problems. Such a thing is possible in heavily mathematical based problems. What then? In that cases, threads should always be used, right?

Thread vs single execution

Let's check our theory.

Our first file is single.py.

```python
import random

def gen_random_data():
    arr = []
    for i in range(100000):
        arr.append(random.random())
    print("job done")
```

We have a simple function that generates random data. This is to simulate a purely mathematical problem. We run this code 4 times:

```python
if __name__ == '__main__':
    gen_random_data()
    gen_random_data()
    gen_random_data()
    gen_random_data()
```

I am running the code four times, to see what effect using threads has (as each call above will have its own thread).

Now, let's open our next file, thread.py.

```python
import threading
import random
```

We are importing the threading module this time.

```
def gen_random_data():
    arr = []
    for i in range(100000):
        arr.append(random.random())
    print "job done"
```

We have the same function as before.

```
if __name__ == '__main__':

    for x in range(4):
        process = threading.Thread(target=gen_random_data)
        process.start()
```

threading.Thread creates a Thread. *target* is the name of the function we want to call. In our case, it's *gen_random_data*. We create a thread and start it off with *process.start()*.

All fairly simple. Since the thread function is running the four threads in parallel, it should be faster, right?

I have a machine with 4 cores. Let's try it out.

I'm going to use the Linux *time* utility to time the code. Your results may vary based on your operating system, your processor, but the pattern should be the same.

I ran the code three times, and took the most common value. First, the single execution code:

```
time ./single.py
job done
job done
job done
job done

real    0m0.601s
user    0m0.564s
sys     0m0.016s
```

real is the value we care about, and we see it took 0.6 seconds. Now, the multithreaded code:

```
time ./thread.py
job done
job done
job done
job done

real    0m1.033s
user    0m1.012s
sys     0m0.304s
```

Wait, that took more than a minute. How's that possible? Why did threading actually slow our code? The threaded code is almost twice as slow!

Welcome to the Global Interpreter Lock (GIL). GIL is a "feature" of Python that prevents threads from running in parallel. The GIL forces each thread to wait for access to the CPU, which means in practice it is the same as running a single threaded code, except you have all the overhead of creating and managing threads. Read more about it here[5].

To get around this problem, the Multiprocessing module was created. It's function calls are similar to the Threading module, so that's it's easier to migrate between the two.

Let's see the same code with multiprocessing.

```python
#!/usr/bin/python
import multiprocessing
import random
from gen_rand import gen_random_data

if __name__ == '__main__':
    for x in range(4):
        process = multiprocessing.Process(target=gen_random_data)
        process.start()
```

You can see the code is similar to threading, except we call the multiprocessing module. Let's give this a spin:

[5]http://www.dabeaz.com/GIL/

```
 time ./process.py
job done
job done
job done
job done

real    0m0.233s
user    0m0.644s
sys     0m0.064s
```

This time, we can see its taking a third of the time of the single execution run. Why a third, when I have four cores? The main reason is that there is an overhead in creating processes. There are also other factors, like what other processes are running in the background, how much disk/network access is required.

But multiprocessing isn't always the answer

The problem with multiprocessing is that there is no shared state. So if you want to pass data between processes, you have to roll your own solution (or use something the operating system provides, like sockets in Linux).

If you have ever written multi-threaded (or multiprocess) code, you will know it can get messy very soon. It's okay for tiny problems, but not for production code, especially if it has been bolted on in a very ugly fashion, like it has for Python.

So I hate to say this in a Python book, but if you must have multithreaded code, I'd recommend using a language with it built in, like Erlang or Go.

Machine Learning with an Amazon like Recommendation Engine

If you have been to Amazon, you must have seen the "also bought" section, which recommends which other movies / books you will like, based on the movies your currently bought/rated.

Amazon is a special case of this, as it hires hundreds of engineers whose job is to tweak this algorithm. And it works great for things like books. I have bought many great books based on Amazon's recommendation. It works for movies as well. Where is fails is for items that are not usually rated/reviewed, at least as much as books/movies. Things like household products. I recently bought a drain cleaner, and was immediately bombarded with toilet cleaning products, even though the cleaner had been a one time buy.

That said, their algorithm works well in the general case, and is the main reason Amazon has become such a powerhouse. Amazon always recommends the items it thinks you will like. If you think that is a big deal, try going to Amazon's rivals for books, like B&N, Apple or Kobo. They always have the same 4-5 books they always recommend, a list that is updated once a month, and usually represents books by big publishers who have paid top dollar to be featured.

