An introduction to Python Programming for Research

James Hetherington

September 30, 2016

Contents

Chapter 1

Introduction

1.1 Why teach Python?

- In this first session, we will introduce Python.
- This course is about programming for data analysis and visualisation in research.
- It's not mainly about Python.
- But we have to use some language.

1.1.1 Why Python?

- Python is quick to program in
- Python is popular in research, and has lots of libraries for science
- Python interfaces well with faster languages
- Python is free, so you'll never have a problem getting hold of it, wherever you go.

1.1.2 Why write programs for research?

- Not just labour saving
- Scripted research can be tested and reproduced

1.1.3 Sensible Input - Reasonable Output

Programs are a rigorous way of describing data analysis for other researchers, as well as for computers.

Computational research suffers from people assuming each other's data manipulation is correct. By sharing codes, which are much more easy for a non-author to understand than spreadsheets, we can avoid the "SIRO" problem. The old saw "Garbage in Garbage out" is not the real problem for science:

- Sensible input
- Reasonable output

1.2 Many kinds of Python

1.2.1 The IPython Notebook

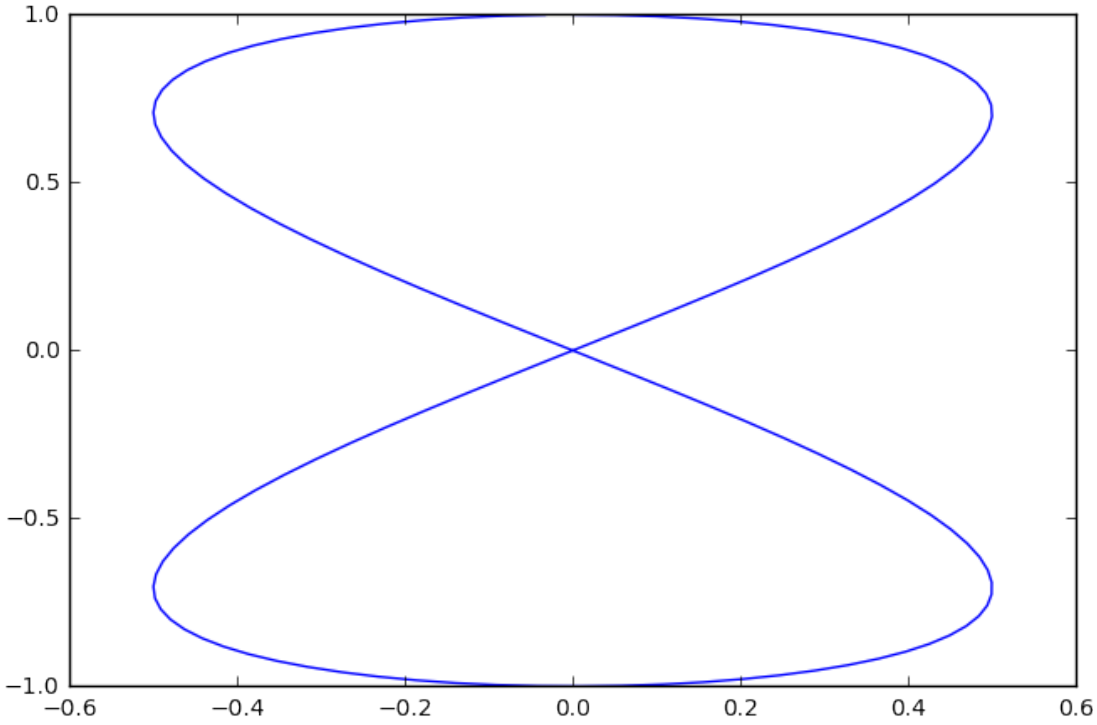
The easiest way to get started using Python, and one of the best for research data work, is the IPython Notebook.

In the notebook, you can easily mix code with discussion and commentary, and mix code with the results of that code; including graphs and other data visualisations.

```
In [1]: ### Make plot
        %matplotlib inline
        import numpy as np
        import math
        import matplotlib.pyplot as plt

        theta=np.arange(0,4*math.pi,0.1)
        eight=plt.figure()
        axes=eight.add_axes([0,0,1,1])
        axes.plot(0.5*np.sin(theta),np.cos(theta/2))
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

```
Out[1]: [<matplotlib.lines.Line2D at 0x2add10bc3b70>]
```



We're going to be mainly working in the IPython notebook in this course. To get hold of a copy of the notebook, follow the setup instructions shown on the course website, or use the installation in UCL teaching cluster rooms.

IPython notebooks consist of discussion cells, referred to as "markdown cells", and "code cells", which contain Python. This document has been created using IPython notebook, and this very cell is a **Markdown Cell**.

```
In [2]: print("This cell is a code cell")
```

```
This cell is a code cell
```

Code cell inputs are numbered, and show the output below.
Markdown cells contain text which uses a simple format to achive pretty layout, for example, to obtain:
bold, *italic*

- Bullet

  Quote

We write:

```
**bold**, *italic*

* Bullet

> Quote
```

See the Markdown documentation at This Hyperlink

1.2.2 Typing code in the notebook

When working with the notebook, you can either be in a cell, typing its contents, or outside cells, moving around the notebook.

- When in a cell, press escape to leave it. When moving around outside cells, press return to enter.
- Outside a cell:
- Use arrow keys to move around.
- Press `b` to add a new cell below the cursor.
- Press `m` to turn a cell from code mode to markdown mode.
- Press `shift+enter` to calculate the code in the block.
- Press `h` to see a list of useful keys in the notebook.
- Inside a cell:
- Press `tab` to suggest completions of variables. (Try it!)

Supplementary material: Learn more about the notebook here. Try these videos

1.2.3 Python at the command line

Data science experts tend to use a "command line environment" to work. You'll be able to learn this at our "Software Carpentry" workshops, which cover other skills for computationally based research.

```
In [3]: %%bash
        # Above line tells Python to execute this cell as *shell code*
        # not Python, as if we were in a command line
        # This is called a 'cell magic'

        python -c "print(2*4)"
```

```
8
```

1.2.4 Python scripts

Once you get good at programming, you'll want to be able to write your own full programs in Python, which work just like any other program on your computer. We'll not cover this in this course, you can learn more about this in MPHYG001. Here are some examples:

```
In [4]: %%writefile eight.py
        print(2*4)
```

```
Writing eight.py
```

```
In [5]: %%bash
        python eight.py
```

```
8
```

We can make the script directly executable (on Linux or Mac) by inserting a [hash-bang](https://en.wikipedia.org/wiki/Shebang_(Unix)

```
In [6]: %%writefile eight.py
        #! /usr/bin/env python
        print(2*7)
```

```
Overwriting eight.py
```

```
In [7]: %%bash
        chmod u+x eight.py
```

```
In [8]: %%bash
        ./eight.py
```

```
14
```

1.2.5 Python Libraries

We can write our own python libraries, called modules which we can import into the notebook and invoke:

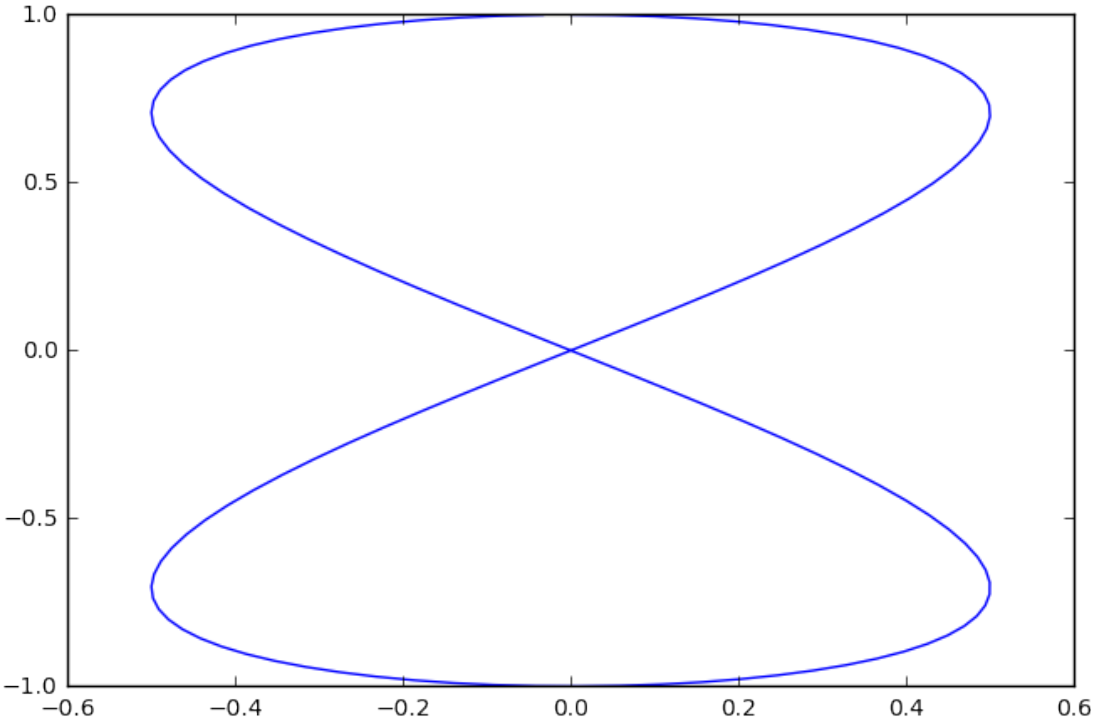
```
In [9]: %%writefile draw_eight.py
        # Above line tells the notebook to treat the rest of this
        # cell as content for a file on disk.

        import numpy as np
        import math
        import matplotlib.pyplot as plt

        def make_figure():
          theta=np.arange(0,4*math.pi,0.1)
          eight=plt.figure()
          axes=eight.add_axes([0,0,1,1])
          axes.plot(0.5*np.sin(theta),np.cos(theta/2))
          return eight
```

```
Writing draw_eight.py
```

In a real example, we could edit the file on disk using a program such as Notepad++ for windows or Atom for Mac.

```
In [10]: import draw_eight # Load the library file we just wrote to disk
```

```
In [11]: image=draw_eight.make_figure()
```



There is a huge variety of available packages to do pretty much anything. For instance, try `import antigravity`

```
In [1]: %matplotlib inline
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

Chapter 2

An example Python data analysis notebook

2.1 Why write software to manage your data and plots?

We can use programs for our entire research pipeline. Not just big scientific simulation codes, but also the small scripts which we use to tidy up data and produce plots. This should be code, so that the whole research pipeline is recorded for reproducibility. Data manipulation in spreadsheets is much harder to share or check.

2.2 A tour of capabilities

We're going to start with a look at some of the awesome things that you can do with programming, for motivation for when it gets difficult. However, you *will not* understand all the detail of the code in this section, and *nor should you*. (If you do, maybe one of our more advanced courses is more appropriate for you!)

2.3 Supplementary materials

You can see another similar demonstration on the software carpentry site at http://swcarpentry.github.io/python-novice-inflammation/01-numpy.html We'll try to give links to other sources of Python training along the way. Part of our "flipped classroom" approach is that we assume you know how to use the internet! If you find something confusing out there, please bring it along to the next session. In this course, we'll always try to draw your attention to other sources of information about what we're learning. Paying attention to as many of these as you need to, is just as important as these core notes.

2.4 The example program: a "graph of green land".

2.4.1 Importing Libraries

Research programming is all about using libraries: tools other people have provided programs that do many cool things. By combining them we can feel really powerful but doing minimum work ourselves. The python syntax to import someone else's library is "import".

```
In [2]: import geopy # A python library for investigating geographic information.
        # https://pypi.python.org/pypi/geopy~
```

Now, if you try to follow along on this example in an IPython notebook, you'll probably find that you just got an error message.

You'll need to wait until we've covered installation of additional python libraries later in the course, then come back to this and try again. For now, just follow along and try get the feel for how programming for data-focused research works.

```
In [3]: geocoder=geopy.geocoders.GoogleV3(domain="maps.google.co.uk")
        geocoder.geocode('Cambridge',exactly_one=False)

Out[3]: [Location(Cambridge, UK, (52.205337, 0.121817, 0.0)),
         Location(Cambridge, Gloucester GL2, UK, (51.73193, -2.3649, 0.0))]
```

2.4.2 Comments

Code after a # symbol doesn't get run.

```
In [4]: print("This runs")# print "This doesn't"
        # print This doesn't either

This runs
```

2.4.3 Functions

We can wrap code up in a **function**, so that we can repeatedly get just the information we want.

```
In [5]: def geolocate(place):
            return geocoder.geocode(place, exactly_one = False)[0][1]
```

Defining **functions** which put together code to make a more complex task seem simple from the outside is the most important thing in programming. The output of the function is stated by "return"; the input comes in in brackets after the function name:

```
In [6]: geolocate('London')

Out[6]: (51.5073509, -0.1277583)
```

2.4.4 Variables

We can store a result in a variable:

```
In [7]: london_location = geolocate("London")
        print(london_location)

(51.5073509, -0.1277583)
```

2.4.5 More complex functions

The google maps API allows us to fetch a map of a place, given a latitude and longitude. The URLs look like: http://maps.googleapis.com/maps/api/staticmap?size=400x400&center=51.51,-0.1275&zoom=12 We'll probably end up working out these URLS quite a bit. So we'll make ourselves another function to build up a URL given our parameters.

```
In [8]: import requests
        def request_map_at(lat,long, satellite=True,zoom=10,size=(400,400),sensor=False):
          base="http://maps.googleapis.com/maps/api/staticmap?"

          params={
            "sensor": str(sensor).lower(),
            "zoom": zoom,
            "size": "x".join(map(str,size)),
            "center" : ",".join(map(str,(lat,long))),
            "style" : "feature:all|element:labels|visibility:off"
          }
          if satellite:
            params["maptype"]="satellite"

          return requests.get(base,params = params)

In [9]: map_response = request_map_at(51.5072, -0.1275)
```

2.4.6 Checking our work

```
In [10]: map_response

Out[10]: <Response [200]>
```

Let's see what URL we ended up with:

```
In [11]: url = map_response.url
         print(url[0:50])
         print(url[50:100])
         print(url[100:])
```

```
http://maps.googleapis.com/maps/api/staticmap?cent
er=51.5072%2C-0.1275&size=400x400&style=feature%3A
all%7Celement%3Alabels%7Cvisibility%3Aoff&zoom=10&sensor=false&maptype=satellite
```

We can write **automated tests** so that if we change our code later, we can check the results are still valid.

```
In [12]: assert "http://maps.googleapis.com/maps/api/staticmap?" in url
         assert "center=51.5072%2C-0.1275" in url
         assert "zoom=10" in url
         assert "size=400x400" in url
         assert "sensor=false" in url
```

Our previous function comes back with an Object representing the web request. In object oriented programming, we use the . operator to get access to a particular **property** of the object, in this case, the actual image at that URL is in the `content` property. It's a big file, so I'll just get the first few chars:

```
In [13]: map_response.content[0:20]

Out[13]: b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x01\x90'
```

2.4.7 Displaying results

I'll need to do this a lot, so I'll wrap up our previous function in another function, to save on typing.

```
In [14]: def map_at(*args, **kwargs):
             return request_map_at(*args, **kwargs).content
```

I can use a library that comes with IPython notebook to display the image. Being able to work with variables which contain images, or documents, or any other weird kind of data, just as easily as we can with numbers or letters, is one of the really powerful things about modern programming languages like Python.

```
In [15]: map_png = map_at(*london_location)
```

```
In [16]: print("The type of our map result is actually a: ", type(map_png))
```

```
The type of our map result is actually a:  <class 'bytes'>
```

```
In [17]: import IPython
         IPython.core.display.Image(map_png)
```

```
Out[17]:
```

2.4.8 Manipulating Numbers

Now we get to our research project: we want to find out how urbanised the world is, based on satellite imagery, along a line between two cites. We expect the satellite image to be greener in the countryside.

We'll use lots more libraries to count how much green there is in an image.

```
In [18]: from io import BytesIO # An object to convert between files and unicode strings
         import numpy as np # A library to deal with matrices
         from matplotlib import image as img # A library to deal with images
```

Let's define what we count as green:

```
In [19]: def is_green(pixels):
             threshold=1.1
             greener_than_red = pixels[:,:,1] > threshold* pixels[:,:,0]
             greener_than_blue = pixels[:,:,1] > threshold*pixels[:,:,2]
             green = np.logical_and(greener_than_red, greener_than_blue)
             return green
```

This code has assumed we have our pixel data for the image as a $400 \times 400 \times 3$ 3-d matrix, with each of the three layers being red, green, and blue pixels.

We find out which pixels are green by comparing, element-by-element, the middle (green, number 1) layer to the top (red, zero) and bottom (blue, 2)

Now we just need to parse in our data, which is a PNG image, and turn it into our matrix format:

```
In [20]: def count_green_in_png(data):
             pixels = img.imread(BytesIO(data)) # Get our PNG image as a numpy array
             return np.sum(is_green(pixels))

In [21]: print(count_green_in_png(map_at(*london_location)))

108024
```

We'll also need a function to get an evenly spaced set of places between two endpoints:

```
In [22]: def location_sequence(start, end, steps):
             lats = np.linspace(start[0], end[0], steps) # "Linearly spaced" data
             longs = np.linspace(start[1],end[1], steps)
             return np.vstack([lats, longs]).transpose()

In [23]: location_sequence(geolocate("London"), geolocate("Cambridge"), 5)

Out[23]: array([[  5.15073509e+01,  -1.27758300e-01],
                [  5.16818474e+01,  -6.53644750e-02],
                [  5.18563439e+01,  -2.97065000e-03],
                [  5.20308405e+01,   5.94231750e-02],
                [  5.22053370e+01,   1.21817000e-01]])
```

2.4.9 Creating Images

We should display the green content to check our work:

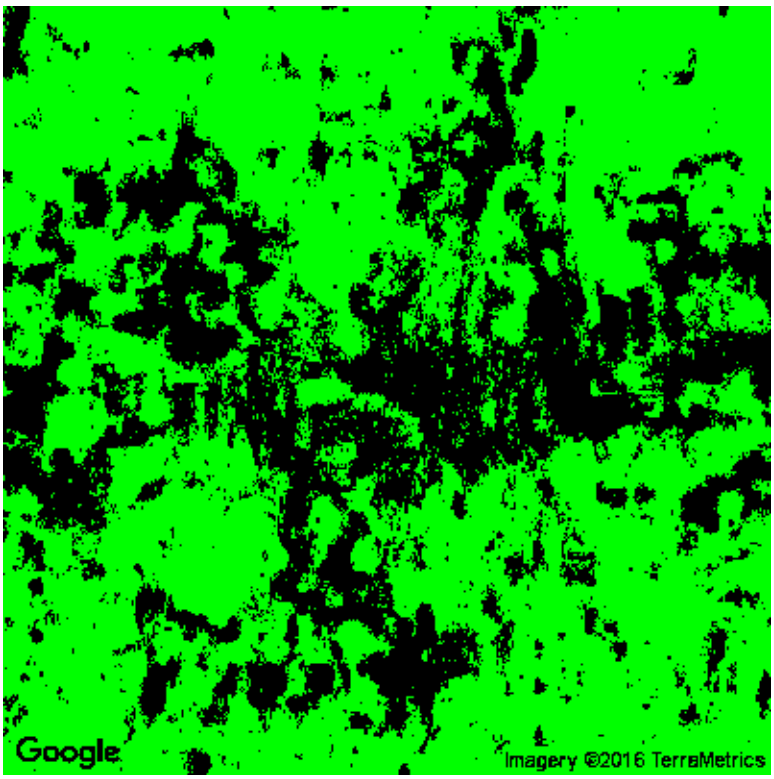
```
In [24]: def show_green_in_png(data):
             pixels=img.imread(BytesIO(data)) # Get our PNG image as rows of pixels
             green = is_green(pixels)

             out = green[:,:,np.newaxis]*np.array([0,1,0])[np.newaxis,np.newaxis,:]


             buffer = BytesIO()
             result = img.imsave(buffer, out, format='png')
             return buffer.getvalue()
```

```
In [25]: IPython.core.display.Image(
             map_at(*london_location)
         )
```

Out[25]:

```
In [26]: IPython.core.display.Image(
             show_green_in_png(
                 map_at(
                     *london_location)))
```

Out[26]:



2.4.10 Looping

We can loop over each element in out list of coordinates, and get a map for that place:

```
In [27]: for location in location_sequence(geolocate("London"),
                                           geolocate("Birmingham"),4):
             IPython.core.display.display( IPython.core.display.Image(
                     map_at(*location)))
```


Google
Imagery ©2016 TerraMetrics


Google
Imagery ©2016 TerraMetrics


Google
Imagery ©2016 TerraMetrics

So now we can count the green from London to Birmingham!

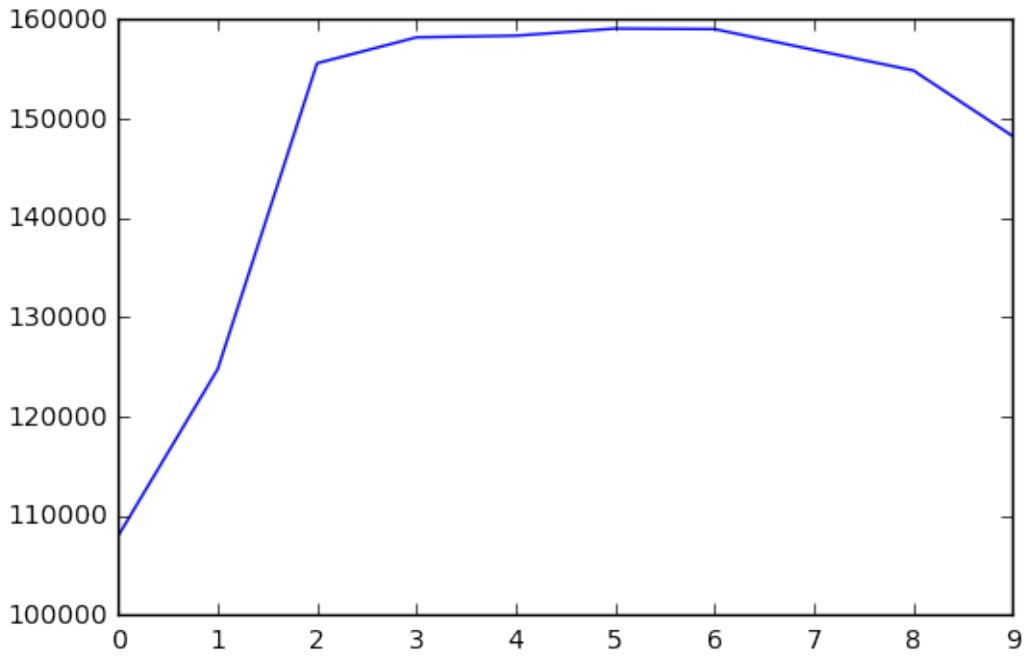
```
In [28]: [count_green_in_png(map_at(*location))

                    for location in location_sequence(geolocate("London"),
                                                      geolocate("Birmingham"),
                                                      10)]
```

```
Out[28]: [108024,
          124796,
          155602,
          158213,
          158378,
          159097,
          159044,
          156943,
          154878,
          148263]
```

2.4.11 Plotting graphs

Let's plot a graph.

```
In [29]: import matplotlib.pyplot as plt

In [30]: plt.plot([count_green_in_png(map_at(*location))
                   for location in location_sequence(geolocate("London"),
                                                     geolocate("Birmingham"),
                                                     10)])

Out[30]: [<matplotlib.lines.Line2D at 0x2ad0d2596a90>]
```



From a research perspective, of course, this code needs a lot of work. But I hope the power of using programming is clear.

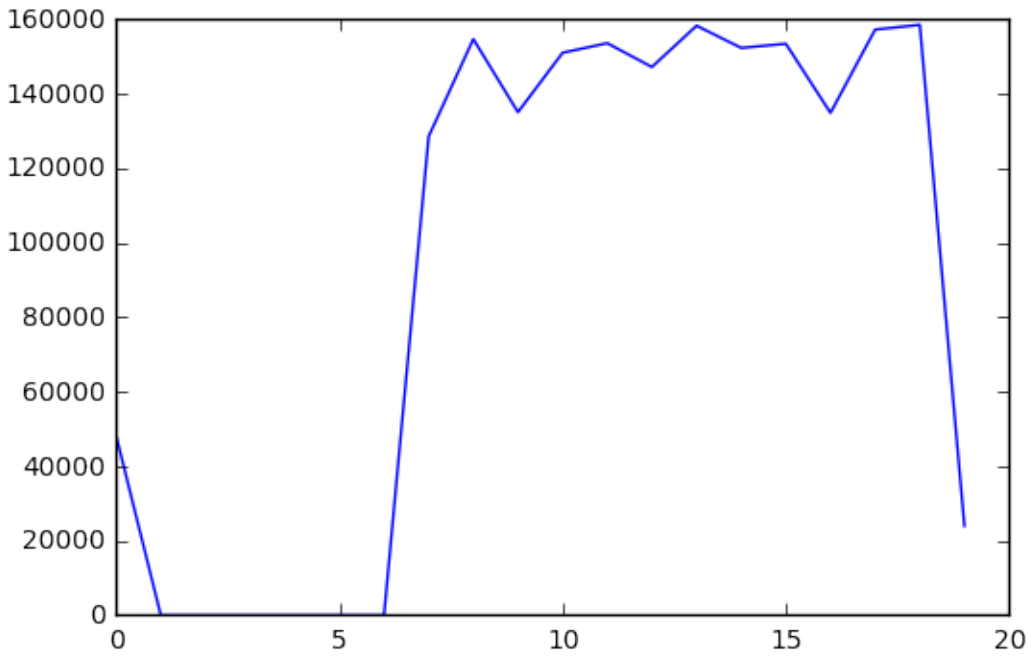
2.4.12 Composing Program Elements

We built little pieces of useful code, to:

- Find latitude and longitude of a place
- Get a map at a given latitude and longitude
- Decide whether a (red,green,blue) triple is mainly green
- Decide whether each pixel is mainly green
- Plot a new image showing the green places
- Find evenly spaced points between two places

By putting these together, we can make a function which can plot this graph automatically for any two places:

```
In [31]: def green_between(start, end,steps):
             return [count_green_in_png(map_at(*location))
                     for location in location_sequence(
                                     geolocate(start),
                                     geolocate(end),
                                     steps)]

In [32]: plt.plot(green_between('Rio de Janeiro', 'Buenos Aires', 20))

Out[32]: [<matplotlib.lines.Line2D at 0x2ad0d2631828>]
```



And that's it! We've covered, very very quickly, the majority of the python language, and much of the theory of software engineering.

Now we'll go back, carefully, through all the concepts we touched on, and learn how to use them properly ourselves.

Chapter 3

Variables

3.1 Variable Assignment

When we generate a result, the answer is displayed, but not kept anywhere.

```
In [1]: 2*3

Out[1]: 6
```

If we want to get back to that result, we have to store it. We put it in a box, with a name on the box. This is a **variable**.

```
In [2]: six = 2*3

In [3]: print(six)

6
```

If we look for a variable that hasn't ever been defined, we get an error.

```
In [4]: print(seven)


        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-4-37b67c799511> in <module>()
    ----> 1 print(seven)


        NameError: name 'seven' is not defined
```

That's **not** the same as an empty box, well labeled:

```
In [5]: nothing = None

In [6]: print(nothing)

None
```

(None is the special python value for a no-value variable.)

Supplementary Materials: There's more on variables at http://swcarpentry.github.io/python-novice-inflammation/01-numpy.html

Anywhere we could put a raw number, we can put a variable label, and that works fine:

```
In [7]: print(5*six)

30


In [8]: scary = six*six*six

In [9]: print(scary)

216
```

3.2 Reassignment and multiple labels

But here's the real scary thing: it seems like we can put something else in that box:

```
In [10]: scary = 25

In [11]: print(scary)

25
```

Note that **the data that was there before has been lost**.

No labels refer to it any more - so it has been "Garbage Collected"! We might imagine something pulled out of the box, and thrown on the floor, to make way for the next occupant.

In fact, though, it is the **label** that has moved. We can see this because we have more than one label refering to the same box:

```
In [12]: name = "James"

In [13]: nom = name

In [14]: print(nom)

James


In [15]: print(name)

James
```

And we can move just one of those labels:

```
In [16]: nom = "Hetherington"

In [17]: print(name)

James


In [18]: print(nom)
```

```
Hetherington
```

So we can now develop a better understanding of our labels and boxes: each box is a piece of space (an *address*) in computer memory. Each label (variable) is a reference to such a place.

When the number of labels on a box ("variables referencing an address") gets down to zero, then the data in the box cannot be found any more.

After a while, the language's "Garbage collector" will wander by, notice a box with no labels, and throw the data away, **making that box available for more data**.

Old fashioned languages like C and Fortran don't have Garbage collectors. So a memory address with no references to it still takes up memory, and the computer can more easily run out.

So when I write:

```
In [19]: name = "Jim"
```

The following things happen:

1. A new text **object** is created, and an address in memory is found for it.
2. The variable "name" is moved to refer to that address.
3. The old address, containing "James", now has no labels.
4. The garbage collector frees the memory at the old address.

