An introduction to Python Programming for Research

James Hetherington

January 17, 2020

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

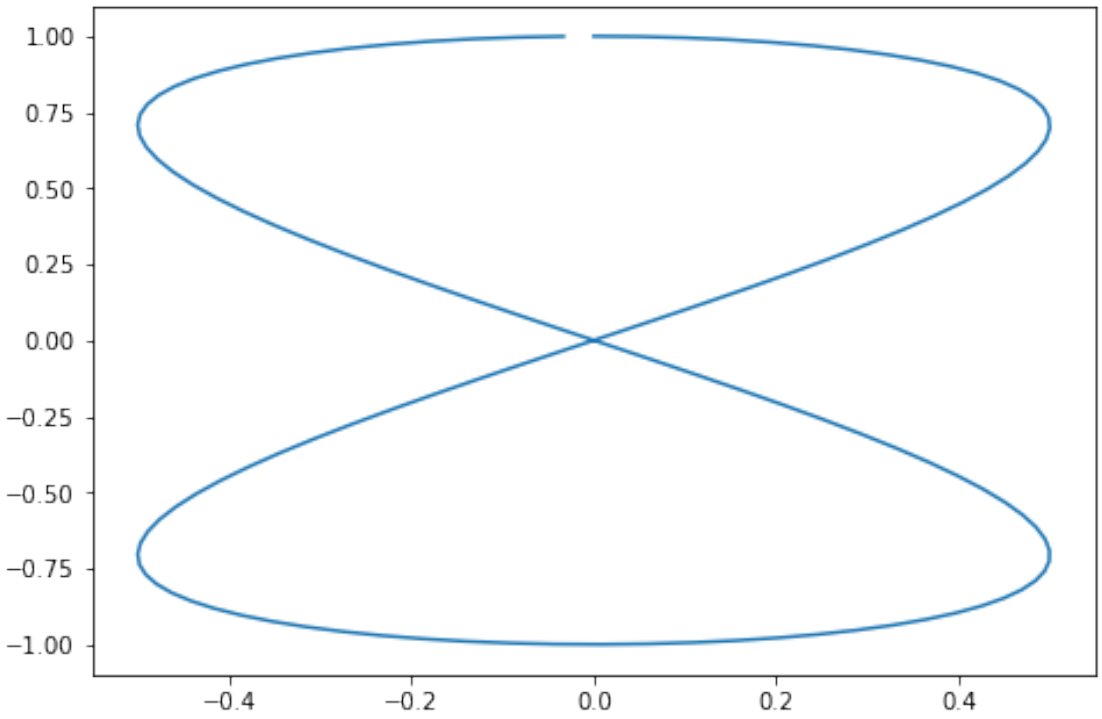
1.2.1 The Jupyter Notebook

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 math

        import numpy as np
        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 0x7f503afd4150>]
```



We're going to be mainly working in the Jupyter 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 Desktop@UCL (available in the teaching cluster rooms or anywhere).

Jupyter notebooks consist of discussion cells, referred to as "markdown cells", and "code cells", which contain Python. This document has been created using Jupyter 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 Jupyter notebooks.

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. Here are some examples:

```
In [4]: %%bash
        echo "print(2 * 4)" > eight.py
        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%29) and setting the permissions to execute.

```
In [5]: %%writefile fourteen.py
        #! /usr/bin/env python
        print(2 * 7)
```

```
Writing fourteen.py
```

```
In [6]: %%bash
        chmod u+x fourteen.py
        ./fourteen.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:

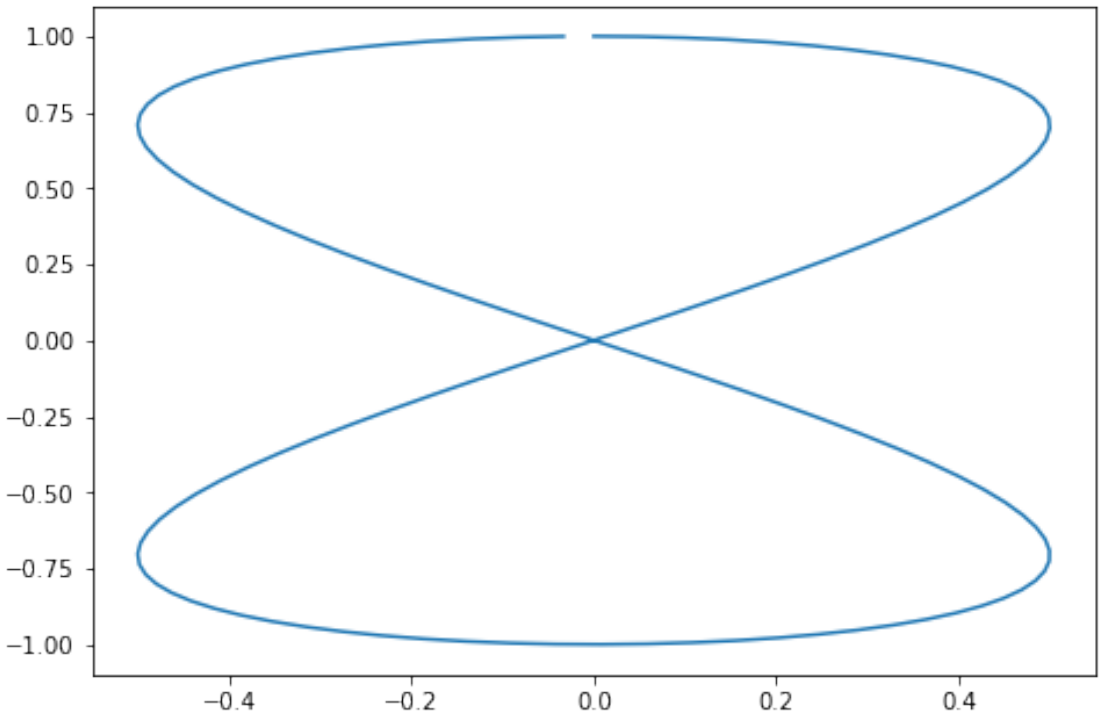
```
In [7]: %%writefile draw_eight.py
        # Above line tells the notebook to treat the rest of this
        # cell as content for a file on disk.
        import math

        import numpy as np
        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 Atom or VS code.

```
In [8]: import draw_eight # Load the library file we just wrote to disk
```

```
In [9]: image = draw_eight.make_figure()
```



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

The `%%` at the beginning of a cell is called *magics*. There's a large list of them available and you can create your own.

1.3 An example Python data analysis notebook

This page illustrates how to use Python to perform a simple but complete analysis: retrieve data, do some computations based on it, and visualise the results.

Don't worry if you don't understand everything on this page! Its purpose is to give you an example of things you can do and how to go about doing them - you are not expected to be able to reproduce an analysis like this in Python at this stage! We will be looking at the concepts and practices introduced on this page as we go along the course.

As we show the code for different parts of the work, we will be touching on various aspects you may want to keep in mind, either related to Python specifically, or to research programming more generally.

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. 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.org/project/geopy/
```

Now, if you try to follow along on this example in an Jupyter 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.Yandex(lang="en_US")
        geocoder.geocode('Cambridge', exactly_one=False)
```

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

HTTPError                                 Traceback (most recent call last)

~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
    356         except Exception as error:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


/opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


/opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
```

```
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-2-7fa4fab2949a> in <module>
      1 geocoder = geopy.geocoders.Yandex(lang="en_US")
----> 2 geocoder.geocode('Cambridge', exactly_one=False)


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                 exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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')
```

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

    HTTPError                                 Traceback (most recent call last)

    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
    356         except Exception as error:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)
```

```
    <ipython-input-5-ccb6d38c8bab> in <module>
----> 1 geolocate('Cambridge')


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                 exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

1.3.5 Variables

We can store a result in a variable:

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


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

        HTTPError                                 Traceback (most recent call last)

        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
        354         try:
    --> 355             page = requester(req, timeout=timeout, **kwargs)
        356         except Exception as error:


        /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
        530             meth = getattr(processor, meth_name)
    --> 531             response = meth(req, response)
        532


        /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
        640             response = self.parent.error(
    --> 641                 'http', request, response, code, msg, hdrs)
```

```
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-6-e33090ca51bc> in <module>
----> 1 london_location = geolocate("London")
      2 print(london_location)


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                 exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)
```

```
GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

1.3.6 More complex functions

The Yandex API allows us to fetch a map of a place, given a longitude and latitude. The URLs look like: https://static-maps.yandex.ru/1.x/?size=400,400&ll=-0.1275,51.51&z=10&l=sat&lang=en_US 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)):
            base = "https://static-maps.yandex.ru/1.x/?"

            params = dict(
                z = zoom,
                size = str(size[0]) + "," + str(size[1]),
                ll = str(long) + "," + str(lat),
                l = "sat" if satellite else "map",
                lang = "en_US"
            )

            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:])
```

```
https://static-maps.yandex.ru/1.x/?z=10&size=400%2
C400&ll=-0.1275%2C51.5072&l=sat&lang=en_US
```