Like I said, Amazon's algorithm is highly tweaked (and secret), and is based on years of experience. While we can't replicate what they did, we can understand the theory of how the algorithm works.

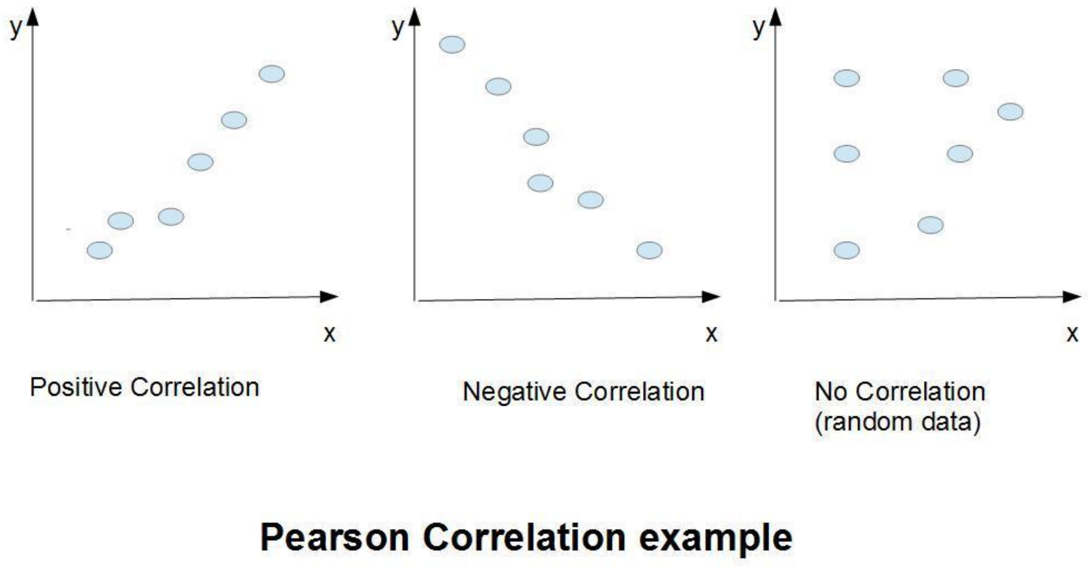
At its heart, the algorithm looks at items you have rated, and tries to find people who rated the same items in the same way as you. It then checks which other books/movies they liked that you haven't bought yet, and recommends it to you.

So if you constantly rate action movies as high, the algorithm will show you other highly recommended action movies. But what if you like both action and scifi? Based on my experience, you will get recommendations for both.

So how does the algorithm know which movies to recommend? In technical terms, it looks which movies are most co-related to the ones you rated.

Pearson Correlation

The Pearson Correlation coefficient measures how strongly related two items are, ie, will increasing one increase the other as well? The easiest way to understand this is via a diagram.



The first image is of positive correlation- the values are moving up in step. The second is an example of negative correlation, ie, increasing one will decrease the other. That means they are inversely related. The third is random data- it shows no correlation. This is actually the most common case, as not everything has correlation to everything else (unless you believe in weird psychic phenomena).

How does this work in practice? Simple. Again, numpy already has functions for Pearson correlation. All you need to do is call them with different inputs.

In fact, numpy has two functions for the Pearson coefficient. One is really slow, and one is fast. I don't know why they have two functions, must be a historical thing. I will only use the faster function. Let me give you a few quick examples on how it works.

```
a = [1, 2, 3, 4, 5]
b = [1, 3, 9, 20, 22]

np.corrcoef(a,b)[1][0]

Out[8]: 0.969954025101608
```

np.corrcoef() is the function we are using. It returns a value from -1 to 1. -1 is strong negative correlation, while +1 is strong positive correlation. 0 means no correlation. You may also note that it returns an array, and I'm reading the *[1][0]*th value. That is because this function can be used to compare multiple arrays, and so it returns a matrix of correlated values. For our case with only two arrays, we just need one of the values returned, and we read this one.

I declare two arrays, *a* and *b*. Notice that both contain increasing numbers. When I call the *np.corrcoef()* function on them, I get a value of 0.9, very high correlation.