Supplementary materials: There's an online python tutor which is great for visualising memory and references. Try the [scenario we just looked at](http://www.pythontutor.com/visualize.html#code=name+%3D+%22James%22%0Anom+%3D+name%0Aprint(nom%
frontend.js&cumulative=false&heapPrimitives=true&textReferences=false&py=3&rawInputLstJSON=%5B%5D&curInstr=

Labels are contained in groups called "frames": our frame contains two labels, 'nom' and 'name'.

3.3 Objects and types

An object, like `name`, has a type. In the online python tutor example, we see that the objects have type "str". `str` means a text object: Programmers call these 'strings'.

```
In [20]: type(name)

Out[20]: str
```

Depending on its type, an object can have different *properties*: data fields Inside the object.

Consider a Python complex number for example:

```
In [21]: z=3+1j

In [22]: type(z)

Out[22]: complex

In [23]: z.real

Out[23]: 3.0

In [24]: z.imag

Out[24]: 1.0

In [25]: x = "Banana"

In [26]: type(x)
```

```
Out[26]: str

In [27]: x.real


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-27-bd0dd11a5d8a> in <module>()
    ----> 1 x.real


        AttributeError: 'str' object has no attribute 'real'
```

A property of an object is accessed with a dot.
The jargon is that the "dot operator" is used to obtain a property of an object.
When we try to access a property that doesn't exist, we get an error:

```
In [28]: z.wrong


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-28-76215e50e85b> in <module>()
    ----> 1 z.wrong


        AttributeError: 'complex' object has no attribute 'wrong'
```

3.4 Reading error messages.

It's important, when learning to program, to develop an ability to read an error message and find, from in amongst all the confusing noise, the bit of the error message which tells you what to change!

We don't yet know what is meant by `AttributeError`, or "Traceback".

```
In [29]: second_complex_number=5-6j
         print("Gets to here")
         print(second_complex_number.wrong)
         print("Didn't get to here")

Gets to here


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-29-3eaebc91b05c> in <module>()
          1 second_complex_number=5-6j
```

```
      2 print("Gets to here")
----> 3 print(second_complex_number.wrong)
      4 print("Didn't get to here")


AttributeError: 'complex' object has no attribute 'wrong'
```

But in the above, we can see that the error happens on the **third** line of our code cell.
We can also see that the error message: > 'complex' object has no attribute 'wrong'
. . . tells us something important. Even if we don't understand the rest, this is useful for debugging!

3.5 Variables and the notebook kernel

When I type code in the notebook, the objects live in memory between cells.

```
In [30]: number = 0

In [31]: print(number)

0
```

If I change a variable:

```
In [32]: number = number +1

In [33]: print(number)

1
```

It keeps its new value for the next cell.
But cells are **not** always evaluated in order.
If I now go back to Input 22, reading `number = number + 1`, and run it again, with shift-enter. Number will change from 2 to 3, then from 3 to 4. Try it!
So it's important to remember that if you move your cursor around in the notebook, it doesn't always run top to bottom.
Supplementary material: (1) https://ipython.org/ipython-doc/3/notebook/index.html (2) http://ipython.org/videos.html

Chapter 4

Using Functions

4.1 Calling functions

We often want to do thing to our objects that are more complicated than just assigning them to variables.

```
In [1]: len("pneumonoultramicroscopicsilicovolcanoconiosis")

Out[1]: 45

In [2]: len([3,5,15])

Out[2]: 3
```

Here we have "called a function".

The function `len` takes one input, and has one output. The output is the length of whatever the input was.

Programmers also call function inputs "parameters" or, confusingly, "arguments".

Here's another example:

```
In [3]: sorted("Python")

Out[3]: ['P', 'h', 'n', 'o', 't', 'y']
```

Which gives us back a *list* of the letters in Python, sorted alphabetically.

The input goes in brackets after the function name, and the output emerges wherever the function is used.

So we can put a function call anywhere we could put a "literal" object or a variable.

```
In [4]: name = 'Jim'

In [5]: len(name)*8

Out[5]: 24

In [6]: x = len('Mike')
        y = len('Bob')
        z = x+y

In [7]: print(z)

7
```

4.2 Using methods

Objects come associated with a bunch of functions designed for working on objects of that type. We access these with a dot, just as we do for data attributes:

```
In [8]: "shout".upper()

Out[8]: 'SHOUT'
```

These are called methods. If you try to use a method defined for a different type, you get an error:

```
In [9]: x = 5
        x.upper()


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-9-5d563274bba7> in <module>()
          1 x = 5
    ----> 2 x.upper()


        AttributeError: 'int' object has no attribute 'upper'
```

If you try to use a method that doesn't exist, you get an error:

```
In [10]: x = 5
         x.wrong


         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-10-131d304742ae> in <module>()
           1 x = 5
     ----> 2 x.wrong


         AttributeError: 'int' object has no attribute 'wrong'
```

Methods and properties are both kinds of **attribute**, so both are accessed with the dot operator. Objects can have both properties and methods:

```
In [11]: z = 1+5j

In [12]: z.real

Out[12]: 1.0

In [13]: z.conjugate()

Out[13]: (1-5j)
```

4.3 Functions are just a type of object!

Now for something that will take a while to understand: don't worry if you don't get this yet, we'll look again at this in much more depth later in the course.

If we forget the (), we realise that a *method is just a property which is a function*!

```
In [14]: z.conjugate
Out[14]: <function complex.conjugate>
In [15]: type(z.conjugate)
Out[15]: builtin_function_or_method
In [16]: somefunc = z.conjugate
In [17]: somefunc()
Out[17]: (1-5j)
```

Functions are just a kind of variable:

```
In [18]: magic = sorted
In [19]: type(magic)
Out[19]: builtin_function_or_method
In [20]: magic(["Technology", "Advanced"])
Out[20]: ['Advanced', 'Technology']
In [21]: def double(data):
             return data*2
In [22]: double(5)
Out[22]: 10
In [23]: double
Out[23]: <function __main__.double>
In [24]: timestwo = double
In [25]: timestwo(8)
Out[25]: 16
In [26]: timestwo(5,6,7)

         ---------------------------------------------------------------------------
         TypeError                                 Traceback (most recent call last)

         <ipython-input-26-2816b5f0c79e> in <module>()
     ----> 1 timestwo(5,6,7)


         TypeError: double() takes 1 positional argument but 3 were given

In [27]: def timesten(input):
             return 5*double(input)
In [28]: timesten(4)
Out[28]: 40
```

4.4 Getting help on functions and methods

The 'help' function, when applied to a function, gives help on it!

```
In [29]: help(sorted)

Help on built-in function sorted in module builtins:

sorted(iterable, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customise the sort order, and the
    reverse flag can be set to request the result in descending order.



In [30]: help(z.conjugate)

Help on built-in function conjugate:

conjugate(...) method of builtins.complex instance
    complex.conjugate() -> complex

    Return the complex conjugate of its argument. (3-4j).conjugate() == 3+4j.
```

The 'dir' function, when applied to an object, lists all its attributes (properties and methods):

```
In [31]: dir("Hexxo")

Out[31]: ['__add__',
          '__class__',
          '__contains__',
          '__delattr__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__getnewargs__',
          '__gt__',
          '__hash__',
          '__init__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mod__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
```

```
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'capitalize',
'casefold',
'center',
'count',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'format_map',
'index',
'isalnum',
'isalpha',
'isdecimal',
'isdigit',
'isidentifier',
'islower',
'isnumeric',
'isprintable',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'maketrans',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
'rpartition',
'rsplit',
'rstrip',
'split',
'splitlines',
'startswith',
'strip',
'swapcase',
'title',
'translate',
'upper',
'zfill']
```

Most of these are confusing methods beginning and ending with __, part of the internals of python.

Again, just as with error messages, we have to learn to read past the bits that are confusing, to the bit we want:

```
In [32]: "Hexxo".replace("x", "l")

Out[32]: 'Hello'
```

4.5 Operators

Now that we know that functions are a way of taking a number of inputs and producing an output, we should look again at what happens when we write:

```
In [33]: x = 2 + 3

In [34]: print(x)

5
```

This is just a pretty way of calling an "add" function. Things would be more symmetrical if add were actually written

```
x = +(2,3)
```

Where '+' is just the name of the name of the adding function.
In python, these functions **do** exist, but they're actually **methods** of the first input: they're the mysterious __ functions we saw earlier (Two underscores.)

```
In [35]: x.__add__(7)

Out[35]: 12
```

We call these symbols, +, – etc, "operators".
The meaning of an operator varies for different types:

```
In [36]: [2, 3, 4] + [5, 6]

Out[36]: [2, 3, 4, 5, 6]
```

Sometimes we get an error when a type doesn't have an operator:

```
In [37]: 7 - 2

Out[37]: 5

In [38]: [2, 3, 4] - [5, 6]


---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-38-4627195e7799> in <module>()
----> 1 [2, 3, 4] - [5, 6]


TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

The word “operand” means “thing that an operator operates on”!
Or when two types can’t work together with an operator:

```
In [39]: [2, 3, 4] + 5
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-39-84117f41979f> in <module>()
----> 1 [2, 3, 4] + 5


TypeError: can only concatenate list (not "int") to list
```

Just as in Mathematics, operators have a built-in precedence, with brackets used to force an order of operations:

```
In [40]: print(2 + 3 * 4)
```

```
14
```

```
In [41]: print((2 + 3) * 4)
```

```
20
```

Supplementary material: http://www.mathcs.emory.edu/~valerie/courses/fall10/155/resources/op_precedence.html

Chapter 5

Types

We have seen that Python objects have a 'type':

```
In [1]: type(5)

Out[1]: int
```

5.1 Floats and integers

Python has two core numeric types, `int` for integer, and `float` for real number.

```
In [2]: one=1
        ten=10
        one_float=1.0
        ten_float=10.

In [3]: tenth=one_float/ten_float

In [4]: tenth

Out[4]: 0.1

In [5]: one//ten

Out[5]: 0

In [6]: 2+3

Out[6]: 5

In [7]: "Hello "+"World"

Out[7]: 'Hello World'
```

With integers, we can be clear about what we want: divide an integer by an integer and return another integer, or divide an integer by an integer and return a floating point. Most other languages do not afford that kind of distinction! It can lead to hard-to-find bugs.

```
In [8]: print(one//ten, one/ten)

0 0.1
```

```
In [9]: print(type(one//ten), type(one/tenth))

<class 'int'> <class 'float'>
```

The divided by operator `/` means divide by for real numbers. Whereas `//` means divide by integers for integers, and divide by floats for floats

```
In [10]: 10//3

Out[10]: 3

In [11]: 10/3

Out[11]: 3.3333333333333335

In [12]: 10//3.0

Out[12]: 3.0

In [13]: str(5)

Out[13]: '5'

In [14]: 10/float(3)

Out[14]: 3.3333333333333335

In [15]: x = 5

In [16]: str(x)

Out[16]: '5'

In [17]: x = "5.2"

         int(float(x))

Out[17]: 5
```

I lied when I said that the `float` type was a real number. It's actually a computer representation of a real number called a "floating point number". Representing $\sqrt{2}$ or $\frac{1}{3}$ perfectly would be impossible in a computer, so we use a finite amount of memory to do it.

Supplementary material:

- https://docs.python.org/2/tutorial/floatingpoint.html
- http://floating-point-gui.de/formats/fp/
- Advanced: http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

5.2 Strings

Python has a built in `string` type, supporting many useful methods.

```
In [18]: given = "James"
         family = "Hetherington"
         full = given + " " + family
```

So + for strings means "join them together" - *concatenate*.

```
In [19]: print(full.upper())

JAMES HETHERINGTON


In [20]: 2*4

Out[20]: 8
```

As for `float` and `int`, the name of a type can be used as a function to convert between types:

```
In [21]: ten

Out[21]: 10

In [22]: one

Out[22]: 1

In [23]: type(ten)

Out[23]: int

In [24]: print(ten+one)

11


In [25]: print(float(str(ten) + str(one)))

101.0
```

We can remove extraneous material from the start and end of a string:

```
In [26]: 5

Out[26]: 5

In [27]: "    Hello  ".strip()

Out[27]: 'Hello'
```

5.3 Lists

Python's basic **container** type is the `list`

We can define our own list with square brackets:

```
In [28]: [1, 3, 7]

Out[28]: [1, 3, 7]

In [29]: type([1, 3, 7])

Out[29]: list
```

Lists *do not* have to contain just one type:

```
In [30]: various_things = [1, 2, "banana", 3.4, [1,2] ]
```

We access an **element** of a list with an `int` in square brackets:

```
In [31]: one = 1
         two = 2
         three = 3

In [32]: my_new_list = [one, two, three]

In [33]: middle_value_in_list = my_new_list[1]

In [34]: middle_value_in_list

Out[34]: 2

In [35]: [1,2,3][1]

Out[35]: 2

In [36]: various_things[2]

Out[36]: 'banana'

In [37]: index = 2
         various_things[index]

Out[37]: 'banana'
```

Note that list indices start from zero.
We can quickly make a list with numbers counted up:

```
In [38]: count_to_five = list(range(5))
         print(count_to_five)

[0, 1, 2, 3, 4]
```

We can use a string to join together a list of strings:

```
In [39]: name = ["James", "Philip", "John", "Hetherington"]
         print(" -> ".join(name))

James -> Philip -> John -> Hetherington
```

And we can split up a string into a list:

```
In [40]: "Ernst Stavro Blofeld".split("o")

Out[40]: ['Ernst Stavr', ' Bl', 'feld']
```

We can an item to a list:

```
In [41]: name.append("Jr")
         print(name)

['James', 'Philip', 'John', 'Hetherington', 'Jr']
```

Or we can add more than one:

```
In [42]: name.extend(["the", "third"])
         print(name)

['James', 'Philip', 'John', 'Hetherington', 'Jr', 'the', 'third']
```

5.4 Sequences

Many other things can be treated like `lists`. Python calls things that can be treated like lists `sequences`.
A string is one such *sequence type*

```
In [43]: print(count_to_five[1])
         print("James"[2])

1
m


In [44]: print(count_to_five[1:3])
         print("Hello World"[4:8])

[1, 2]
o Wo


In [45]: print(len(various_things))
         print(len("Python"))

5
6


In [46]: len([[1, 2], 4])

Out[46]: 2

In [47]: print("John" in name)
         print(9 in count_to_five)

True
False
```

5.5 Unpacking

Multiple values can be **unpacked** when assigning from sequences, like dealing out decks of cards.

```
In [48]: mylist = ['Goodbye', 'Cruel']
         a, b = mylist
         print(a)

Goodbye


In [49]: a = mylist[0]
         b = mylist[1]
```

Chapter 6

Containers

6.1 Checking for containment.

The `list` we saw is a container type: its purpose is to hold other objects. We can ask python whether or not a container contains a particular item:

```
In [1]: 'Dog' in ['Cat', 'Dog', 'Horse']

Out[1]: True

In [2]: 'Bird' in ['Cat', 'Dog', 'Horse']

Out[2]: False

In [3]: 2 in range(5)

Out[3]: True

In [4]: 99 in range(5)

Out[4]: False
```

6.2 Mutability

An array can be modified:

```
In [5]: name =  "James Philip John Hetherington".split(" ")
        print(name)

['James', 'Philip', 'John', 'Hetherington']


In [6]: name[0] = "Dr"
        name[1:3] = ["Griffiths-"]
        name.append("PhD")

        print(" ".join(name))

Dr Griffiths- Hetherington PhD
```

6.3 Tuples

A `tuple` is an immutable sequence:

```
In [7]: my_tuple = ("Hello", "World")

In [8]: my_tuple

Out[8]: ('Hello', 'World')

In [9]: my_tuple[0]="Goodbye"

        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-9-98e64e9effd8> in <module>()
    ----> 1 my_tuple[0]="Goodbye"


        TypeError: 'tuple' object does not support item assignment
```

`str` is immutable too:

```
In [10]: fish = "Hake"
         fish[0] = 'R'

         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-10-fe8069275347> in <module>()
           1 fish = "Hake"
     ----> 2 fish[0] = 'R'


         TypeError: 'str' object does not support item assignment
```

But note that container reassignment is moving a label, **not** changing an element:

```
In [11]: fish = "Rake" ## OK!
```

Supplementary material: Try the online memory visualiser for this one.

6.4 Memory and containers

The way memory works with containers can be important:

```
In [12]: x = list(range(3))
         print(x)

[0, 1, 2]
```

```
In [13]: y = x
         print(y)

[0, 1, 2]


In [14]: z = x[0:3]
         y[1] = "Gotcha!"
         print(x)
         print(y)
         print(z)

[0, 'Gotcha!', 2]
[0, 'Gotcha!', 2]
[0, 1, 2]


In [15]: z[2] = "Really?"
         print(x)
         print(y)
         print(z)

[0, 'Gotcha!', 2]
[0, 'Gotcha!', 2]
[0, 1, 'Really?']
```

Supplementary material: This one works well at the [memory visualiser](http://www.pythontutor.com/visualize.html#co
frontend.js&cumulative=false&heapPrimitives=true&textReferences=false&py=2&rawInputLstJSON=%5B%5D&curInstr=

```
In [16]: x = ["What's", "Going", "On?"]
         y = x
         z = x[0:3]

         y[1] = "Gotcha!"
         z[2] = "Really?"

In [17]: x

Out[17]: ["What's", 'Gotcha!', 'On?']
```

The explanation: While `y` is a second label on the *same object*, `z` is a separate object with the same data.
Nested objects make it even more complicated:

```
In [18]: x = [['a', 'b'], 'c']
         y = x
         z = x[0:2]

         x[0][1] ='d'
         z[1] ='e'

In [19]: x

Out[19]: [['a', 'd'], 'c']

In [20]: y
```

```
Out[20]: [['a', 'd'], 'c']

In [21]: z

Out[21]: [['a', 'd'], 'e']
```

Try the [visualiser](http://www.pythontutor.com/visualize.html#code=x%3D%5B%5B'a','b'%5D,'c'%5D%0Ay%3Dx%0
frontend.js&cumulative=false&heapPrimitives=true&textReferences=false&py=2&rawInputLstJSON=%5B%5D&curInstr=
again.

6.5 Identity vs Equality

Having the same data is different from being the same actual object in memory:

```
In [22]: print([1, 2] == [1, 2])
         print([1, 2] is [1, 2])

True
False
```

The == operator checks, element by element, that two containers have the same data. The `is` operator checks that they are actually the same object.

```
In [23]: [0,1,2] == range(3)

Out[23]: False

In [24]: [0, 1, 2] is range(3)

Out[24]: False
```

But, and this point is really subtle, for immutables, the python language might save memory by reusing a single instantiated copy. This will always be safe.

```
In [25]: print("Hello" == "Hello")
         print("Hello" is "Hello")

True
True
```

Chapter 7

Dictionaries

7.1 The Python Dictionary

Python supports a container type called a dictionary.
This is also known as an "associative array", "map" or "hash" in other languages.
In a list, we use a number to look up an element:

```
In [1]: names="Martin Luther King".split(" ")
        names[1]

Out[1]: 'Luther'
```

In a dictionary, we look up an element using **another object of our choice**:

```
In [2]: me = { "name": "James", "age": 39, "Jobs": ["Programmer", "Teacher"] }

In [3]: print(me)

{'Jobs': ['Programmer', 'Teacher'], 'age': 39, 'name': 'James'}


In [4]: print(me['Jobs'])

['Programmer', 'Teacher']


In [5]: print(type(me))

<class 'dict'>
```

7.2 Keys and Values

The things we can use to look up with are called **keys**:

```
In [6]: me.keys()

Out[6]: dict_keys(['Jobs', 'age', 'name'])
```

The things we can look up are called **values**:

```
In [7]: me.values()
```

```
Out[7]: dict_values([['Programmer', 'Teacher'], 39, 'James'])
```

When we test for containment on a `dict` we test on the **keys**:

```
In [8]: 'Jobs' in me

Out[8]: True

In [9]: 'James' in me

Out[9]: False

In [10]: 'James' in me.values()

Out[10]: True
```

7.3 Immutable Keys Only

The way in which dictionaries work is one of the coolest things in computer science: the "hash table". This is way beyond the scope of this course, but it has a consequence:

You can only use **immutable** things as keys.

```
In [11]: good_match = {("Lamb", "Mint"): True, ("Bacon", "Chocolate"): False}
```

but:

```
In [12]: illegal = {[1,2]: 3}


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-12-cca03b227ff4> in <module>()
       ----> 1 illegal = {[1,2]: 3}


         TypeError: unhashable type: 'list'
```

Supplementary material: You can start to learn about the 'hash table' here This material is **very** advanced, but, I think, really interesting!

7.4 No guarantee of order

Another consequence of the way dictionaries work is that there's no guaranteed order among the elements:

```
In [13]: my_dict = {'0': 0, '1':1, '2': 2, '3': 3, '4': 4}
         print(my_dict)
         print(my_dict.values())

{'0': 0, '4': 4, '3': 3, '2': 2, '1': 1}
dict_values([0, 4, 3, 2, 1])
```

7.5 Sets

A set is a `list` which cannot contain the same element twice.

```
In [14]: name = "James Hetherington"
         unique_letters = set(name)

In [15]: unique_letters

Out[15]: {' ', 'H', 'J', 'a', 'e', 'g', 'h', 'i', 'm', 'n', 'o', 'r', 's', 't'}

In [16]: print("".join(unique_letters))

ghiomesa trnHJ


In [17]: "".join(['a', 'b', 'c'])

Out[17]: 'abc'
```

It has no particular order, but is really useful for checking or storing **unique** values.

```
In [18]: alist = [1, 2, 3]
         is_unique = len(set(alist)) == len(alist)
         print(is_unique)

True
```

7.6 Safe Lookup

```
In [19]: x = {'a':1, 'b':2}

In [20]: x['a']

Out[20]: 1

In [21]: x['fish']


         ---------------------------------------------------------------------------

         KeyError                                  Traceback (most recent call last)

         <ipython-input-21-81af752ce649> in <module>()
    ----> 1 x['fish']


         KeyError: 'fish'


In [22]: x.get('a')

Out[22]: 1

In [23]: x.get('fish')

In [24]: x.get('fish', 'tuna') == 'tuna'

Out[24]: True
```

Chapter 8

Data structures

8.1 Nested Lists and Dictionaries

In research programming, one of our most common tasks is building an appropriate *structure* to model our complicated data. Later in the course, we'll see how we can define our own types, with their own attributes, properties, and methods. But probably the most common approach is to use nested structures of lists, dictionaries, and sets to model our data. For example, an address might be modelled as a dictionary with appropriately named fields:

```
In [1]: UCL={'City': 'London',
             'Street': 'Gower Street',
             'Postcode': 'WC1E 6BT'}

In [2]: James={
            'City': 'London',
            'Street': 'Waterson Street',
            'Postcode': 'E2 8HH'
        }
```

A collection of people's addresses is then a list of dictionaries:

```
In [3]: addresses=[UCL, James]

In [4]: addresses

Out[4]: [{'City': 'London', 'Postcode': 'WC1E 6BT', 'Street': 'Gower Street'},
         {'City': 'London', 'Postcode': 'E2 8HH', 'Street': 'Waterson Street'}]
```

A more complicated data structure, for example for a census database, might have a list of residents or employees at each address:

```
In [5]: UCL['people']=['Clare','James', 'Owain']

In [6]: James['people']=['Sue', 'James']

In [7]: addresses

Out[7]: [{'City': 'London',
          'Postcode': 'WC1E 6BT',
          'Street': 'Gower Street',
          'people': ['Clare', 'James', 'Owain']},
         {'City': 'London',
          'Postcode': 'E2 8HH',
          'Street': 'Waterson Street',
          'people': ['Sue', 'James']}]
```

Which is then a list of dictionaries, with keys which are strings or lists.
We can go further, e.g.:

```
In [8]: UCL['Residential']=False
```

And we can write code against our structures:

```
In [9]: leaders =  [place['people'][0] for place in addresses]
        print(leaders)
```

```
['Clare', 'Sue']
```

This was an example of a 'list comprehension', which have used to get data of this structure, and which we'll see more of in a moment...

8.2 Exercise: a Maze Model.

Work with a partner to design a data structure to represent a maze using dictionaries and lists.

- Each place in the maze has a name, which is a string.
- Each place in the maze has zero or more people currently standing at it, by name.
- Each place in the maze has a maximum capacity of people that can fit in it.
- From each place in the maze, you can go from that place to a few other places, using a direction like 'up', 'north', or 'sideways'

Create an example instance, in a notebook, of a simple structure for your maze:

- The front room can hold 2 people. James is currently there. You can go outside to the garden, or upstairs to the bedroom, or north to the kitchen.
- From the kitchen, you can go south to the front room. It fits 1 person.
- From the garden you can go inside to living room. It fits 3 people. Sue is currently there.
- From the bedroom, you can go downstairs. You can also jump out of the window to the garden. It fits 2 people.

Make sure that your model:

- Allows empty rooms
- Allows you to jump out of the upstairs window, but not to fly back up.
- Allows rooms which people can't fit in.

```
In [10]: cities = [{'name': "London", "capacity": 8, "residents": ["Me","Sue"]},
                  {'name': "Edinburgh", "capacity": 1, "residents": ["Dave"]},
                  {'name': "Cardiff", "capacity": 1, "residents": []}]

In [11]: len(cities[2]['residents'])

Out[11]: 0

In [12]:
```

8.2.1 Solution: my Maze Model

Here's one possible solution to the Maze model. Yours will probably be different, and might be just as good. That's the artistry of software engineering: some solutions will be faster, others use less memory, while others will be easier for other people to understand. Optimising and balancing these factors is fun!

```
In [1]: house = {
            'living' : {
                'exits': {
                    'north' : 'kitchen',
                    'outside' : 'garden',
                    'upstairs' : 'bedroom'
                },
                'people' : ['James'],
                'capacity' : 2
            },
            'kitchen' : {
                'exits': {
                    'south' : 'living'
                },
                'people' : [],
                'capacity' : 1
            },
            'garden' : {
                'exits': {
                    'inside' : 'living'
                },
                'people' : ['Sue'],
                'capacity' : 3
            },
            'bedroom' : {
                'exits': {
                    'downstairs' : 'living',
                    'jump' : 'garden'
                },
                'people' : [],
                'capacity' : 1
            }
        }
```

Some important points:

- The whole solution is a complete nested structure.
- I used indenting to make the structure easier to read.
- Python allows code to continue over multiple lines, so long as sets of brackets are not finished.
- There is an **Empty** person list in empty rooms, so the type structure is robust to potential movements of people.
- We are nesting dictionaries and lists, with string and integer data.

```
In [2]: people_so_far = 0

        for room_name in house:
            people_so_far = people_so_far + len(house[room_name]['people'])

        print people_so_far
```

```
  File "<ipython-input-2-a3d2f066e414>", line 6
    print people_so_far
                      ^
SyntaxError: Missing parentheses in call to 'print'
```

In [3]:

Chapter 9

Control and Flow

9.1 Turing completeness

Now that we understand how we can use objects to store and model our data, we only need to be able to control the flow of our program in order to have a program that can, in principle, do anything!

Specifically we need to be able to:

- Control whether a program statement should be executed or not, based on a variable. "Conditionality"
- Jump back to an earlier point in the program, and run some statements again. "Branching"

Once we have these, we can write computer programs to process information in arbitrary ways: we are *Turing Complete*!

9.2 Conditionality

Conditionality is achieved through Python's `if` statement:

```
In [1]: x = -3
        if x < 0:
            print(x, " is negative")
            print("This is controlled")
        print("Always run this")
```

```
-3  is negative
This is controlled
Always run this
```

The first time through, the print statement never happened.

The **controlled** statements are indented. Once we remove the indent, the statements will once again happen regardless.

9.3 Else and Elif

Python's if statement has optional elif (else-if) and else clauses:

```
In [2]: x = -3
        if x < 0:
            print("x is negative")
```

```
        else:
            print("x is positive")
```

```
x is negative
```

```
In [3]: x = 5
        if x < 0:
            print("x is negative")
        elif x == 0:
            print("x is zero")
        else:
            print("x is positive")
```

```
x is positive
```

Try editing the value of x here, and note that other sections are found.

```
In [4]: choice ='dlgkhdglkhgkjhdkjgh'

        if choice == 'high':
            print(1)
        elif choice == 'medium':
            print(2)
        else:
            print(3)
```

```
3
```

9.4 Comparison

`True` and `False` are used to represent **boolean** (true or false) values.

```
In [5]: 1 > 2
```

```
Out[5]: False
```

Comparison on strings is alphabetical.

```
In [6]: "UCL" > "King's"
```

```
Out[6]: True
```

There's no automatic conversion of the **string** True to true:

```
In [7]: True == "True"
```

```
Out[7]: False
```

Be careful not to compare values of different types. At best, the language won't allow it and an issue an error, at worst it will allow it and do some behind-the-scene conversion that may be surprising.

```
In [8]: '1' < 2
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-8-b67fbc3e6cdc> in <module>()
----> 1 '1' < 2


TypeError: unorderable types: str() < int()
```

Any statement that evaluates to `True` or `False` can be used to control an `if` Statement.

9.5 Automatic Falsehood

Various other things automatically count as true or false, which can make life easier when coding:

```
In [9]: mytext = "Hello"
        if mytext:
            print("Mytext is not empty")

        mytext2 = ""
        if mytext2:
            print("Mytext2 is not empty")
```

```
Mytext is not empty
```

We can use logical not and logical and to combine true and false:

```
In [10]: x=3.2
         if not (x>0 and type(x) == int):
             print(x,"is not a positive integer")
```

```
3.2 is not a positive integer
```

`not` also understands magic conversion from false-like things to True or False.