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

```
In [10]: assert "https://static-maps.yandex.ru/1.x/?" in url
         assert "ll=-0.1275%2C51.5072" in url
         assert "z=10" in url
         assert "size=400%2C400" 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 [11]: map_response.content[0:20]
```

```
Out[11]: b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H\x00H\x00\x00'
```

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 Jupyter 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)
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-13-d69fb2ebab72> in <module>
  1 import IPython
----> 2 map_png = map_at(*london_location)


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

```
In [14]: print("The type of our map result is actually a: ", type(map_png))
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-14-6c68dc8fc8d1> in <module>
----> 1 print("The type of our map result is actually a: ", type(map_png))


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

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

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

NameError                                 Traceback (most recent call last)

<ipython-input-15-ac666969e449> in <module>
----> 1 IPython.core.display.Image(map_png)


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

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

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

    HTTPError                                 Traceback (most recent call last)

    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
    356         except Exception as error:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-16-9124101779f1> in <module>
----> 1 IPython.core.display.Image(map_at(*geolocate("New Delhi")))


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
```

```
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
117             exactly_one,
118         )


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
371                                 exc_info=False)
372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
374                 except KeyError:
375                     raise GeocoderServiceError(message)


GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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
         import imageio # 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 = imageio.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) ))
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-20-1df2d88d5544> in <module>
----> 1 print(count_green_in_png( map_at(*london_location) ))


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

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)
```

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

HTTPError                                 Traceback (most recent call last)

~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
356         except Exception as error:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
532


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
642


/opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
570


/opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
504             if result is not None:
```

```
    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-22-ed53afe2376e> in <module>
----> 1 location_sequence(geolocate("London"), geolocate("Cambridge"), 5)


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                 exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

1.3.10 Creating Images

We should display the green content to check our work:

```
In [23]: def show_green_in_png(data):
             pixels = imageio.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 = imageio.imwrite(buffer, out, format='png')
            return buffer.getvalue()
```

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

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-24-84d560d5795b> in <module>
          1 IPython.core.display.Image(
    ----> 2     map_at(*london_location, satellite=True)
          3 )


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

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

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-25-ba1938f843d6> in <module>
          2     show_green_in_png(
          3         map_at(
    ----> 4             *london_location,
          5             satellite=True)))


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

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)))
```

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

        HTTPError                                 Traceback (most recent call last)
```

```
    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
    356         except Exception as error:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-26-b3877d0d28cf> in <module>
----> 1 for location in location_sequence(geolocate("London"),
      2                                   geolocate("Birmingham"),
      3                                   4):
      4     IPython.core.display.display(
      5         IPython.core.display.Image(map_at(*location)))


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
```

```
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
117             exactly_one,
118         )


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
371                                 exc_info=False)
372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
374                 except KeyError:
375                     raise GeocoderServiceError(message)


GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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)]


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

HTTPError                                 Traceback (most recent call last)

~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
356         except Exception as error:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
532


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
642


/opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
```

```
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-27-b5d8a75e50ec> in <module>
      1 [count_green_in_png(map_at(*location))
      2             for location in
----> 3                  location_sequence(geolocate("London"),
      4                                    geolocate("Birmingham"),
      5                                    10)]


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                     exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)
```

```
GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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)])


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

         HTTPError                                 Traceback (most recent call last)

         ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
         354         try:
     --> 355             page = requester(req, timeout=timeout, **kwargs)
         356         except Exception as error:


         /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
         530             meth = getattr(processor, meth_name)
     --> 531             response = meth(req, response)
         532


         /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
         640             response = self.parent.error(
     --> 641                 'http', request, response, code, msg, hdrs)
         642


         /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
         568             args = (dict, 'default', 'http_error_default') + orig_args
     --> 569             return self._call_chain(*args)
         570


         /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
         502             func = getattr(handler, meth_name)
     --> 503             result = func(*args)
         504             if result is not None:


         /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
         648     def http_error_default(self, req, fp, code, msg, hdrs):
     --> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
         650
```

```
    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-29-e7d26b5362c3> in <module>
      1 plt.plot([count_green_in_png(map_at(*location))
      2             for location in
----> 3               location_sequence(geolocate("London"),
      4                                 geolocate("Birmingham"),
      5                                 10)])


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                 exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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))
```

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

HTTPError                                 Traceback (most recent call last)

~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
356         except Exception as error:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
532


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
642


/opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
570


/opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
504             if result is not None:


/opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
650


HTTPError: HTTP Error 403: Forbidden
```

```
During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-31-eea54607c6ae> in <module>
----> 1 plt.plot(green_between('New York', 'Chicago', 20))


    <ipython-input-30-7a2c39ca1229> in green_between(start, end, steps)
      2     return [count_green_in_png( map_at(*location) )
      3             for location in location_sequence(
----> 4                             geolocate(start),
      5                             geolocate(end),
      6                             steps)]


    <ipython-input-4-48a1a3c91ee7> in geolocate(place)
      1 def geolocate(place):
----> 2     return geocoder.geocode(place, exactly_one = False)[0][1]


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                   exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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-4-25c0309421cb> 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 Software Carpentry's Python lesson.
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 = "Eric"
In [14]: nom = name
In [15]: print(nom)
Eric

In [16]: print(name)
Eric
```

And we can move just one of those labels:

```
In [17]: nom = "Idle"
In [18]: print(name)
Eric

In [19]: print(nom)
Idle
```

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 = "Michael"
```

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
```

We can see what properties and methods an object has available using the `dir` function:

```
In [23]: dir(z)
```

```
Out[23]: ['__abs__',
          '__add__',
          '__bool__',
          '__class__',
          '__delattr__',
          '__dir__',
          '__divmod__',
          '__doc__',
          '__eq__',
          '__float__',
          '__floordiv__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getnewargs__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__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']
```

You can see that there are several methods whose name starts and ends with `__` (e.g. `__init__`): these are special methods that Python uses internally, and we will discuss some of them later on in this course. The others (in this case, `conjugate`, `img` and `real`) are the methods and fields through which we can interact with this object.

```
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-27-0cc5a8ef8f99> 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-f92e96af0737> 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'
```

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 31, reading `number = number + 1`, I can run it again, with Shift-Enter. The value of `number` will change from 2 to 3, then from 3 to 4 - but the output of the next cell (containing the `print` statement) will not change unless I rerun that too. 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) Jupyter notebook documentation.

1.5 Using Functions

1.5.1 Calling functions

We often want to do things 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 (more specifically, according to their Unicode order).

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-9-328ac508ff1b> 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-10-29321da545fa> 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(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 customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

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__',
          '__init_subclass__',
          '__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',
 'isascii',
 '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(old, new, count=-1, /) method of builtins.str instance
    Return a copy with all occurrences of substring old replaced by new.

      count
        Maximum number of occurrences to replace.
        -1 (the default value) means replace all occurrences.

    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-33-5b64b789ad11> 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-34-67b01a5c24ab> 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: Python operator precedence.

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:

- Python's documentation about floating point arithmetic;
- How floating point numbers work;
- Advanced: What Every Computer Scientist Should Know About Floating-Point Arithmetic.

1.6.2 Strings

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

```
In [17]: given = "Terry"
         family = "Jones"
         full = given + " " + family
```

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

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

TERRY JONES
```

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'
```

Note that you can write strings in Python using either single (`' ... '`) or double (`" ... "`) quote marks. The two ways are equivalent. However, if your string includes a single quote (e.g. an apostrophe), you should use double quotes to surround it:

```
In [23]: "Terry's animation"
```

```
Out[23]: "Terry's animation"
```

And vice versa: if your string has a double quote inside it, you should wrap the whole string in single quotes.

```
In [24]: '"Wow!", said John.'
```

```
Out[24]: '"Wow!", said John.'
```

1.6.3 Lists

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

We can define our own list with square brackets:

```
In [25]: [1, 3, 7]
```

```
Out[25]: [1, 3, 7]
```

```
In [26]: type([1, 3, 7])
```

```
Out[26]: list
```

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

```
In [27]: various_things = [1, 2, "banana", 3.4, [1,2] ]
```

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

```
In [28]: various_things[2]
```

```
Out[28]: 'banana'
```

```
In [29]: index = 0
         various_things[index]
```

```
Out[29]: 1
```

Note that list indices start from zero.

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

```
In [30]: name = ["Sir", "Michael", "Edward", "Palin"]
         print("==".join(name))
```

```
Sir==Michael==Edward==Palin
```

And we can split up a string into a list:

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

Out[31]: ['Ernst', 'Stavro', 'Blofeld']

In [32]: "Ernst Stavro Blofeld".split("o")

Out[32]: ['Ernst Stavr', ' Bl', 'feld']
```

And combine these:

```
In [33]: "->".join("John Ronald Reuel Tolkein".split(" "))

Out[33]: 'John->Ronald->Reuel->Tolkein'
```

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

```
In [34]: identity = [[1, 0], [0, 1]]

In [35]: identity[0][0]

Out[35]: 1
```

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

1.6.4 Ranges

Another useful type is range, which gives you a sequence of consecutive numbers. In contrast to a list, ranges generate the numbers as you need them, rather than all at once.

