An introduction to Python Programming for Research

James Hetherington

February 2, 2017

Contents

Chapter 1

Introduction to Python

1.1 Introduction

1.1.1 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.

1.1.2 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.

1.1.3 Why write programs for research?

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

1.1.4 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

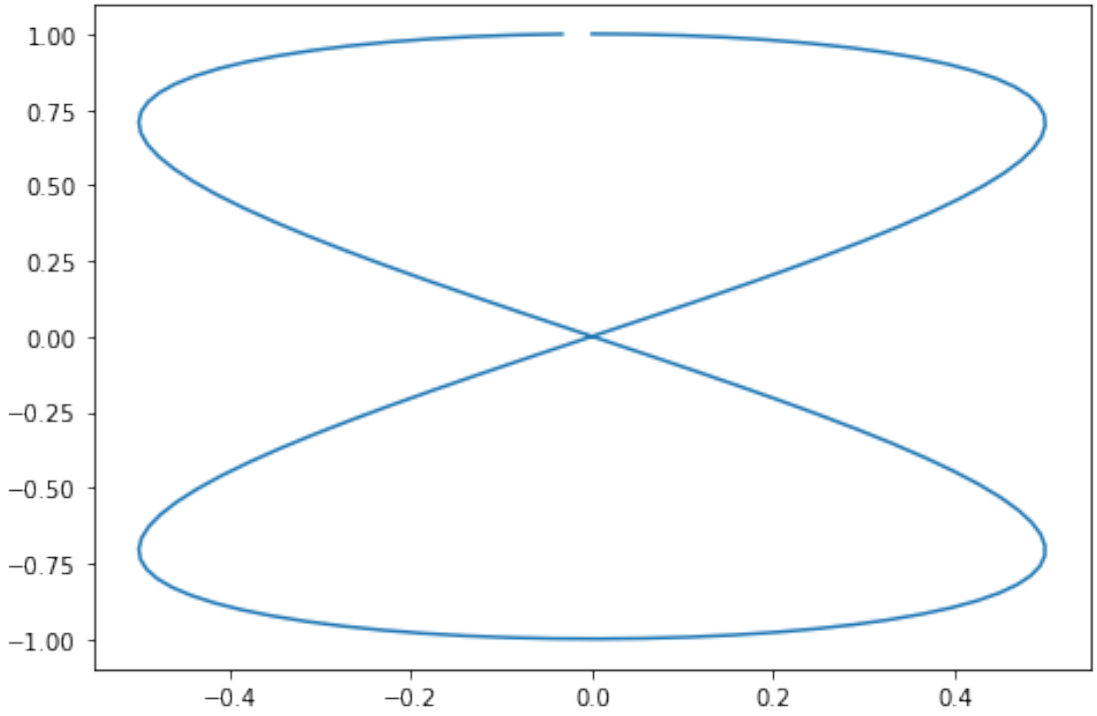
1.2.1 The Jupyter Notebook

The easiest way to get started using Python, and one of the best for research data work, is the Jupyter 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))
```

Out[1]:
```
[<matplotlib.lines.Line2D at 0x2b9cdd30e668>]
```



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]: %%bash
        echo "print(2*4)" > eight.py
        python eight.py
```

```
8
```

```
In [5]: %%bash
        echo '#!/usr/bin/env python' > eight
        echo "print(2*4)" >> eight
        chmod u+x eight
        ./eight
```

```
8
```

1.2.5 Python Libraries

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

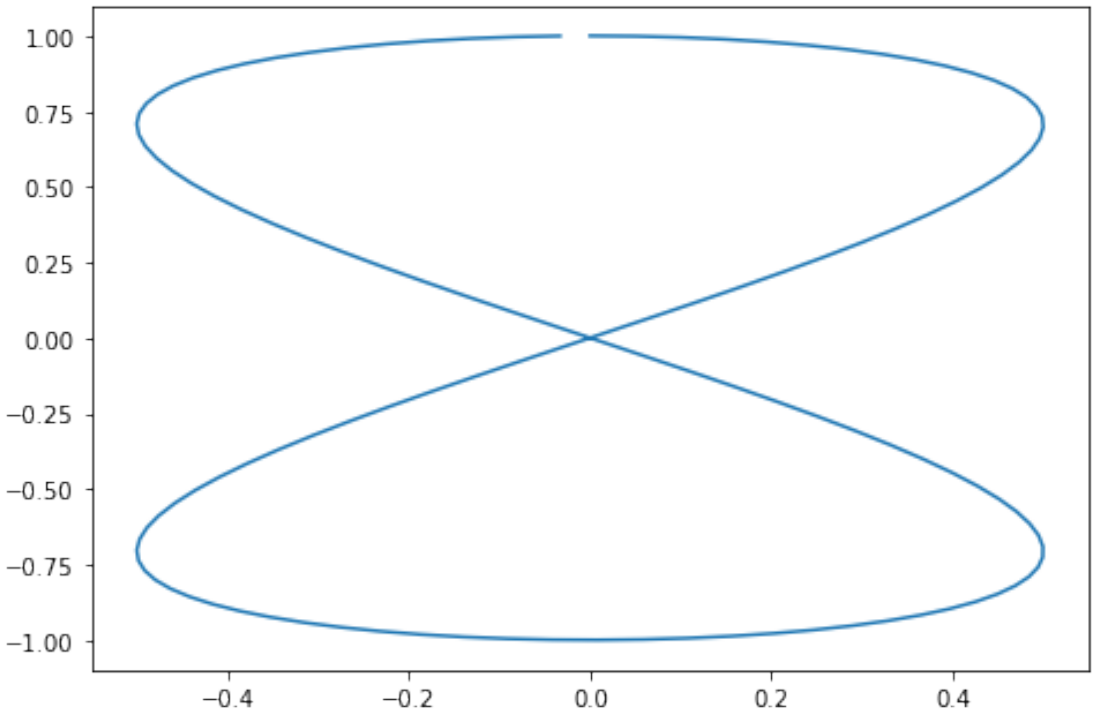
```
In [6]: %%writefile draw_eight.py
        # Above line tells the notebook to treat the rest of this
        # cell as content for a file on disk.

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