```
In [11]: not not "Who's there!" # Thanks to Mysterious Student
```

```
Out[11]: True
```

```
In [12]: bool("")
```

```
Out[12]: False
```

```
In [13]: bool("James")
```

```
Out[13]: True
```

```
In [14]: bool([])
```

```
Out[14]: False
```

```
In [15]: bool(['a'])
```

```
Out[15]: True

In [16]: bool({})

Out[16]: False

In [17]: bool({'name': 'James'})

Out[17]: True

In [18]: bool(0)

Out[18]: False

In [19]: bool(1)

Out[19]: True

In [20]: not 2==3

Out[20]: True
```

But subtly, although these quantities evaluate True or False in an if statement, they're not themselves actually True or False under ==:

```
In [21]: [] == False

Out[21]: False

In [22]: bool([]) == bool(False)

Out[22]: True
```

9.6 Indentation

In Python, indentation is semantically significant. You can choose how much indentation to use, so long as you are consistent, but four spaces is conventional. Please do not use tabs.

In the notebook, and most good editors, when you press <tab>, you get four spaces.

```
In [23]: if x>0:
             print(x)

3.2
```

9.7 Pass

A statement expecting identation must have some indented code. This can be annoying when commenting things out. (With #)

```
In [24]: if x>0:
             # print x
         print("Hello")
```

```
  File "<ipython-input-24-ffabc0ff1732>", line 3
    print("Hello")
        ^
IndentationError: expected an indented block
```

So the `pass` statement is used to do nothing.

```
In [25]: if x>0:
             pass
         print("Hello")

Hello
```

9.8 Iteration

Our other aspect of control is looping back on ourselves.

We use `for ... in` to "iterate" over lists:

```
In [1]: mylist = [3, 7, 15, 2]
        for element in mylist:
            print(element**2)

9
49
225
4
```

Each time through the loop, the variable in the `value` slot is updated to the **next** element of the sequence.

9.9 Iterables

Any sequence type is iterable:

```
In [2]: vowels="aeiou"

        sarcasm = []

        for letter in "Okay":
            if letter.lower() in vowels:
                repetition = 3
            else:
                repetition = 1
            sarcasm.append(letter*repetition)

        "".join(sarcasm)

Out[2]: 'OOOkaaay'
```

The above is a little puzzle, work through it to understand why it does what it does

9.10 Dictionaries are Iterables

All sequences are iterables. Some iterables (things you can `for` loop over) are not sequences (things with you can do `x[5]` to), for example sets.

```
In [3]: import datetime
        now = datetime.datetime.now()

        founded = {"James": 1976, "UCL": 1826, "Cambridge": 1209}

        current_year = now.year

        for x in founded:
            print(x, " is ", current_year -  founded[x], "years old.")
```

```
Cambridge  is  807 years old.
UCL  is  190 years old.
James  is  40 years old.
```

```
In [4]: thing = "UCL"

        founded[thing]
```

```
Out[4]: 1826
```

```
In [5]: founded
```

```
Out[5]: {'Cambridge': 1209, 'James': 1976, 'UCL': 1826}
```

```
In [6]: founded.items()
```

```
Out[6]: dict_items([('Cambridge', 1209), ('UCL', 1826), ('James', 1976)])
```

9.11 Unpacking and Iteration

Unpacking can be useful with iteration:

```
In [7]: triples=[
            [4,11,15],
            [39,4,18]
        ]
```

```
In [8]: for whatever in triples:
            print(whatever)
```

```
[4, 11, 15]
[39, 4, 18]
```

```
In [9]: a,b = [36, 7]
```

```
In [10]: b
```

```
Out[10]: 7
```

```
In [11]: for first, middle, last in triples:
             print(middle)
```

```
11
4
```

```
In [12]: # A reminder that the words you use for variable names are arbitrary:
         for hedgehog, badger, fox in triples:
             print(badger)
```

```
11
4
```

for example, to iterate over the items in a dictionary as pairs:

```
In [13]: for name, year in founded.items():
             print(name, " is ", current_year - year, "years old.")
```

```
Cambridge  is  807 years old.
UCL  is  190 years old.
James  is  40 years old.
```

```
In [14]: for name in founded:
             print(name, " is ", current_year - founded[name], "years old.")
```

```
Cambridge  is  807 years old.
UCL  is  190 years old.
James  is  40 years old.
```

9.12 Break, Continue

- Continue skips to the next turn of a loop
- Break stops the loop early

```
In [15]: for n in range(50):
             if n == 20:
                 break
             if n % 2 == 0:
                 continue
             print(n)
```

```
1
3
5
7
9
11
13
15
17
19
```

These aren't useful that often, but are worth knowing about. There's also an optional `else` clause on loops, executed only if you don't `break`, but I've never found that useful.

9.13 Exercise: the Maze Population

Take your maze data structure. Write a program to count the total number of people in the maze, and also determine the total possible occupants.

9.13.1 Solution: counting people in the maze

With this maze structure:

```
In [1]: house = {
            'living' : {
                'exits': {
                    'north' : 'kitchen',
                    'outside' : 'garden',
                    'upstairs' : 'bedroom'
                },
                'people' : ['James', 'Frank'],
                'capacity' : 2
            },
            'kitchen' : {
                'exits': {
                    'south' : 'living'
                },
                'people' : [],
                'capacity' : 1
            },
            'garden' : {
                'exits': {
                    'inside' : 'living'
                },
                'people' : ['Sue'],
                'capacity' : 3
            },
            'bedroom' : {
                'exits': {
                    'downstairs' : 'living',
                    'jump' : 'garden'
                },
                'people' : [],
                'capacity' : 1
            }
        }
```

We can count the occupants and capacity like this:

```
In [2]: for room_name in house:
            print(room_name)

garden
kitchen
living
bedroom


In [3]: house
```

```
Out[3]: {'bedroom': {'capacity': 1,
          'exits': {'downstairs': 'living', 'jump': 'garden'},
          'people': []},
         'garden': {'capacity': 3, 'exits': {'inside': 'living'}, 'people': ['Sue']},
         'kitchen': {'capacity': 1, 'exits': {'south': 'living'}, 'people': []},
         'living': {'capacity': 2,
          'exits': {'north': 'kitchen', 'outside': 'garden', 'upstairs': 'bedroom'},
          'people': ['James', 'Frank']}}

In [4]: house.values()

Out[4]: dict_values([{'people': ['Sue'], 'capacity': 3, 'exits': {'inside': 'living'}}, {'p

In [5]: running_total = 0

        for room_data in house.values():
            running_total += len(room_data['people'])

        print(running_total)

3
```

Chapter 10

Comprehensions

10.1 The list comprehension

If you write a for loop **inside** a pair of square brackets for a list, you magic up a list as defined. This can make for concise but hard to read code, so be careful.

```
In [1]: result = []
        for x in range(10):
            result.append(2**x)

        print(result)
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```
In [2]: [2**x for x in range(10)]
```

```
Out[2]: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

You can do quite weird and cool things with comprehensions:

```
In [3]: [len(str(2**x)) for x in range(20)]
```

```
Out[3]: [1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6]
```

You can write an `if` statement in comprehensions too:

```
In [4]: [2**x for x in range(30) if x%3 ==0 ]
```

```
Out[4]: [1, 8, 64, 512, 4096, 32768, 262144, 2097152, 16777216, 134217728]
```

Consider the following, and make sure you understand why it works:

```
In [5]: "".join([letter for letter in "James Hetherington" if letter.lower() not in 'aeiou
```

```
Out[5]: 'Jms Hthrngtn'
```

10.2 Comprehensions versus building lists with append:

This code:

```
In [6]: result=[]
        for x in range(30):
            if x%3 == 0:
                result.append(2**x)
        print(result)
```

```
[1, 8, 64, 512, 4096, 32768, 262144, 2097152, 16777216, 134217728]
```

Does the same as the comprehension above. The comprehension is generally considered more readable.

Comprehensions are therefore an example of what we call 'syntactic sugar': they do not increase the capabilities of the language.

Instead, they make it possible to write the same thing in a more readable way.

Everything we learn from now on will be either syntactic sugar or interaction with something other than idealised memory, such as a storage device or the internet. Once you have variables, conditionality, and branching, your language can do anything. (And this can be proved.)

10.3 Nested comprehensions

If you write two `for` statements in a comprehension, you get a single array generated over all the pairs:

```
In [7]: [x - y for x in range(4) for y in range(4)]
```

```
Out[7]: [0, -1, -2, -3, 1, 0, -1, -2, 2, 1, 0, -1, 3, 2, 1, 0]
```

You can select on either, or on some combination:

```
In [8]: [x - y for x in range(4) for y in range(4) if x>=y]
```

```
Out[8]: [0, 1, 0, 2, 1, 0, 3, 2, 1, 0]
```

If you want something more like a matrix, you need to do *two nested* comprehensions!

```
In [9]: [[x - y for x in range(4)] for y in range(4)]
```

```
Out[9]: [[0, 1, 2, 3], [-1, 0, 1, 2], [-2, -1, 0, 1], [-3, -2, -1, 0]]
```

Note the subtly different square brackets.

Note that the list order for multiple or nested comprehensions can be confusing:

```
In [10]: [x+y for x in ['a','b','c'] for y in ['1','2','3']]
```

```
Out[10]: ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

```
In [11]: [[x+y for x in ['a','b','c']] for y in ['1','2','3']]
```

```
Out[11]: [['a1', 'b1', 'c1'], ['a2', 'b2', 'c2'], ['a3', 'b3', 'c3']]
```

10.4 Dictionary Comprehensions

You can automatically build dictionaries, by using a list comprehension syntax, but with curly brackets and a colon:

```
In [12]: { (str(x))*3: x for x in range(3) }
```

```
Out[12]: {'000': 0, '111': 1, '222': 2}
```

10.5 List-based thinking

Once you start to get comfortable with comprehensions, you find yourself working with containers, nested groups of lists and dictionaries, as the 'things' in your program, not individual variables.

Given a way to analyse some dataset, we'll find ourselves writing stuff like:

```
analysed_data = [analyze(datum) for datum in data]

analysed_data = map(analyse, data)
```

There are lots of built-in methods that provide actions on lists as a whole:

```
In [13]: any([True, False, True])

Out[13]: True

In [14]: all([True, False, True])

Out[14]: False

In [15]: max([1, 2, 3])

Out[15]: 3

In [16]: sum([1, 2, 3])

Out[16]: 6
```

One common method is `map`, which is syntactic sugar for a simple list comprehension that applies one function to every member of a list:

```
In [17]: [str(x) for x in range(10)]

Out[17]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [18]: map(str, range(10))

Out[18]: <map at 0x2b89027b7cf8>
```

So I can write:

```
analysed_data = map(analyse, data)
```

10.6 Exercise: Occupancy Dictionary

Take your maze data structure. Write a program to print out a new dictionary, which gives, for each room's name, the number of people in it. Don't add in a zero value in the dictionary for empty rooms.

The output should look similar to:

```
In [19]: {'bedroom': 1, 'garden': 3, 'kitchen': 1, 'living': 2}

Out[19]: {'bedroom': 1, 'garden': 3, 'kitchen': 1, 'living': 2}
```

10.6.1 Solution

With this maze structure:

```
In [1]: house = {
            'living' : {
                'exits': {
                    'north' : 'kitchen',
                    'outside' : 'garden',
                    'upstairs' : 'bedroom'
                },
                'people' : ['James'],
                'capacity' : 2
            },
            'kitchen' : {
                'exits': {
                    'south' : 'living'
                },
                'people' : [],
                'capacity' : 1
            },
            'garden' : {
                'exits': {
                    'inside' : 'living'
                },
                'people' : ['Sue'],
                'capacity' : 3
            },
            'bedroom' : {
                'exits': {
                    'downstairs' : 'living',
                    'jump' : 'garden'
                },
                'people' : [],
                'capacity' : 1
            }
        }
```

We can get a simpler dictionary with just capacities like this:

```
In [2]: result = {
            name: room['capacity']
            for name, room in house.items()
            if room['capacity'] > 0
        }
        print(result)
```

```
{'kitchen': 1, 'living': 2, 'bedroom': 1, 'garden': 3}
```

Chapter 11

Functions

11.1 Definition

We use `def` to define a function, and `return` to pass back a value:

```
In [1]: def double(x):
            return x*2

In [2]: double(5)

Out[2]: 10

In [3]: double([5])

Out[3]: [5, 5]

In [4]: double('five')

Out[4]: 'fivefive'
```

11.2 Default Parameters

We can specify default values for parameters:

```
In [5]: def jeeves(name = "Sir"):
            return "Very good, "+ name

In [6]: jeeves()

Out[6]: 'Very good, Sir'

In [7]: jeeves('James')

Out[7]: 'Very good, James'
```

If you have some parameters with defaults, and some without, those with defaults **must** go later.

```
In [8]: def product(x=5, y=7):
            return x*y

In [9]: product(9)

Out[9]: 63

In [10]: product(y=11)

Out[10]: 55

In [11]: product()

Out[11]: 35
```

11.3 Side effects

Functions can do things to change their **mutable** arguments, so `return` is optional.

```
In [12]: def double_inplace(vec):
             vec[:] = [element*2 for element in vec]

         z=[1, 2, 3, 4]
         double_inplace(z)
         print(z)
```

```
[2, 4, 6, 8]
```

In this example, we're using `[:]` to access into the same list, and write it's data.

```
vec = [element*2 for element in vec]
```

would just move a local label, not change the input.
Let's remind ourselves of this behaviour with a simple array:

```
In [13]: x=5
         x=7
         x=['a','b','c']
         y=x

In [14]: x

Out[14]: ['a', 'b', 'c']

In [15]: x[:]=["Hooray!", "Yippee"]

In [16]: y

Out[16]: ['Hooray!', 'Yippee']
```

11.4 Early Return

```
In [17]: def isbigger(x, limit =20):
             if x>limit:
                 return True
             print("Gothere")
             return False

         isbigger(25, 15)

Out[17]: True

In [18]: isbigger(40, 15)

Out[18]: True
```

Return without arguments can be used to exit early from a function

```
In [19]: def extend(to, vec, pad):
             if len(vec) >= to:
                 return
             vec[:] = vec + [pad] * (to - len(vec))
```

```
In [20]: x = [1, 2, 3]
         extend(6, x, 'a')
         print(x)
```

```
[1, 2, 3, 'a', 'a', 'a']
```

```
In [21]: z = list(range(9))
         extend(6, z, 'a')
         print(z)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

11.5 Unpacking arguments

If a vector is supplied to a function with a '*', its elements are used to fill each of a function's arguments.

```
In [22]: def arrow(before, after):
             return str(before) + " -> " + str(after)

         print(arrow(1, 3))
```

```
1 -> 3
```

```
In [23]: x = [1 , -1]

         print(arrow(*x))
```

```
1 -> -1
```

This can be quite powerful:

```
In [24]: charges={"neutron": 0, "proton": 1, "electron": -1}
```

```
In [25]: charges.items()
```

```
Out[25]: dict_items([('neutron', 0), ('electron', -1), ('proton', 1)])
```

```
In [26]: for particle in charges.items():
             print(arrow(*particle))
```

```
neutron -> 0
electron -> -1
proton -> 1
```

11.6 Sequence Arguments

Similiarly, if a $\star$ is used in the **definition** of a function, multiple arguments are absorbed into a list **inside** the function:

```
In [27]: def doubler(*sequence):
             return [x*2 for x in sequence]

         print(doubler(1,2,3, 'four'))
```

```
[2, 4, 6, 'fourfour']
```

11.7 Keyword Arguments

If two asterisks are used, named arguments are supplied as a dictionary:

```
In [28]: def arrowify(**args):
             for key, value in args.items():
                 print(key + " -> " + value)

         arrowify(neutron="n", proton="p", electron="e")
```

```
neutron -> n
electron -> e
proton -> p
```

Chapter 12

Using Libraries

12.1 Import

To use a function or type from a python library, rather than a **built-in** function or type, we have to import the library.

```
In [1]: math.sin(1.6)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-1-ecc9cee3d19a> in <module>()
    ----> 1 math.sin(1.6)


        NameError: name 'math' is not defined
```

```
In [2]: import math

In [3]: math.sin(1.6)

Out[3]: 0.9995736030415051
```

We call these libraries **modules**:

```
In [4]: type(math)

Out[4]: module
```

The tools supplied by a module are *attributes* of the module, and as such, are accessed with a dot.

```
In [5]: dir(math)

Out[5]: ['__doc__',
         '__file__',
         '__loader__',
         '__name__',
         '__package__',
         '__spec__',
```

```
        'acos',
        'acosh',
        'asin',
        'asinh',
        'atan',
        'atan2',
        'atanh',
        'ceil',
        'copysign',
        'cos',
        'cosh',
        'degrees',
        'e',
        'erf',
        'erfc',
        'exp',
        'expm1',
        'fabs',
        'factorial',
        'floor',
        'fmod',
        'frexp',
        'fsum',
        'gamma',
        'gcd',
        'hypot',
        'inf',
        'isclose',
        'isfinite',
        'isinf',
        'isnan',
        'ldexp',
        'lgamma',
        'log',
        'log10',
        'log1p',
        'log2',
        'modf',
        'nan',
        'pi',
        'pow',
        'radians',
        'sin',
        'sinh',
        'sqrt',
        'tan',
        'tanh',
        'trunc']
```

They include properties as well as functions:

```
In [6]: math.pi

Out[6]: 3.141592653589793
```

You can always find out where on your storage medium a library has been imported from:

```
In [7]: print(math.__file__)

/home/travis/virtualenv/python3.5.0/lib/python3.5/lib-dynload/math.cpython-35m-x86_64-linux
```

Note that `import` does *not* install libraries from PyPI. It just makes them available to your current notebook session, assuming they are already installed. Installing libraries is harder, and we'll cover it later. So what libraries are available? Until you install more, you might have just the modules that come with Python, the *standard library*

Supplementary Materials: Review the list of standard library modules: https://docs.python.org/2/library/

If you installed via Anaconda, then you also have access to a bunch of modules that are commonly used in research.

Supplementary Materials: Review the list of modules that are packaged with Anaconda by default: http://docs.continuum.io/anaconda/pkg-docs.html (The green ticks)

We'll see later how to add more libraries to our setup.

12.2 Why bother?

Why bother with modules? Why not just have everything available all the time?

The answer is that there are only so many names available! Without a module system, every time I made a variable whose name matched a function in a library, I'd lose access to it. In the olden days, people ended up having to make really long variable names, thinking their names would be unique, and they still ended up with "name clashes". The module mechanism avoids this.

12.3 Importing from modules

Still, it can be annoying to have to write `math.sin(math.pi)` instead of `sin(pi)`. Things can be imported *from* modules to become part of the current module:

```
In [8]: from math import sin
        sin(math.pi)

Out[8]: 1.2246467991473532e-16
```

Importing one-by-one like this is a nice compromise between typing and risk of name clashes.

It *is* possible to import **everything** from a module, but you risk name clashes.

```
In [9]: pi = "pie"
        def sin(x):
            print("eat " + x)

        from math import *
        sin(pi)

Out[9]: 1.2246467991473532e-16
```

12.4 Import and rename

You can rename things as you import them to avoid clashes or for typing convenience

```
In [10]: import math as m
         m.cos(0)
```

```
Out[10]: 1.0

In [11]: pi=3
         from math import pi as realpi
         print(sin(pi), sin(realpi))

0.1411200080598672 1.2246467991473532e-16
```

Chapter 13

Loading data from files

13.1 Loading data

An important part of this course is about using Python to analyse and visualise data. Most data, of course, is supplied to us in various *formats*: spreadsheets, database dumps, or text files in various formats (csv, tsv, json, yaml, hdf5, netcdf) It is also stored in some *medium*: on a local disk, a network drive, or on the internet in various ways. It is important to distinguish the data format, how the data is structured into a file, from the data's storage, where it is put.

We'll look first at the question of data *transport*: loading data from a disk, and at downloading data from the internet. Then we'll look at data *parsing*: building Python structures from the data. These are related, but separate questions.

13.2 An example datafile

Let's write an example datafile to disk so we can investigate it. We'll just use a plain-text file. IPython notebook provides a way to do this: if we put `%%writefile` at the top of a cell, instead of being interpreted as python, the cell contents are saved to disk.

```
In [1]: %%writefile mydata.txt
        A poet once said, 'The whole universe is in a glass of wine.'
        We will probably never know in what sense he meant it,
        for poets do not write to be understood.
        But it is true that if we look at a glass of wine closely enough we see the entire
        There are the things of physics: the twisting liquid which evaporates depending
        on the wind and weather, the reflection in the glass;
        and our imagination adds atoms.
        The glass is a distillation of the earth's rocks,
        and in its composition we see the secrets of the universe's age, and the evolution
        What strange array of chemicals are in the wine? How did they come to be?
        There are the ferments, the enzymes, the substrates, and the products.
        There in wine is found the great generalization; all life is fermentation.
        Nobody can discover the chemistry of wine without discovering,
        as did Louis Pasteur, the cause of much disease.
        How vivid is the claret, pressing its existence into the consciousness that watches
        If our small minds, for some convenience, divide this glass of wine, this universe,
        into parts --
        physics, biology, geology, astronomy, psychology, and so on --
        remember that nature does not know it!
```

```
        So let us put it all back together, not forgetting ultimately what it is for.
        Let it give us one more final pleasure; drink it and forget it all!
           - Richard Feynman

Writing mydata.txt
```

Where did that go? It went to the current folder, which for a notebook, by default, is where the notebook is on disk.

```
In [2]: import os # The 'os' module gives us all the tools we need to search in the file sy
        os.getcwd() # Use the 'getcwd' function from the 'os' module to find where we are c

Out[2]: '/home/travis/build/UCL-RITS/doctoral-programming-intro/notebooks'
```

Can we see it is there?

```
In [3]: os.listdir(os.getcwd())

Out[3]: ['028dictionaries.html',
         '015variables.html',
         '030MazeSolution.nbconvert.ipynb',
         '016using_functions.html',
         '016using_functions.ipynb',
         '120Counting.ipynb',
         '064JsonYamlXML.ipynb',
         '023types.ipynb',
         '038SolutionComprehension.nbconvert.ipynb',
         '032conditionality.ipynb',
         '036MazeSolution2.nbconvert.ipynb',
         '04functions.html',
         '015variables.ipynb',
         '037comprehensions.html',
         '029structures.ipynb',
         '030MazeSolution.html',
         '038SolutionComprehension.html',
         '062csv.ipynb',
         '028dictionaries.nbconvert.ipynb',
         '060files.ipynb',
         '061internet.ipynb',
         '030MazeSolution.ipynb',
         '010exemplar.ipynb',
         '072plotting.ipynb',
         '023types.html',
         '00pythons.ipynb',
         '050import.ipynb',
         '00pythons.nbconvert.ipynb',
         'eight.py',
         '036MazeSolution2.ipynb',
         '__pycache__',
         '068QuakesSolution.ipynb',
         '091Libraries.ipynb',
         '037comprehensions.ipynb',
         '050import.nbconvert.ipynb',
         '110Capstone.ipynb',
         '028dictionaries.ipynb',
```

```
 '066QuakeExercise.ipynb',
 '010exemplar.html',
 '015variables.nbconvert.ipynb',
 '065MazeSaved.ipynb',
 '035looping.html',
 '037comprehensions.nbconvert.ipynb',
 '036MazeSolution2.html',
 '04functions.ipynb',
 '032conditionality.html',
 '082NumPy.ipynb',
 '103WritingLibraries.ipynb',
 '023types.nbconvert.ipynb',
 '035looping.nbconvert.ipynb',
 '050import.html',
 '025containers.nbconvert.ipynb',
 '016using_functions.nbconvert.ipynb',
 '029structures.html',
 '029structures.nbconvert.ipynb',
 '084Boids.ipynb',
 '010exemplar.nbconvert.ipynb',
 'index.md',
 '025containers.html',
 '04functions.nbconvert.ipynb',
 'draw_eight.py',
 '032conditionality.nbconvert.ipynb',
 'mydata.txt',
 '00pythons.html',
 '038SolutionComprehension.ipynb',
 '060files.nbconvert.ipynb',
 '101Classes.ipynb',
 '025containers.ipynb',
 '035looping.ipynb']
```

```
In [4]: [x for x in os.listdir(os.getcwd()) if '.txt' in x ]
```

```
Out[4]: ['mydata.txt']
```

Yep! Note how we used a list comprehension to filter all the extraneous files.

13.3 Path independence and `os`

We can use `dirname` to get the parent folder for a folder, in a platform independent-way.

```
In [5]: os.path.dirname(os.getcwd())
```

```
Out[5]: '/home/travis/build/UCL-RITS/doctoral-programming-intro'
```

We could do this manually using `split`:

```
In [6]: "/".join(os.getcwd().split("/")[:-1])
```

```
Out[6]: '/home/travis/build/UCL-RITS/doctoral-programming-intro'
```

But this would not work on windows, where path elements are separated with a `\` instead of a `/`. So it's important to use `os.path` for this stuff.

Supplementary Materials: If you're not already comfortable with how files fit into folders, and folders form a tree, with folders containing subfolders, then look at http://swcarpentry.github.io/shell-novice/01-filedir.html.

Satisfy yourself that after using `%%writedir`, you can then find the file on disk with Windows Explorer, OSX Finder, or the Linux Shell.

We can see how in Python we can investigate the file system with functions in the `os` module, using just the same programming approaches as for anything else.

We'll gradually learn more features of the `os` module as we go, allowing us to move around the disk, `walk` around the disk looking for relevant files, and so on. These will be important to master for automating our data analyses.

13.4 The python `file` type

So, let's read our file:

```
In [7]: myfile = open('mydata.txt')

In [8]: type(myfile)

Out[8]: _io.TextIOWrapper
```

We can go line-by-line, by treating the file as an iterable:

```
In [9]: [x for x in myfile]

Out[9]: ["A poet once said, 'The whole universe is in a glass of wine.'\n",
         'We will probably never know in what sense he meant it, \n',
         'for poets do not write to be understood. \n',
         'But it is true that if we look at a glass of wine closely enough we see the entir
         'There are the things of physics: the twisting liquid which evaporates depending\n
         'on the wind and weather, the reflection in the glass;\n',
         'and our imagination adds atoms.\n',
         "The glass is a distillation of the earth's rocks,\n",
         "and in its composition we see the secrets of the universe's age, and the evolutio
         'What strange array of chemicals are in the wine? How did they come to be? \n',
         'There are the ferments, the enzymes, the substrates, and the products.\n',
         'There in wine is found the great generalization; all life is fermentation.\n',
         'Nobody can discover the chemistry of wine without discovering, \n',
         'as did Louis Pasteur, the cause of much disease.\n',
         'How vivid is the claret, pressing its existence into the consciousness that watch
         'If our small minds, for some convenience, divide this glass of wine, this univers
         'into parts -- \n',
         'physics, biology, geology, astronomy, psychology, and so on -- \n',
         'remember that nature does not know it!\n',
         '\n',
         'So let us put it all back together, not forgetting ultimately what it is for.\n',
         'Let it give us one more final pleasure; drink it and forget it all!\n',
         '   - Richard Feynman']
```

If we do that again, the file has already finished, there is no more data.

```
In [10]: [x for x in myfile]

Out[10]: []
```

We need to 'rewind' it!

```
In [11]: myfile.seek(0)
         [len(x) for x in myfile if 'ut' in x]

Out[11]: [94, 94, 64, 78]
```

It's really important to remember that a file is a *different* built in type than a string.

13.5 Working with files.

We can read one line at a time with `readline`:

```
In [12]: myfile.seek(0)
         first = myfile.readline()

In [13]: first

Out[13]: "A poet once said, 'The whole universe is in a glass of wine.'\n"

In [14]: second = myfile.readline()

In [15]: second

Out[15]: 'We will probably never know in what sense he meant it, \n'
```

We can read the whole remaining file with `read`:

```
In [16]: rest = myfile.read()

In [17]: rest

Out[17]: "for poets do not write to be understood. \nBut it is true that if we look at a gl
```

Which means that when a file is first opened, read is useful to just get the whole thing as a string:

```
In [18]: feynman_quote = open('mydata.txt')

In [19]: open('mydata.txt').read()

Out[19]: "A poet once said, 'The whole universe is in a glass of wine.'\nWe will probably n
```

You can also read just a few characters:

```
In [20]: myfile.seek(1335)

Out[20]: 1335

In [21]: myfile.read(15)

Out[21]: '\n   - Richard F'
```

13.6 Converting Strings to Files

Because files and strings are different types, we CANNOT just treat strings as if they were files:

```
In [22]: mystring = "Hello World\n My name is James"

In [23]: mystring

Out[23]: 'Hello World\n My name is James'

In [24]: mystring.readline()


         ---------------------------------------------------------------------------

         AttributeError                            Traceback (most recent call last)

         <ipython-input-24-8c85154fa12a> in <module>()
     ----> 1 mystring.readline()


         AttributeError: 'str' object has no attribute 'readline'
```

This is important, because some file format parsers expect input from a **file** and not a string. We can convert between them using the StringIO module in the standard library:

```
In [25]: from io import StringIO

In [26]: mystringasafile = StringIO(mystring)

In [27]: mystringasafile.readline()

Out[27]: 'Hello World\n'

In [28]: mystringasafile.readline()

Out[28]: ' My name is James'
```

Note that in a string, `\n` is used to represent a newline.