If you try to print a range, you'll see something that looks a little strange:

```
In [36]: range(5)

Out[36]: range(0, 5)
```

We don't see the contents, because *they haven't been generatead yet.* Instead, Python gives us a description of the object - in this case, its type (range) and its lower and upper limits.

We can quickly make a list with numbers counted up by converting this range:

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

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

Ranges in Python can be customised in other ways, such as by specifying the lower limit or the step (that is, the difference between successive elements). You can find more information about them in the official Python documentation.

1.6.5 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.*

Sequences support various useful operations, including: - Accessing a single element at a particular index: `sequence[index]` - Accessing multiple elements (a *slice*): `sequence[start:end_plus_one]` - Getting the length of a sequence: `len(sequence)` - Checking whether the sequence contains an element: `element in sequence`

The following examples illustrate these operations with lists, strings and ranges.

```
In [38]: print(count_to_five[1])
```

```
1
```

```
In [39]: print("Palin"[2])
```

```
l
```

```
In [40]: count_to_five = range(5)
```

```
In [41]: count_to_five[1:3]
```

```
Out[41]: range(1, 3)
```

```
In [42]: "Hello World"[4:8]
```

```
Out[42]: 'o Wo'
```

```
In [43]: len(various_things)
```

```
Out[43]: 5
```

```
In [44]: len("Python")
```

```
Out[44]: 6
```

```
In [45]: name
```

```
Out[45]: ['Sir', 'Michael', 'Edward', 'Palin']
```

```
In [46]: "Edward" in name
```

```
Out[46]: True
```

```
In [47]: 3 in count_to_five
```

```
Out[47]: True
```

1.6.6 Unpacking

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

```
In [48]: mylist = ['Hello', 'World']
         a, b = mylist
         print(b)
```

```
World
```

```
In [49]: range(4)
```

```
Out[49]: range(0, 4)
```

```
In [50]: zero, one, two, three = range(4)
```

```
In [51]: two
```

```
Out[51]: 2
```

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

```
In [52]: zero, one, two, three = range(7)
```

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

<ipython-input-52-3331a3ab5222> in <module>
----> 1 zero, one, two, three = range(7)


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

```
In [53]: zero, one, two, three = range(2)
```

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

<ipython-input-53-8575e9410b1d> in <module>
----> 1 zero, one, two, three = range(2)


ValueError: not enough values to unpack (expected 4, got 2)
```

Python provides some handy syntax to split a sequence into its first element ("head") and the remaining ones (its "tail"):

```
In [54]: head, *tail = range(4)
         print("head is", head)
         print("tail is", tail)
```

```
head is 0
tail is [1, 2, 3]
```

Note the syntax with the *. The same pattern can be used, for example, to extract the middle segment of a sequence whose length we might not know:

```
In [55]: one, *two, three = range(10)
```

```
In [56]: print("one is", one)
         print("two is", two)
         print("three is", three)
```

```
one is 0
two is [1, 2, 3, 4, 5, 6, 7, 8]
three is 9
```

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 = "Sir Michael Edward Palin".split(" ")
        print(name)
```

```
['Sir', 'Michael', 'Edward', 'Palin']
```

```
In [6]: name[0] = "Knight"
        name[1:3] = ["Mike-"]
        name.append("FRGS")

        print(" ".join(name))
```

```
Knight Mike- Palin FRGS
```

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-8-242e9dae76d3> 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-10-7127277fc72e> 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.

The explanation: While `y` is a second label on the *same object*, `z` is a separate object with the same data. Writing `x[:]` creates a new list containing all the elements of `x` (remember: `[:]` is equivalent to `[0:<last>]`). This is the case whenever we take a slice from a list, not just when taking all the elements with `[:]`.

The difference between `y=x` and `z=x[:]` is important!

Nested objects make it even more complicated:

```
In [22]: x = [['a', 'b'] , 'c']
         y = x
         z = x[0:2]

In [23]: x[0][1] = 'd'
         z[1] = 'e'

In [24]: x

Out[24]: [['a', 'd'], 'c']

In [25]: y

Out[25]: [['a', 'd'], 'c']

In [26]: z

Out[26]: [['a', 'd'], 'e']
```

Try the visualiser again.

Supplementary material: The copies that we make through slicing are called *shallow copies*: we don't copy all the objects they contain, only the references to them. This is why the nested list in `x[0]` is not copied, so `z[0]` still refers to it. It is possible to actually create copies of all the contents, however deeply nested they are - this is called a *deep copy*. Python provides methods for that in its standard library, in the `copy` module. You can read more about that, as well as about shallow and deep copies, in the library reference.

1.7.5 Identity vs Equality

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

```
In [27]: [1, 2] == [1, 2]

Out[27]: True

In [28]: [1, 2] is [1, 2]

Out[28]: 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 [29]: "Hello" == "Hello"

Out[29]: True

In [30]: "Hello" is "Hello"

Out[30]: True
```

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

```
In [31]: x = range(3)
         y = x
         z = x[:]

In [32]: x == y

Out[32]: True

In [33]: x is y

Out[33]: True

In [34]: x == z

Out[34]: True

In [35]: x is z

Out[35]: False
```

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]: chapman = {"name": "Graham", "age": 48,
                   "Jobs": ["Comedian", "Writer"] }

In [4]: chapman

Out[4]: {'name': 'Graham', 'age': 48, 'Jobs': ['Comedian', 'Writer']}

In [5]: chapman['Jobs']

Out[5]: ['Comedian', 'Writer']

In [6]: chapman['age']

Out[6]: 48

In [7]: type(chapman)

Out[7]: dict
```

1.8.2 Keys and Values

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

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

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

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

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

Out[9]: dict_values(['Graham', 48, ['Comedian', 'Writer']])
```

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

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

Out[10]: True

In [11]: 'Graham' in chapman

Out[11]: False

In [12]: 'Graham' in chapman.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-14-514a4c981e6d> 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 `tuple`s.

1.8.4 No guarantee of order (before Python 3.7)

Another consequence of the way dictionaries used to work is that there was no guaranteed order among the elements. However, since Python 3.7, it's guaranteed that dictionaries return elements in the order in which they were inserted. Read more about why that changed and how it is still fast.

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

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

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 = "Graham Chapman"
         unique_letters = set(name)
```

```
In [17]: unique_letters

Out[17]: {' ', 'C', 'G', 'a', 'h', 'm', 'n', 'p', 'r'}
```

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]: {' ', 'C', 'G', 'a', 'h', 'm', 'n', 'p', 'r'}
```

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]: 'hrGp Cnma'
```

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]: Chapman = {
            'City': 'London',
            'Street': 'Southwood ln',
            'Postcode': 'N6 5TB'
        }
```

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

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

In [4]: addresses

Out[4]: [{'City': 'London', 'Street': 'Gower Street', 'Postcode': 'WC1E 6BT'},
         {'City': 'London', 'Street': 'Southwood ln', 'Postcode': 'N6 5TB'}]
```

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'] = ['Jeremy','Leonard', 'James', 'Henry']

In [6]: Chapman['people'] = ['Graham', 'David']

In [7]: addresses

Out[7]: [{'City': 'London',
          'Street': 'Gower Street',
          'Postcode': 'WC1E 6BT',
          'people': ['Jeremy', 'Leonard', 'James', 'Henry']},
         {'City': 'London',
          'Street': 'Southwood ln',
          'Postcode': 'N6 5TB',
          'people': ['Graham', 'David']}]
```

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]: ['Jeremy', 'Graham']
```

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. Graham 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. David 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' : ['Graham'],
                'capacity' : 2
            },
            'kitchen' : {
                'exits': {
                    'south' : 'living'
                },
                'people' : [],
                'capacity' : 1
            },
            'garden' : {
                'exits': {
                    'inside' : 'living'
                },
                'people' : ['David'],
                '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(f"{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(f"{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-10-2ae56e567bff> in <module>
    ----> 1 '1' < 2


        TypeError: '<' not supported between instances of 'str' and 'int'
```

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

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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-11-4b266c2a1d9b> in <module>
    ----> 1 '5' < 2


        TypeError: '<' not supported between instances of 'str' and 'int'
```

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

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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-12-142f2d5d83a7> in <module>
    ----> 1 '1' > 2


        TypeError: '>' not supported between instances of 'str' and '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 = ""

In [16]: if mytext2:
             print("Mytext2 is not empty")
```

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

```
In [17]: x = 3.2
         if not (x > 0 and isinstance(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 [18]: not not "Who's there!" # Thanks to Mysterious Student

Out[18]: True

In [19]: bool("")

Out[19]: False

In [20]: bool("Graham")

Out[20]: True

In [21]: bool([])

Out[21]: False

In [22]: bool(['a'])

Out[22]: True

In [23]: bool({})

Out[23]: False

In [24]: bool({'name': 'Graham'})

Out[24]: True

In [25]: bool(0)

Out[25]: False

In [26]: bool(1)

Out[26]: True
```

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

```
In [27]: [] == False

Out[27]: False

In [28]: bool([]) == False

Out[28]: True
```

1.10.6 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.