```
In [8]: image=draw_eight.make_figure()
```



1.3 An example Python data analysis notebook

1.3.1 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.

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 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.

1.3.2 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 [1]: 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 [2]: geocoder=geopy.geocoders.GoogleV3(domain="maps.google.co.uk")
        geocoder.geocode('Cambridge', exactly_one=False)

Out[2]: [Location(Cambridge, UK, (52.205337, 0.121817, 0.0))]
```

The results come out as a **list** inside a list: `[Name, [Latitude, Longitude]]`. Programs represent data in a variety of different containers like this.

1.3.3 Comments

Code after a # symbol doesn't get run.

```
In [3]: print("This runs") # print "This doesn't"
        # print This doesn't either

This runs
```

1.3.4 Functions

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

```
In [4]: 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 [5]: geolocate('Cambridge')

Out[5]: (52.205337, 0.121817)
```

1.3.5 Variables

We can store a result in a variable:

```
In [6]: london_location = geolocate("London")
        print(london_location)

(51.5073509, -0.1277583)
```

1.3.6 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 [7]: 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=dict(
            sensor= str(sensor).lower(),
            zoom= zoom,
            size= str(size[0])+"x"+str(size[1]),
            center = str(lat)+","+str(long),
            style="feature:all|element:labels|visibility:off"
          )
          if satellite:
            params["maptype"]="satellite"

          return requests.get(base,params=params)

In [8]: map_response=request_map_at(51.5072, -0.1275)
```

1.3.7 Checking our work

Let's see what URL we ended up with:

```
In [9]: url=map_response.url
        print(url[0:50])
        print(url[50:100])
        print(url[100:])
```

```
http://maps.googleapis.com/maps/api/staticmap?styl
e=feature%3Aall%7Celement%3Alabels%7Cvisibility%3A
off&sensor=false&size=400x400&center=51.5072%2C-0.1275&zoom=10&maptype=satellite
```

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

```
In [10]: from nose.tools import assert_in
         assert_in("http://maps.googleapis.com/maps/api/staticmap?", url)
         assert_in("center=51.5072%2C-0.1275", url)
         assert_in("zoom=10", url)
         assert_in("size=400x400", url)
         assert_in("sensor=false", 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 [11]: map_response.content[0:20]
```

```
Out[11]: b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR\x00\x00\x01\x90'
```

1.3.8 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 [12]: 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 [13]: import IPython
         map_png=map_at(*london_location)

In [14]: 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 [15]: IPython.core.display.Image(map_png)
```

Out[15]:


```
In [16]: IPython.core.display.Image(map_at(*geolocate("New Delhi")))
```

Out[16]:

1.3.9 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 [17]: from io import BytesIO # A library to convert between files and 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 [18]: 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 [19]: def count_green_in_png(data):
             f=BytesIO(data)
             pixels=img.imread( f ) # Get our PNG image as a numpy array
             return np.sum( is_green(pixels) )

In [20]: print(count_green_in_png( map_at(*london_location) ))

108030
```

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

```
In [21]: 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 [22]: location_sequence(geolocate("London"), geolocate("Cambridge"), 5)