Now let's try a more random input:

```
a = [1, 2, 3, 4, 5]
c = [2 , 7, 9, 1, 0]
np.corrcoef(a,c)[1][0]
Out[9]: -0.39904344223381105
```

This time I get -0.3, which makes sense, as there is no correlation between random data.

This will become more clear as we look at the example.

Dive into the code

Okay, now that we know the theory, let's dive into the code.

I have generated some random data for a few movie ratings. Let's have a look at it (ml_data1.py):

```
{'Terminator': {'Tom': 4.0,
  'Jack': 1.5,
  'Lisa': 3.0,
  'Sally': 2.0},

 'Terminator 2': {'Tom': 5.0,
  'Jack' : 1.0,
  'Lisa': 3.5,
  'Sally': 2.0},

 'It happened one night': {'Tom': 3.5,
  'Jack': 3.5,
  'Tiger': 4.0,
  'Lisa': 5.0,
  'Michele': 3.0,
  'Sally': 4.0,},

 '27 Dresses': {'Tom': 3.0,
  'Jack': 3.5,
  'Tiger': 3.0,
  'Lisa': 5.0,
  'Michele': 4.0,
  'Sally': 4.0},

  'Poirot': {'Tom': 4.0,
  'Jack': 3.0,
  'Tiger': 5.0,
  'Lisa': 4.0,
  'Michele': 3.5,
  'Sally': 3.0,
  },

 'Sherlock Holmes': {'Tom': 4.0,
  'Jack': 3.0,
  'Tiger': 3.5,
  'Lisa': 3.5,
  'Sally': 2.0,
  }}
```

The movie data is stored as a dictionary. Each dictionary has its own sub-dictionary. Let's have a look at the first movie:

```
'Terminator': {'Tom': 4.0,
  'Jack': 1.5,
  'Lisa': 3.0,
  'Sally': 2.0},
```

The movie Terminator has been rated by four people: Tom has given it a score of 4.0, while Jack has given it 1.5, and so on. These numbers are random (ie, I just made them up).

You will notice that not all people have rated all the movies. This is something we will need to take into account when we are calculating the correlation.

Let's open up the file *calc_correlation.py* Ignore the function for now, let's see the main code:

```
if len(sys.argv) < 2:
    print("Usage: python calc_correlation.py <data file.py>")
    exit(1)
```

We want to give the script a data file to calculate the correlation on, in this case, ml_data1.py, the file we looked at before. If the file is not given, we print the usage and exit.

```
with open(sys.argv[1], 'r') as f:
    temp = f.read()
    movies_list = ast.literal_eval(temp)
    print(movies_list)
```

If you have never used the *with* function in Python, it's a cool fairly new feature. Normally, when you open a file, you have to close it, deal with any errors etc. *with* does all that for you. It will open the file, close it at the end, and handle any errors that may arise.

Looking at the code line by line.

```
with open(sys.argv[1], 'r') as f:
```

We open the file passed as first argument as read only.

```
temp = f.read()
movies_list = ast.literal_eval(temp)
```

The first thing we do is read the file into a *temp* variable. The next line may require some explanation.

A nifty feature in Python is the *eval()* function. It allows you to generate Python code in real time and run it. Obviously, this is very dangerous, as you don't know what you are reading. Someone may put in code to delete all your files. For this reason, *eval* is almost never used in practice.

The *literal_eval()* function in the *ast* library gets around the risks of *eval. literal_eval* will only allow Python data structures like lists, dictionaries, or any other Python data structure to be read. If you try to read anything else, it will throw an error.

But why do I need it anyway? The file I am reading, *ml_data1.py* contains a Python dictionary. However, when we read the file, it is read as a text file. I need to convert it to a Python dictionary. That is what *literal_eval* will do. It takes the data it read and converts it to a Python dictionary I can use. If I try to pass in something dangerous, the function will throw an error.

There are other ways to do this, like using the *json* module, which I will show you later.