No indentation when it is expected, results in an error:

```
In [29]: x = 2

In [30]: if x > 0:
         print(x)


          File "<ipython-input-30-61c7132d9aa5>", line 2
        print(x)
            ^
    IndentationError: expected an indented block
```

but:

```
In [31]: if x > 0:
             print(x)

2
```

1.10.7 Pass

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

```
In [32]: if x > 0:
             # print x

         print("Hello")


          File "<ipython-input-32-1045ed7694fb>", line 4
        print("Hello")
            ^
    IndentationError: expected an indented block
```

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

```
In [33]: if x > 0:
             # print x
             pass

         print("Hello")

Hello
```

1.10.8 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]

In [2]: for whatever in mylist:
            print(whatever ** 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.

1.10.9 Iterables

Any sequence type is iterable:

```
In [3]: vowels = "aeiou"
        sarcasm = []

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

            sarcasm.append(letter * repetition)

        "".join(sarcasm)

Out[3]: 'OOOkaaay'
```

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

1.10.10 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 and dictionaries.

```
In [4]: import datetime
        now = datetime.datetime.now()

        founded = {"Eric": 1943, "UCL": 1826, "Cambridge": 1209}

        current_year = now.year

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

Eric  is  77 years old.
UCL  is  194 years old.
Cambridge  is  811 years old.
```

1.10.11 Unpacking and Iteration

Unpacking can be useful with iteration:

```
In [5]: triples = [
            [4, 11, 15],
            [39, 4, 18]
        ]
```

```
In [6]: for whatever in triples:
            print(whatever)
```

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

```
In [7]: for first, middle, last in triples:
            print(middle)
```

```
11
4
```

```
In [8]: # 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 [9]: things = {"Eric": [1943, 'South Shields'],
                  "UCL": [1826, 'Bloomsbury'],
                  "Cambridge": [1209, 'Cambridge']}

        print(things.items())
```

```
dict_items([('Eric', [1943, 'South Shields']), ('UCL', [1826, 'Bloomsbury']), ('Cambridge', [1209, 'Cam
```

```
In [10]: for name, year in founded.items():
             print(name, " is ", current_year - year, "years old.")
```

```
Eric  is  77 years old.
UCL  is  194 years old.
Cambridge  is  811 years old.
```

1.10.12 Break, Continue

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

```
In [11]: 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.

1.10.13 Classroom 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.

1.10.14 Solution: counting people in the maze

With this maze structure:

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

```
            }
        }
```

We can count the occupants and capacity like this:

```
In [2]: capacity = 0
        occupancy = 0
        for name, room in house.items():
            capacity += room['capacity']
            occupancy += len(room['people'])
        print(f"House can fit {capacity} people, and currently has: {occupancy}.")
```

```
House can fit 7 people, and currently has: 2.
```

As a side note, note how we included the values of `capacity` and `occupancy` in the last line. This is a handy syntax for building strings that contain the values of variables. You can read more about it at this Python String Formatting Best Practices guide or in the official documentation.

1.11 Comprehensions

1.11.1 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]: [2 ** x for x in range(10)]
```

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

Which is equivalent to the following code without using comprehensions:

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

        result
```

```
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(10)]
```

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

1.11.2 Selection in comprehensions

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 "Eric Idle"
                 if letter.lower() not in 'aeiou'])
```

```
Out[5]: 'rc dl'
```

1.11.3 Comprehensions versus building lists with `append`:

This code:

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

```
Out[6]: [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.

Almost 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.)

1.11.4 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']]
```

1.11.5 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}
```

1.11.6 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]
```

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
```

My favourite is `map`, which, similar to a list comprehension, 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]: list(map(str, range(10)))

Out[18]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

So I can write:

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

We'll learn more about `map` and similar functions when we discuss functional programming later in the course.

1.11.7 Classroom Exercise: Occupancy Dictionary

Take your maze data structure. First write an expression to print out a new dictionary, which holds, for each room, that room's capacity. The output should look like:

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

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

Now, 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 [20]: {'garden': 1, 'living': 1}

Out[20]: {'garden': 1, 'living': 1}
```

1.11.8 Solution

With this maze structure:

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

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

```
In [2]: {name: room['capacity'] for name, room in house.items()}
```

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

To get the current number of occupants, we can use a similar dictionary comprehension. Remember that we can *filter* (only keep certain rooms) by adding an `if` clause:

```
In [3]: {name: len(room['people']) for name, room in house.items() if len(room['people']) > 0}
```

```
Out[3]: {'living': 1, 'garden': 1}
```

1.12 Functions

1.12.1 Definition

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

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

        print(double(5), double([5]), double('five'))
```

```
10 [5, 5] fivefive
```

1.12.2 Default Parameters

We can specify default values for parameters:

```
In [2]: def jeeves(name = "Sir"):
            return f"Very good, {name}"
```

```
In [3]: jeeves()
```

```
Out[3]: 'Very good, Sir'
```

```
In [4]: jeeves('John')
```

```
Out[4]: 'Very good, John'
```

If you have some parameters with defaults, and some without, those with defaults **must** go later.
If you have multiple default arguments, you can specify neither, one or both:

```
In [5]: def jeeves(greeting="Very good", name="Sir"):
            return f"{greeting}, {name}"
```

```
In [6]: jeeves()
```

```
Out[6]: 'Very good, Sir'
```

```
In [7]: jeeves("Hello")
```

```
Out[7]: 'Hello, Sir'
```

```
In [8]: jeeves(name = "John")
```

```
Out[8]: 'Very good, John'
```

```
In [9]: jeeves(greeting="Suits you")
```

```
Out[9]: 'Suits you, Sir'
```

```
In [10]: jeeves("Hello", "Producer")
```

```
Out[10]: 'Hello, Producer'
```

1.12.3 Side effects

Functions can do things to change their **mutable** arguments, so `return` is optional.
This is pretty awful style, in general, functions should normally be side-effect free.
Here is a contrived example of a function that makes plausible use of a side-effect

```
In [11]: def double_inplace(vec):
             vec[:] = [element * 2 for element in vec]

         z = list(range(4))
         double_inplace(z)
         print(z)
```

```
[0, 2, 4, 6]
```

```
In [12]: letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
         letters[:] = []
```

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.
But I'd usually just write this as a function which **returned** the output:

```
In [13]: def double(vec):
             return [element * 2 for element in vec]
```

Let's remind ourselves of the behaviour for modifying lists in-place using `[:]` with a simple array:

```
In [14]: x = 5
         x = 7
         x = ['a', 'b', 'c']
         y = x

In [15]: x

Out[15]: ['a', 'b', 'c']

In [16]: x[:] = ["Hooray!", "Yippee"]

In [17]: y

Out[17]: ['Hooray!', 'Yippee']
```

1.12.4 Early Return

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

Here's a slightly more plausibly useful function-with-side-effects to extend a list with a specified padding datum.