Out[22]: 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]])
```

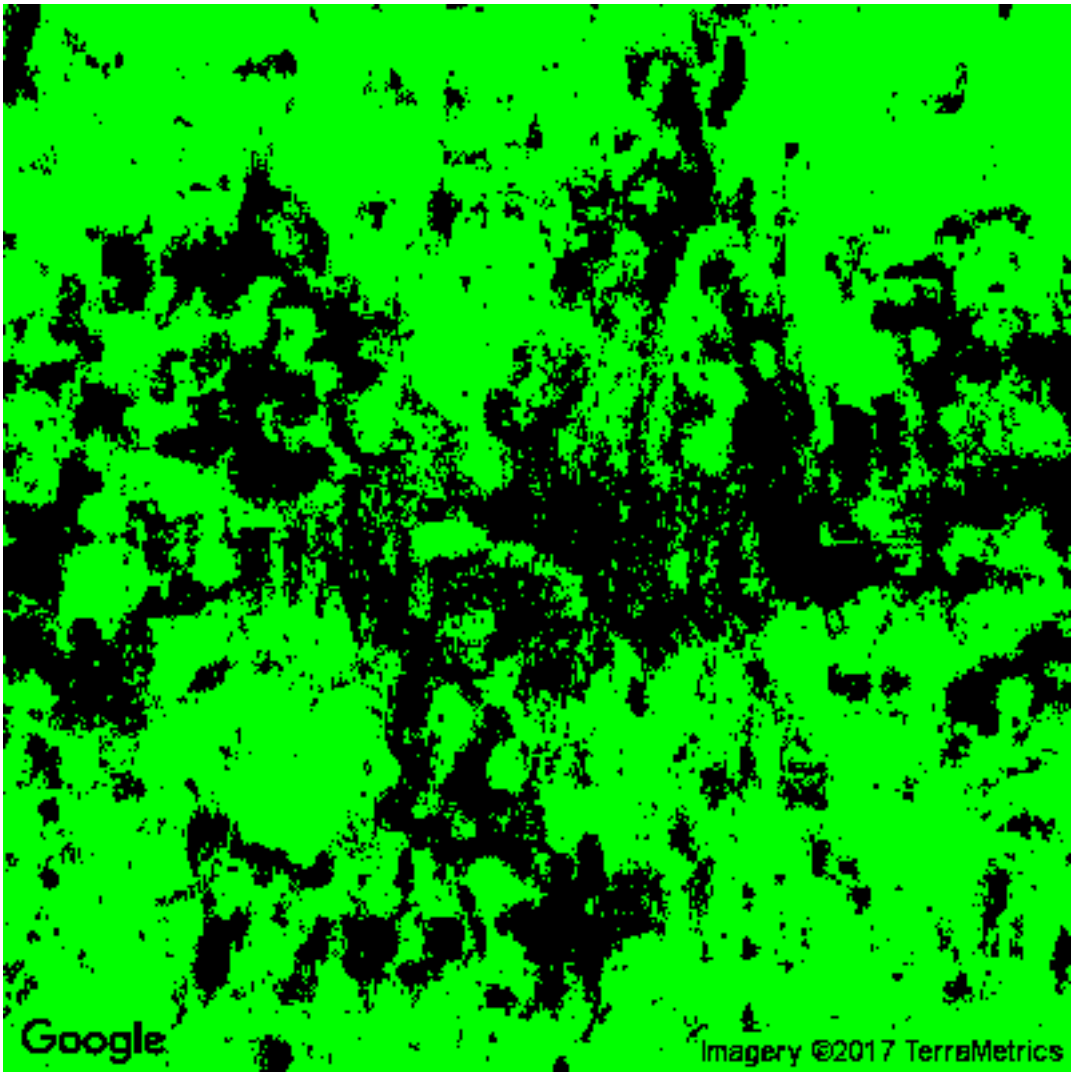
1.3.10 Creating Images

We should display the green content to check our work:

```
In [23]: 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 [24]: IPython.core.display.Image(
             map_at(*london_location, satellite=True)
         )

Out[24]:
```


```
In [25]: IPython.core.display.Image(
             show_green_in_png(
                 map_at(
                     *london_location,
                     satellite=True)))
```

Out[25]:



1.3.11 Looping

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

```
In [26]: for location in location_sequence(
                geolocate("London"), geolocate("Birmingham"),
                4):
             IPython.core.display.display(
                IPython.core.display.Image(map_at(*location)))
```


Google
Imagery ©2017 TerraMetrics


Google
Imagery ©2017 TerraMetrics


Google
Imagery ©2017 TerraMetrics

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