13.7 Closing files

We really ought to close files when we've finished with them, as it makes the computer more efficient. (On a shared computer, this is particularly important)

```
In [29]: feynman_quote.close()
```

Because it's so easy to forget this, python provides a **context manager** to open a file, then close it automatically at the end of an indented block:

```
In [30]: somefile = open('mydata.txt')
         content=somefile.read()
         somefile.close()

In [31]: with open('mydata.txt') as somefile:
             content = somefile.read()

         print(content)
```

```
A poet once said, 'The whole universe is in a glass of wine.'
We will probably never know in what sense he meant it,
for poets do not write to be understood.
But it is true that if we look at a glass of wine closely enough we see the entire universe
There are the things of physics: the twisting liquid which evaporates depending
on the wind and weather, the reflection in the glass;
and our imagination adds atoms.
The glass is a distillation of the earth's rocks,
and in its composition we see the secrets of the universe's age, and the evolution of stars
What strange array of chemicals are in the wine? How did they come to be?
There are the ferments, the enzymes, the substrates, and the products.
There in wine is found the great generalization; all life is fermentation.
Nobody can discover the chemistry of wine without discovering,
as did Louis Pasteur, the cause of much disease.
How vivid is the claret, pressing its existence into the consciousness that watches it!
If our small minds, for some convenience, divide this glass of wine, this universe,
into parts --
physics, biology, geology, astronomy, psychology, and so on --
remember that nature does not know it!

So let us put it all back together, not forgetting ultimately what it is for.
Let it give us one more final pleasure; drink it and forget it all!
   - Richard Feynman
```

The code to be done while the file is open is indented, just like for an `if` statement.

```
In [32]: print(somefile)
```

```
<_io.TextIOWrapper name='mydata.txt' mode='r' encoding='UTF-8'>
```

You should pretty much **always** use this syntax for working with files.

13.8 Writing files

We might want to create a file from a string in memory. We can't do this with the notebook's `%%writefile` – this is just a notebook convenience, and isn't very programmable.

When we open a file, we can specify a 'mode', in this case, 'w' for writing. ('r' for reading is the default.)

```
In [33]: with open('mywrittenfile', 'w') as target:
             target.write('Hello')
             target.write('World')

In [34]: with open('mywrittenfile', 'r') as source:
             print(source.read())
```

```
HelloWorld
```

```
In [35]: import os
         print(os.getcwd())
```

```
/home/travis/build/UCL-RITS/doctoral-programming-intro/notebooks
```

And we can "append" to a file with mode 'a':

```
In [36]: with open('mywrittenfile', 'a') as target:
             target.write('Hello')
             target.write('James')

In [37]: with open('mywrittenfile','r') as source:
             print(source.read())

HelloWorldHelloJames
```

Chapter 14

Getting data from the Internet

We've seen about obtaining data from our local file system.

The other common place today that we might want to obtain data is from the internet.

It's very common today to treat the web as a source and store of information; we need to be able to programmatically download data, and place it in python objects.

We may also want to be able to programmatically *upload* data, for example, to automatically fill in forms.

This can be really powerful if we want to, for example, do automated metaanalysis across a selection of research papers.

14.1 URLs

All internet resources are defined by a Uniform Resource Locator.

```
In [1]: "http://maps.googleapis.com:80/maps/api/staticmap?size=400x400&center=51.51,-0.127
```

```
Out[1]: 'http://maps.googleapis.com:80/maps/api/staticmap?size=400x400&center=51.51,-0.127
```

A url consists of:

- A *scheme* (http, https, ssh, ...)
- A *host* (maps.googleapis.com, the name of the remote computer you want to talk to)
- A *port* (optional, most protocols have a typical port associated with them, e.g. 80 for http)
- A *path* (Like a file path on the machine, here it is maps/api/staticmap)
- A *query* part after a ?, (optional, usually ampersand-separated *parameters* e.g. size=400x400, or zoom=12)

Supplementary materials: These can actually be different for different protocols, the above is a simplification, you can see more, for example, at https://en.wikipedia.org/wiki/URI_scheme

URLs are not allowed to include all characters; we need to, for example, "escape" a space that appears inside the URL, replacing it with %20, so e.g. a request of http://some example.com/ would need to be http://some%20example.com/

Supplementary materials: The code used to replace each character is the ASCII code for it.

Supplementary materials: The escaping rules a are quite subtle. See https://en.wikipedia.org/wiki/Percent-encoding

14.2 Requests

The python requests library can help us manage and manipulate URLs. It is easier to use than the 'urllib' library that is part of the standard library, and is included with anaconda and canopy. It sorts out escaping, parameter encoding, and so on for us.

To request the above URL, for example, we write:

```
In [2]: import requests

In [3]: response = requests.get("http://maps.googleapis.com/maps/api/staticmap",
                                params={
                    'size' : '400x400',
                    'center' : '51.51,-0.1275',
                    'zoom' : 12
                })

In [4]: response

Out[4]: <Response [200]>

In [5]: response.url

Out[5]: 'http://maps.googleapis.com/maps/api/staticmap?size=400x400&zoom=12&center=51.51%2C
```

When we do a request, the result comes back as text. For the png image in the above, this isn't very readable:

```
In [6]: response.content[0:10]

Out[6]: b'\x89PNG\r\n\x1a\n\x00\x00'
```

Just as for file access, therefore, we will need to send the text we get to a python module which understands that file format.

Again, it is important to separate the *transport* model, (e.g. a file system, or an "http request" for the web), from the data model of the data that is returned.

14.3 Example: Sunspots

Let's try to get something scientific: the sunspot cycle data from http://sidc.be/silso/home:

```
In [7]: spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php').text

In [8]: spots[-81:]

Out[8]: '86;0\n2016;07;2016.540;  32.5;  3.7;  910;0\n2016;08;2016.624;  50.7;  4.4;  879;0
```

This looks like semicolon-separated data, with different records on different lines. (Line separators come out as `\n`)

There are many many scientific datasets which can now be downloaded like this - integrating the download into your data pipeline can help to keep your data flows organised.

14.4 Writing our own Parser

We'll need a python library to handle semicolon-separated data like the sunspot data.

You might be thinking: "But I can do that myself!":

```
In [9]: lines = spots.split("\n")
        lines[0:5]

Out[9]: ['1749;01;1749.042;  96.7; -1.0;   -1;1',
         '1749;02;1749.123; 104.3; -1.0;   -1;1',
         '1749;03;1749.204; 116.7; -1.0;   -1;1',
         '1749;04;1749.288;  92.8; -1.0;   -1;1',
         '1749;05;1749.371; 141.7; -1.0;   -1;1']
```

```
In [10]: years = [line.split(";")[0] for line in lines]

In [11]: years[0:15]

Out[11]: ['1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1749',
          '1750',
          '1750',
          '1750']
```

But **don't**: what if, for example, one of the records contains a separator inside it; most computers will put the content in quotes, so that, for example,

```
"something; something"; something; something
```

has three fields, the first of which is

```
something; something
```

The naive code above would give four fields, of which the first is

```
"Something
```

You'll never manage to get all that right; so you'll be better off using a library to do it.

14.5 Writing data to the internet

Note that we're using `requests.get`. `get` is used to receive data from the web. You can also use `post` to fill in a web-form programmatically.

Supplementary material: Learn about using `post` with requests.

Supplementary material: Learn about the different kinds of http request: Get, Post, Put, Delete...

This can be used for all kinds of things, for example, to programmatically add data to a web resource. It's all well beyond our scope for this course, but it's important to know it's possible, and start to think about the scientific possibilities.

Chapter 15

Field and Record Data

15.1 Separated Value Files

Let's carry on with our sunspots example:

```
In [1]: import requests
        spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php').text
        spots.split('\n')[0]
```

```
Out[1]: '1749;01;1749.042;  96.7; -1.0;   -1;1'
```

We want to work programmatically with *Separated Value* files.
These are files which have:

- Each *record* on a line
- Each record has multiple *fields*
- Fields are separated by some *separator*

Typical separators are the `space`, `tab`, `comma`, and `semicolon` separated values files, e.g.:

- Space separated value (e.g. `field1, "field two", field3`)
- Comma separated value (e.g. 'field1, another field, "wow, another field")

Comma-separated-value is abbreviated CSV, and tab separated value TSV.
CSV is also used to refer to all the different sub-kinds of separated value files, i.e. some people use csv to refer to tab, space and semicolon separated files.

CSV is not a particularly superb data format, because it forces your data model to be a list of lists. Richer file formats describe "serialisations" for dictionaries and for deeper-than-two nested list structures as well.

Nevertheless, because you can always export *spreadsheets* as CSV files, (each cell is a field, each row is a record) CSV files are very popular.

15.2 CSV variants.

Some CSV formats define a comment character, so that rows beginning with, e.g., a #, are not treated as data, but give a human index.

Some CSV formats define a three-deep list structure, where a double-newline separates records into blocks.

Some CSV formats assume that the first line defines the names of the fields, e.g.:

```
name, age, job
James, 38, Programmer
Elizabeth, 89, Queen
```

15.3 Python CSV readers

The Python standard library has a `csv` module. However, it's less powerful than the CSV capabilities in `numpy`, the main scientific python library for handling data. Numpy is destributed with Anaconda and Canopy, so we recommend you just use that.

Numpy has powerful capabilities for handling matrices, and other fun stuff, and we'll learn about these later in the course, but for now, we'll just use numpy's CSV reader, and assume it makes us lists and dictionaries, rather than it's more exciting `array` type.

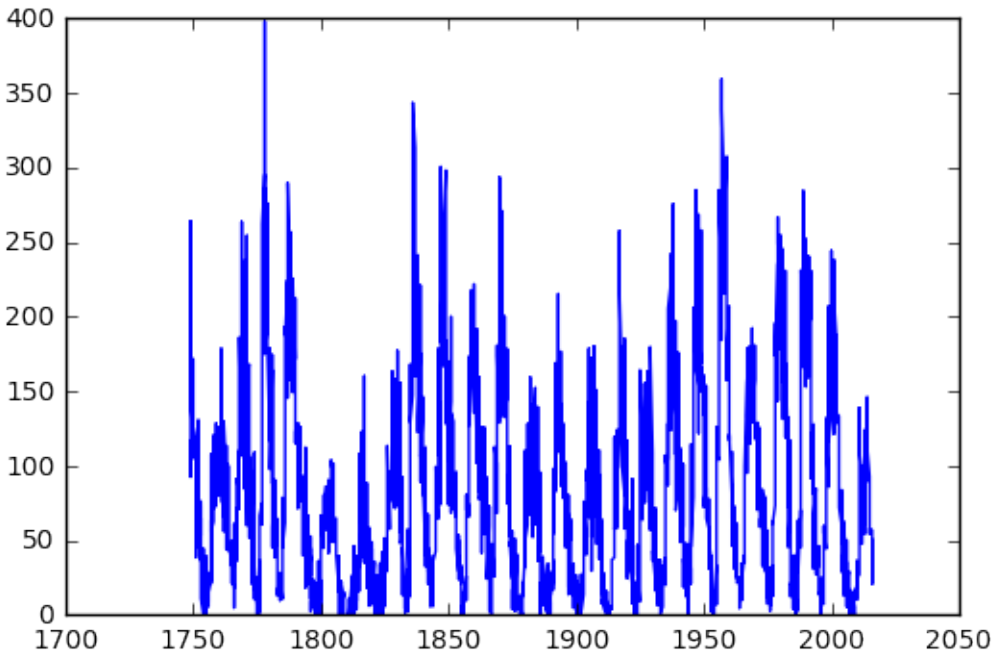
```
In [2]: import numpy as np
        from io import BytesIO

In [3]: sunspots = np.genfromtxt(BytesIO(spots.encode()), delimiter=';')
```

`genfromtxt` is a powerful CSV reader. I used the `delimiter` optional argument to specify the delimeter. I could also specify `names=True` if I had a first line naming fields, and `comments=#` if I had comment lines.

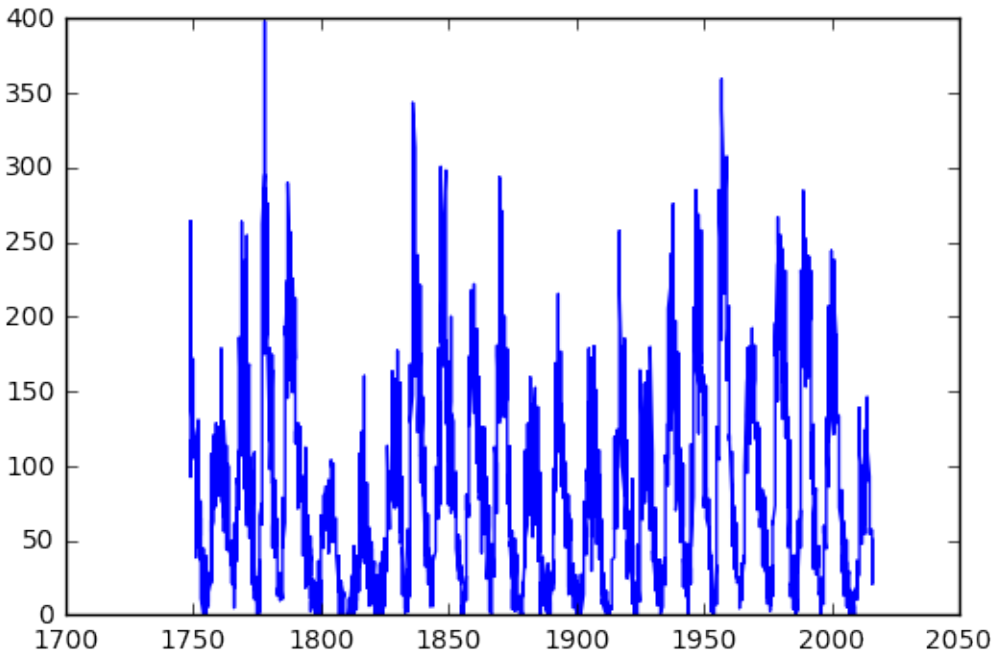
```
In [4]: sunspots

Out[4]: array([[  1.74900000e+03,   1.00000000e+00,   1.74904200e+03, ...,
                 -1.00000000e+00,  -1.00000000e+00,   1.00000000e+00],
                [  1.74900000e+03,   2.00000000e+00,   1.74912300e+03, ...,
                 -1.00000000e+00,  -1.00000000e+00,   1.00000000e+00],
                [  1.74900000e+03,   3.00000000e+00,   1.74920400e+03, ...,
                 -1.00000000e+00,  -1.00000000e+00,   1.00000000e+00],
                ...,
                [  2.01600000e+03,   6.00000000e+00,   2.01645600e+03, ...,
                   2.20000000e+00,   8.86000000e+02,   0.00000000e+00],
                [  2.01600000e+03,   7.00000000e+00,   2.01654000e+03, ...,
                   3.70000000e+00,   9.10000000e+02,   0.00000000e+00],
                [  2.01600000e+03,   8.00000000e+00,   2.01662400e+03, ...,
                   4.40000000e+00,   8.79000000e+02,   0.00000000e+00]])
```

We can now plot the "Sunspot cycle":

```
In [5]: %matplotlib inline
        from matplotlib import pyplot as plt
        plt.plot(sunspots[:,0], sunspots[:,3]) # Numpy syntax to access all
                                               #rows, specified column.

/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer


Out[5]: [<matplotlib.lines.Line2D at 0x2af6acc97da0>]
```




```
In [6]: %matplotlib inline
        import requests
        import numpy as np
        from io import BytesIO
        from matplotlib import pyplot as plt

        spots =  requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php').text
        sunspots = np.genfromtxt(BytesIO(spots.encode()), delimiter=';')
        plt.plot(sunspots[:,0], sunspots[:,3])

Out[6]: [<matplotlib.lines.Line2D at 0x2af6acd7ec88>]
```



The plot command accepted an array of 'X' values and an array of 'Y' values. We used a special NumPy ":" syntax, which we'll learn more about later.

15.4 Naming Columns

I happen to know that the columns here are defined as follows:
From http://www.sidc.be/silso/infosnmtot:

> CSV
>
> Filename: SN_m_tot_V2.0.csv Format: Comma Separated values (adapted for import in spreadsheets) The separator is the semicolon ';'.
>
> Contents: * Column 1-2: Gregorian calendar date - Year - Month * Column 3: Date in fraction of year. * Column 4: Monthly mean total sunspot number. * Column 5: Monthly mean standard deviation of the input sunspot numbers. * Column 6: Number of observations used to compute the monthly mean total sunspot number. * Column 7: Definitive/provisional marker. '1' indicates that the value is definitive. '0' indicates that the value is still provisional.

I can actually specify this to the formatter:

```
In [7]: sunspots = np.genfromtxt(BytesIO(spots.encode()), delimiter=';',
                                 names=['year','month','date',
                                 'mean','deviation','observations','definitive'])

In [8]: sunspots

Out[8]: array([(1749.0, 1.0, 1749.042, 96.7, -1.0, -1.0, 1.0),
               (1749.0, 2.0, 1749.123, 104.3, -1.0, -1.0, 1.0),
               (1749.0, 3.0, 1749.204, 116.7, -1.0, -1.0, 1.0), ...,
```

```
               (2016.0, 6.0, 2016.456, 20.9, 2.2, 886.0, 0.0),
               (2016.0, 7.0, 2016.54, 32.5, 3.7, 910.0, 0.0),
               (2016.0, 8.0, 2016.624, 50.7, 4.4, 879.0, 0.0)],
              dtype=[('year', '<f8'), ('month', '<f8'), ('date', '<f8'), ('mean', '<f8'),
```

```
In [9]: sunspots['year']
```

```
Out[9]: array([ 1749.,  1749.,  1749., ...,  2016.,  2016.,  2016.])
```

15.5 Typed Fields

It's also often good to specify the datatype of each field.

```
In [10]: sunspots = np.genfromtxt(BytesIO(spots.encode()), delimiter=';',
                                  names=['year','month','date',
                                  'mean','deviation','observations','definitive'],
                                  dtype=[int, int, float, float, float, int, int])
```

```
In [11]: sunspots
```

```
Out[11]: array([(1749, 1, 1749.042, 96.7, -1.0, -1, 1),
                (1749, 2, 1749.123, 104.3, -1.0, -1, 1),
                (1749, 3, 1749.204, 116.7, -1.0, -1, 1), ...,
                (2016, 6, 2016.456, 20.9, 2.2, 886, 0),
                (2016, 7, 2016.54, 32.5, 3.7, 910, 0),
                (2016, 8, 2016.624, 50.7, 4.4, 879, 0)],
               dtype=[('year', '<i8'), ('month', '<i8'), ('date', '<f8'), ('mean', '<f8'),
```

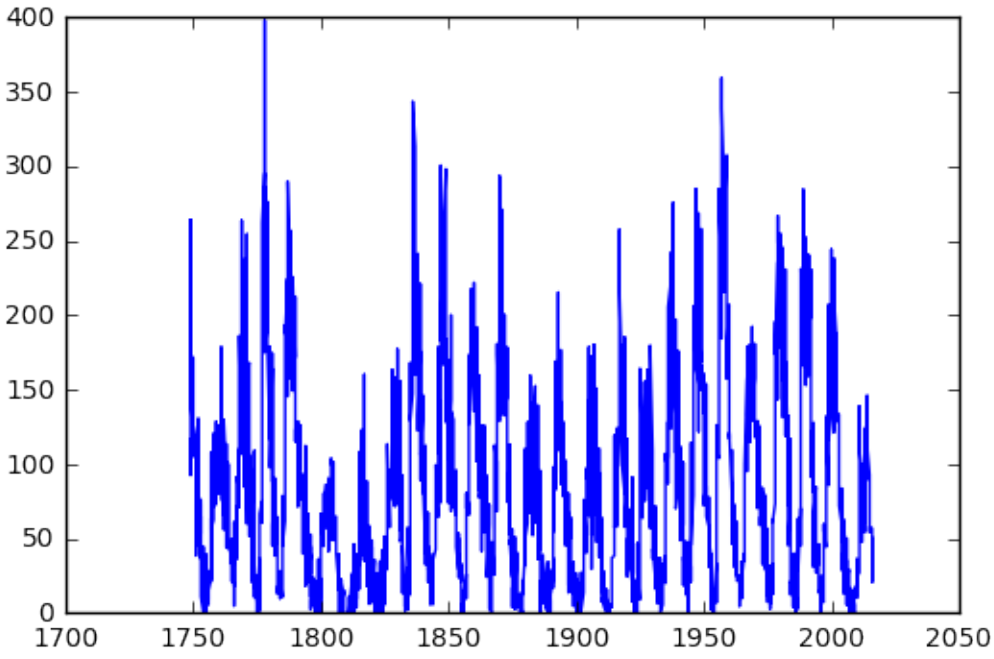
Now, NumPy understands the names of the columns, so our plot command is more readable:

```
In [12]: sunspots['year']
```

```
Out[12]: array([1749, 1749, 1749, ..., 2016, 2016, 2016])
```

```
In [13]: plt.plot(sunspots['year'],sunspots['mean'])
```

```
Out[13]: [<matplotlib.lines.Line2D at 0x2af6ace99c50>]
```



```
In [1]: from pip import main as pip
        pip(['install', 'pyaml'])

Collecting pyaml
Collecting PyYAML (from pyaml)
Installing collected packages: PyYAML, pyaml
Successfully installed PyYAML-3.12 pyaml-16.9.0


Out[1]: 0
```

Chapter 16

Structured Data

16.1 Structured data

CSV files can only model data where each record has several fields, and each field is a simple datatype, a string or number.

We often want to store data which is more complicated than this, with nested structures of lists and dictionaries. Structured data formats like Json, YAML, and XML are designed for this.

16.2 Json

A very common structured data format is JSON.

This allows us to represent data which is combinations of lists and dictionaries as a text file which looks a bit like a Javascript (or Python) data literal.

```
In [2]: import json
```

Any nested group of dictionaries and lists can be saved:

```
In [3]: mydata =  {'key': ['value1', 'value2'], 'key2': {'key4':'value3'}}

In [4]: text = json.dumps(mydata)

In [5]: with open('myfile.json', 'w') as f:
            f.write(text)
```

Loading data is also really easy:

```
In [6]: file_as_string = open('myfile.json').read()

In [7]: file_as_string

Out[7]: '{"key2": {"key4": "value3"}, "key": ["value1", "value2"]}'

In [8]: mydata = json.loads(file_as_string)

In [9]: mydata['key2']

Out[9]: {'key4': 'value3'}
```

This is a very nice solution for loading and saving python datastructures.

It's a very common way of transferring data on the internet, and of saving datasets to disk.

There's good support in most languages, so it's a nice inter-language file interchange format.

16.3 Unicode

Supplementary Material: Why do the strings come back with 'u' everywhere? These are **Unicode Strings**, designed to hold hold all the world's characters.

16.4 Yaml

Yaml is a very similar dataformat to Json, with some nice additions:

- You don't need to quote strings if they don't have funny characters in
- You can have comment lines, beginning with a #
- You can write dictionaries without the curly brackets: it just notices the colons.
- You can write lists like this:

```
In [10]: %%writefile myfile.yaml

         somekey:
             # Look, this is a list
             - a list
             - with values
             - [1, 2, 4] # and another list of integers
```

```
Writing myfile.yaml
```

```
In [11]: import yaml
```

```
In [12]: yaml.load(open('myfile.yaml'))
```

```
Out[12]: {'somekey': ['a list', 'with values', [1, 2, 4]]}
```

Yaml is a very versatile format for ad-hoc datafiles, but the library doesn't ship with default Python, (though it is part of Anaconda and Canopy) so some people still prefer Json for it's univerality.

Because Yaml gives the **option** of serialising a list either as newlines with dashes, *or* with square brackets, you can control this choice:

```
In [13]: print(yaml.dump(mydata))
```

```
key: [value1, value2]
key2: {key4: value3}
```

```
In [14]: print(yaml.dump(mydata, default_flow_style=False))
```

```
key:
- value1
- value2
key2:
  key4: value3
```

16.5 XML

Supplementary material: XML is another popular choice when saving nested data structures. It's very careful, but verbose. If your field uses XML data, you'll need to learn a python XML parser, (there are a few), and about how XML works.

16.6 Exercise:

Use YAML or JSON to save your maze datastructure to disk and load it again.

```
In [15]: %%writefile foo.txt
         Write this text

Writing foo.txt
```

```
In [16]: myfile = open('foo.txt', 'w')

In [17]: myfile.write("Write THIS text")

Out[17]: 15
```

16.7 Solution: Saving and Loading a Maze

```
In [1]: house = {
            'living' : {
                'exits': {
                    'north' : 'kitchen',
                    'outside' : 'garden',
                    'upstairs' : 'bedroom'
                },
                'people' : ['James'],
                'capacity' : 2
            },
            'kitchen' : {
                'exits': {
                    'south' : 'living'
                },
                'people' : [],
                'capacity' : 1
            },
            'garden' : {
                'exits': {
                    'inside' : 'living'
                },
                'people' : ['Sue'],
                'capacity' : 3
            },
            'bedroom' : {
                'exits': {
                    'downstairs' : 'living',
                    'jump' : 'garden'
                },
                'people' : [],
                'capacity' : 1
            }
        }
```

Save the maze with json:

```
In [2]: import json
```

```
In [3]: with open('maze.json','w') as json_maze_out:
            json_maze_out.write(json.dumps(house))
```

Consider the file on the disk:

```
In [4]: %%bash
        cat 'maze.json'
```

```
{"bedroom": {"exits": {"jump": "garden", "downstairs": "living"}, "capacity": 1, "people":
```

and now load it into a different variable:

```
In [5]: with open('maze.json') as json_maze_in:
            maze_again = json.load(json_maze_in)

In [6]: maze_again

Out[6]: {'bedroom': {'capacity': 1,
          'exits': {'downstairs': 'living', 'jump': 'garden'},
          'people': []},
         'garden': {'capacity': 3, 'exits': {'inside': 'living'}, 'people': ['Sue']},
         'kitchen': {'capacity': 1, 'exits': {'south': 'living'}, 'people': []},
         'living': {'capacity': 2,
          'exits': {'north': 'kitchen', 'outside': 'garden', 'upstairs': 'bedroom'},
          'people': ['James']}}
```

Or with YAML:

```
In [7]: import yaml

In [8]: with open('maze.yaml','w') as yaml_maze_out:
            yaml_maze_out.write(yaml.dump(house))

In [9]: %%bash
        cat 'maze.yaml'
```

```
bedroom:
  capacity: 1
  exits: {downstairs: living, jump: garden}
  people: []
garden:
  capacity: 3
  exits: {inside: living}
  people: [Sue]
kitchen:
  capacity: 1
  exits: {south: living}
  people: []
living:
  capacity: 2
  exits: {north: kitchen, outside: garden, upstairs: bedroom}
  people: [James]
```

```
In [10]: with open('maze.yaml') as yaml_maze_in:
             maze_again = yaml.load(yaml_maze_in)
```

```
In [11]: maze_again

Out[11]: {'bedroom': {'capacity': 1,
            'exits': {'downstairs': 'living', 'jump': 'garden'},
            'people': []},
           'garden': {'capacity': 3, 'exits': {'inside': 'living'}, 'people': ['Sue']},
           'kitchen': {'capacity': 1, 'exits': {'south': 'living'}, 'people': []},
           'living': {'capacity': 2,
            'exits': {'north': 'kitchen', 'outside': 'garden', 'upstairs': 'bedroom'},
            'people': ['James']}}
```

Chapter 17

Extended Exercise: the biggest Earthquake in the UK this Century

17.1 The Problem

GeoJSON is a json-based file format for sharing geographic data. One example dataset is the USGS earthquake data:

```
In [1]: import requests
        quakes=requests.get("http://earthquake.usgs.gov/fdsnws/event/1/query.geojson",
                            params={
                    'starttime':"2000-01-01",
                    "maxlatitude":"58.723",
                    "minlatitude":"50.008",
                    "maxlongitude":"1.67",
                    "minlongitude":"-9.756",
                    "minmagnitude":"1",
                    "endtime":"2015-11-22",
                    "orderby":"time-asc"}
                            )

In [2]: quakes.text[0:100]

Out[2]: '{"type":"FeatureCollection","metadata":{"generated":1475242118000,"url":"http://ea
```

Your exercise: determine the location of the largest magnitude earthquake in the UK this century.