```
In [18]: def extend(to, vec, pad):
             if len(vec) >= to:
                 return # Exit early, list is already long enough.

             vec[:] = vec + [pad] * (to - len(vec))

In [19]: x = list(range(3))
         extend(6, x, 'a')
         print(x)

[0, 1, 2, 'a', 'a', 'a']


In [20]: z = range(9)
         extend(6, z, 'a')
         print(z)

range(0, 9)
```

1.12.5 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 [21]: def arrow(before, after):
             return f"{before} -> {after}"

         arrow(1, 3)
```

```
Out[21]: '1 -> 3'
```

```
In [22]: x = [1, -1]
         arrow(*x)
```

```
Out[22]: '1 -> -1'
```

This can be quite powerful:

```
In [23]: charges = {"neutron": 0, "proton": 1, "electron": -1}
         for particle in charges.items():
             print(arrow(*particle))
```

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

1.12.6 Sequence Arguments

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

```
In [24]: def doubler(*sequence):
             return [x * 2 for x in sequence]
```

```
In [25]: doubler(1, 2, 3)
```

```
Out[25]: [2, 4, 6]
```

```
In [26]: doubler(5, 2, "Wow!")
```

```
Out[26]: [10, 4, 'Wow!Wow!']
```

1.12.7 Keyword Arguments

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

```
In [27]: def arrowify(**args):
             for key, value in args.items():
                 print(f"{key} -> {value}")

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

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

These different approaches can be mixed:

```
In [28]: def somefunc(a, b, *args, **kwargs):
             print("A:", a)
             print("B:", b)
             print("args:", args)
             print("keyword args", kwargs)
```

```
In [29]: somefunc(1, 2, 3, 4, 5, fish="Haddock")
```

```
A: 1
B: 2
args: (3, 4, 5)
keyword args {'fish': 'Haddock'}
```

1.13 Using Libraries

1.13.1 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-1-12dcc3af2e0c> 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',
'remainder',
'sin',
'sinh',
'sqrt',
'tan',
'tanh',
'tau',
'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__[0:50])
        print(math.__file__[50:])

/home/travis/virtualenv/python3.7.5/lib/python3.7/
lib-dynload/math.cpython-37m-x86_64-linux-gnu.so
```

Note that `import` does *not* install libraries. 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.

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 on different architectures (modules installed by default are shown with ticks).

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

1.13.2 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.

1.13.3 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]: import math
        math.sin(math.pi)

Out[8]: 1.2246467991473532e-16

In [9]: from math import sin
        sin(math.pi)

Out[9]: 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 [10]: from math import *
         sin(pi)

Out[10]: 1.2246467991473532e-16
```

1.13.4 Import and rename

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

```
In [11]: import math as m
         m.cos(0)
```

```
Out[11]: 1.0
```

```
In [12]: pi = 3
         from math import pi as realpi
         print(sin(pi), sin(realpi))
```

```
0.1411200080598672 1.2246467991473532e-16
```

1.14 Defining your own classes

1.14.1 User Defined Types

A **class** is a user-programmed Python type (since Python 2.2!)

It can be defined like:

```
In [1]: class Room(object):
            pass
```

Or:

```
In [2]: class Room():
            pass
```

Or:

```
In [3]: class Room:
            pass
```

What's the difference? Before Python 2.2 a class was distinct from all other Python types, which caused some odd behaviour. To fix this, classes were redefined as user programmed types by extending `object`, e.g., class `room(object)`.

So most Python 2 code will use this syntax as very few people want to use old style python classes. Python 3 has formalised this by removing old-style classes, so they can be defined without extending `object`, or indeed without braces.

Just as with other python types, you use the name of the type as a function to make a variable of that type:

```
In [4]: zero = int()
        type(zero)
```

```
Out[4]: int
```

```
In [5]: myroom = Room()
        type(myroom)
```

```
Out[5]: __main__.Room
```

In the jargon, we say that an **object** is an **instance** of a particular **class**.

`__main__` is the name of the scope in which top-level code executes, where we've defined the class `Room`.

Once we have an object with a type of our own devising, we can add properties at will:

```
In [6]: myroom.name = "Living"

In [7]: myroom.name

Out[7]: '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 [8]: myroom.capacity = 3
        myroom.occupants = ["Graham", "Eric"]
```

1.14.2 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 [9]: class Room:
            def overfull(self):
                return len(self.occupants) > self.capacity

In [10]: myroom = Room()
         myroom.capacity = 3
         myroom.occupants = ["Graham", "Eric"]

In [11]: myroom.overfull()

Out[11]: False

In [12]: myroom.occupants.append(['TerryG'])

In [13]: myroom.occupants.append(['John'])

In [14]: myroom.overfull()

Out[14]: 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".

This is just a convention for this variable name, not a keyword. You could call it something else if you wanted.

1.14.3 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 [15]: class Room:
             def __init__(self, name, exits, capacity, occupants=[]):
                 self.name = name
                 self.occupants = occupants  # Note the default argument, occupants start empty
                 self.exits = exits
                 self.capacity = capacity

             def overfull(self):
                 return len(self.occupants) > self.capacity
```

```
In [16]: living = Room("Living Room", {'north': 'garden'}, 3)

In [17]: living.capacity

Out[17]: 3
```

Methods which begin and end with **two underscores** in their names fulfil special capabilities in Python, such as constructors.

1.14.4 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 [18]: class Maze:
             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):
                 self.describe()
                 print("")
                 self.wander()
                 print("")

             def simulate(self, steps):
                 for _ in range(steps):
                     self.step()
```

And a "Room" class with exits, and people:

```
In [19]: class Room:
             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(f"{self.name}: " + " ".join(self.occupants.keys()))
```

We define a "Person" class for room occupants:

```
In [20]: class Person:
        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("{some} goes {action} to the {where}".format(some=self.name,
                                                               action=exit,
                                                               where=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 [21]: graham = Person('Graham')
         eric = Person('Eric')
```

```
terryg = Person('TerryG')
john = Person('John')
```

```
In [22]: living = Room('livingroom', {'outside': 'garden',
                                       'upstairs': 'bedroom', 'north': 'kitchen'}, 2)
         kitchen = Room('kitchen', {'south': 'livingroom'}, 1)
         garden = Room('garden', {'inside': 'livingroom'}, 3)
         bedroom = Room('bedroom', {'jump': 'garden', 'downstairs': 'livingroom'}, 1)
```

```
In [23]: house = Maze('My House')
```

```
In [24]: for room in [living, kitchen, garden, bedroom]:
             house.add_room(room)
```

```
In [25]: living.add_occupant(graham)
```

```
In [26]: garden.add_occupant(eric)
         garden.add_occupant(terryg)
```

```
In [27]: bedroom.add_occupant(john)
```

And we can run a "simulation" of our model:

```
In [28]: house.simulate(3)
```

```
livingroom: Graham
garden: Eric TerryG
bedroom: John

Graham goes north to the kitchen
Eric goes inside to the livingroom
TerryG goes inside to the livingroom
John goes jump to the garden

livingroom: Eric TerryG
kitchen: Graham
garden: John

Eric goes upstairs to the bedroom
TerryG goes outside to the garden
Graham goes south to the livingroom
John goes inside to the livingroom

livingroom: Graham John
garden: TerryG
bedroom: Eric

Graham goes north to the kitchen
John goes outside to the garden
TerryG goes inside to the livingroom
Eric goes jump to the garden
```

1.14.5 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 [29]: class Maze:
             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 [30]: class Room:
             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 [31]: class Person:
            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("{some} goes {action} to the {where}".format(some=self.name,
                                                                   action=exit.name,
                                                                   where=destination.name))

            def wander(self):
                exit = self.room.random_valid_exit()
                if exit:
                    self.use(exit)

            def describe(self):
                print("{who} is in the {where}".format(who=self.name,
                                                       where=self.room.name))
```

```
In [32]: class Exit:
            def __init__(self, name, target):
                self.name = name
                self.target = target

            def valid(self):
                return self.target.has_space()
```

```
In [33]: house = Maze('My New House')
```

```
In [34]: 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 [35]: house.add_exit('north', living, kitchen, 'south')
```

```
In [36]: house.add_exit('upstairs', living, bed, 'downstairs')
```

```
In [37]: house.add_exit('outside', living, garden, 'inside')
```

```
In [38]: house.add_exit('jump', bed, garden)

In [39]: house.add_occupant('Graham', living)
         house.add_occupant('Eric', garden)
         house.add_occupant('TerryJ', bed)
         house.add_occupant('John', garden)

In [40]: house.simulate(3)

Graham is in the livingroom
Eric is in the garden
TerryJ is in the bedroom
John is in the garden

Graham goes outside to the garden
Eric goes inside to the livingroom
TerryJ goes jump to the garden
John goes inside to the livingroom

Graham is in the garden
Eric is in the livingroom
TerryJ is in the garden
John is in the livingroom

Eric goes upstairs to the bedroom
TerryJ goes inside to the livingroom
John goes north to the kitchen

Graham is in the garden
Eric is in the bedroom
TerryJ is in the livingroom
John is in the kitchen

Graham goes inside to the livingroom
Eric goes jump to the garden
TerryJ goes outside to the garden
John goes south 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 an important part of this course.

1.14.6 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 2

Working with Data

2.1 Loading data from files

2.1.1 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.

2.1.2 An example datafile

Let's write an example datafile to disk so we can investigate it. We'll just use a plain-text file. Jupyter 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 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

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 system
        os.getcwd() # Use the 'getcwd' function from the 'os' module to find where we are on disk.

Out[2]: '/home/travis/build/UCL/rsd-engineeringcourse/ch01data'
```

Can we see if it is there?

```
In [3]: import os
        [x for x in os.listdir(os.getcwd()) if ".txt" in x]

Out[3]: ['mydata.txt']
```

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

2.1.3 Path independence and os

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

```
In [4]: os.path.dirname(os.getcwd())

Out[4]: '/home/travis/build/UCL/rsd-engineeringcourse'
```

We could do this manually using `split`:

```
In [5]: "/".join(os.getcwd().split("/")[:-1])

Out[5]: '/home/travis/build/UCL/rsd-engineeringcourse'
```

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 this Software Carpentry lesson on navigating the file system.