```
In [27]: [count_green_in_png(map_at(*location))
                      for location in
                           location_sequence(geolocate("London"),
                                             geolocate("Birmingham"),
                                             10)]
```

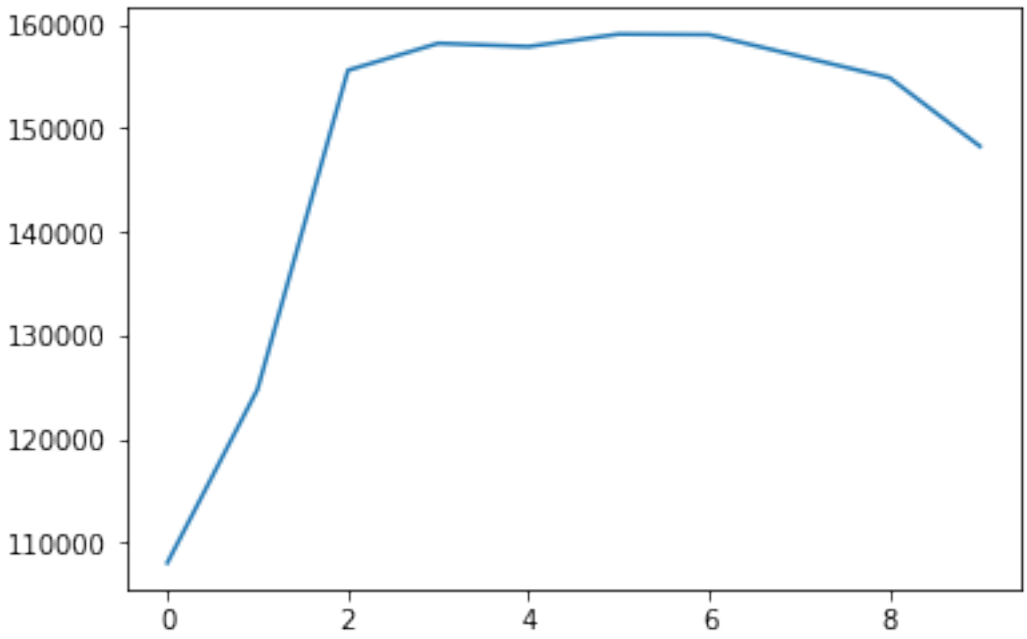
```
Out[27]: [108030,
          124815,
          155611,
          158217,
          157903,
          159104,
          159053,
          156974,
          154885,
          148267]
```

1.3.12 Plotting graphs

Let's plot a graph.

```
In [28]: import matplotlib.pyplot as plt
         %matplotlib inline

In [29]: plt.plot([count_green_in_png(map_at(*location))
                   for location in
                     location_sequence(geolocate("London"),
                                       geolocate("Birmingham"),
                                       10)])
```

```
Out[29]: [<matplotlib.lines.Line2D at 0x2ab407784780>]
```



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

1.3.13 Composing Program Elements

We built little pieces of useful code, to:

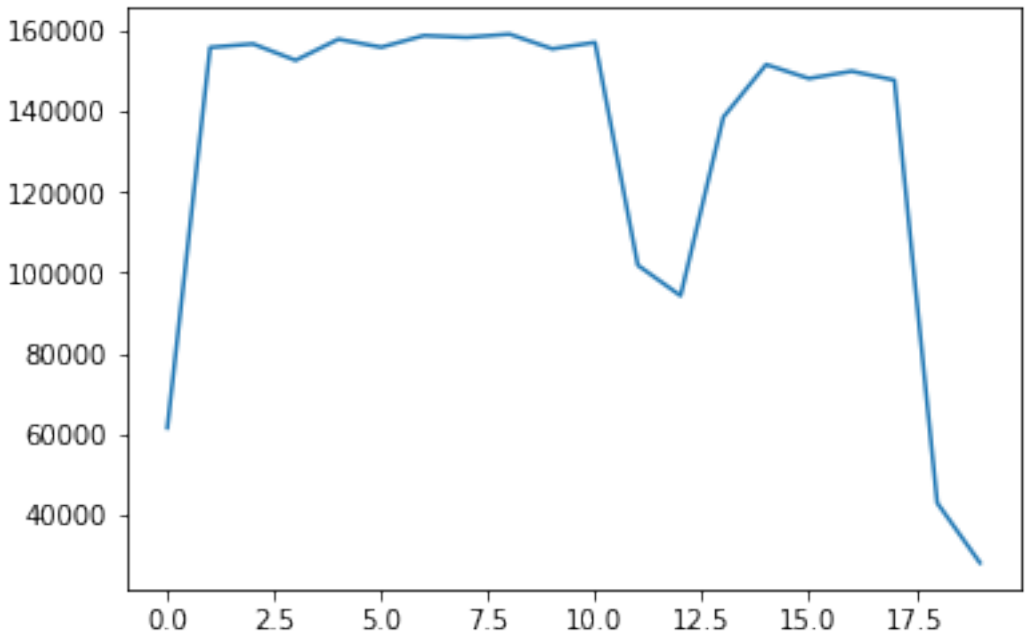
- Find latitude and longitude of a place
- Get a map at a given latitude and longitude
- Decide whether a (red,green,blue) triple is mainly green
- Decide whether each pixel is mainly green
- Plot a new image showing the green places
- Find evenly spaced points between two places

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

```
In [30]: 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 [31]: plt.plot(green_between('New York', 'Chicago', 20))
```