Anyway, now we have read the data into a file.

```
correlated_dict = {}
for movie in movies_list:
    correlated_dict[movie] =  find_correlation(movies_list, movie)
```

movies_list is what we read in from the file. We loop over that and find the correlation for each movie. Let's looks at how the function *find_correlation()* does that.

```
def find_correlation(movie_list, movie_for_correlation):
    '''
    Input:
    movie_list - List of movies
    movie_for_correlation: The movie to calculate the correlation for

    Return:
    Dictionary of correlation for movie_for_correlation
    '''
    correlate_dict = {}
```

The function takes two parameters: A list of movies, and the movie to find the correlation for (within that list). It returns a dictionary of correlated values.

```
correlate_dict = {}
for movie in movie_list:
```

We declare *correlate_dict*, the final dictionary we will return. We then loop over the movie list.

```
# Don't include current movie in correlation, as you can't compare a movie to itself!
if movie != movie_for_correlation:

    movie_for_correlation_list = []
    movie_to_compare_list = []
```

When doing the calculations, we don't want to compare a movie to itself (as that would always show perfect correlation). So we check for that, and then declare a few empty arrays.

Remember our data file?

```
{'Terminator': {'Tom': 4.0,
  'Jack': 1.5,
  'Lisa': 3.0,
  'Sally': 2.0},
```

See that we have the movie, and then the people who reviewed it? Now, we loop over the reviewers.

```
# Loop through the people who reviewed the movie
for reviewer_name in movie_list[movie_for_correlation]:

    # Check that the reviewer has reviewed the current movie.
    # If so, calculate the correlation coefficient.
    # If they haven't reviewed the movie, then it makes no sense doing a correlation.

    if reviewer_name in movie_list[movie]:
```

One check we do (in the last line above) is to check that *this* reviewer (the one we are looping over), reviewed the current movie. If s/he didn't, we move to the next one.

```
if reviewer_name in movie_list[movie]:
        movie_for_correlation_list.append(movie_list[movie_for_correlation][reviewer\
_name])
        movie_to_compare_list.append(movie_list[movie][reviewer_name])
```

We are still in the for loop. If the reviewer reviewed the movie, we save their score for the current movie, as well as their score for the movie under consideration.

To make it clear, let's look at the whole for loop:

```
# Loop through the people who reviewed the movie
for reviewer_name in movie_list[movie_for_correlation]:

    # Check that the reviewer has reviewed the current movie.
    # If so, calculate the correlation coefficient.
    # If they haven't reviewed the movie, then it makes no sense doing a correlation.

    if reviewer_name in movie_list[movie]:
        movie_for_correlation_list.append(movie_list[movie_for_correlation][reviewer\
_name])
        movie_to_compare_list.append(movie_list[movie][reviewer_name])
```

This for loop will loop over the reviewers, and if they reviewed both the movies, will store their scores in two arrays. That's all we are doing. The complicated code is to compensate for the fact that not all people reviewed all movies.

```
correlate_dict[movie] = np.corrcoef(movie_for_correlation_list,movie_to_compare_list\
)[1][0]

return correlate_dict
```

Finally, we calculate the correlation using the *np.corrcoef()* function we saw earlier. The function creates a dictionary with correlation scores for the current movie. For example, if the movie is *Terminator*, the dictionary would store what correlation score *Terminator* has to *Poirot*, to *Sherlock Holmes* etc.

Each movie will have its own list. So *Poirot* will have its own dictionary which will store its relation to *Terminator* and other movies.

We return the dictionary.

Back in the main loop:

```
for movie in movies_list:
    correlated_dict[movie] =  find_correlation(movies_list, movie)
print(correlated_dict)
```

As I said, we are creating a correlation dictionary for each movie. The goal is to create a correlation index we can use later to make recommendations.

```
json.dump(correlated_dict, open("corr_dict.py",'w'))
```

Remember I said I'd show you another way to write Python dictionaries to file? This is the easier way, using the json module. It opens the file and dumps our dictionary in one go. Let's open our file *corr_dict.py* and look at it. I've cleaned it a bit.

```
"27 Dresses": {"Poirot": -0.30434782608695654, "Terminator 2": -0.12547286652195427,\
 "It happened one night": 0.5791405068790082, "Sherlock Holmes": -0.2758386421836852\
6, "Terminator": -0.15404159684748153},
```

So *27 Dresses* has a -0.3 correlation to *Poirot*, but a +0.5 to *It happened one night*, which makes sense as both as romantic movies (or not, as I deliberately made up these examples to have this correlation).

Building the recommendation engine

To understand what I'm doing, let's work through a few examples.

Using correlation to rate movies

Say you have two movies, A and B. You have given a rating of 4.0 to A.

The correlation between A and B is 1.0. Since 1.0 is the highest value, that means the movies are perfectly correlated, and the person who likes one will like the other.