Satisfy yourself that after using `%%writefile`, 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.

2.1.4 Opening files in Python

So, let's read our file:

```
In [6]: myfile = open('mydata.txt')
```

```
In [7]: type(myfile)
```

```
Out[7]: _io.TextIOWrapper
```

Even though the name of this type is not very clear, it offers various ways of accessing the file.
We can go line-by-line, by treating the file as an iterable:

```
In [8]: [x for x in myfile]
```

```
Out[8]: ["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 entire universe. \
         '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 evolution of stars. \
         '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 watches it!\n',
         'If our small minds, for some convenience, divide this glass of wine, this universe, \n',
         '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\n']
```

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

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

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

We need to 'rewind' it!

```
In [10]: myfile.seek(0)
         [len(x) for x in myfile if 'know' in x]
```

```
Out[10]: [56, 39]
```

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

2.1.5 Working with files

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

```
In [11]: myfile.seek(0)
         first = myfile.readline()

In [12]: first

Out[12]: "A poet once said, 'The whole universe is in a glass of wine.'\n"

In [13]: second = myfile.readline()

In [14]: second

Out[14]: 'We will probably never know in what sense he meant it, \n'
```

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

```
In [15]: rest = myfile.read()

In [16]: rest

Out[16]: "for poets do not write to be understood. \nBut it is true that if we look at a glass of wine
```

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

```
In [17]: open('mydata.txt').read()

Out[17]: "A poet once said, 'The whole universe is in a glass of wine.'\nWe will probably never know in
```

You can also read just a few characters:

```
In [18]: myfile.seek(1335)

Out[18]: 1335

In [19]: myfile.read(15)

Out[19]: '\n   - Richard F'
```

2.1.6 Converting strings to files

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

```
In [20]: mystring = "Hello World\n My name is James"

In [21]: mystring

Out[21]: 'Hello World\n My name is James'

In [22]: mystring.readline()

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

AttributeError                            Traceback (most recent call last)

<ipython-input-22-8fadd4a635f7> 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` class of the io module in the standard library:

```
In [23]: from io import StringIO

In [24]: mystringasafile = StringIO(mystring)

In [25]: mystringasafile.readline()

Out[25]: 'Hello World\n'

In [26]: mystringasafile.readline()

Out[26]: ' My name is James'
```

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

2.1.7 Closing files

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

```
In [27]: myfile.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 [28]: with open('mydata.txt') as somefile:
             content = somefile.read()

         content

Out[28]: "A poet once said, 'The whole universe is in a glass of wine.'\nWe will probably never know in
```

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

You should pretty much **always** use this syntax for working with files. We will see more about context managers in a later chapter.

2.1.8 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 [29]: with open('mywrittenfile', 'w') as target:
             target.write('Hello')
             target.write('World')

In [30]: with open('mywrittenfile','r') as source:
             print(source.read())

HelloWorld
```

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

```
In [31]: with open('mywrittenfile', 'a') as target:
             target.write('Hello')
             target.write('James')
```

```
In [32]: with open('mywrittenfile','r') as source:
             print(source.read())

HelloWorldHelloJames
```

If a file already exists, mode 'w' will overwrite it.

2.2 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.

2.2.1 URLs

All internet resources are defined by a Uniform Resource Locator.

```
In [1]: "https://static-maps.yandex.ru/1.x/?size=400,400&ll=-0.1275,51.51&z=10&l=sat&lang=en_US"

Out[1]: 'https://static-maps.yandex.ru/1.x/?size=400,400&ll=-0.1275,51.51&z=10&l=sat&lang=en_US'
```

A url consists of:

- A *scheme* (`http`, `https`, `ssh`, ...)
- A *host* (`static-maps.yandex.ru`, 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, 443 for https)
- A *path* (Like a file path on the machine, here it is `1.x/`)
- A *query* part after a `?`, (optional, usually ampersand-separated *parameters* e.g. `size=400x400`, or `z=10`)

Supplementary materials: These can actually be different for different protocols, the above is a simplification. You can see more, for example, at the wikipedia article about the 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 are quite subtle. See the wikipedia article for more detail. The standard library provides the urlencode function that can take care of this for you.

2.2.2 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("https://static-maps.yandex.ru/1.x/?size=400,400&ll=-0.1275,51.51&z=10&
                                params={
                                    'size': '400,400',
                                    'll': '-0.1275,51.51',
                                    'zoom': 10,
                                    'l': 'sat',
                                    'lang': 'en_US'
                                })
```

```
In [4]: response.content[0:50]
```

```
Out[4]: b'\xff\xd8\xff\xe0\x00\x10JFIF\x00\x01\x01\x01\x00H\x00H\x00\x00\xff\xdb\x00C\x00\x08\x06\x06\x
```

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

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.

2.2.3 Example: Sunspots

Let's try to get something scientific: the sunspot cycle data from SILSO:

```
In [5]: spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php').text
```

```
In [6]: spots[0:80]
```

```
Out[6]: '1749;01;1749.042;  96.7; -1.0;   -1;1\n1749;02;1749.123; 104.3; -1.0;   -1;1\n1749'
```

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.

2.2.4 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 [7]: lines = spots.split("\n")
        lines[0:5]
```

```
Out[7]: ['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 [8]: years = [line.split(";")[0] for line in lines]
```

```
In [9]: years[0:15]
```

```
Out[9]: ['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.

2.2.5 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.

2.3 Field and Record Data

2.3.1 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')
        spots.text.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 where:

- Each **record** is on its own line
- Each record has multiple **fields**
- Fields are separated by some **separator**

Typical separators are the `space`, `tab`, `comma`, and `semicolon`, leading to correspondingly-named file formats, 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 row is a record, each cell is a field), CSV files are very popular.

2.3.2 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 comment.

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 (also called a header) defines the names of the fields, e.g.:

```
name, age
James, 39
Will, 2
```

2.3.3 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 gives us lists and dictionaries, rather than its more exciting `array` type.

```
In [2]: import numpy as np
        import requests
```

```
In [3]: spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php', stream=True)
```

`stream=True` delays loading all of the data until it is required.

```
In [4]: sunspots = np.genfromtxt(spots.raw, 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.

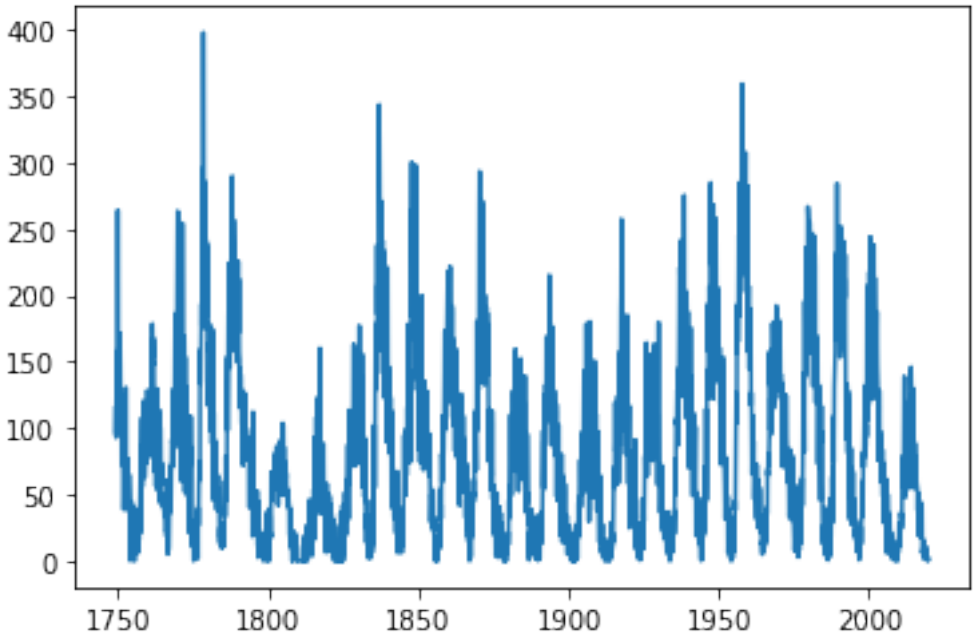
```
In [5]: sunspots[0][3]
```

```
Out[5]: 96.7
```

We can now plot the "Sunspot cycle":

```
In [6]: %matplotlib inline

        from matplotlib import pyplot as plt
        plt.plot(sunspots[:,2], sunspots[:,3]) # Numpy syntax to access all
                                               # rows, specified column.
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7f4d6d3df550>]
```



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. Don't worry about the `%matplotlib` magic command for now - we'll also look at this later.