```
Out[31]: [<matplotlib.lines.Line2D at 0x2ab4078209e8>]
```



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.

1.4 Variables

1.4.1 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
```

```
In [7]: type(None)
```

```
Out[7]: NoneType
```

(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 [8]: print(5*six)
```

```
30
```

```
In [9]: scary = six*six*six
```

```
In [10]: print(scary)
```

```
216
```

1.4.2 Reassignment and multiple labels

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

```
In [11]: scary = 25
```

```
In [12]: 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 [13]: name = "James"

In [14]: nom = name

In [15]: print(nom)

James


In [16]: print(name)

James
```

And we can move just one of those labels:

```
In [17]: nom = "Hetherington"

In [18]: print(name)

James


In [19]: 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 [20]: 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

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

1.4.3 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 [21]: type(name)

Out[21]: str
```

Depending on its type, an object can have different *properties*: data fields Inside the object. Consider a Python complex number for example:

```
In [22]: z=3+1j

In [23]: dir(z)

Out[23]: ['__abs__',
          '__add__',
          '__bool__',
          '__class__',
          '__delattr__',
          '__dir__',
          '__divmod__',
          '__doc__',
          '__eq__',
          '__float__',
          '__floordiv__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getnewargs__',
          '__gt__',
          '__hash__',
          '__init__',
          '__int__',
          '__le__',
          '__lt__',
          '__mod__',
          '__mul__',
          '__ne__',
          '__neg__',
          '__new__',
          '__pos__',
          '__pow__',
          '__radd__',
          '__rdivmod__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__rfloordiv__',
          '__rmod__',
          '__rmul__',
          '__rpow__',
          '__rsub__',
          '__rtruediv__',
          '__setattr__',
```

```
 '__sizeof__',
 '__str__',
 '__sub__',
 '__subclasshook__',
 '__truediv__',
 'conjugate',
 'imag',
 'real']
```

```
In [24]: type(z)
```

```
Out[24]: complex
```

```
In [25]: z.real
```

```
Out[25]: 3.0
```

```
In [26]: z.imag
```

```
Out[26]: 1.0
```

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 [27]: z.wrong
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-27-76215e50e85b> in <module>()
----> 1 z.wrong


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

1.4.4 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 [28]: z2=5-6j
         print("Gets to here")
         print(z.wrong)
         print("Didn't get to here")
```

```
Gets to here
```

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

AttributeError                            Traceback (most recent call last)
```

```
<ipython-input-28-755782d0a041> in <module>()
  1 z2=5-6j
  2 print("Gets to here")
----> 3 print(z.wrong)
  4 print("Didn't get to here")


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

```
In [29]:
```

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!

1.4.5 Variables and the notebook kernel

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

```
In [29]: number = 0

In [30]: print(number)

0
```

If I change a variable:

```
In [31]: number = number +1

In [32]: 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

1.5 Using Functions

1.5.1 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
```

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 [2]: sorted("Python")

Out[2]: ['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 [3]: len('Jim')*8

Out[3]: 24

In [4]: x=len('Mike')
        y=len('Bob')
        z=x+y

In [5]: print(z)

7
```

1.5.2 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 [6]: "shout".upper()

Out[6]: 'SHOUT'
```

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

```
In [7]: x = 5

In [8]: type(x)

Out[8]: int

In [9]: x.upper()


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

        AttributeError                            Traceback (most recent call last)

        <ipython-input-9-2262f8ba8d84> in <module>()
    ----> 1 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.wrong
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-10-83048ffb4512> in <module>()
----> 1 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)

In [14]: z.conjugate

Out[14]: <function complex.conjugate>
```

1.5.3 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 [15]: z.conjugate

Out[15]: <function complex.conjugate>

In [16]: type(z.conjugate)

Out[16]: builtin_function_or_method

In [17]: somefunc=z.conjugate

In [18]: somefunc()

Out[18]: (1-5j)
```

Functions are just a kind of variable, and we can assign new labels to them:

```
In [19]: sorted([1,5,3,4])

Out[19]: [1, 3, 4, 5]

In [20]: magic = sorted

In [21]: type(magic)

Out[21]: builtin_function_or_method

In [22]: magic(["Technology", "Advanced"])

Out[22]: ['Advanced', 'Technology']
```

1.5.4 Getting help on functions and methods

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

```
In [23]: help(sorted)
```

```
Help on built-in function sorted in module builtins:

sorted(...)
    sorted(iterable, key=None, reverse=False) --> new sorted list
```

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

```
In [24]: dir("Hexxo")
```

```
Out[24]: ['__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 [25]: "Hexxo".replace("x", "l")

Out[25]: 'Hello'

In [26]: help("FIsh".replace)

Help on built-in function replace:

replace(...) method of builtins.str instance
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new.  If the optional argument count is
```