You'll need to: * Get the text of the web result * Parse the data as JSON * Understand how the data is structured into dictionaries and lists * Where is the magnitude? * Where is the place description or coordinates? * Program a search through all the quakes to find the biggest quake. * Find the place of the biggest quake * Form a URL for Google Maps at that latitude and longitude: look back at the introductory example * Display that image

```
In [3]: quakes

Out[3]: <Response [200]>

In [4]: type(quakes.text)

Out[4]: str

In [5]: import json
```

```
In [6]: data=json.loads(quakes.text)

In [7]: type(data)

Out[7]: dict

In [8]: data.keys()

Out[8]: dict_keys(['type', 'features', 'metadata', 'bbox'])

In [9]: type(data['features'])

Out[9]: list

In [10]: type(data['features'][0])

Out[10]: dict

In [11]: data['features'][0].keys()

Out[11]: dict_keys(['type', 'properties', 'id', 'geometry'])

In [12]: data['features'][0]['properties'].keys()

Out[12]: dict_keys(['status', 'gap', 'felt', 'type', 'sig', 'tsunami', 'time', 'updated',

In [13]: data['features'][0]['properties']['mag']

Out[13]: 2.6

In [14]: data['features'][0]['geometry']

Out[14]: {'coordinates': [-2.81, 54.77, 14], 'type': 'Point'}

In [15]: data['features'][0]['geometry']['coordinates'][0]

Out[15]: -2.81

In [16]: for quake in data['features']:
             print("This quake has magnitude", quake['properties']['mag'])
             print(" ... and is at longitude",quake['geometry']['coordinates'][0])
             print(" ... and latitude", quake['geometry']['coordinates'][1])
             print()

This quake has magnitude 2.6
 ... and is at longitude -2.81
 ... and latitude 54.77

This quake has magnitude 4
 ... and is at longitude -1.61
 ... and latitude 52.28

This quake has magnitude 4
 ... and is at longitude 1.564
 ... and latitude 53.236

This quake has magnitude 3.3
 ... and is at longitude 0.872
 ... and latitude 58.097
```

```
This quake has magnitude 2.9
 ... and is at longitude -1.845
 ... and latitude 51.432

This quake has magnitude 2.9
 ... and is at longitude -3.639
 ... and latitude 55.102

This quake has magnitude 4
 ... and is at longitude -4.684
 ... and latitude 50.995

This quake has magnitude 2.6
 ... and is at longitude 1.144
 ... and latitude 51.76

This quake has magnitude 2.6
 ... and is at longitude 1.094
 ... and latitude 51.332

This quake has magnitude 3.5
 ... and is at longitude -3.205
 ... and latitude 51.552

This quake has magnitude 2.5
 ... and is at longitude -3.25
 ... and latitude 51.7

This quake has magnitude 4.2
 ... and is at longitude -0.856
 ... and latitude 52.846

This quake has magnitude 3
 ... and is at longitude -3.14
 ... and latitude 51.63

This quake has magnitude 3.4
 ... and is at longitude 1.288
 ... and latitude 53.168

This quake has magnitude 3.5
 ... and is at longitude -3.255
 ... and latitude 51.7

This quake has magnitude 3.4
 ... and is at longitude -3.081
 ... and latitude 51.567

This quake has magnitude 3
 ... and is at longitude -0.009
 ... and latitude 50.048

This quake has magnitude 2.3
```

```
 ... and is at longitude -5.749
 ... and latitude 56.596

This quake has magnitude 2.1
 ... and is at longitude -3.588
 ... and latitude 51.713

This quake has magnitude 4.8
 ... and is at longitude -2.15
 ... and latitude 52.52

This quake has magnitude 3.2
 ... and is at longitude -2.136
 ... and latitude 52.522

This quake has magnitude 1.2
 ... and is at longitude -2.138
 ... and latitude 52.521

This quake has magnitude 3.7
 ... and is at longitude -2
 ... and latitude 53.475

This quake has magnitude 4.3
 ... and is at longitude -2.219
 ... and latitude 53.478

This quake has magnitude 2.9
 ... and is at longitude -2.219
 ... and latitude 53.463

This quake has magnitude 3.5
 ... and is at longitude -2.146
 ... and latitude 53.473

This quake has magnitude 3.3
 ... and is at longitude -2.157
 ... and latitude 53.477

This quake has magnitude 3.8
 ... and is at longitude -2.179
 ... and latitude 53.485

This quake has magnitude 2.8
 ... and is at longitude -2.197
 ... and latitude 53.482

This quake has magnitude 2.5
 ... and is at longitude -2.204
 ... and latitude 53.481

This quake has magnitude 2.5
 ... and is at longitude -2.213
 ... and latitude 53.488
```

```
This quake has magnitude 2.6
 ... and is at longitude -2.188
 ... and latitude 53.477

This quake has magnitude 3.1
 ... and is at longitude -2.198
 ... and latitude 53.481

This quake has magnitude 2.3
 ... and is at longitude 1.582
 ... and latitude 51.055

This quake has magnitude 3
 ... and is at longitude -4.416
 ... and latitude 56.169

This quake has magnitude 2.8
 ... and is at longitude -4.439
 ... and latitude 56.181

This quake has magnitude 2.5
 ... and is at longitude -4.427
 ... and latitude 56.167

This quake has magnitude 3.2
 ... and is at longitude -1.013
 ... and latitude 53.481

This quake has magnitude 3.3
 ... and is at longitude -2.98
 ... and latitude 51.089

This quake has magnitude 3.6
 ... and is at longitude -2.98
 ... and latitude 51.089

This quake has magnitude 3.7
 ... and is at longitude -2.98
 ... and latitude 51.089

This quake has magnitude 3.4
 ... and is at longitude -1.999
 ... and latitude 53.566

This quake has magnitude 2.7
 ... and is at longitude 0.765
 ... and latitude 58.146

This quake has magnitude 3
 ... and is at longitude -0.602
 ... and latitude 53.936

This quake has magnitude 3.8
```

```
 ... and is at longitude -3.853
 ... and latitude 53.265

This quake has magnitude 2.6
 ... and is at longitude -2.055
 ... and latitude 53.092

This quake has magnitude 2.9
 ... and is at longitude -0.392
 ... and latitude 51.008

This quake has magnitude 3
 ... and is at longitude -5.224
 ... and latitude 56.839

This quake has magnitude 2.4
 ... and is at longitude -0.395
 ... and latitude 51.772

This quake has magnitude 3.2
 ... and is at longitude -5.641
 ... and latitude 53.001

This quake has magnitude 2.7
 ... and is at longitude -5.685
 ... and latitude 56.68

This quake has magnitude 2.4
 ... and is at longitude -5.662
 ... and latitude 56.668

This quake has magnitude 1.7
 ... and is at longitude -3.736
 ... and latitude 56.245

This quake has magnitude 2.5
 ... and is at longitude -3.761
 ... and latitude 56.277

This quake has magnitude 2.6
 ... and is at longitude -1.22
 ... and latitude 51.31

This quake has magnitude 1.9
 ... and is at longitude -5.233
 ... and latitude 56.676

This quake has magnitude 2.9
 ... and is at longitude -5.642
 ... and latitude 57.531

This quake has magnitude 2.5
 ... and is at longitude -2.988
 ... and latitude 51.094
```

```
This quake has magnitude 2.7
 ... and is at longitude -3.032
 ... and latitude 52.031

This quake has magnitude 1.4
 ... and is at longitude -5.251
 ... and latitude 56.705

This quake has magnitude 2.7
 ... and is at longitude -2.569
 ... and latitude 52.352

This quake has magnitude 2.8
 ... and is at longitude -4.512
 ... and latitude 50.346

This quake has magnitude 3.6
 ... and is at longitude -3.634
 ... and latitude 55.085

This quake has magnitude 3.6
 ... and is at longitude 0.998
 ... and latitude 53.666

This quake has magnitude 1.3
 ... and is at longitude -5.624
 ... and latitude 56.942

This quake has magnitude 1.7
 ... and is at longitude -1.254
 ... and latitude 53.458

This quake has magnitude 1.4
 ... and is at longitude -1.248
 ... and latitude 53.461

This quake has magnitude 1.7
 ... and is at longitude -1.225
 ... and latitude 53.453

This quake has magnitude 1.2
 ... and is at longitude -1.239
 ... and latitude 53.461

This quake has magnitude 1.6
 ... and is at longitude -1.23
 ... and latitude 53.46

This quake has magnitude 1.3
 ... and is at longitude -1.212
 ... and latitude 53.476

This quake has magnitude 1.4
```

```
 ... and is at longitude -1.223
 ... and latitude 53.453

This quake has magnitude 4.6
 ... and is at longitude 1.009
 ... and latitude 51.085

This quake has magnitude 2.6
 ... and is at longitude -0.957
 ... and latitude 52.801

This quake has magnitude 2.5
 ... and is at longitude -2.185
 ... and latitude 53.488

This quake has magnitude 2.2
 ... and is at longitude -2.178
 ... and latitude 53.482

This quake has magnitude 2.3
 ... and is at longitude 1.415
 ... and latitude 51.399

This quake has magnitude 2.3
 ... and is at longitude -3.206
 ... and latitude 55.804

This quake has magnitude 3.3
 ... and is at longitude -3.277
 ... and latitude 52.866

This quake has magnitude 2.3
 ... and is at longitude -3.221
 ... and latitude 55.788

This quake has magnitude 3.1
 ... and is at longitude 1.032
 ... and latitude 58.204

This quake has magnitude 4.8
 ... and is at longitude -0.332
 ... and latitude 53.403

This quake has magnitude 2.8
 ... and is at longitude -6.04
 ... and latitude 50.382

This quake has magnitude 3.1
 ... and is at longitude -0.351
 ... and latitude 53.357

This quake has magnitude 2.5
 ... and is at longitude -2.952
 ... and latitude 54.691
```

```
This quake has magnitude 1.5
 ... and is at longitude -2.574
 ... and latitude 53.38

This quake has magnitude 3.5
 ... and is at longitude -5.557
 ... and latitude 56.829

This quake has magnitude 3.9
 ... and is at longitude -2.512
 ... and latitude 52.212

This quake has magnitude 2.7
 ... and is at longitude 1.18
 ... and latitude 50.44

This quake has magnitude 3.5
 ... and is at longitude 1.178
 ... and latitude 51.116

This quake has magnitude 3
 ... and is at longitude -0.252
 ... and latitude 53.687

This quake has magnitude 3.7
 ... and is at longitude -3.017
 ... and latitude 54.167

This quake has magnitude 3.5
 ... and is at longitude -3.146
 ... and latitude 54.39

This quake has magnitude 3.6
 ... and is at longitude -1.652
 ... and latitude 54.169

This quake has magnitude 3.5
 ... and is at longitude -5.784
 ... and latitude 56.822

This quake has magnitude 2.7
 ... and is at longitude -3.734
 ... and latitude 50.571

This quake has magnitude 3.9
 ... and is at longitude -0.743
 ... and latitude 50.122

This quake has magnitude 2.9
 ... and is at longitude -1.25
 ... and latitude 52.801

This quake has magnitude 2.3
```

```
 ... and is at longitude -4.231
 ... and latitude 53.125

This quake has magnitude 2.9
 ... and is at longitude -5.715
 ... and latitude 56.776

This quake has magnitude 3.8
 ... and is at longitude -4.719
 ... and latitude 52.883

This quake has magnitude 2.8
 ... and is at longitude -4.72
 ... and latitude 52.879

This quake has magnitude 3.2
 ... and is at longitude -3.403
 ... and latitude 53.887

This quake has magnitude 4.1
 ... and is at longitude -4.164
 ... and latitude 51.363

This quake has magnitude 3.5
 ... and is at longitude -0.732
 ... and latitude 52.722

This quake has magnitude 2.6
 ... and is at longitude -1.191
 ... and latitude 53.057

This quake has magnitude 2.9
 ... and is at longitude -1.299
 ... and latitude 51.072

This quake has magnitude 3.8
 ... and is at longitude -0.717
 ... and latitude 52.727

This quake has magnitude 3.7
 ... and is at longitude 1.438
 ... and latitude 51.304

This quake has magnitude 3
 ... and is at longitude -4.355
 ... and latitude 53.116

This quake has magnitude 2.8
 ... and is at longitude -0.693
 ... and latitude 52.714
```

```
In [1]: %matplotlib inline
        from matplotlib import pyplot as plt
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

17.2 Download the data

```
In [2]: import requests
        quakes_response=requests.get("http://earthquake.usgs.gov/fdsnws/event/1/query.geojs
                                params={
                        'starttime':"2000-01-01",
                        "maxlatitude":"58.723",
                        "minlatitude":"50.008",
                        "maxlongitude":"1.67",
                        "minlongitude":"-9.756",
                        "minmagnitude":"1",
                        "endtime":"2015-07-13",
                        "orderby":"time-asc"}
                                )
```

```
In [3]: quakes_response
```

```
Out[3]: <Response [200]>
```

```
In [4]: type(quakes_response)
```

```
Out[4]: requests.models.Response
```

17.3 Parse the data as JSON

```
In [5]: import json
```

```
In [6]: requests_json = json.loads(quakes_response.text)
```

17.4 Investigate the data to discover how it is structured.

```
In [7]: type(requests_json)
```

```
Out[7]: dict
```

```
In [8]: requests_json.keys()
```

```
Out[8]: dict_keys(['bbox', 'features', 'metadata', 'type'])
```

```
In [9]: type(requests_json['features'])
```

```
Out[9]: list
```

```
In [10]: len(requests_json['features'])
```

```
Out[10]: 110
```

```
In [11]: type(requests_json['features'][0])
```

```
Out[11]: dict
```

```
In [12]: requests_json['features'][0].keys()
```

```
Out[12]: dict_keys(['geometry', 'id', 'type', 'properties'])
```

```
In [13]: requests_json['features'][0]['properties']['mag']
```

```
Out[13]: 2.6
```

```
In [14]: requests_json['features'][0]['geometry']['coordinates']
```

```
Out[14]: [-2.81, 54.77, 14]
```

17.5 Find the largest quake

```
In [15]: quakes = requests_json['features']
```

```
In [16]: largest_so_far = quakes[0]

         for quake in quakes:
             if quake['properties']['mag'] > largest_so_far['properties']['mag']:
                 largest_so_far = quake

         largest_so_far
```

```
Out[16]: {'geometry': {'coordinates': [-2.15, 52.52, 9.4], 'type': 'Point'},
          'id': 'usp000bcxg',
          'properties': {'alert': None,
           'cdi': None,
           'code': 'p000bcxg',
           'detail': 'http://earthquake.usgs.gov/fdsnws/event/1/query?eventid=usp000bcxg&fo
           'dmin': None,
           'felt': None,
           'gap': None,
           'ids': ',usp000bcxg,atlas20020922235314,',
           'mag': 4.8,
           'magType': 'mb',
           'mmi': None,
           'net': 'us',
           'nst': 268,
           'place': 'England, United Kingdom',
           'rms': None,
           'sig': 354,
           'sources': ',us,atlas,',
           'status': 'reviewed',
           'time': 1032738794600,
           'title': 'M 4.8 - England, United Kingdom',
           'tsunami': 0,
           'type': 'earthquake',
           'types': ',impact-text,origin,phase-data,',
           'tz': None,
           'updated': 1426874123970,
           'url': 'http://earthquake.usgs.gov/earthquakes/eventpage/usp000bcxg'},
          'type': 'Feature'}
```

```
In [17]: print(largest_so_far['properties']['mag'])
```

```
4.8
```

```
In [18]: lat=largest_so_far['geometry']['coordinates'][1]
         long=largest_so_far['geometry']['coordinates'][0]
         print("Latitude:", lat, "Longitude:", long)
```

```
Latitude: 52.52 Longitude: -2.15
```

17.6 Get a map at the point of the quake

```
In [19]: import requests
         def request_map_at(lat,long, satellite=False,zoom=12,size=(400,400),sensor=False):
           base="http://maps.googleapis.com/maps/api/staticmap?"

           params=dict(
             sensor= str(sensor).lower(),
             zoom= zoom,
             size= "x".join(map(str,size)),
             center= ",".join(map(str,(lat,long)))
           )
           if satellite:
             params["maptype"]="satellite"

           return requests.get(base,params=params)
```

```
In [20]: import IPython
         map_png=request_map_at(lat, long,zoom=10)
```

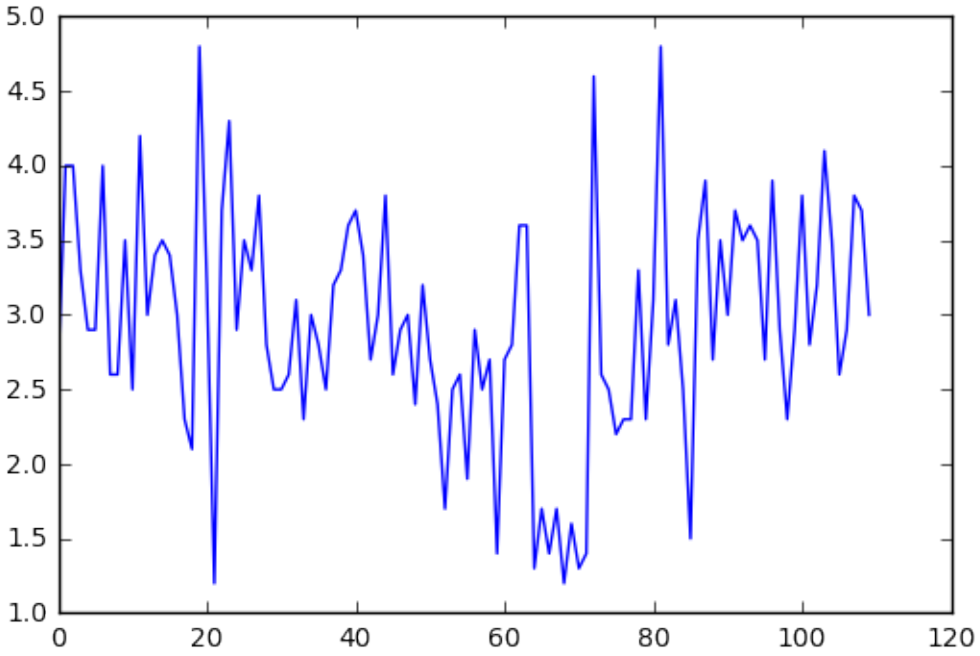
17.7 Display the map

```
In [21]: IPython.core.display.Image(map_png.content)
```

```
Out[21]:
```


```
In [22]: plt.plot([  quake['properties']['mag'] for quake in quakes  ])
```

```
Out[22]: [<matplotlib.lines.Line2D at 0x2b5c52ae67f0>]
```




```
In [1]: %matplotlib inline

/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

Chapter 18

Plotting with Matplotlib

18.1 Importing Matplotlib

We import the 'pyplot' object from Matplotlib, which provides us with an interface for making figures. We usually abbreviate it.

```
In [2]: from matplotlib import pyplot as plt
```

18.2 Notebook magics

When we write:

```
In [3]: %matplotlib inline
```

We tell the IPython notebook to show figures we generate alongside the code that created it, rather than in a separate window. Lines beginning with a single percent are not python code: they control how the notebook deals with python code.

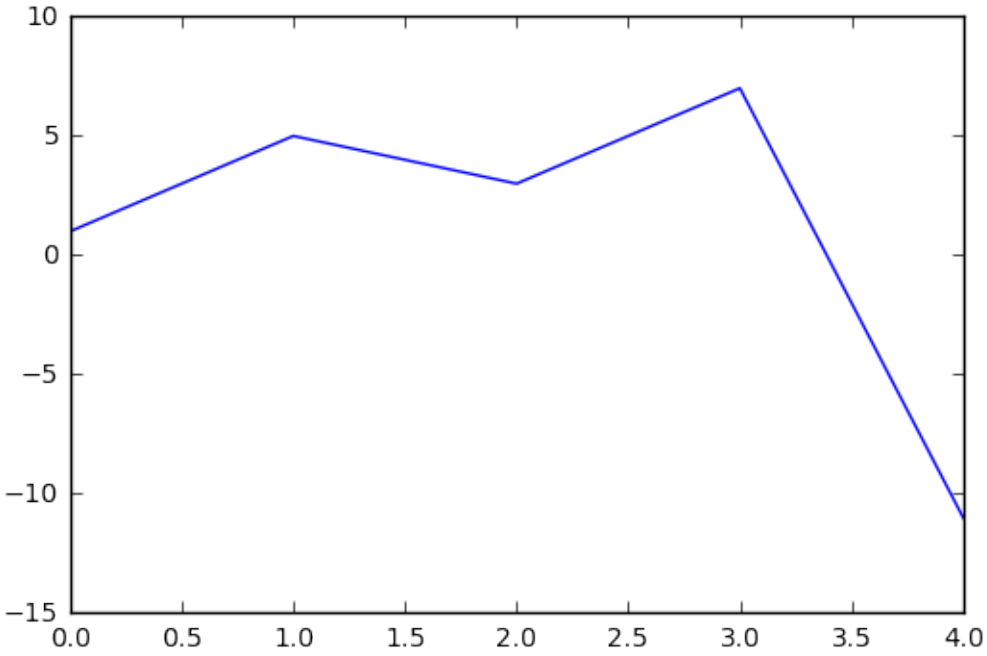
Lines beginning with two percents are "cell magics", that tell IPython notebook how to interpret the particular cell; we've seen `%%writefile`, for example.

On MacOS, in some corner cases (virtual environments), `%matplotlib inline` may need to be the first line in the notebook.

18.3 A basic plot

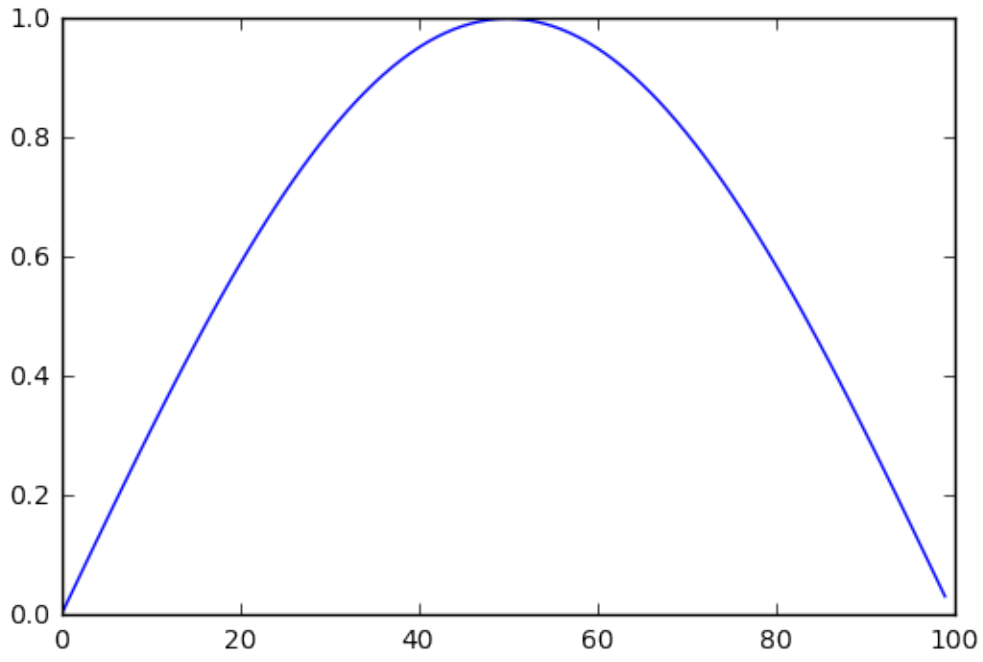
When we write:

```
In [4]: plt.plot([1,5,3,7,-11])

Out[4]: [<matplotlib.lines.Line2D at 0x2b30f11a30f0>]
```




```
In [5]: from math import sin, cos, pi
        plt.plot([sin(pi*x/100.0) for x in range(100)])
```

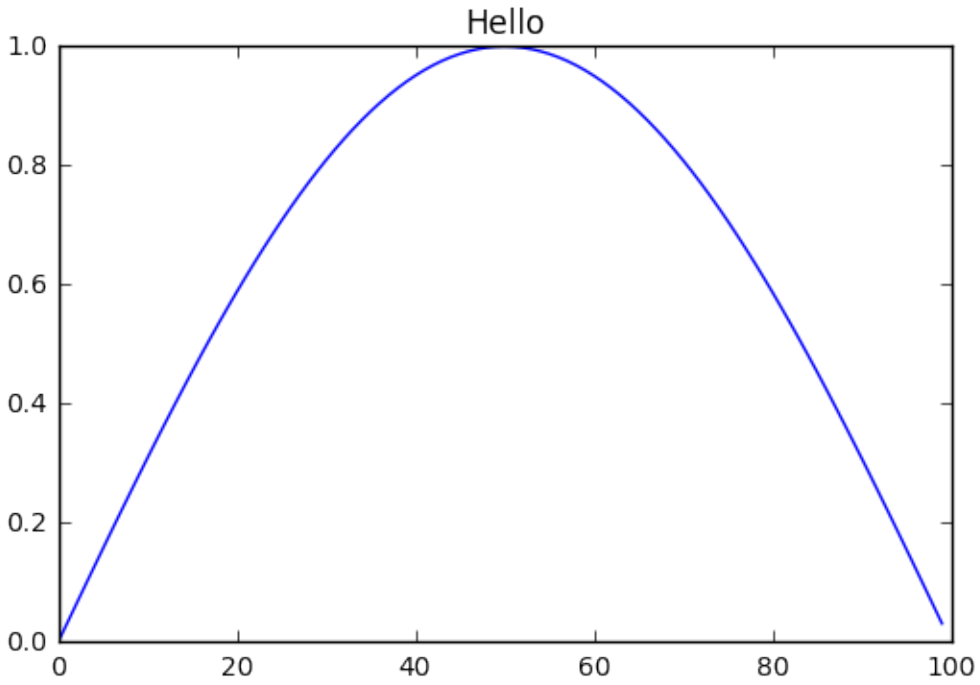
```
Out[5]: [<matplotlib.lines.Line2D at 0x2b30f11c2978>]
```



The plot command *returns* a figure, just like the return value of any function. The notebook then displays this.

To add a title, axis labels etc, we need to get that figure object, and manipulate it. For convenience, matplotlib allows us to do this just by issuing commands to change the "current figure":

```
In [6]: plt.plot([sin(pi*x/100.0) for x in range(100)])
        plt.title("Hello")

Out[6]: <matplotlib.text.Text at 0x2b30f1208c50>
```



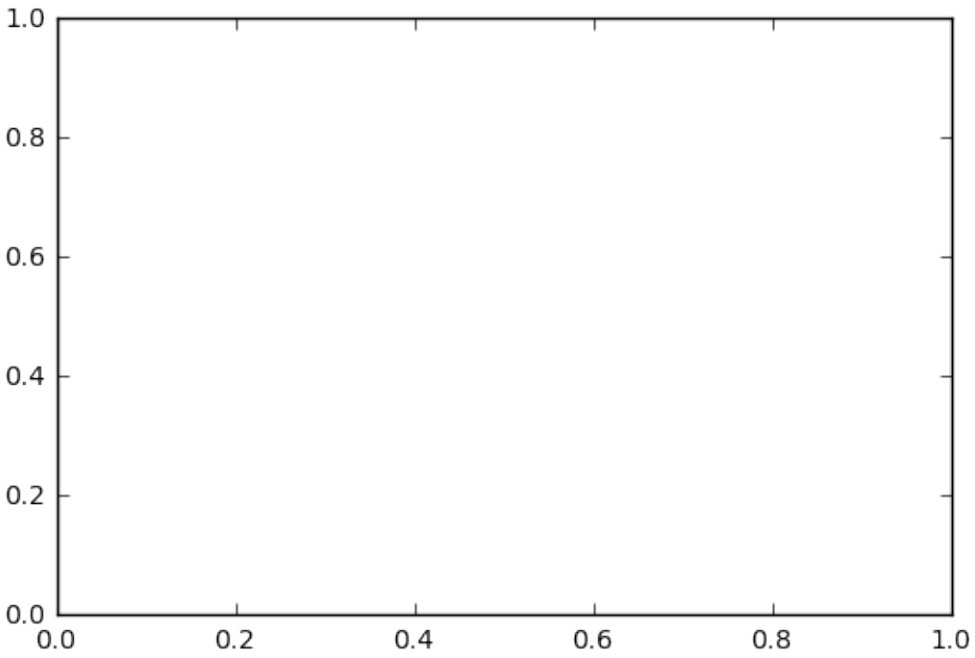
But this requires us to keep all our commands together in a single cell, and makes use of a "global" single "current plot", which, while convenient for quick exploratory sketches, is a bit cumbersome. To produce from our notebook proper plots to use in papers, Python's plotting library, matplotlib, defines some types we can use to treat individual figures as variables, and manipulate this.

18.4 Figures and Axes

We often want multiple graphs in a single figure (e.g. for figures which display a matrix of graphs of different variables for comparison).