So your estimated score for B will be:

4.0 x 1.0 = 4.0

So you would give a rating of 4.0 to movie B as well (at least according to the algorithm).

What if the correlation was 0.5? Your calculated score for B would be:

4.0 x 0.5 = 2.0

And what if it was -0.5? Your score would be *-2.0* , which means you would hate the movie.

Remember, what I'm doing is looking at the movies I've rated, what their correlation is to movies I haven't rated, and then try to find my anticipated score. And I do this for all the 3 movies I've rated.

Before we start on that, I've cleaned up corr_dict.py for humans (as it's originally only used by machines). The cleaned copy is in *corr_dict_cleaned.py*:

```
{
"27 Dresses": {"Poirot": -0.30434782608695654, "Terminator 2": -0.13998740998253317,\
 "It happened one night": 0.5791405068790082, "Sherlock Holmes": -0.2758386421836852\
6, "Terminator": -0.16796775328675631},

"It happened one night": {"Terminator 2": 0.25275763912268084, "27 Dresses": 0.57914\
05068790082, "Terminator": 0.42437890059987993, "Sherlock Holmes": 0.0, "Poirot": 0.\
2895702534395041},

"Terminator 2": {"Poirot": 0.87845527687491742, "27 Dresses": -0.13998740998253317, \
"It happened one night": 0.25275763912268084, "Sherlock Holmes": 0.73889576951817515\
, "Terminator": 0.92184471452179317},

"Terminator": {"Terminator 2": 0.92184471452179317, "27 Dresses": -0.167967753286756\
31, "It happened one night": 0.42437890059987993, "Sherlock Holmes": 0.7702079842374\
0755, "Poirot": 0.72664794872022465},

"Poirot": {"Terminator 2": 0.87845527687491742, "27 Dresses": -0.30434782608695654, \
"It happened one night": 0.2895702534395041, "Sherlock Holmes": 0.6698938453032357, \
"Terminator": 0.72664794872022465},

"Sherlock Holmes": {"Poirot": 0.6698938453032357, "Terminator 2": 0.7388957695181751\
5, "27 Dresses": -0.27583864218368526, "It happened one night": 0.0, "Terminator": 0\
.77020798423740755}

}
```

Let's start with 27 dresses, and try to calculate its anticipated score:

My Movie (A)	**My rating for movie (B)**	**Correlation of (B) with 27 dresses (C)**	**My calculated score: (B) * (C)**
Terminator	5.0	-0.16	-0.8
Sherlock Holmes	4.0	-0.27	-1.08
Poirot	4.5	-0.3	-1.35
			Total: -3.23

So table A lists the movies I rated, B is the score I gave them. C is the correlation the current movie (27 Dresses) to the movie I rated (Terminator). I multiply B by C to get the score I would be expected to give to the current movie (27 dresses).

So for the very first movie, Terminator, we get a score of *5.0 x -0.16 = -0.8*.

I then do this for all three movies I have rated, and total them up to get one score.

The total score is -3.23. There are many ways to get the average score, but I will use the simplest: Average. I divide the above with number of movies I rated (three) to get -1.07. This is my expected rating for *27 Dresses*. Clearly, this movie is not recommended.

Let's do the same for Terminator 2:

My Movie (A)	**My rating for movie (B)**	**Correlation of (B) with 27 dresses (C)**	**My calculated score: (B) * (C)**
Terminator	5.0	0.92	4.6
Sherlock Holmes	4.0	0.73	2.92
Poirot	4.5	0.87	3.9
			Total: 11.43

Dividing by 3, the average score is 3.8 for *Terminator 2*. This movie would be strong recommended.

Let's now look at the code in *ml_main.py*:

```
import pdb
import json


# My ratings for movies.
my_movies = {'Terminator': 5.0,
      'Sherlock Holmes' : 4.0,
      'Poirot' : 4.5
      }
```

After importing the modules we need, we declare a dictionary called *my_movies*, which contains a few movies that I have rated. This will be used to drive our recommendation engine.

```
# Read the data from the correlation dictionary we calculated earlier
correlated_dict = json.load(open("corr_dict.py"))

# A dictionary to store the total of my calculated votes
total_my_votes = {}

# A running total  to store intermediate results.
running_total = 0
```

We open the dictionary we created, *corr_dict.py*, and declare some variables. We are using the *json* library we used last time.

```
# Loop over rated movies
for movie_key in my_movies.keys():
```