```
    given, only the first count occurrences are replaced.
```

1.5.5 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 [27]: x = 2 + 3

In [28]: 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 [29]: x.__add__(7)

Out[29]: 12
```

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

```
In [30]: "Hello" + "Goodbye"

Out[30]: 'HelloGoodbye'

In [31]: [2, 3, 4] + [5, 6]

Out[31]: [2, 3, 4, 5, 6]
```

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

```
In [32]: 7-2

Out[32]: 5

In [33]: [2, 3, 4] - [5, 6]


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-33-4627195e7799> 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 [34]: [2, 3, 4] + 5


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

         TypeError                                 Traceback (most recent call last)

         <ipython-input-34-84117f41979f> in <module>()
     ----> 1 [2, 3, 4] + 5


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

To do this, put:

```
In [35]: [2, 3, 4] + [5]

Out[35]: [2, 3, 4, 5]
```

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

```
In [36]: print(2+3*4)

14


In [37]: print((2+3)*4)

20
```

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

1.6 Types

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

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

Out[1]: int
```

1.6.1 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.
```

Zero after a point is optional. But the **Dot** makes it a float.

```
In [3]: tenth= one_float/ten_float

In [4]: tenth

Out[4]: 0.1

In [5]: type(one)

Out[5]: int

In [6]: type(one_float)

Out[6]: float
```

The meaning of an operator varies depending on the type it is applied to! (And on the python version.)

```
In [7]: print(one//ten)

0


In [8]: one_float/ten_float

Out[8]: 0.1

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

<class 'float'>


In [10]: type(tenth)

Out[10]: float
```

The divided by operator when applied to floats, means divide by for real numbers. But when applied to integers, it means divide then round down:

```
In [11]: 10//3

Out[11]: 3

In [12]: 10.0/3

Out[12]: 3.3333333333333335

In [13]: 10/3.0

Out[13]: 3.3333333333333335
```

So if I have two integer variables, and I want the `float` division, I need to change the type first.

There is a function for every type name, which is used to convert the input to an output of the desired type.

```
In [14]: x = float(5)
         type(x)

Out[14]: float

In [15]: 10/float(3)
```

```
Out[15]: 3.3333333333333335
```

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.

```
In [16]: N=10000.0
         sum([1/N]*int(N))
```

```
Out[16]: 0.9999999999999062
```

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

1.6.2 Strings

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

```
In [17]: given = "James"
         family = "Hetherngton"
         full = given + " " + family
```

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

```
In [18]: print(full.upper())
```

```
JAMES HETHERNGTON
```

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

```
In [19]: ten, one
```

```
Out[19]: (10, 1)
```

```
In [20]: print(ten + one)
```

```
11
```

```
In [21]: print(float( str(ten)+str(one) ))
```

```
101.0
```

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

```
In [22]: "    Hello  ".strip()
```

```
Out[22]: 'Hello'
```

1.6.3 Lists

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

We can define our own list with square brackets:

```
In [23]: [1, 3, 7]
```

```
Out[23]: [1, 3, 7]
```

```
In [24]: type([1, 3, 7])
```

```
Out[24]: list
```

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

```
In [25]: various_things = [1, 2, "banana", 3.4, [1,2] ]
```

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

```
In [26]: various_things[2]
```

```
Out[26]: 'banana'
```

```
In [27]: index = 0
         various_things[index]
```

```
Out[27]: 1
```

Note that list indices start from zero.

We can quickly make a list with numbers counted up:

```
In [28]: count_to_five = range(5)
         print(list(count_to_five))
```

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

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

```
In [29]: name = ["James", "Philip", "John", "Hetherington"]
         print("==".join(name))
```

```
James==Philip==John==Hetherington
```

And we can split up a string into a list:

```
In [30]: "Ernst Stavro Blofeld".split(" ")
```

```
Out[30]: ['Ernst', 'Stavro', 'Blofeld']
```

```
In [31]: "Ernst Stavro Blofeld".split("o")
```

```
Out[31]: ['Ernst Stavr', ' Bl', 'feld']
```

And combine these:

```
In [32]: "->".join("John Ronald Reuel Tolkein".split(" "))
```

```
Out[32]: 'John->Ronald->Reuel->Tolkein'
```

A matrix can be represented by **nesting** lists – putting lists inside other lists.