So Matplotlib divides a `figure` object up into axes: each pair of axes is one 'subplot'. To make a boring figure with just one pair of axes, however, we can just ask for a default new figure, with brand new axes

```
In [7]: sine_graph, sine_graph_axes=plt.subplots();
```



Once we have some axes, we can plot a graph on them:

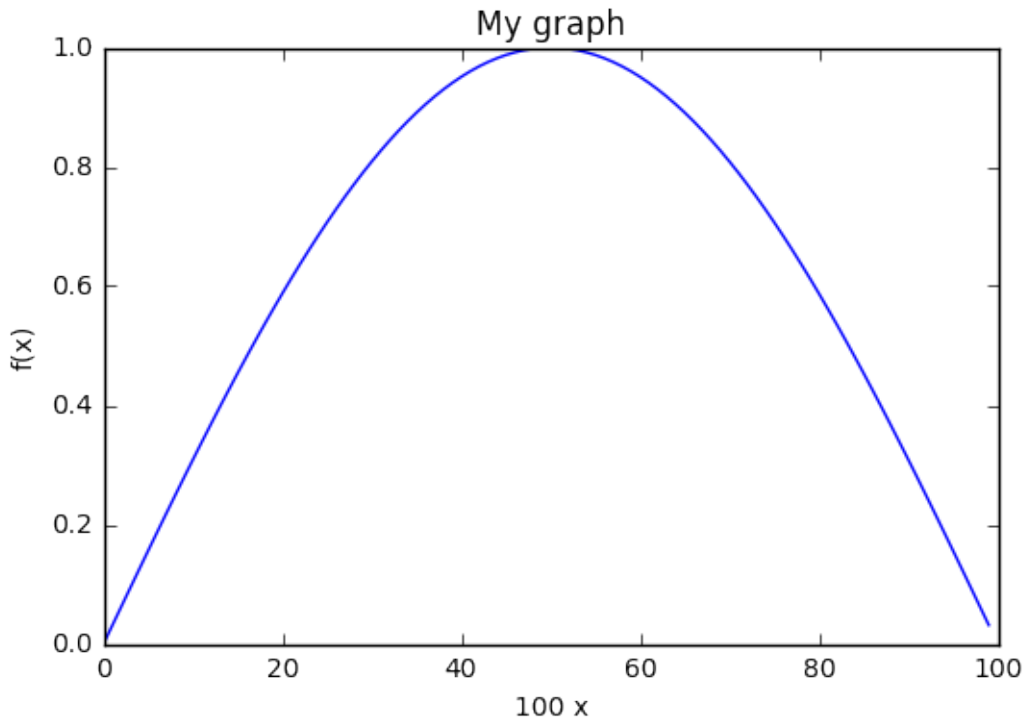
```
In [8]: sine_graph_axes.plot(
            [sin(pi*x/100.0) for x in range(100)],
            label='sin(x)')

Out[8]: [<matplotlib.lines.Line2D at 0x2b30f12632b0>]
```

We can add a title to a pair of axes:

```
In [9]: sine_graph_axes.set_title("My graph")

Out[9]: <matplotlib.text.Text at 0x2b30f1266438>

In [10]: sine_graph_axes.set_ylabel("f(x)")

Out[10]: <matplotlib.text.Text at 0x2b30f1246b70>

In [11]: sine_graph_axes.set_xlabel("100 x")

Out[11]: <matplotlib.text.Text at 0x2b30f11e9978>
```

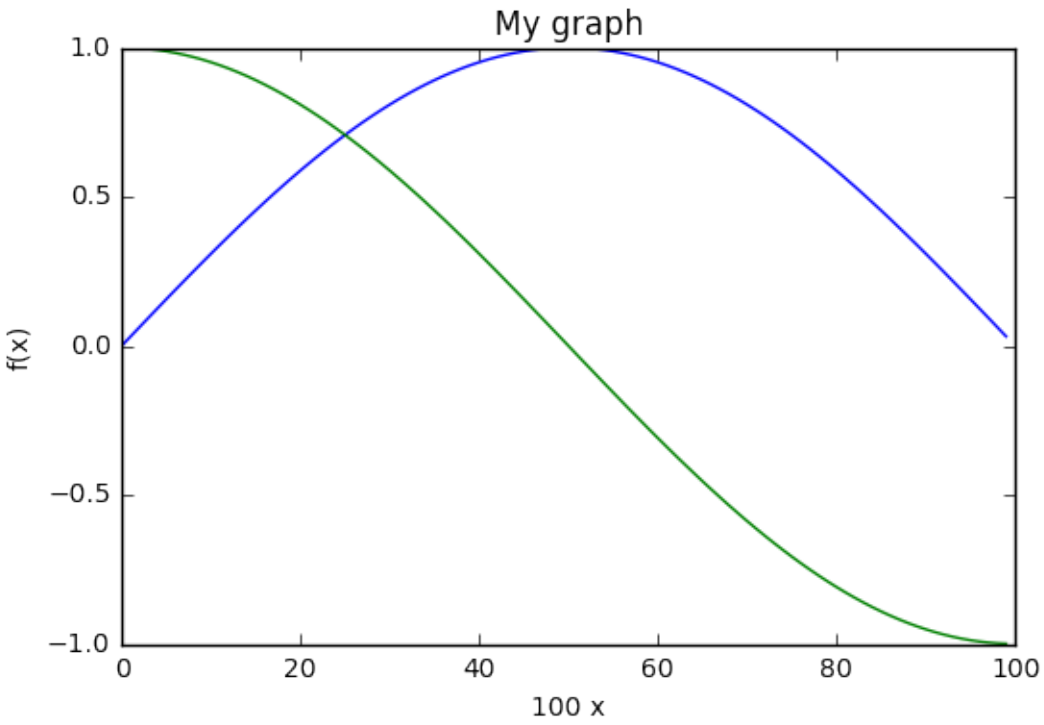
Now we need to actually display the figure. As always with the notebook, if we make a variable be returned by the last line of a code cell, it gets displayed:

```
In [12]: sine_graph

Out[12]:
```



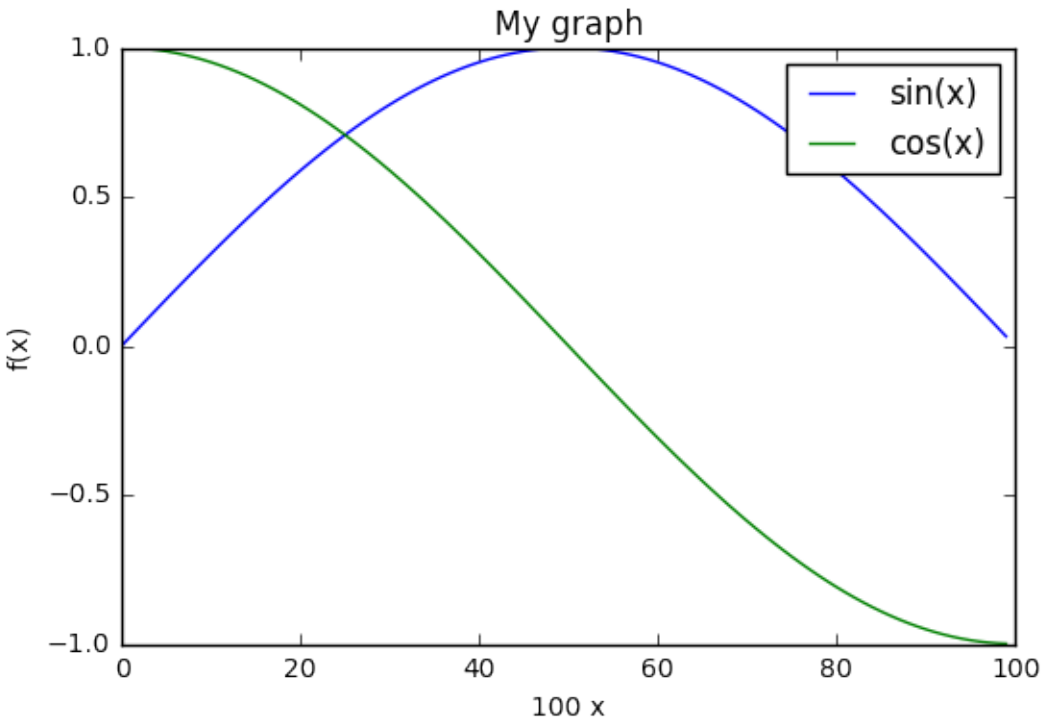
We can add another curve:

```
In [13]: sine_graph_axes.plot([cos(pi*x/100.0) for x in range(100)], label='cos(x)')

Out[13]: [<matplotlib.lines.Line2D at 0x2b30f12408d0>]

In [14]: sine_graph

Out[14]:
```



A legend will help us distinguish the curves:

```
In [15]: sine_graph_axes.legend()

Out[15]: <matplotlib.legend.Legend at 0x2b30f1337f60>

In [16]: sine_graph

Out[16]:
```



18.5 Saving figures.

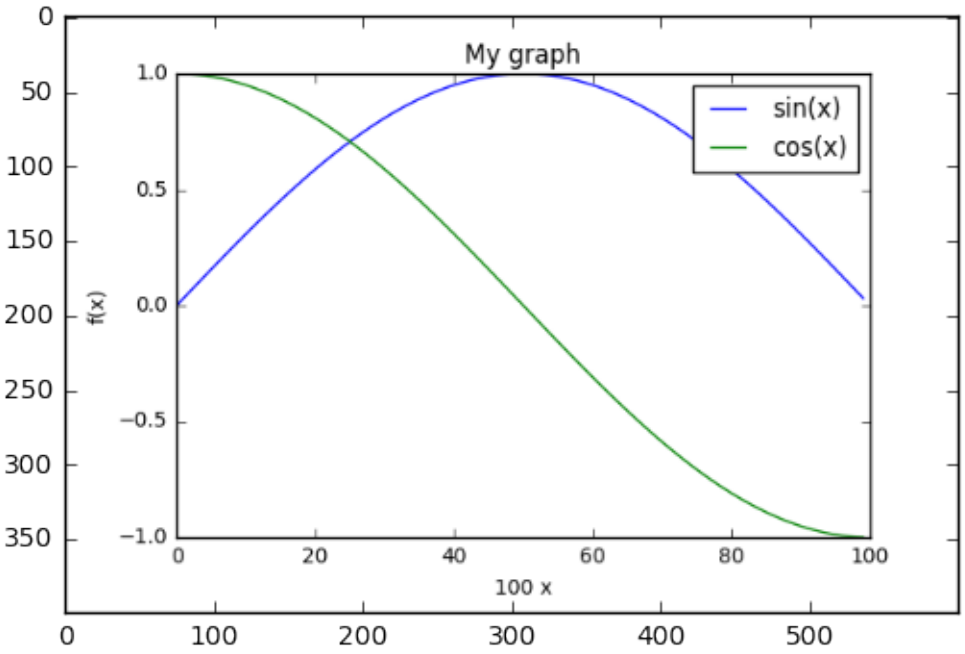
We must be able to save figures to disk, in order to use them in papers. This is really easy:

```
In [17]: sine_graph.savefig('my_graph.png')
```

In order to be able to check that it worked, we need to know how to display an arbitrary image in the notebook. You can also save in different formats (eps, svg, pdf...)

The programmatic way is like this:

```
In [18]: import matplotlib.image as mpimg
         img = mpimg.imread('my_graph.png')
         imgplot = plt.imshow(img)
```

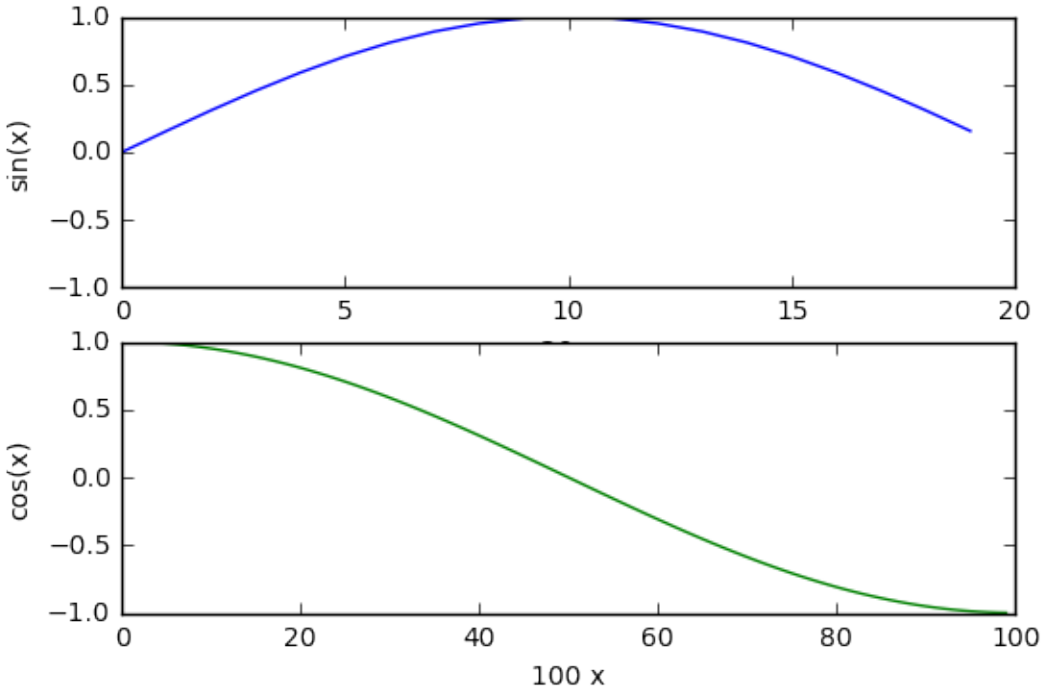


18.6 Subplots

We might have wanted the sin and cos graphs on separate axes:

```
In [19]: double_graph=plt.figure()

<matplotlib.figure.Figure at 0x2b30f1765ac8>

In [20]: sin_axes=double_graph.add_subplot(2,1,1)

In [21]: cos_axes=double_graph.add_subplot(2,1,2)

In [22]: sin_axes.plot([sin(pi*x/20.0) for x in range(20)])

Out[22]: [<matplotlib.lines.Line2D at 0x2b30f1bb2da0>]

In [23]: sin_axes.set_ylabel("sin(x)")

Out[23]: <matplotlib.text.Text at 0x2b30f1781e80>

In [24]: cos_axes.plot([cos(pi*x/100.0) for x in range(100)], 'g')

Out[24]: [<matplotlib.lines.Line2D at 0x2b30f1bb9940>]

In [25]: cos_axes.set_ylabel("cos(x)")

Out[25]: <matplotlib.text.Text at 0x2b30f1b78f28>

In [26]: cos_axes.set_xlabel("100 x")
```

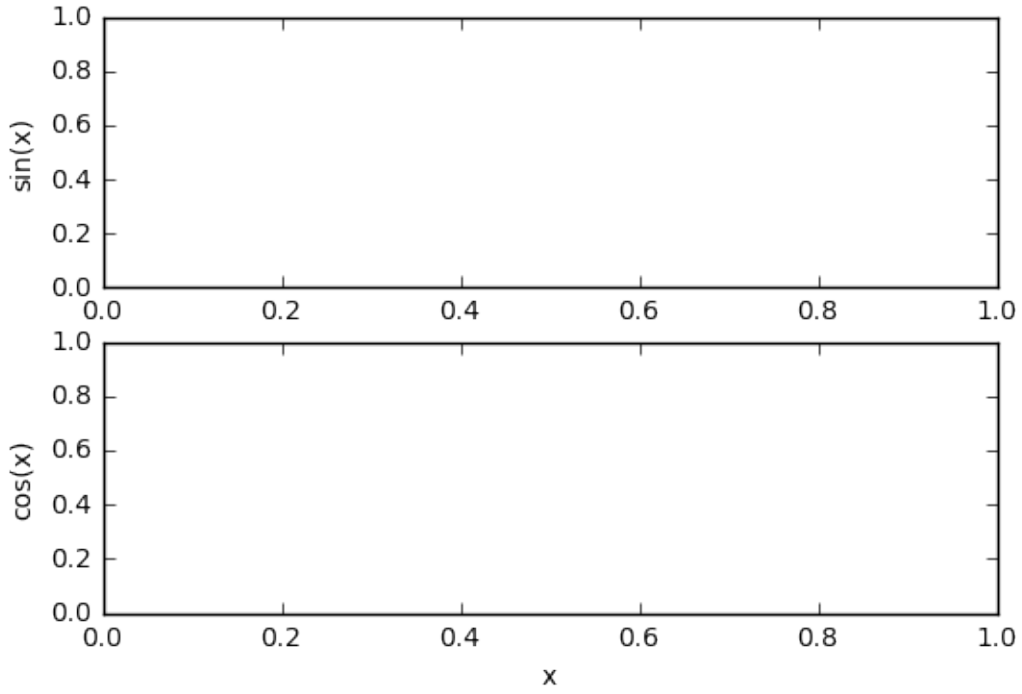
```
Out[26]: <matplotlib.text.Text at 0x2b30f1b718d0>

In [27]: sin_axes.set_xlabel("20 x")

Out[27]: <matplotlib.text.Text at 0x2b30f1715b00>

In [28]: sin_axes.set_ylim([-1,1])
         cos_axes.set_ylim([-1,1])

Out[28]: (-1, 1)

In [29]: double_graph

Out[29]:
```



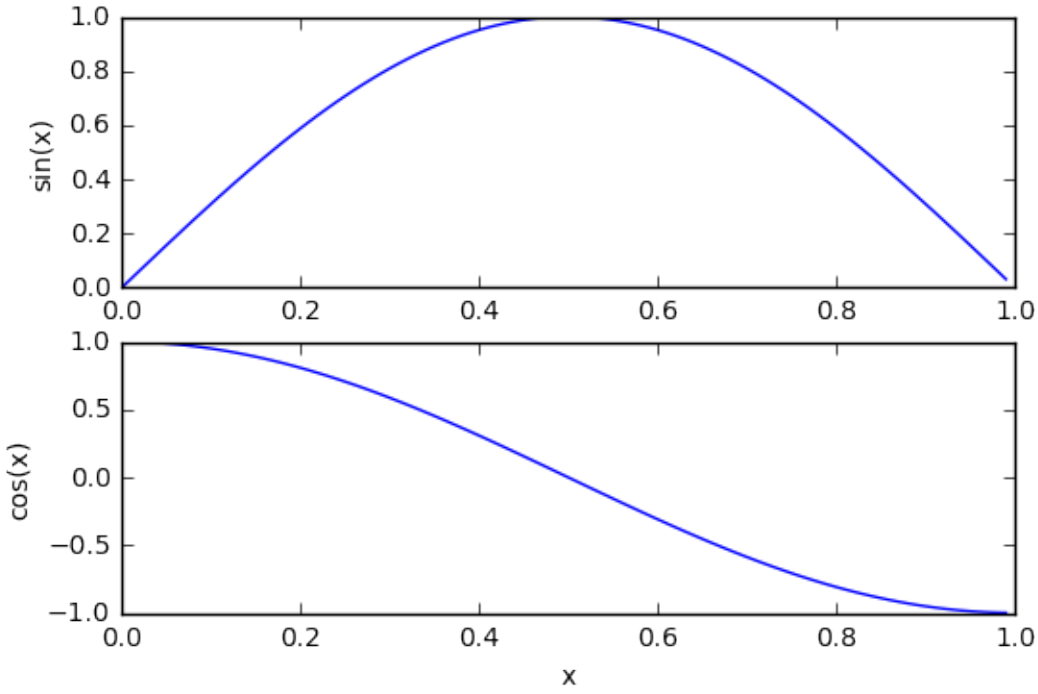
18.7 Versus plots

When we specify a single `list` to `plot`, the x-values are just the array index number. We usually want to plot something more meaningful:

```
In [30]: double_graph=plt.figure()
         sin_axes=double_graph.add_subplot(2,1,1)
         cos_axes=double_graph.add_subplot(2,1,2)
         cos_axes.set_ylabel("cos(x)")
         sin_axes.set_ylabel("sin(x)")
         cos_axes.set_xlabel("x")

Out[30]: <matplotlib.text.Text at 0x2b30f1c2c358>
```




```
In [31]: sin_axes.plot([x/100.0 for x in range(100)],
                       [sin(pi*x/100.0) for x in range(100)])
         cos_axes.plot([x/100.0 for x in range(100)],
                       [cos(pi*x/100.0) for x in range(100)])

Out[31]: [<matplotlib.lines.Line2D at 0x2b30f1c85cf8>]

In [32]: double_graph

Out[32]:
```



18.8 Learning More

There's so much more to learn about matplotlib: pie charts, bar charts, heat maps, 3-d plotting, animated plots, and so on. You can learn all this via the Matplotlib Website. You should try to get comfortable with all this, so please use some time in class, or at home, to work your way through a bunch of the examples.

Chapter 19

NumPy

19.1 The Scientific Python Trilogy

Why is Python so popular for research work?

MATLAB has typically been the most popular "language of technical computing", with strong built-in support for efficient numerical analysis with matrices (the *mat* in MATLAB is for Matrix, not Maths), and plotting.

Other dynamic languages have cleaner, more logical syntax (Ruby, Scheme)

But Python users developed three critical libraries, matching the power of MATLAB for scientific work:

- Matplotlib, the plotting library created by John D. Hunter
- NumPy, a fast matrix maths library created by Travis Oliphant
- Pandas, data structures and data analysis tools
- IPython, the precursor of the notebook, created by Fernando Perez

By combining a plotting library, a matrix maths library, and an easy-to-use interface allowing live plotting commands in a persistent environment, the powerful capabilities of MATLAB were matched by a free and open toolchain.

We've learned about Matplotlib and IPython in this course already. NumPy is the last part of the trilogy.

19.2 Limitations of Python Lists

The normal Python List is just one dimensional. To make a matrix, we have to nest Python arrays:

```
In [1]: x= [range(5) for something_unused in range(5)]

In [2]: x

Out[2]: [range(0, 5), range(0, 5), range(0, 5), range(0, 5), range(0, 5)]
```

Applying an operation to every element is a pain:

```
In [3]: x + 5


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-3-057023a07318> in <module>()
    ----> 1 x + 5
```

```
        TypeError: can only concatenate list (not "int") to list
```

```
In [4]: [[elem +5 for elem in row] for row in x]
```

```
Out[4]: [[5, 6, 7, 8, 9],
         [5, 6, 7, 8, 9],
         [5, 6, 7, 8, 9],
         [5, 6, 7, 8, 9],
         [5, 6, 7, 8, 9]]
```

Common useful operations like transposing a matrix or reshaping a 10 by 10 matrix into a 20 by 5 matrix are not easy to code in raw Python lists.

19.3 The NumPy array

NumPy's array type represents a multidimensional matrix $M_{i,j,k\ldots n}$

The NumPy array seems at first to be just like a list:

```
In [5]: import numpy as np
        my_array = np.array(range(5))
```

```
In [6]: my_array
```

```
Out[6]: array([0, 1, 2, 3, 4])
```

```
In [7]: my_array[2]
```

```
Out[7]: 2
```

```
In [8]: for element in my_array:
            print("Hello" * element)
```

```
Hello
HelloHello
HelloHelloHello
HelloHelloHelloHello
```

We can also see our first weakness of NumPy arrays versus Python lists:

```
In [9]: my_array.append(4)
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-9-ad82621ab44a> in <module>()
    ----> 1 my_array.append(4)


        AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

For NumPy arrays, you typically don't change the data size once you've defined your array, whereas for Python lists, you can do this efficiently. However, you get back lots of goodies in return...

19.4 Elementwise Operations

But most operations can be applied element-wise automatically!

```
In [10]: my_array * 2
```

```
Out[10]: array([0, 2, 4, 6, 8])
```

These "vectorized" operations are very fast:

```
In [11]: big_list = range(10000)
         big_array = np.arange(10000)
```

```
In [12]: %%timeit
         [x**2 for x in big_list]
```

```
100 loops, best of 3: 4.46 ms per loop
```

```
In [13]: %%timeit
         big_array**2
```

```
The slowest run took 5.02 times longer than the fastest. This could mean that an intermedia
100000 loops, best of 3: 14.8 µs per loop
```

19.5 Arange and linspace

NumPy has two easy methods for defining floating-point evenly spaced arrays:

```
In [14]: x=np.arange(0,10,0.1)
         x
```

```
Out[14]: array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
                 1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
                 2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
                 3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
                 4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
                 5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,  6.5,
                 6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,  7.6,
                 7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
                 8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
                 9.9])
```

We can quickly define non-integer ranges of numbers for graph plotting:

```
In [15]: import math
```

```
In [16]: values = np.linspace(0, math.pi, 100) # Start, stop, number of steps
```

```
In [17]: values
```

```
Out[17]: array([ 0.        ,  0.03173326,  0.06346652,  0.09519978,  0.12693304,
                 0.1586663 ,  0.19039955,  0.22213281,  0.25386607,  0.28559933,
                 0.31733259,  0.34906585,  0.38079911,  0.41253237,  0.44426563,
                 0.47599889,  0.50773215,  0.53946541,  0.57119866,  0.60293192,
                 0.63466518,  0.66639844,  0.6981317 ,  0.72986496,  0.76159822,
```

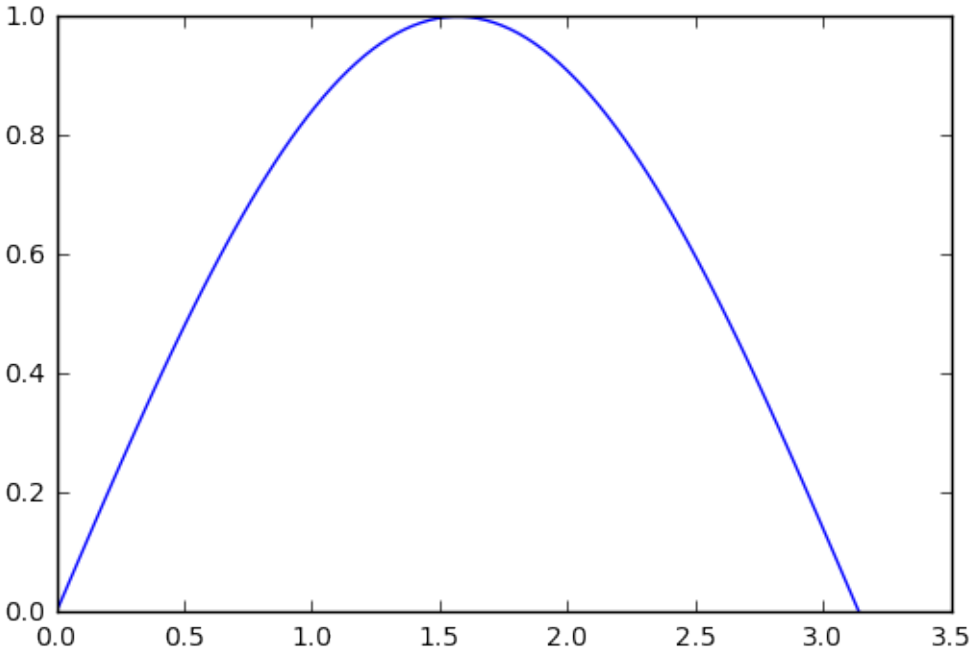
```
0.79333148,  0.82506474,  0.856798  ,  0.88853126,  0.92026451,
0.95199777,  0.98373103,  1.01546429,  1.04719755,  1.07893081,
1.11066407,  1.14239733,  1.17413059,  1.20586385,  1.23759711,
1.26933037,  1.30106362,  1.33279688,  1.36453014,  1.3962634 ,
1.42799666,  1.45972992,  1.49146318,  1.52319644,  1.5549297 ,
1.58666296,  1.61839622,  1.65012947,  1.68186273,  1.71359599,
1.74532925,  1.77706251,  1.80879577,  1.84052903,  1.87226229,
1.90399555,  1.93572881,  1.96746207,  1.99919533,  2.03092858,
2.06266184,  2.0943951 ,  2.12612836,  2.15786162,  2.18959488,
2.22132814,  2.2530614 ,  2.28479466,  2.31652792,  2.34826118,
2.37999443,  2.41172769,  2.44346095,  2.47519421,  2.50692747,
2.53866073,  2.57039399,  2.60212725,  2.63386051,  2.66559377,
2.69732703,  2.72906028,  2.76079354,  2.7925268 ,  2.82426006,
2.85599332,  2.88772658,  2.91945984,  2.9511931 ,  2.98292636,
3.01465962,  3.04639288,  3.07812614,  3.10985939,  3.14159265])
```

NumPy comes with 'vectorised' versions of common functions which work element-by-element when applied to arrays:

```
In [18]: %matplotlib inline
         from matplotlib import pyplot as plt
         plt.plot(values, np.sin(values))
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

```
Out[18]: [<matplotlib.lines.Line2D at 0x2ad35def10b8>]
```



So we don't have to use awkward list comprehensions when using these.