2.3.4 Naming Columns

I happen to know that the columns here are defined as follows:

From the data provider's own documentation:

> 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]: spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php', stream=True)

        sunspots = np.genfromtxt(spots.raw, delimiter=';',
                                 names=['year','month','date',
                                 'mean','deviation','observations','definitive'])
```

```
In [8]: sunspots
```

```
Out[8]: array([(1749.,  1., 1749.042,  96.7, -1. ,  -1., 1.),
               (1749.,  2., 1749.123, 104.3, -1. ,  -1., 1.),
               (1749.,  3., 1749.204, 116.7, -1. ,  -1., 1.), …,
               (2019., 10., 2019.79 ,   0.4,  0.1, 857., 0.),
               (2019., 11., 2019.873,   0.5,  0.1, 693., 0.),
               (2019., 12., 2019.958,   1.6,  0.6, 719., 0.)],
              dtype=[('year', '<f8'), ('month', '<f8'), ('date', '<f8'), ('mean', '<f8'), ('deviation',
```

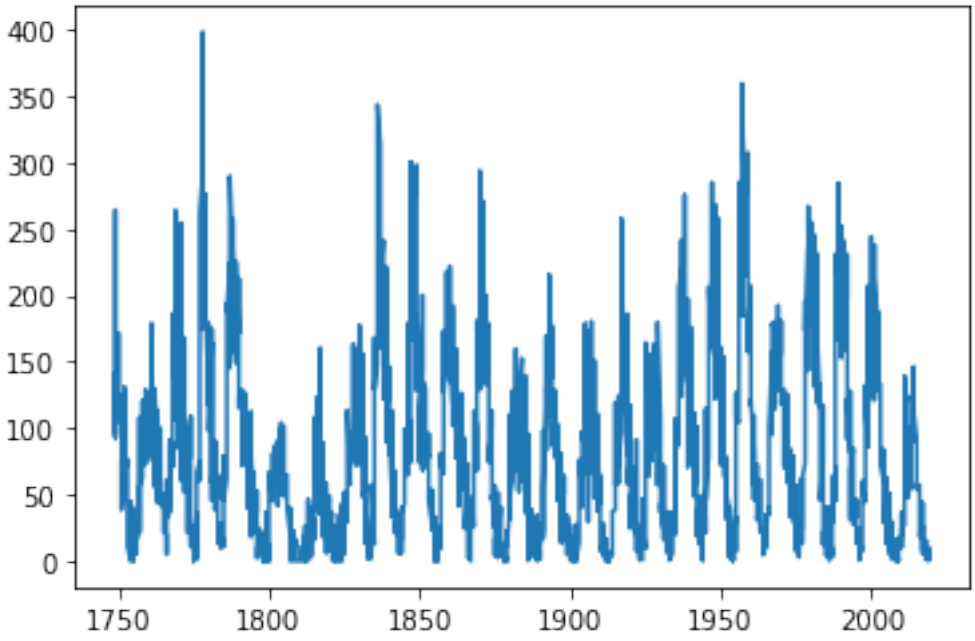
2.3.5 Typed Fields

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

```
In [9]: spots = requests.get('http://www.sidc.be/silso/INFO/snmtotcsv.php', stream=True)

        sunspots = np.genfromtxt(spots.raw, delimiter=';',
                                 names=['year','month','date',
                                 'mean','deviation','observations','definitive'],
                                 dtype=[int, int, float, float, float, int, int])

In [10]: sunspots

Out[10]: array([(1749,  1, 1749.042,  96.7, -1. ,  -1, 1),
                (1749,  2, 1749.123, 104.3, -1. ,  -1, 1),
                (1749,  3, 1749.204, 116.7, -1. ,  -1, 1), …,
                (2019, 10, 2019.79 ,   0.4,  0.1, 857, 0),
                (2019, 11, 2019.873,   0.5,  0.1, 693, 0),
                (2019, 12, 2019.958,   1.6,  0.6, 719, 0)],
               dtype=[('year', '<i8'), ('month', '<i8'), ('date', '<f8'), ('mean', '<f8'), ('deviation'
```

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

```
In [11]: sunspots['year']

Out[11]: array([1749, 1749, 1749, …, 2019, 2019, 2019])

In [12]: plt.plot(sunspots['year'], sunspots['mean'])

Out[12]: [<matplotlib.lines.Line2D at 0x7f4d456fc450>]
```



2.4 Structured Data

2.4.1 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.

2.4.2 JSON

JSON is a very common open-standard data format that is used to store structured data in a human-readable way.

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 [1]: import json
```

Any nested group of dictionaries and lists can be saved:

```
In [2]: mydata = {'key': ['value1', 'value2'],
                  'key2': {'key4':'value3'}}
```

```
In [3]: json.dumps(mydata)
```

```
Out[3]: '{"key": ["value1", "value2"], "key2": {"key4": "value3"}}'
```

If you would like a more readable output, you can use the `indent` argument.

```
In [4]: print(json.dumps(mydata, indent=4))
```

```
{
    "key": [
        "value1",
        "value2"
    ],
    "key2": {
        "key4": "value3"
    }
}
```

Loading data is also really easy:

```
In [5]: %%writefile myfile.json
        {
            "somekey": ["a list", "with values"]
        }

Writing myfile.json


In [6]: with open('myfile.json', 'r') as json_file:
            my_data_as_string = json_file.read()

In [7]: my_data_as_string

Out[7]: '{\n    "somekey": ["a list", "with values"]\n}\n'

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

In [9]: mydata['somekey']

Out[9]: ['a list', 'with values']
```

This is a very nice solution for loading and saving Python data structures.
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.

2.4.3 YAML

YAML is a very similar data format 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:
             - a list # Look, this is a list
             - with values

Writing myfile.yaml


In [11]: import yaml  # This may need installed as pyyaml
```

```
In [12]: with open('myfile.yaml') as yaml_file:
             my_data = yaml.safe_load(yaml_file)
         print(mydata)

{'somekey': ['a list', 'with values']}
```

YAML is a popular format for ad-hoc data files, but the library doesn't ship with default Python (though it is part of Anaconda and Canopy), so some people still prefer JSON for its universality.

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.safe_dump(mydata))

somekey:
- a list
- with values
```

```
In [14]: print(yaml.safe_dump(mydata, default_flow_style=True))

{somekey: [a list, with values]}
```

`default_flow_style=False` (the default) uses a "block style" (rather than an "inline" or "flow style") to delineate data structures. See the YAML docs for more details.

2.4.4 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.

2.4.5 Exercise: Saving and loading data

Use YAML or JSON to save your maze data structure to disk and load it again.

```
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'
```

```
{"living": {"exits": {"north": "kitchen", "outside": "garden", "upstairs": "bedroom"}, "people": ["James
```

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]: {'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}}
```

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.safe_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']}}
```

2.5 Classroom exercise: the biggest earthquake in the UK this century

2.5.1 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": "2018-10-11",
                                    "orderby": "time-asc"}
                                )
```

```
In [2]: quakes.text[0:100]
```

```
Out[2]: '{"type":"FeatureCollection","metadata":{"generated":1579287393000,"url":"https://earthquake.us
```

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 an online map service at that latitude and longitude: look back at the introductory example * Display that image

2.6 Solution to the earthquake exercise

NOTE: This is intended as a reference for **after** you have attempted the problem (notebook version) yourself!

2.6.1 Download the 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": "2018-10-11",
                                  "orderby": "time-asc"}
                              )
```

2.6.2 Parse the data as JSON

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

```
In [3]: requests_json = json.loads(quakes.text)
```

2.6.3 Investigate the data to discover how it is structured

There is no foolproof way of doing this. A good first step is to see the type of our data!

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

```
Out[4]: dict
```

Now we can navigate through this dictionary to see how the information is stored in the nested dictionaries and lists. The `keys` method can indicate what kind of information each dictionary holds, and the `len` function tells us how many entries are contained in a list. How you explore is up to you!

```
In [5]: requests_json.keys()

Out[5]: dict_keys(['type', 'metadata', 'features', 'bbox'])

In [6]: len(requests_json['features'])

Out[6]: 120

In [7]: requests_json['features'][0].keys()

Out[7]: dict_keys(['type', 'properties', 'geometry', 'id'])

In [8]: requests_json['features'][0]['properties'].keys()

Out[8]: dict_keys(['mag', 'place', 'time', 'updated', 'tz', 'url', 'detail', 'felt', 'cdi', 'mmi', 'ale

In [9]: requests_json['features'][0]['properties']['mag']

Out[9]: 2.6

In [10]: requests_json['features'][0]['geometry']

Out[10]: {'type': 'Point', 'coordinates': [-2.81, 54.77, 14]}
```

Also note that some IDEs display JSON in a way that makes its structure easier to understand. Try saving this data in a text file and opening it in an IDE or a browser.