```
In [33]: identity = [[1, 0], [0, 1]]
```

```
In [34]: identity[0][0]
```

```
Out[34]: 1
```

... but later we will learn about a better way of representing matrices.

1.6.4 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 [35]: print(count_to_five[1])

1


In [36]: print("James"[2])

m


In [37]: count_to_five = list(range(5))

In [38]: count_to_five[1:3]

Out[38]: [1, 2]

In [39]: "Hello World"[4:8]

Out[39]: 'o Wo'

In [40]: len(various_things)

Out[40]: 5

In [41]: len("Python")

Out[41]: 6

In [42]: name

Out[42]: ['James', 'Philip', 'John', 'Hetherington']

In [43]: "John" in name

Out[43]: True

In [44]: 3 in count_to_five

Out[44]: True

In [45]: "James's Class"

Out[45]: "James's Class"

In [46]: '"Wow!", said Bob.'

Out[46]: '"Wow!", said Bob.'
```

1.6.5 Unpacking

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

```
In [47]: mylist = ['Hello', 'World']
         a, b = mylist
         print(b)
```

```
World
```

```
In [48]: range(4)
```

```
Out[48]: range(0, 4)
```

```
In [49]: zero, one, two, three = range(4)
```

```
In [50]: two
```

```
Out[50]: 2
```

If there is too much or too little data, an error results:

```
In [51]: zero, one, two, three = range(7)
```

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

        ValueError                                Traceback (most recent call last)

        <ipython-input-51-6b7ac0213dc4> in <module>()
    ----> 1 zero, one, two, three = range(7)


        ValueError: too many values to unpack (expected 4)
```

```
In [52]: zero, one, two, three = range(2)
```

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

        ValueError                                Traceback (most recent call last)

        <ipython-input-52-0f89676d3fd9> in <module>()
    ----> 1 zero, one, two, three = range(2)


        ValueError: need more than 2 values to unpack
```

1.7 Containers

1.7.1 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
```

1.7.2 Mutability

A list 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
```

1.7.3 Tuples

A `tuple` is an immutable sequence. It is like a list, execpt it cannot be changed. It is defined with round brackets.

```
In [7]: x = 0,
        type(x)
```

```
Out[7]: tuple
```

```
In [8]: my_tuple = ("Hello", "World")
        my_tuple[0]="Goodbye"
```

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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-8-d3ad0c7e33f1> in <module>()
          1 my_tuple = ("Hello", "World")
    ----> 2 my_tuple[0]="Goodbye"


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

```
In [9]: type(my_tuple)

Out[9]: tuple
```

`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.

1.7.4 Memory and containers

The way memory works with containers can be important:

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

Out[12]: [0, 1, 2]

In [13]: y = x
         y

Out[13]: [0, 1, 2]

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

In [15]: x

Out[15]: [0, 'Gotcha!', 2]

In [16]: y

Out[16]: [0, 'Gotcha!', 2]

In [17]: z

Out[17]: [0, 1, 2]

In [18]: z[2] = "Really?"
```

```
In [19]: x

Out[19]: [0, 'Gotcha!', 2]

In [20]: y

Out[20]: [0, 'Gotcha!', 2]

In [21]: z

Out[21]: [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=

The explanation: While `y` is a second label on the *same object*, `z` is a separate object with the same data.

The difference between `y=x` and `z=x[:]` is important. (Remember `[:]` is equivalent to `[0:<last>]`)

```
In [22]: x

Out[22]: [0, 'Gotcha!', 2]

In [23]: y = x

In [24]: y = x[:]
```

Nested objects make it even more complicated:

```
In [25]: x = [ ['a', 'b'] , 'c']
         y = x
         z = x[0:2]

In [26]: x[0][1] = 'd'
         z[1] = 'e'

In [27]: x

Out[27]: [['a', 'd'], 'c']

In [28]: y

Out[28]: [['a', 'd'], 'c']

In [29]: z

Out[29]: [['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.

1.7.5 Identity vs Equality

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

```
In [30]: [1, 2] == [1, 2]

Out[30]: True

In [31]: [1, 2] is [1, 2]
```

```
Out[31]: 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.

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 [32]: "Hello" == "Hello"

Out[32]: True

In [33]: "Hello" is "Hello"

Out[33]: True
```

This can be useful in understanding problems like the one above:

```
In [34]: x = range(3)
         y=x
         z=x[:]

In [35]: x == y

Out[35]: True

In [36]: x is y

Out[36]: True

In [37]: x == z

Out[37]: True

In [38]: x is z

Out[38]: False

In [39]: {
             'a': 2,
             'b': 3
         }

Out[39]: {'a': 2, 'b': 3}
```

1.8 Dictionaries

1.8.1 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(" ")

In [2]: names[1]

Out[2]: 'Luther'
```

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

```
In [3]: me = { "name": "James", "age": 39,
               "Jobs": ["Programmer", "Teacher"] }

In [4]: me

Out[4]: {'Jobs': ['Programmer', 'Teacher'], 'age': 39, 'name': 'James'}

In [5]: me['Jobs']

Out[5]: ['Programmer', 'Teacher']

In [6]: me['age']

Out[6]: 39

In [7]: type(me)

Out[7]: dict
```

1.8.2 Keys and Values

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

```
In [8]: me.keys()

Out[8]: dict_keys(['age', 'Jobs', 'name'])
```

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

```
In [9]: me.values()

Out[9]: dict_values([39, ['Programmer', 'Teacher'], 'James'])
```

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

```
In [10]: 'Jobs' in me

Out[10]: True

In [11]: 'James' in me

Out[11]: False

In [12]: 'James' in me.values()

Out[12]: True
```

1.8.3 Immutable Keys Only

The way in which dictionaries work is one of the coolest things in computer science: the "hash table". The details of this are beyond the scope of this course, but we will consider some aspects in the section on performance programming.