19.6 Multi-Dimensional Arrays

NumPy's true power comes from multi-dimensional arrays:

```
In [19]: np.zeros([3,4,2])

Out[19]: array([[[ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.]],

                [[ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.]],

                [[ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.],
                 [ 0.,  0.]]])
```

Unlike a list-of-lists in Python, we can reshape arrays:

```
In [20]: x=np.array(range(40))
         y=x.reshape([4,5,2])
         y

Out[20]: array([[[ 0,  1],
                 [ 2,  3],
                 [ 4,  5],
                 [ 6,  7],
                 [ 8,  9]],

                [[10, 11],
                 [12, 13],
                 [14, 15],
                 [16, 17],
                 [18, 19]],

                [[20, 21],
                 [22, 23],
                 [24, 25],
                 [26, 27],
                 [28, 29]],

                [[30, 31],
                 [32, 33],
                 [34, 35],
                 [36, 37],
                 [38, 39]]])
```

And index multiple columns at once:

```
In [21]: y[3,2,1]

Out[21]: 35
```

Including selecting on inner axes while taking all from the outermost:

```
In [22]: y[:,2,1]

Out[22]: array([ 5, 15, 25, 35])
```

And subselecting ranges:

```
In [23]: y[2:,:1,:]

Out[23]: array([[[20, 21]],

                [[30, 31]]])
```

And transpose arrays:

```
In [24]: y.transpose()

Out[24]: array([[[ 0, 10, 20, 30],
                 [ 2, 12, 22, 32],
                 [ 4, 14, 24, 34],
                 [ 6, 16, 26, 36],
                 [ 8, 18, 28, 38]],

                [[ 1, 11, 21, 31],
                 [ 3, 13, 23, 33],
                 [ 5, 15, 25, 35],
                 [ 7, 17, 27, 37],
                 [ 9, 19, 29, 39]]])
```

You can get the dimensions of an array with `shape`

```
In [25]: y.shape

Out[25]: (4, 5, 2)

In [26]: y.transpose().shape

Out[26]: (2, 5, 4)
```

Some numpy functions apply by default to the whole array, but can be chosen to act only on certain axes:

```
In [27]: x=np.arange(12).reshape(4,3)
         x

Out[27]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])

In [28]: x.sum(1) # Sum along the second axis, leaving the first.

Out[28]: array([ 3, 12, 21, 30])

In [29]: x.sum(0) # Sum along the first axis, leaving the second.

Out[29]: array([18, 22, 26])

In [30]: x.sum() # Sum all axes

Out[30]: 66
```

19.7 Array Datatypes

A Python `list` can contain data of mixed type:

```
In [31]: x = ['hello', 2, 3.4]

In [32]: type(x[2])

Out[32]: float

In [33]: type(x[1])

Out[33]: int
```

A NumPy array always contains just one datatype:

```
In [34]: np.array(x)

Out[34]: array(['hello', '2', '3.4'],
               dtype='<U5')
```

NumPy will choose the least-generic-possible datatype that can contain the data:

```
In [35]: y=np.array([2, 3.4])

In [36]: y

Out[36]: array([ 2. ,  3.4])

In [37]: type(y[0])

Out[37]: numpy.float64
```

19.8 Broadcasting

This is another really powerful feature of NumPy

By default, array operations are element-by-element:

```
In [38]: np.arange(5) * np.arange(5)

Out[38]: array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with non-matching shapes we get an error:

```
In [39]: np.arange(5) * np.arange(6)


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-39-66b7c967724c> in <module>()
    ----> 1 np.arange(5) * np.arange(6)


        ValueError: operands could not be broadcast together with shapes (5,) (6,)
```

```
In [40]: np.zeros([2,3]) * np.zeros([2,4])

         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-40-bf6e403cd465> in <module>()
         ----> 1 np.zeros([2,3]) * np.zeros([2,4])


         ValueError: operands could not be broadcast together with shapes (2,3) (2,4)


In [41]: m1 = np.arange(100).reshape([10, 10])

In [42]: m2 = np.arange(100).reshape([10, 5, 2])

In [43]: m1 + m2

         ---------------------------------------------------------------------------

         ValueError                                Traceback (most recent call last)

         <ipython-input-43-24e1f0a857c4> in <module>()
         ----> 1 m1 + m2


         ValueError: operands could not be broadcast together with shapes (10,10) (10,5,2)
```

Arrays must match in all dimensions in order to be compatible:

```
In [44]: np.ones([3,3]) * np.ones([3,3]) # Note elementwise multiply, *not* matrix multipl

Out[44]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])
```

Except, that if one array has any Dimension 1, then the data is **REPEATED** to match the other.

```
In [45]: m1 = np.arange(10).reshape([10,1])
         m1

Out[45]: array([[0],
                [1],
                [2],
                [3],
                [4],
                [5],
                [6],
                [7],
                [8],
                [9]])

In [46]: m2 = m1.transpose()
         m2
```

```
Out[46]: array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])

In [47]: m1.shape # "Column Vector"

Out[47]: (10, 1)

In [48]: m2.shape # "Row Vector"

Out[48]: (1, 10)

In [49]: m1 + m2

Out[49]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
                [ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11],
                [ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
                [ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13],
                [ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14],
                [ 6,  7,  8,  9, 10, 11, 12, 13, 14, 15],
                [ 7,  8,  9, 10, 11, 12, 13, 14, 15, 16],
                [ 8,  9, 10, 11, 12, 13, 14, 15, 16, 17],
                [ 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]])

In [50]: 10 * m1 + m2

Out[50]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
                [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
                [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
                [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
                [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
                [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

This works for arrays with more than one unit dimension.

19.9 Newaxis

Broadcasting is very powerful, and numpy allows indexing with `np.newaxis` to temporarily create new one-long dimensions on the fly.

```
In [51]: x = np.arange(10).reshape(2,5)
         y = np.arange(8).reshape(2,2,2)

In [52]: x.reshape(2,5,1,1)

Out[52]: array([[[[0]],

                 [[1]],

                 [[2]],

                 [[3]],
```

```
                [[4]]],


               [[[5]],

                [[6]],

                [[7]],

                [[8]],

                [[9]]]])
In [53]: x[:,:,np.newaxis,np.newaxis].shape

Out[53]: (2, 5, 1, 1)

In [54]: y[:,np.newaxis,:,:].shape

Out[54]: (2, 1, 2, 2)

In [55]: res = x[:,:,np.newaxis,np.newaxis] * y[:,np.newaxis,:,:]

In [56]: res.shape

Out[56]: (2, 5, 2, 2)

In [57]: np.sum(res)

Out[57]: 830
```

Note that `newaxis` works because a $3 \times 1 \times 3$ array and a 3×3 array contain the same data, differently shaped:

```
In [58]: threebythree = np.arange(9).reshape(3,3)
         threebythree

Out[58]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [59]: threebythree[:, np.newaxis, :]

Out[59]: array([[[0, 1, 2]],

                [[3, 4, 5]],

                [[6, 7, 8]]])
```

19.10 Dot Products

NumPy multiply is element-by-element, not a dot-product:

```
In [60]: np.arange(9).reshape(3,3) * np.arange(3,12).reshape(3,3)

Out[60]: array([[ 0,  4, 10],
                [18, 28, 40],
                [54, 70, 88]])
```

To get a dot-product, do this:

```
In [61]: np.dot(np.arange(9).reshape(3,3), np.arange(3,12).reshape(3,3))

Out[61]: array([[ 24,  27,  30],
                [ 78,  90, 102],
                [132, 153, 174]])

In [62]: x = np.ones([3,5])
         y = np.ones([5,4])

In [63]: (x[:,:,np.newaxis] * y[np.newaxis,:,:]).sum(1)

Out[63]: array([[ 5.,  5.,  5.,  5.],
                [ 5.,  5.,  5.,  5.],
                [ 5.,  5.,  5.,  5.]])
```

19.11 Array DTypes

Arrays have a "dtype" which specifies their datatype:

```
In [64]: x=[2, 3.4, 7.2, 0]

In [65]: np.array(x)

Out[65]: array([ 2. ,  3.4,  7.2,  0. ])

In [66]: np.array(x).dtype

Out[66]: dtype('float64')
```

These are, when you get to know them, fairly obvious string codes for datatypes: NumPy supports all kinds of datatypes beyond the python basics.

NumPy will convert python type names to dtypes:

```
In [67]: int_array = np.array(x, dtype=int)

In [68]: float_array = np.array(x, dtype=float)

In [69]: int_array

Out[69]: array([2, 3, 7, 0])

In [70]: float_array

Out[70]: array([ 2. ,  3.4,  7.2,  0. ])

In [71]: int_array.dtype

Out[71]: dtype('int64')

In [72]: float_array.dtype

Out[72]: dtype('float64')
```

19.12 Record Arrays

These are a special array structure designed to match the CSV "Record and Field" model. It's a very different structure from the normal numPy array, and different fields *can* contain different datatypes. We saw this when we looked at CSV files:

```
In [73]: x = np.arange(50).reshape([10,5])

In [74]: record_x = x.view(dtype={'names': ["col1", "col2", "another", "more",
                                            "last"],
                                  'formats': [int]*5 } )

In [75]: record_x

Out[75]: array([[(0, 1, 2, 3, 4)],
                [(5, 6, 7, 8, 9)],
                [(10, 11, 12, 13, 14)],
                [(15, 16, 17, 18, 19)],
                [(20, 21, 22, 23, 24)],
                [(25, 26, 27, 28, 29)],
                [(30, 31, 32, 33, 34)],
                [(35, 36, 37, 38, 39)],
                [(40, 41, 42, 43, 44)],
                [(45, 46, 47, 48, 49)]],
               dtype=[('col1', '<i8'), ('col2', '<i8'), ('another', '<i8'), ('more', '<i8'
```

Record arrays can be addressed with field names like they were a dictionary:

```
In [76]: record_x['col1']

Out[76]: array([[ 0],
                [ 5],
                [10],
                [15],
                [20],
                [25],
                [30],
                [35],
                [40],
                [45]])
```

We've seen these already when we used NumPy's CSV parser.

19.13 Logical arrays, masking, and selection

Numpy defines operators like == and < to apply to arrays *element by element*

```
In [77]: x = np.zeros([3,4])
         x

Out[77]: array([[ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.]])

In [78]: y = np.arange(-1,2)[:,np.newaxis] * np.arange(-2,2)[np.newaxis,:]
         y
```

```
Out[78]: array([[ 2,  1,  0, -1],
                [ 0,  0,  0,  0],
                [-2, -1,  0,  1]])

In [79]: iszero =  x == y
         iszero

Out[79]: array([[False, False,  True, False],
                [ True,  True,  True,  True],
                [False, False,  True, False]], dtype=bool)
```

A logical array can be used to select elements from an array:

```
In [80]: y[np.logical_not(iszero)]

Out[80]: array([ 2,  1, -1, -2, -1,  1])
```

Although when printed, this comes out as a flat list, if assigned to, the *selected elements of the array are changed!*

```
In [81]: y[iszero] = 5

In [82]: y

Out[82]: array([[ 2,  1,  5, -1],
                [ 5,  5,  5,  5],
                [-2, -1,  5,  1]])
```

19.14 Numpy memory

Numpy memory management can be tricksy:

```
In [83]: x = np.arange(5)
         y = x[:]

In [84]: y[2] = 0
         x

Out[84]: array([0, 1, 0, 3, 4])

In [85]: x = np.arange(5)
         x = x + 2
         x

Out[85]: array([2, 3, 4, 5, 6])
```

We must use `np.copy` to force separate memory. Otherwise NumPy tries it's hardest to make slices be *views* on data.

Now, this has all been very theoretical, but let's go through a practical example, and see how powerful NumPy can be.

Chapter 20

The Boids!

20.1 Flocking

> The aggregate motion of a flock of birds, a herd of land animals, or a school of fish is a beautiful and familiar part of the natural world... The aggregate motion of the simulated flock is created by a distributed behavioral model much like that at work in a natural flock; the birds choose their own course. Each simulated bird is implemented as an independent actor that navigates according to its local perception of the dynamic environment, the laws of simulated physics that rule its motion, and a set of behaviors programmed into it... The aggregate motion of the simulated flock is the result of the dense interaction of the relatively simple behaviors of the individual simulated birds.

– Craig W. Reynolds, "Flocks, Herds, and Schools: A Distributed Behavioral Model", *Computer Graphics* **21** *4* 1987, pp 25-34 See the original paper

- Collision Avoidance: avoid collisions with nearby flockmates
- Velocity Matching: attempt to match velocity with nearby flockmates
- Flock Centering: attempt to stay close to nearby flockmates

20.2 Setting up the Boids

Our boids will each have an x velocity and a y velocity, and an x position and a y position.

We'll build this up in NumPy notation, and eventually, have an animated simulation of our flying boids.

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

Let's start with simple flying in a straight line.

Our locations, for each of our N boids, will be an array, shape $2 \times N$, with the x positions in the first row, and y positions in the second row.

```
In [2]: boid_count = 5
```

We'll want to be able to seed our Boids in a random position.

We'd better define the edges of our simulation area:

```
In [3]: limits = np.array([2000,4000])
```

```
In [4]: np.random.rand(2, boid_count)
```

```
Out[4]: array([[ 0.37764087,  0.14746797,  0.61508551,  0.29696232,  0.56354435],
               [ 0.42454391,  0.23172286,  0.64167284,  0.96184223,  0.0308127 ]])
```

```
In [5]: limits.shape

Out[5]: (2,)

In [6]: positions = np.random.rand(2,boid_count)*limits[:,np.newaxis]
        positions

Out[6]: array([[ 1541.65854722,   462.71392074,   869.45149676,    42.01077647,
                 1477.32553263],
               [ 1352.75633102,    23.61351242,  1957.70764292,  3051.53270395,
                 1238.28124949]])
```

We used **broadcasting** with np.newaxis to apply our upper limit to each boid. `rand` gives us a random number between 0 and 1. We multiply by our limits to get a number up to that limit.

Let's put that in a function:

```
In [7]: def new_flock(count, lower_limits, upper_limits):
            width=upper_limits-lower_limits
            return (lower_limits[:,np.newaxis] +
                 np.random.rand(2, count)*width[:,np.newaxis])
```

But each bird will also need a starting velocity. Let's make these random too:

```
In [8]: velocities = new_flock(boid_count, np.array([0, -20]), np.array([10, 20]))
        velocities

Out[8]: array([[  3.69869418,   5.33047988,   6.72728785,   1.53601681,
                  5.82559298],
               [ 12.04268965,   3.94119964,   9.2196863 ,   4.70663959,
                 -8.00413431]])
```

20.3 Flying in a Straight Line

Now we see the real amazingness of NumPy: if we want to move our *whole flock* according to $\delta_x = \delta_t \cdot \frac{dv}{dt}$

We just do:

```
In [9]: delta_t = 1 # Arbitrary choice; defines unit of velocity
        positions += velocities * delta_t
```

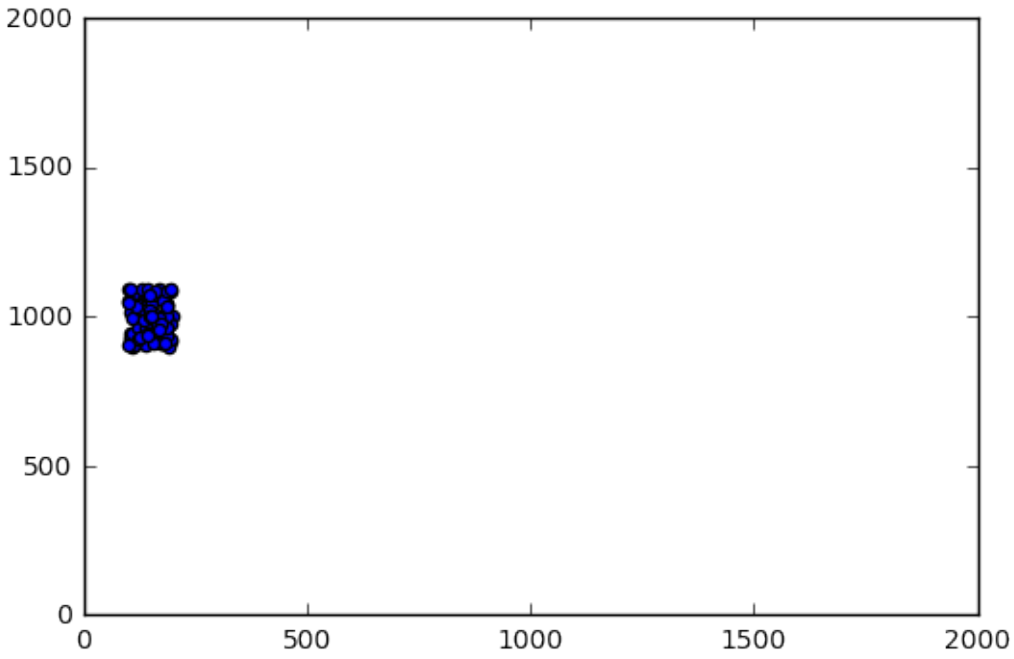
20.4 Matplotlib Animations

So now we can animate our Boids using the matplotlib animation tools. All we have to do is import the relevant libraries:

```
In [10]: from matplotlib import animation
         from matplotlib import pyplot as plt
         %matplotlib inline

/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

Then, we make a static plot, showing our first frame:

```
In [11]: # create a simple animation
         positions=new_flock(100, np.array([100,900]), np.array([200,1100]))
         velocities=new_flock(100, np.array([0,-20]), np.array([10,20]))

         figure = plt.figure()
         axes = plt.axes(xlim=(0, limits[0]), ylim=(0, limits[0]))
         scatter=axes.scatter(positions[0,:],positions[1,:])
         scatter

Out[11]: <matplotlib.collections.PathCollection at 0x2ab9c5edf2b0>
```



Then, we define a function which **updates** the figure for each timestep

```
In [12]: def update_boids(positions, velocities):
             positions += velocities

         def animate(frame):
             update_boids(positions, velocities)
             scatter.set_offsets(positions.transpose())
```

Call `FuncAnimation`, and specify how many frames we want:

```
In [13]: anim=animation.FuncAnimation(figure, animate,
                                      frames=50, interval=50)
```

Save out the figure:

```
In [14]: anim.save('boids_1.mp4')
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/animation.py:784
  warnings.warn("MovieWriter %s unavailable" % writer)


        ---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call last)

        <ipython-input-14-8c77dee640ac> in <module>()
    ----> 1 anim.save('boids_1.mp4')


        /home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/animatio
        836                         # TODO: See if turning off blit is really necessary
        837                         anim._draw_next_frame(d, blit=False)
    --> 838                     writer.grab_frame(**savefig_kwargs)
        839
        840         # Reconnect signal for first draw if necessary


        /opt/python/3.5.0/lib/python3.5/contextlib.py in __exit__(self, type, value, traceb
         64         if type is None:
         65             try:
    ---> 66                 next(self.gen)
         67             except StopIteration:
         68                 return


        /home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/animatio
        200         self.setup(*args)
        201         yield
    --> 202         self.finish()
        203
        204     def _run(self):


        /home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/animatio
        400             raise RuntimeError('Error creating movie, return code: '
        401                                + str(self._proc.returncode)
    --> 402                                + ' Try running with --verbose-debug')
        403
        404     def cleanup(self):


        RuntimeError: Error creating movie, return code: 1 Try running with --verbose-debug
```

And download the saved animation

You can even use an external library to view the results directly in the notebook. If you're on your own computer, you can download it from https://github.com/jakevdp/JSAnimation.git (See the notes on installing libraries...)

Unfortunately, if you're on the teaching cluster, you won't be able to install it there.

```
In [15]: from JSAnimation import IPython_display
         # Inline animation tool; needs manual install via
```

```
# If you don't have this, you need to save animations as MP4.
positions=new_flock(100, np.array([100,900]), np.array([200,1100]))
anim
```

```
Out[15]: <matplotlib.animation.FuncAnimation at 0x2ab9c5e72ef0>
```

20.5 Fly towards the middle

Boids try to fly towards the middle:

```
In [16]: positions=new_flock(4, np.array([100,900]), np.array([200,1100]))
         velocities=new_flock(4, np.array([0,-20]), np.array([10,20]))

In [17]: positions

Out[17]: array([[  195.37867361,   100.61410233,   194.86030285,   141.17154861],
                [  951.74480973,  1033.22391927,  1091.73221827,   934.78489654]])

In [18]: velocities

Out[18]: array([[  7.70543161,   8.48639093,   1.47765243,   3.83338279],
                [  9.47307994,  -7.60489275,  -7.98393739,  18.29932464]])

In [19]: middle=np.mean(positions, 1)
         middle

Out[19]: array([  158.00615685,  1002.87146095])

In [20]: direction_to_middle = positions-middle[:, np.newaxis]

In [21]: move_to_middle_strength = 0.01
         velocities = velocities - direction_to_middle * move_to_middle_strength
```

Let's update our function, and animate that:

```
In [22]: def update_boids(positions, velocities):
             move_to_middle_strength = 0.1
             middle=np.mean(positions, 1)
             direction_to_middle = positions - middle[:, np.newaxis]
             velocities -= direction_to_middle * move_to_middle_strength
             positions += velocities

In [23]: def animate(frame):
             update_boids(positions, velocities)
             scatter.set_offsets(positions.transpose())

In [24]: anim=animation.FuncAnimation(figure, animate,
                                      frames=50, interval=50)

In [25]: from JSAnimation import IPython_display
         positions=new_flock(100, np.array([100,900]), np.array([200,1100]))
         velocities=new_flock(100, np.array([0,-20]), np.array([10,20]))
         anim

Out[25]: <matplotlib.animation.FuncAnimation at 0x2ab9c5f98198>
```

20.6 Avoiding collisions

We'll want to add our other flocking rules to the behaviour of the Boids.

We'll need a matrix giving the distances between each bird. This should be $N \times N$.

```
In [26]: positions=new_flock(4, np.array([100,900]), np.array([200,1100]))
         velocities=new_flock(4, np.array([0,-20]), np.array([10,20]))
```

We might think that we need to do the X-distances and Y-distances separately:

```
In [27]: xpos=positions[0,:]

In [28]: xpos

Out[28]: array([ 161.54568578,  137.22329823,  191.20196978,  157.33367923])

In [29]: xpos[:,np.newaxis]

Out[29]: array([[ 161.54568578],
                [ 137.22329823],
                [ 191.20196978],
                [ 157.33367923]])

In [30]: xpos[np.newaxis,:]

Out[30]: array([[ 161.54568578,  137.22329823,  191.20196978,  157.33367923]])

In [31]: xsep_matrix = xpos[:,np.newaxis] - xpos[np.newaxis,:]

In [32]: xsep_matrix

Out[32]: array([[  0.        ,  24.32238755, -29.656284  ,   4.21200655],
                [-24.32238755,   0.        , -53.97867155, -20.110381  ],
                [ 29.656284  ,  53.97867155,   0.        ,  33.86829055],
                [ -4.21200655,  20.110381  , -33.86829055,   0.        ]])

In [33]: xsep_matrix.shape

Out[33]: (4, 4)

In [34]: xsep_matrix

Out[34]: array([[  0.        ,  24.32238755, -29.656284  ,   4.21200655],
                [-24.32238755,   0.        , -53.97867155, -20.110381  ],
                [ 29.656284  ,  53.97867155,   0.        ,  33.86829055],
                [ -4.21200655,  20.110381  , -33.86829055,   0.        ]])
```

But in NumPy we can be cleverer than that, and make a 2 by N by N matrix of separations:

```
In [35]: separations = positions[:,np.newaxis,:] - positions[:,:,np.newaxis]

In [36]: separations.shape

Out[36]: (2, 4, 4)
```

And then we can get the sum-of-squares $\delta_x^2 + \delta_y^2$ like this:

```
In [37]: squared_displacements = separations * separations
```

```
In [38]: square_distances = np.sum(squared_displacements, 0)

In [39]: square_distances

Out[39]: array([[    0.        ,  6373.15962006,  4228.30049314,  9744.65378653],
                [ 6373.15962006,     0.        ,  3243.76760094,   914.66339974],
                [ 4228.30049314,  3243.76760094,     0.        ,  2808.13296105],
                [ 9744.65378653,   914.66339974,  2808.13296105,     0.        ]])
```

Now we need to find birds that are too close:

```
In [40]: alert_distance = 2000
         close_birds = square_distances < alert_distance
         close_birds

Out[40]: array([[ True, False, False, False],
                [False,  True, False,  True],
                [False, False,  True, False],
                [False,  True, False,  True]], dtype=bool)
```

Find the direction distances **only** to those birds which are too close:

```
In [41]: separations_if_close = np.copy(separations)
         far_away = np.logical_not(close_birds)

In [42]: separations_if_close[0,:,:][far_away] = 0
         separations_if_close[1,:,:][far_away] = 0
         separations_if_close

Out[42]: array([[[  0.        ,   0.        ,   0.        ,   0.        ],
                 [  0.        ,   0.        ,   0.        ,  20.110381  ],
                 [  0.        ,   0.        ,   0.        ,   0.        ],
                 [  0.        , -20.110381  ,   0.        ,   0.        ]],

                [[  0.        ,   0.        ,   0.        ,   0.        ],
                 [  0.        ,   0.        ,   0.        ,  22.58840357],
                 [  0.        ,   0.        ,   0.        ,   0.        ],
                 [  0.        , -22.58840357,   0.        ,   0.        ]]])
```

And fly away from them:

```
In [43]: velocities = velocities + np.sum(separations_if_close,2)
```

Now we can update our animation:

```
In [44]: def update_boids(positions, velocities):
             move_to_middle_strength = 0.01
             middle = np.mean(positions, 1)
             direction_to_middle = positions - middle[:,np.newaxis]
             velocities -= direction_to_middle * move_to_middle_strength

             separations = positions[:,np.newaxis,:] - positions[:,:,np.newaxis]
             squared_displacements = separations * separations
             square_distances = np.sum(squared_displacements, 0)
             alert_distance = 100
             far_away = square_distances > alert_distance
             separations_if_close = np.copy(separations)
```

```
            separations_if_close[0,:,:][far_away] =0
            separations_if_close[1,:,:][far_away] =0
            velocities += np.sum(separations_if_close,1)

            positions += velocities
```

```
In [45]: def animate(frame):
            update_boids(positions, velocities)
            scatter.set_offsets(positions.transpose())

        anim = animation.FuncAnimation(figure, animate,
                                       frames=50, interval=50)

        from JSAnimation import IPython_display
        positions = new_flock(100, np.array([100,900]), np.array([200,1100]))
        velocities = new_flock(100, np.array([0,-20]), np.array([10,20]))
        anim
```

```
Out[45]: <matplotlib.animation.FuncAnimation at 0x2ab9c5fa49e8>
```

20.7 Match speed with nearby birds

This is pretty similar:

```
In [46]: def update_boids(positions, velocities):
            move_to_middle_strength = 0.01
            middle = np.mean(positions, 1)
            direction_to_middle = positions-middle[:,np.newaxis]
            velocities -= direction_to_middle*move_to_middle_strength

            separations = positions[:,np.newaxis,:] - positions[:,:,np.newaxis]
            squared_displacements = separations*separations
            square_distances = np.sum(squared_displacements, 0)
            alert_distance = 100
            far_away=square_distances > alert_distance
            separations_if_close = np.copy(separations)
            separations_if_close[0,:,:][far_away] =0
            separations_if_close[1,:,:][far_away] =0
            velocities += np.sum(separations_if_close,1)

            velocity_differences = velocities[:,np.newaxis,:] - velocities[:,:,np.newaxis]
            formation_flying_distance = 10000
            formation_flying_strength = 0.125
            very_far=square_distances > formation_flying_distance
            velocity_differences_if_close = np.copy(velocity_differences)
            velocity_differences_if_close[0,:,:][very_far] =0
            velocity_differences_if_close[1,:,:][very_far] =0
            velocities -= np.mean(velocity_differences_if_close, 1) * formation_flying_st

            positions += velocities
```

```
In [47]: def animate(frame):
            update_boids(positions, velocities)
```

```
    scatter.set_offsets(positions.transpose())

anim=animation.FuncAnimation(figure, animate,
                             frames=200, interval=50)

from JSAnimation import IPython_display

positions=new_flock(100, np.array([100,900]), np.array([200,1100]))
velocities=new_flock(100, np.array([0,-20]), np.array([10,20]))
anim
```

```
Out[47]: <matplotlib.animation.FuncAnimation at 0x2ab9c5faeba8>
```

Hopefully the power of NumPy should be pretty clear now. This would be **enormously slower** and, I think, harder to understand using traditional lists.

Chapter 21

Installing Libraries

21.1 Installing Libraries

We've seen that PyPI carries lots of python libraries. But how do we install them?

In the UCL teaching clusters, we cannot, as these need to be installed by administrators.

So you'll need to follow this lesson on a computer of your own.

The main problem is this: *libraries need other libraries*

So you can't just install a library by copying code to the computer: you'll find yourself wandering down a tree of "dependencies"; libraries needed by libraries needed by the library you want.

This is actually a good thing; it means that people are making use of each others' code. There's a real problem in scientific programming, of people who think they're really clever writing their own twenty-fifth version of the same thing.

So using other people's libraries is good.

Why don't we do it more? Because it can often be quite difficult to **install** other peoples' libraries!

Python has developed a good tool for avoiding this: **pip**.

21.2 Installing Geopy using Pip

On a computer you control, on which you have installed python via Anaconda, you will need to open a **terminal** to invoke the library-installer program, `pip`.

- On windows, go to start->all programs->Anaconda->Anaconda Command Prompt
- On mac, start *terminal*.
- On linux, open a bash shell.

Into this shell, type:

```
pip install geopy
```

The computer will install the package automatically from PyPI.

Now, close IPython notebook if you have it open, and reopen it. Check your new library is installed with:

```
In [1]: import geopy
        geocoder = geopy.geocoders.GoogleV3(domain="maps.google.co.uk")
        geocoder.geocode('Cambridge',exactly_one=False)

Out[1]: [Location(Cambridge, UK, (52.205337, 0.121817, 0.0)),
         Location(Cambridge, Gloucester GL2, UK, (51.73193, -2.3649, 0.0))]
```

That was actually pretty easy, I hope. This is how you'll install new libraries when you need them.

Troubleshooting:

On mac or linux, you *might* get a complaint that you need "superuser", "root", or "administrator" access. If so type:

- `sudo pip install geopy`

and enter your password.
If you get a complaint like: 'pip is not recognized as an internal or external command', try the following:

- `conda install pip` (Windows)
- `sudo easy_install pip` (Mac, Linux)

Ask me over email if you run into trouble.

21.3 Installing binary dependencies with Conda

`pip` is the usual Python tool for installing libraries. But there's one area of library installation that is still awkward: some python libraries depend not on other **python** libraries, but on libraries in C++ or Fortran.

This can cause you to run into difficulties installing some libraries. Fortunately, for lots of these, Continuum, the makers of Anaconda, provide a carefully managed set of scripts for installing these awkward non-python libraries too. You can do this with the `conda` command line tool, if you're using Anaconda.

Simply type

- `conda install <whatever>`

instead of `pip install`. This will fetch the python package not from PyPI, but from Anaconda's distribution for your platform, and manage any non-python dependencies too.

Typically, if you're using Anaconda, whenever you come across a python package you want, you should check if Anaconda package it first using this list: http://docs.continuum.io/anaconda/pkg-docs.html. (Or just by trying `conda install` and hoping!) If you can `conda install` it, you'll likely have less problems. But Continuum don't package everything, so you'll need to `pip install` from time to time.

21.4 Where do these libraries go?