We loop over the movies we rated.

```
# Loop over the dictionary of correlation coefficients
for movie_to_compare in correlated_dict[movie_key]:
    running_total = 0
```

For the current movie we are looping over, we look at the correlation coefficients. To remind you what that means, for our first movie *Terminator*, these are the coefficients:

```
"Terminator":
{
"Terminator 2": 0.92184471452179317,
"27 Dresses": -0.16796775328675631,
"It happened one night": 0.42437890059987993,
"Sherlock Holmes": 0.77020798423740755,
"Poirot": 0.72664794872022465
 }
```

The above shows that Terminator has a 0.92 correlation with Terminator 2. Since 1.0 is the max value, this shows a very strong correlation.

In the next line, we loop over all the movies we have a correlation for. Obviously, this will include movies we have already rated. So our next step is to get rid of them, as we only want the correlation for movies we have not seen or rated.

```
if movie_to_compare not in my_movies.keys():
```

Next line:

```
if movie_to_compare not in total_my_votes:
    # Line below creates a new dictionary element for total_my_votes and gives it a \
value.
    total_my_votes.setdefault(movie_to_compare, (correlated_dict[movie_key][movie_to\
_compare] * my_movies[movie_key]) )

else:
    # If this is not the first time, merely update the values we have created before
    total_my_votes[movie_to_compare] += correlated_dict[movie_key][movie_to_compare]\
 * my_movies[movie_key]
```

What I am doing is calculating the correlation for all the movies. The first time we enter the loop, this code is called:

```
if movie_to_compare not in total_my_votes:
    # Line below creates a new dictionary element for total_my_votes and gives it a \
value.
    total_my_votes.setdefault(movie_to_compare, (correlated_dict[movie_key][movie_to\
_compare] * my_movies[movie_key]) )
```

It merely says that if this is the first time, create a new dictionary element for *total_my_-votes[movie_to_compare]* , and give it the value of *correlated_dict[movie_key][movie_to_compare] x my_movies[movie_key]*, which is the same as the calculations I showed you earlier. That's all the function *setdefault* does: It creates a dictionary element if none exists.

The second time we enter the loop, we update the dictionary value:

```
else:
    # If this is not the first time, merely update the values we have created before
    total_my_votes[movie_to_compare] += correlated_dict[movie_key][movie_to_compare]\
 * my_movies[movie_key]
```

As you can see, I am keeping the total. As I explained in the example above, I keep the total, and later average it. We do it like this:

```
recommended_movies = {}
for movie_key in total_my_votes.keys():
    recommended_movies[movie_key] =  total_my_votes[movie_key] / len(total_my_votes.\
keys())
```

Finally, we run our code to see which movies would be recommended. I'm using > 3.0 as strong recommendation, >0.0 as normal recommendation, 0 or less as not recommended.

```
for movie_key in recommended_movies:
    if recommended_movies[movie_key] > 3.0:
        print ("Strongly recommended for you: ", movie_key)
    elif recommended_movies[movie_key] > 0.0:
        print("Recommended for you: ", movie_key)
    else :
        print("Not recommended: ", movie_key)
```

Running the code:

```
$$ python ml_main.py

total_my_votes =  {u'Terminator 2': 11.517855396618794, u'27 Dresses': -3.3127585525\
59827, u'It happened one night': 3.424960643477168}

{u'Terminator 2': 3.8392851322062644, u'27 Dresses': -1.1042528508532756, u'It happe\
ned one night': 1.1416535478257226}

Strongly recommended for you:  Terminator 2
Not recommended:  27 Dresses
Recommended for you:  It happened one night
```

We are getting the same results for Terminator 2 and 27 dresses as we calculated by hand.

In real life

You will have seen that the I kept the code to calculate the correlation coefficients separate from the main code. That's because these would change every day and every hour, based on what users were buying.

You would then have to run *calc_correlation.py* regularly with the updated data. This would typically be done at night, when the servers were not loaded. If you are someone like Amazon, you'd have millions and millions of entries in your dataset, and this could take a fair bit of time. Of course, then you'd be using optimised database technologies to handle this large amount of data.

And, that's it. Hopefully, you know a bit about machine learning now. If you want some challenge, or want to improve your machine learning skills, the next step is to take some real life data, like the MoviesLens database, and build a recommendation engine based on that.

Thank you for reading!