One consequence of this implementation is that you can only use **immutable** things as keys.

```
In [13]: good_match = {
             ("Lamb", "Mint"): True,
             ("Bacon", "Chocolate"): False
            }
```

but:

```
In [14]: illegal = {
             ["Lamb", "Mint"]: True,
             ["Bacon", "Chocolate"]: False
           }
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-14-ae372ef2f311> in <module>()
  1 illegal = {
----> 2     ["Lamb", "Mint"]: True,
  3     ["Bacon", "Chocolate"]: False
  4    }


TypeError: unhashable type: 'list'
```

Remember – square brackets denote lists, round brackets denote `tuples`.

1.8.4 No guarantee of order

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

```
In [15]: my_dict = {'0': 0, '1':1, '2': 2, '3': 3, '4': 4}
         print(my_dict)
         print(my_dict.values())
```

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

1.8.5 Sets

A set is a `list` which cannot contain the same element twice. We make one by calling `set()` on any sequence, e.g. a list or string.

```
In [16]: name = "James Hetherington"
         unique_letters = set(name)

In [17]: unique_letters

Out[17]: {' ', 'H', 'J', 'a', 'e', 'g', 'h', 'i', 'm', 'n', 'o', 'r', 's', 't'}
```

Or by defining a literal like a dictionary, but without the colons:

```
In [18]: primes_below_ten = { 2, 3, 5, 7}

In [19]: type(unique_letters)

Out[19]: set

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

```
Out[20]: set

In [21]: unique_letters

Out[21]: {' ', 'H', 'J', 'a', 'e', 'g', 'h', 'i', 'm', 'n', 'o', 'r', 's', 't'}
```

This will be easier to read if we turn the set of letters back into a string, with `join`:

```
In [22]: "".join(unique_letters)

Out[22]: 'JHhemngaso itr'
```

A set has no particular order, but is really useful for checking or storing **unique** values.
Set operations work as in mathematics:

```
In [23]: x = set("Hello")
         y = set("Goodbye")

In [24]: x & y # Intersection

Out[24]: {'e', 'o'}

In [25]: x | y # Union

Out[25]: {'G', 'H', 'b', 'd', 'e', 'l', 'o', 'y'}

In [26]: y - x # y intersection with complement of x: letters in Goodbye but not in Hello

Out[26]: {'G', 'b', 'd', 'y'}
```

Your programs will be faster and more readable if you use the appropriate container type for your data's meaning. Always use a set for lists which can't in principle contain the same data twice, always use a dictionary for anything which feels like a mapping from keys to values.

1.9 Data structures

1.9.1 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]
        leaders

Out[9]: ['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...

1.9.2 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 one 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 front room. It fits 3 people. Sue is currently there.

- From the bedroom, you can go downstairs to the front room. 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.

myhouse = ["Your answer here"]

1.9.3 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.

1.10 Control and Flow

1.10.1 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*!

1.10.2 Conditionality

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

```
In [1]: x = 5

        if x < 0:
            print(x, " is negative")
```

The absence of output here means the if clause prevented the print statement from running.

```
In [2]: x=-10

        if x < 0:
            print(x, " is negative")
```

```
-10  is negative
```

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.

1.10.3 Else and Elif

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

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

```
x is positive
```

```
In [4]: 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 [5]: choice =  'high'

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

```
1
```

1.10.4 Comparison

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

```
In [6]: 1 > 2
```

```
Out[6]: False
```

Comparison on strings is alphabetical.

```
In [7]: "UCL" > "KCL"
```

```
Out[7]: True
```

But case sensitive:

```
In [8]: "UCL" > "kcl"
```

```
Out[8]: False
```

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

```
In [9]: True == "True"
```

```
Out[9]: False
```

In python two there were subtle implied order comparisons between types, but it was bad style to rely on these. In python three, you cannot compare these.

```
In [10]: '1' < 2
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-10-b67fbc3e6cdc> in <module>()
----> 1 '1' < 2


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

```
In [11]: '5' < 2
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-11-df18900bf463> in <module>()
----> 1 '5' < 2


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

```
In [12]: '1' > 2
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-12-6857a5c3feea> 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.

1.10.5 Automatic Falsehood

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

```
In [13]: mytext = "Hello"
```

```
In [14]: if mytext:
             print("Mytext is not empty")
```

```
Mytext is not empty
```

```
In [15]: mytext2 = ""
```