```
In [2]: import numpy

In [3]: numpy.__path__

Out[3]: ['/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/numpy']
```

Your computer will be configured to keep installed Python packages in a particular place.

Python knows where to look for possible library installations in a list of places, called the "PythonPath". It will try each of these places in turn, until it finds a matching library name.

```
In [4]: import sys
        sys.path[-4:] # Just list the last few

Out[4]: ['/opt/python/3.5.0/lib/python3.5/plat-linux',
         '/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages',
         '/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/IPython/extensio
         '/home/travis/.ipython']
```

21.5 Libraries not in PyPI

Sometimes, such as for the Animation library in the Boids example, you'll need to download the source code directly. This won't automatically follow the dependency tree, but for simple standalone libraries, is sometimes necessary.

To install these on windows, download and unzip the library into a folder of your choice, e.g. `my_python_libs`.

On windows, a reasonable choice is the folder you end up in when you open the Anaconda terminal. You can get a graphical view on this folder by typing: `explorer .`

Make a new folder for your download and unzip the library there.

Now, you need to move so you're inside your download in the terminal:

- `cd my_python_libs`
- `cd <library name>` (e.g. `cd JSAnimation-master`)

Now, manually install the library in your PythonPath:

- `python setup.py install`

[You might need to do

- `sudo python setup.py install`

if you get prompted for 'root' or 'admin' access.]

This is all pretty awkward, but it is worth practicing this stuff, as most of the power of using programming for research resides in all the libraries that are out there.

Finally, writing your own libraries is beyond the scope of this course, but we cover it in Our More Advanced Python Course

Chapter 22

Defining your own classes

22.1 User Defined Types

A **class** is a user-programmed Python type.
It is defined like this:

```
In [1]: class Room(object):
            pass
```

Just as with other python types, you use the name of the type as a function to make a variable of that type:

```
In [2]: zero = int()

In [3]: zero

Out[3]: 0

In [4]: print(type(zero))

<class 'int'>


In [5]: myroom = Room()

In [6]: myroom

Out[6]: <__main__.Room at 0x2b71b2a9b668>

In [7]: print(type(myroom))

<class '__main__.Room'>
```

In the jargon, we say that an **object** is an **instance** of a particular **class**.
Once we have an object with a type of our own devising, we can add properties at will:

```
In [8]: myroom.name = "Living"

In [9]: print(myroom.name)

Living
```

The most common use of a class is to allow us to group data into an object in a way that is easier to read and understand than organising data into lists and dictionaries.

```
In [10]: myroom.capacity = 3
         myroom.occupants = ["James", "Sue"]

In [11]: mystring = "Hello"
         mystring.upper()

Out[11]: 'HELLO'

In [12]: x = "Hello"
         x = 'Hello'

In [13]: y = "James's String"

In [14]: z = "7'"+'2"'

In [15]: z

Out[15]: '7\'2"'
```

22.2 Methods

So far, our class doesn't do much!

We define functions **inside** the definition of a class, in order to give them capabilities, just like the methods on built-in types.

```
In [16]: class Room(object):
             def overfull(self):
                 return len(self.occupants) > self.capacity

In [17]: myroom = Room()
         myroom.capacity = 3
         myroom.occupants = ["James", "Sue"]

In [18]: myroom.overfull()

Out[18]: False

In [19]: myroom.occupants.append(['Clare'])

In [20]: myroom.occupants.append(['Bob'])

In [21]: myroom.overfull()

Out[21]: True
```

When we write methods, we always write the first function argument as `self`, to refer to the object instance itself, the argument that goes "before the dot".

22.3 Constructors

Normally, though, we don't want to add data to the class attributes on the fly like that. Instead, we define a **constructor** that converts input data into an object.

```
In [22]: class Room(object):
             def __init__(self, name, exits, capacity, occupants=[]):
                 self.name = name
                 self.occupants = occupants # Note the default argument, occupants start em
                 self.exits = exits
                 self.capacity = capacity
             def overfull(self):
                 return len(self.occupants) > self.capacity

In [23]: living = Room("Living Room", {'north': 'garden'}, 3)

In [24]: living.capacity

Out[24]: 3
```

Methods which begin and end with **two underscores** in their names fulfil special capabilities in Python, such as constructors.

22.4 Object-oriented design

In building a computer system to model a problem, therefore, we often want to make:

- classes for each *kind of thing* in our system
- methods for each *capability* of that kind
- properties (defined in a constructor) for each *piece of information describing* that kind

For example, the below program might describe our "Maze of Rooms" system:
We define a "Maze" class which can hold rooms:

```
In [25]: class Maze(object):
             def __init__(self, name):
                 self.name = name
                 self.rooms = {}

             def add_room(self, room):
                 room.maze = self # The Room needs to know which Maze it is a part of
                 self.rooms[room.name] = room

             def occupants(self):
                 return  [occupant for room in self.rooms.values()
                          for occupant in room.occupants.values()]

             def wander(self):
                 "Move all the people in a random direction"
                 for occupant in self.occupants():
                     occupant.wander()

             def describe(self):
                 for room in self.rooms.values():
                     room.describe()
```

```
            def step(self):
                house.describe()
                print
                house.wander()
                print

            def simulate(self, steps):
                for _ in range(steps):
                    self.step()
```

And a "Room" class with exits, and people:

```
In [26]: class Room(object):
            def __init__(self, name, exits, capacity, maze = None):
                self.maze = maze
                self.name = name
                self.occupants = {} # Note the default argument, occupants start empty
                self.exits = exits # Should be a dictionary from directions to room names
                self.capacity = capacity

            def has_space(self):
                return len(self.occupants) < self.capacity

            def available_exits(self):
                return [exit for exit, target in self.exits.items()
                        if self.maze.rooms[target].has_space() ]

            def random_valid_exit(self):
                import random
                if not self.available_exits():
                    return None
                return random.choice(self.available_exits())

            def destination(self, exit):
                return self.maze.rooms[ self.exits[exit] ]

            def add_occupant(self, occupant):
                occupant.room = self # The person needs to know which room it is in
                self.occupants[occupant.name] = occupant

            def delete_occupant(self, occupant):
                del self.occupants[occupant.name]

            def describe(self):
                if self.occupants:
                    print(self.name, ": ", " ".join(self.occupants.keys()))
```

We define a "Person" class for room occupants:

```
In [27]: class Person(object):
            def __init__(self, name, room = None):
                self.name=name

            def use(self, exit):
```

```
            self.room.delete_occupant(self)
            destination=self.room.destination(exit)
            destination.add_occupant(self)
            print(self.name, "goes", exit, "to the", destination.name)

        def wander(self):
            exit = self.room.random_valid_exit()
            if exit:
                self.use(exit)
```

And we use these classes to define our people, rooms, and their relationships:

```
In [28]: james=Person('James')
         sue=Person('Sue')
         bob=Person('Bob')
         clare=Person('Clare')

In [29]: living=Room('livingroom', {'outside':'garden', 'upstairs':'bedroom', 'north':'kit
         kitchen=Room('kitchen', {'south':'livingroom'}, 1)
         garden=Room('garden', {'inside':'livingroom'}, 3)
         bedroom=Room('bedroom', {'jump':'garden', 'downstairs': 'livingroom'}, 1)

In [30]: house=Maze('My House')

In [31]: for room in [living, kitchen, garden, bedroom]:
             house.add_room(room)

In [32]: living.add_occupant(james)

In [33]: garden.add_occupant(sue)
         garden.add_occupant(clare)

In [34]: bedroom.add_occupant(bob)
```

And we can run a "simulation" of our model:

```
In [35]: house.simulate(3)
```

```
livingroom :  James
garden :  Clare Sue
bedroom :  Bob
James goes outside to the garden
Clare goes inside to the livingroom
Sue goes inside to the livingroom
Bob goes jump to the garden
livingroom :  Clare Sue
garden :  James Bob
Clare goes outside to the garden
Sue goes north to the kitchen
James goes inside to the livingroom
Bob goes inside to the livingroom
livingroom :  James Bob
kitchen :  Sue
garden :  Clare
James goes outside to the garden
Bob goes outside to the garden
Sue goes south to the livingroom
Clare goes inside to the livingroom
```

22.5 Object oriented design

There are many choices for how to design programs to do this. Another choice would be to separately define exits as a different class from rooms. This way, we can use arrays instead of dictionaries, but we have to first define all our rooms, then define all our exits.

```
In [36]: class Maze(object):
             def __init__(self, name):
                 self.name = name
                 self.rooms = []
                 self.occupants = []

             def add_room(self, name, capacity):
                 result = Room(name, capacity)
                 self.rooms.append(result)
                 return result

             def add_exit(self, name, source, target, reverse= None):
                 source.add_exit(name, target)
                 if reverse:
                     target.add_exit(reverse, source)

             def add_occupant(self, name, room):
                 self.occupants.append(Person(name, room))
                 room.occupancy += 1

             def wander(self):
                 "Move all the people in a random direction"
                 for occupant in self.occupants:
                     occupant.wander()

             def describe(self):
                 for occupant in self.occupants:
                     occupant.describe()

             def step(self):
                 house.describe()
                 print()
                 house.wander()
                 print()

             def simulate(self, steps):
                 for _ in range(steps):
                     self.step()
```

```
In [37]: class Room(object):
             def __init__(self, name, capacity):
                 self.name = name
                 self.capacity = capacity
                 self.occupancy = 0
                 self.exits = []

             def has_space(self):
                 return self.occupancy < self.capacity
```

```
            def available_exits(self):
                return [exit for exit in self.exits if exit.valid() ]

            def random_valid_exit(self):
                import random
                if not self.available_exits():
                    return None
                return random.choice(self.available_exits())

            def add_exit(self, name, target):
                self.exits.append(Exit(name, target))
```

```
In [38]: class Person(object):
            def __init__(self, name, room = None):
                self.name=name
                self.room=room

            def use(self, exit):
                self.room.occupancy -= 1
                destination=exit.target
                destination.occupancy +=1
                self.room=destination
                print(self.name, "goes", exit.name, "to the", destination.name)

            def wander(self):
                exit = self.room.random_valid_exit()
                if exit:
                    self.use(exit)

            def describe(self):
                print(self.name, "is in the", self.room.name)
```

```
In [39]: class Exit(object):
            def __init__(self, name, target):
                self.name = name
                self.target = target

            def valid(self):
                return self.target.has_space()
```

```
In [40]: house=Maze('My New House')
```

```
In [41]: living=house.add_room('livingroom', 2)
         bed = house.add_room('bedroom', 1)
         garden = house.add_room('garden', 3)
         kitchen = house.add_room('kitchen', 1)
```

```
In [42]: house.add_exit('north', living, kitchen, 'south')
```

```
In [43]: house.add_exit('upstairs', living, bed, 'downstairs')
```

```
In [44]: house.add_exit('outside', living, garden, 'inside')
```

```
In [45]: house.add_exit('jump',bed, garden)
```

```
In [46]: house.add_occupant('James', living)
         house.add_occupant('Sue', garden)
         house.add_occupant('Bob', bed)
         house.add_occupant('Clare', garden)

In [47]: house.simulate(3)

James is in the livingroom
Sue is in the garden
Bob is in the bedroom
Clare is in the garden

James goes outside to the garden
Sue goes inside to the livingroom
Bob goes jump to the garden
Clare goes inside to the livingroom

James is in the garden
Sue is in the livingroom
Bob is in the garden
Clare is in the livingroom

Sue goes upstairs to the bedroom
Bob goes inside to the livingroom
Clare goes outside to the garden

James is in the garden
Sue is in the bedroom
Bob is in the livingroom
Clare is in the garden

James goes inside to the livingroom
Sue goes jump to the garden
Bob goes north to the kitchen
Clare goes inside to the livingroom
```

This is a huge topic, about which many books have been written. The differences between these two designs are important, and will have long-term consequences for the project. That is the how we start to think about **software engineering**, as opposed to learning to program, and is where this course ends, and future courses begin!

22.6 Exercise: Your own solution

Compare the two solutions above. Discuss with a partner which you like better, and why. Then, starting from scratch, design your own. What choices did you make that are different from mine?

Chapter 23

Writing your Own Libraries

We will often want to save our Python classes, for use in multiple Notebooks. We can do this by writing text files with a .py extension, and then `importing` them.

23.1 Writing Python in Text Files

You can use a text editor like Atom for Mac or Notepad++ for windows to do this. If you create your own Python files ending in .py, then you can import them with `import` just like external libraries.

You can also maintain your library code in a Notebook, and use %%writefile to create your library.

Libraries are usually structured with multiple files, one for each class.

We group our modules into packages, by putting them together into a folder. You can do this with explorer, or using a shell, or even with Python:

```
In [1]: import os
        if 'mazetool' not in os.listdir(os.getcwd()):
            os.mkdir('mazetool')

In [2]: %%writefile mazetool/maze.py
        from .room import Room
        from .person import Person

        class Maze(object):
            def __init__(self, name):
                self.name = name
                self.rooms = []
                self.occupants = []

            def add_room(self, name, capacity):
                result = Room(name, capacity)
                self.rooms.append(result)
                return result

            def add_exit(self, name, source, target, reverse= None):
                source.add_exit(name, target)
                if reverse:
                    target.add_exit(reverse, source)

            def add_occupant(self, name, room):
                self.occupants.append(Person(name, room))
                room.occupancy += 1
```

```
            def wander(self):
                "Move all the people in a random direction"
                for occupant in self.occupants:
                    occupant.wander()

            def describe(self):
                for occupant in self.occupants:
                    occupant.describe()

            def step(self):
                house.describe()
                print()
                house.wander()
                print()

            def simulate(self, steps):
                for _ in range(steps):
                    self.step()
```

```
Writing mazetool/maze.py
```

```
In [3]: %%writefile mazetool/room.py
        from .exit import Exit


        class Room(object):
            def __init__(self, name, capacity):
                self.name = name
                self.capacity = capacity
                self.occupancy = 0
                self.exits = []

            def has_space(self):
                return self.occupancy < self.capacity

            def available_exits(self):
                return [exit for exit in self.exits if exit.valid() ]

            def random_valid_exit(self):
                import random
                if not self.available_exits():
                    return None
                return random.choice(self.available_exits())

            def add_exit(self, name, target):
                self.exits.append(Exit(name, target))
```

```
Writing mazetool/room.py
```

```
In [4]: %%writefile mazetool/person.py
```

```
        class Person(object):
            def __init__(self, name, room = None):
                self.name=name
                self.room=room

            def use(self, exit):
                self.room.occupancy -= 1
                destination=exit.target
                destination.occupancy +=1
                self.room=destination
                print(self.name, "goes", exit.name, "to the", destination.name)

            def wander(self):
                exit = self.room.random_valid_exit()
                if exit:
                    self.use(exit)

            def describe(self):
                print(self.name, "is in the", self.room.name)
```

```
Writing mazetool/person.py
```

```
In [5]: %%writefile mazetool/exit.py

        class Exit(object):
            def __init__(self, name, target):
                self.name = name
                self.target = target

            def valid(self):
                return self.target.has_space()
```

```
Writing mazetool/exit.py
```

In order to tell Python that our "mazetool" folder is a Python package, we have to make a special file called `__init__.py`. If you import things in there, they are imported as part of the package:

```
In [6]: %%writefile mazetool/__init__.py
        from .maze import Maze
```

```
Writing mazetool/__init__.py
```

23.2 Loading Our Package

We just wrote the files, there is no "Maze" class in this notebook yet:

```
In [7]: myhouse = Maze()
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)
```

```
        <ipython-input-7-34937f5f90f1> in <module>()
    ----> 1 myhouse = Maze()


        NameError: name 'Maze' is not defined
```

But now, we can import Maze, (and the other files will get imported via the chained Import statements, starting from the `__init__.py` file.

```
In [8]: from mazetool import Maze

In [9]: house=Maze('My New House')
        living=house.add_room('livingroom', 2)
```

Note the files we have created are on the disk in the folder we made:

```
In [10]: import os

In [11]: os.listdir(os.path.join(os.getcwd(),'mazetool') )

Out[11]: ['room.py', 'exit.py', '__pycache__', 'maze.py', 'person.py', '__init__.py']
```

`.pyc` files are "Compiled" temporary python files that the system generates to speed things up. They'll be regenerated on the fly when your `.py` files change.

23.3 The Python Path

We want to `import` these from notebooks elsewhere on our computer: it would be a bad idea to keep all our Python work in one folder.

Supplementary material The best way to do this is to learn how to make our code into a proper module that we can install. That's beyond the scope of this course, but read about setuptools if you want to know more.

Alternatively, we can add a folder to the "Python Path", where python searches for modules:

```
In [12]: import sys
         for path in sys.path[::-1]:
             print(path)

/home/travis/.ipython
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/IPython/extensions
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages
/opt/python/3.5.0/lib/python3.5/plat-linux
/opt/python/3.5.0/lib/python3.5
/home/travis/virtualenv/python3.5.0/lib/python3.5/lib-dynload
/home/travis/virtualenv/python3.5.0/lib/python3.5/plat-linux
/home/travis/virtualenv/python3.5.0/lib/python3.5
/home/travis/virtualenv/python3.5.0/lib/python35.zip



In [13]: sys.path.append('/home/jamespjh/devel/libraries/python')

In [14]: print(sys.path[-1])
```

```
/home/jamespjh/devel/libraries/python
```

I've thus added a folder to the list of places searched. If you want to do this permanently, you should set the PYTHONPATH Environment Variable, which you can learn about in a shell course, or can read about online for your operating system.

Chapter 24

Understanding the "Greengraph" Example

We now know enough to understand everything we did in the initial example chapter on the "Greengraph". Go back to that part of the notes, and re-read the code.

Now, we can even write it up into a class, and save it as a module.

24.1 Classes for Greengraph

```
In [1]: %%bash
        mkdir -p greengraph # Create the folder for the module (on mac or linux)
```

```
In [2]: %%writefile greengraph/graph.py
        import numpy as np
        import geopy
        from .map import Map

        class Greengraph(object):
            def __init__(self, start, end):
                self.start=start
                self.end=end
                self.geocoder=geopy.geocoders.GoogleV3(domain="maps.google.co.uk")

            def geolocate(self, place):
                return self.geocoder.geocode(place, exactly_one=False)[0][1]

            def location_sequence(self, start,end,steps):
              lats = np.linspace(start[0], end[0], steps)
              longs = np.linspace(start[1],end[1], steps)
              return np.vstack([lats, longs]).transpose()

            def green_between(self, steps):
                return [Map(*location).count_green()
                        for location in self.location_sequence(
                            self.geolocate(self.start),
                            self.geolocate(self.end),
                            steps)]
```

```
Writing greengraph/graph.py
```

```
In [3]: %%writefile greengraph/map.py

        import numpy as np
        from io import BytesIO
        from matplotlib import image as img
        import requests

        class Map(object):
            def __init__(self, lat, long, satellite=True, zoom=10, size=(400,400), sensor=F
                base="http://maps.googleapis.com/maps/api/staticmap?"

                params = dict(
                    sensor= str(sensor).lower(),
                    zoom= zoom,
                    size= "x".join(map(str, size)),
                    center= ",".join(map(str, (lat, long) )),
                    style="feature:all|element:labels|visibility:off"
                )

                if satellite:
                    params["maptype"]="satellite"

                self.image = requests.get(base, params=params).content # Fetch our PNG imag
                self.pixels = img.imread(BytesIO(self.image))

            def green(self, threshold):
                # Use NumPy to build an element-by-element logical array
                greener_than_red = self.pixels[:,:,1] > threshold* self.pixels[:,:,0]
                greener_than_blue = self.pixels[:,:,1] > threshold*self.pixels[:,:,2]
                green = np.logical_and(greener_than_red, greener_than_blue)
                return green

            def count_green(self, threshold = 1.1):
                return np.sum(self.green(threshold))

            def show_green(data, threshold = 1.1):
                green = self.green(threshold)
                out = green[:,:,np.newaxis]*array([0,1,0])[np.newaxis,np.newaxis,:]
                buffer = BytesIO()
                result = img.imsave(buffer, out, format='png')
                return buffer.getvalue()

Writing greengraph/map.py


In [4]: %%writefile greengraph/__init__.py
        from .graph import Greengraph

Writing greengraph/__init__.py
```

24.2 Invoking our code and making a plot

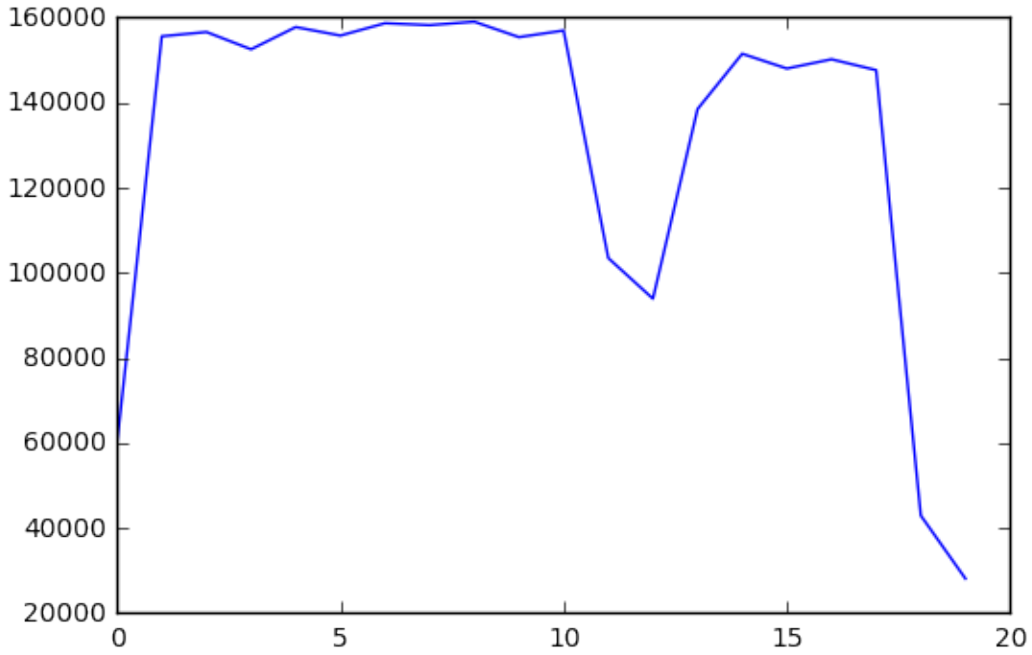
```
In [5]: %matplotlib inline
        from matplotlib import pyplot as plt
```

```
from greengraph import Greengraph


mygraph = Greengraph('New York','Chicago')
data = mygraph.green_between(20)
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```

In [6]:
```
plt.plot(data)
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x2b1b90e1d518>]
```



In [1]:
```
from IPython.display import HTML


HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<form action="javascript:code_toggle()"><input type="submit" value="Click here to s
```

```
Out[1]: <IPython.core.display.HTML object>
```

One of the least effective way to figure out π is to use a Monte-Carlo sampling method.

We will sample points randomly in the interval $[-\frac{1}{2}, \frac{1}{2}]^2$ (2D). The ratio of points that fall in a circle of radius $R = \frac{1}{2}$ vs those in the square $[-\frac{1}{2}, \frac{1}{2}]^2$ is equal to $\frac{\pi R^2}{1} = \frac{1}{4}\pi$

Implement the following algorithm:

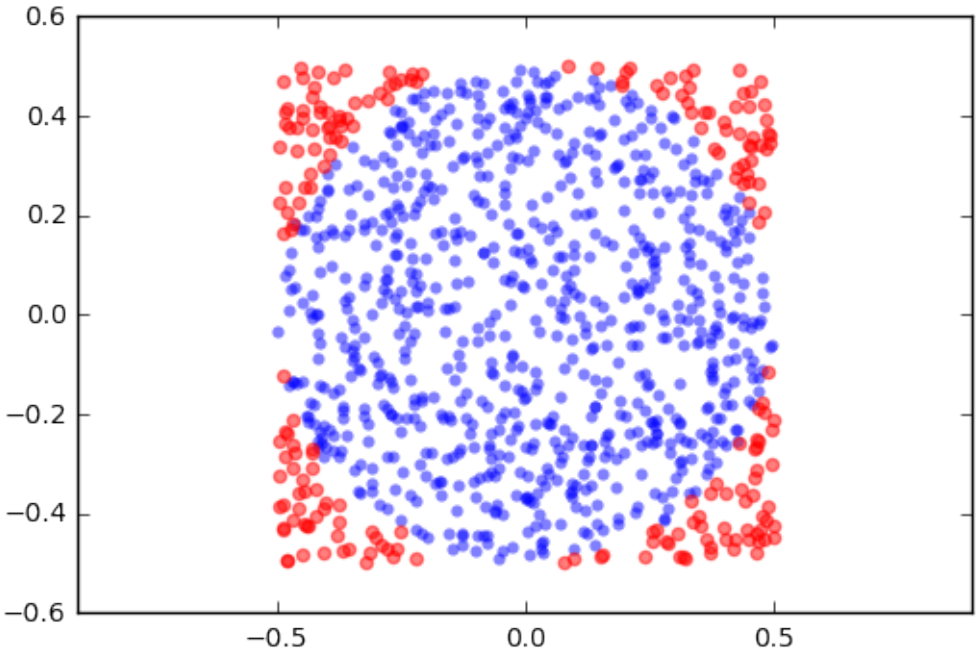
- loop for n iterations
- pick two numbers randomly between $-\frac{1}{2}$ and $\frac{1}{2}$
- if the numbers fall inside the circle then add it to list `in_circle` That means $x^2 + y^2 \leq R^2$
- otherwise add it to another `out_of_circle`
- Compute pi using the lengths of the two lists, e.g $\frac{4n_0}{n_0+n_1}$, with n_0 the length `in_circle` and n_1 the length of `out_of_circle`.
- plot the two sets of points in different colors

```
In [2]: from numpy.random import random

        N = 1000
        in_circle, out_circle = [], []
        radius = 0.5

        def add_sample(in_circle, out_circle, radius):
            point = random() - 0.5, random() - 0.5
            if point[0] * point[0] + point[1] * point[1] <= radius * radius:
                in_circle.append(point)
            else:
                out_circle.append(point)

        for i in range(N):
            add_sample(in_circle, out_circle, radius)

        assert N == len(in_circle) + len(out_circle)
        print("Pi is PI? ", 4 * float(len(in_circle)) / N)
```

```
Pi is PI?  3.156
```

```
In [3]: %matplotlib inline
        from matplotlib import pyplot as plt
        from numpy import array

        in_circle = array(in_circle)
        out_circle = array(out_circle)

        plt.scatter(in_circle[:, 0], in_circle[:, 1], linewidth=0, alpha=0.5)
        plt.scatter(out_circle[:, 0], out_circle[:, 1], alpha=0.5, color='red')
        plt.axis('equal')
        plt.show()
```

```
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
/home/travis/virtualenv/python3.5.0/lib/python3.5/site-packages/matplotlib/font_manager.py
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a momer
```



We could also use the numpy library to do the work for us:

```
In [4]: from numpy import random, sum
        N = 1000
        radius = 0.5
        all_points = random.random((N, 2)) - 0.5
        norms = sum(all_points * all_points, axis=1)
        n_in_circle = len(all_points[norms < radius * radius])

        print("Pi is ", 4 * float(n_in_circle) / len(all_points))
```

```
Pi is  3.12
```

Chapter 25

Compute Factorial N

```
In [5]: def factorial(N):
            if N == 1:
                return 1
            else:
                return N * factorial(N - 1)

In [6]: assert factorial(1) == 1
        assert factorial(2) == 2
        assert factorial(5) == 5 * 4 * 3 * 2 * 1
```

Chapter 26

Ceasar Cipher

https://en.wikipedia.org/wiki/Caesar_cipher

```
In [7]: def caesar_salad(input, n=13):
            alphabet = "abcdefghijklmnopqrstuvwxyz"
            rotabet = alphabet[n:] + alphabet[:n]

            result = ""
            for letter in input:
                index = alphabet.find(letter.lower())
                if index == -1:
                    result += letter
                elif letter.islower():
                    result += rotabet[index]
                else:
                    result += rotabet[index].upper()


            return result
```

```
In [8]: assert caesar_salad("caesar", n=0) == "caesar"
        assert caesar_salad("a", n = 1) == "b"
        assert caesar_salad("az", n = 1) == "ba"
        assert caesar_salad("AaZz", n = 1) == "BbAa"
        assert caesar_salad("Aa!Z z", n = 1) == "Bb!A a"
```

```
In [9]: caesar_salad(caesar_salad("Anything!?", n =12), n = 26 - 12)
```

```
Out[9]: 'Anything!?'
```