2.6.4 Find the largest quake

```
In [11]: quakes = requests_json['features']

In [12]: 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['properties']['mag']

Out[12]: 4.8

In [13]: lat = largest_so_far['geometry']['coordinates'][1]
         long = largest_so_far['geometry']['coordinates'][0]
         print("Latitude: {} Longitude: {}".format(lat, long))

Latitude: 52.52 Longitude: -2.15
```

2.6.5 Get a map at the point of the quake

We saw something similar in the Greengraph example (notebook version) of the previous chapter.

```
In [14]: import requests


         def request_map_at(lat, long, satellite=True,
                            zoom=10, size=(400, 400)):
             base = "https://static-maps.yandex.ru/1.x/?"

             params = dict(
```

```
            z=zoom,
            size="{},{}".format(size[0], size[1]),
            ll="{},{}".format(long, lat),
            l="sat" if satellite else "map",
            lang="en_US"
        )

        return requests.get(base, params=params)
```

```
In [15]: map_png = request_map_at(lat, long, zoom=10, satellite=False)
```

2.6.6 Display the map

```
In [16]: from IPython.display import Image
         Image(map_png.content)
```

Out[16]:

2.7 Scientific File Formats

CSV, JSON and YAML are very common formats for representing general-purpose data, but their simplicity sometimes makes then inconvenient for scientific applications. A common drawback, for example, is that reading very large amounts of data from a CSV or JSON file can be inefficient. This has led to to the use of more targetted file formats which better address scientists' requirements for storing, accessing or manipulating data.

In this section, we will see an example of such a file format, and how to interact with files written in it programmatically.

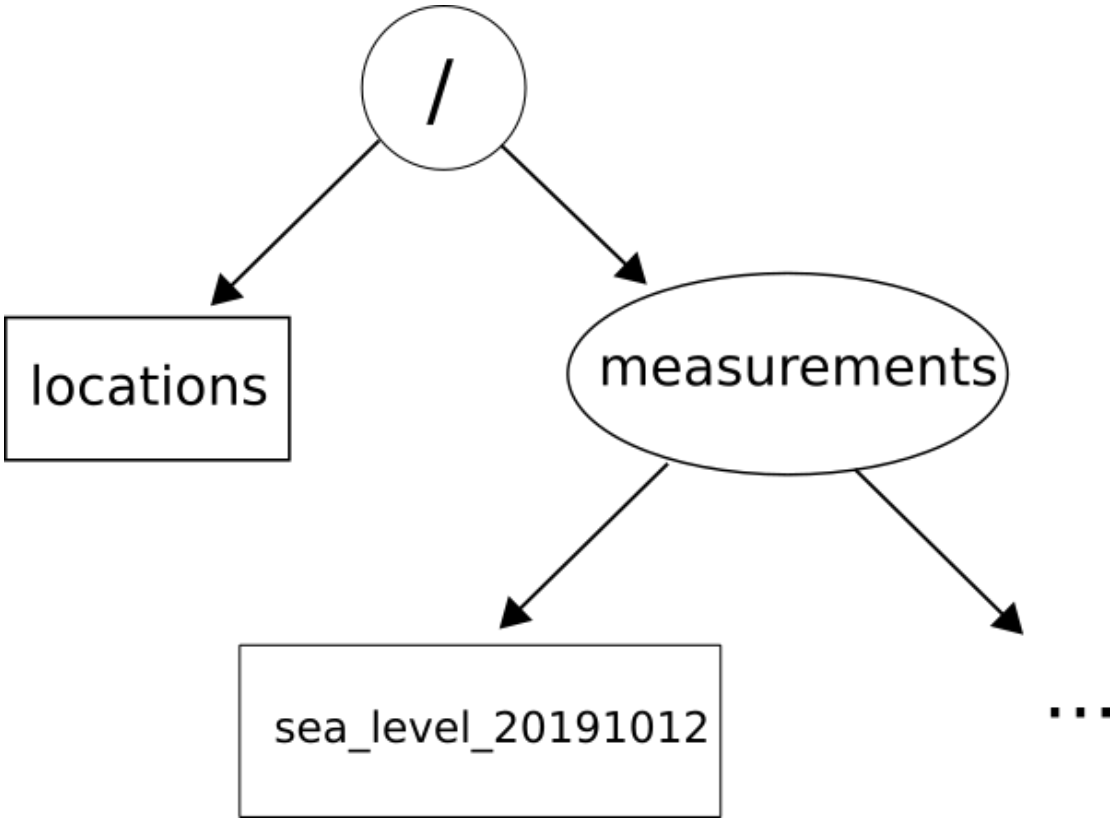
2.7.1 HDF5

HDF5 is the current version of the Hierachical Data Format (HDF), and is commonly used to store large volumes of scientific data, such as experimental results or measurements. An HDF5 file contains two kinds of entities organised in a hierarchy, similar to a filesystem.

- **Datasets** contain scalar or array values. Each dataset has a type, such as integer, floating-point or string.
- **Groups** contain datasets or other groups, much like directories contain files and directories.

Both datasets and groups can have **attributes** associated with them, which provide metadata about the contents.

For example, let's imagine we are trying to store some measurements of sea level at different locations and dates. One way to organise it is shown in the image below:



We will store the locations of our sampling points in a dataset called `locations`, and the actual results in a group called `measurements`. Within that group, we will have a dataset for each date we took samples

on, which will contain results for all locations on that date. For instance, if we are collecting data from N locations at T times per day, each dataset will be a $N \times T$ array of numerical values (integer or floating-point, depending on how we want to record it).

One of the strengths of the HDF5 format is that a file can contain disparate kinds of data, of arbitrary size and types. The attributes provide additional information about the meaning or provenance of the data, and can even link to other datasets and groups within the file.

Working with HDF5 files

Unlike CSV or JSON files, which contain plain text, HDF5 is a **binary file** format. This means that the information stored there is encoded in a more complex way, and cannot be shown or edited using a simple text editor. Instead, to inspect the contents of an HDF5 file, we must use a more specialised application which "knows" how to to read the encoded information. One such application is HDFView.

An alternative is to interact with files **programmatically** - that is, use some code to read or write HDF5 files. Doing this from scratch would be tricky, but there are various libraries that let you interact with an HDF5 file from within your program. You can see examples of basic tasks in various programming languages, including Python, in the documentation pages of the HDF5 standard.

Accessing HDF5 files with Python

Let's now see an example of creating and reading an HDF5 file with Python. In line with the above, we will use the `h5py` library that gives us all the functionality we need.

We'll be creating a file that follows the structure of the climate example mentioned earlier.

The first thing we need to do is install the library. This can be done from the terminal, with the command

```
pip install h5py
```

Some distributions (like Anaconda) already include this library by default, in which case this command will not do anything except report that the library is already installed.

Once installed, we must import it in our file like any other library:

```
In [1]: import h5py
```

Let's create a new HDF5 file that mirrors the structure of the above example. We start by creating an object that will represent this file in our program.

```
In [2]: new_file = h5py.File('my_file.hdf5', 'w')
```

In the example, the file contains a dataset named `locations` and a group called `measurements` at the root level. We can add these to our empty file using some of the methods that the file object provides.

```
In [3]: new_file.create_dataset('locations', data=[[55.9548, -3.11], [38.045, 23.999]])

Out[3]: <HDF5 dataset "locations": shape (2, 2), type "<f8">

In [4]: new_file.create_group('measurements')

Out[4]: <HDF5 group "/measurements" (0 members)>
```

Note that the library lets us create empty datasets, which can be populated later. In this case, however, we initialise the dataset with some values at creation using the `data` argument.

The HDF5 file objects behave somewhat like Python dictionaries: we can access the new group with the usual indexing syntax (`[...]`). This next section shows how to do that and how to add a dataset to the group. Here, we add 4 measurements for each location for that day.

```
In [5]: group = new_file['measurements']
        group.create_dataset("sea_level_20191012", data=[[10, 12, 7, 9], [20, 18, 23, 22]])
```

```
Out[5]: <HDF5 dataset "sea_level_20191012": shape (2, 4), type "<i8">
```

When we are done with writing to the file, we must make sure to close it, so that all the changes are written to it (if they have not been already) and any used memory is released:

```
In [6]: new_file.close()
```

There is a different style for reading and writing files, which is safer and saves you the need to close the file after you are finished. We can use this to read a file and iterate over its contents:

```
In [7]: with h5py.File('my_file.hdf5', 'r') as hdf_file:
            # Print out contents of root group
            print("/ contains...")
            for name in hdf_file:
                print(name)
            # Now print out the contents of the measurements group:
            print("/measurements contains...")
            for name in hdf_file['/measurements']:
                print(name)

/ contains…
locations
measurements
/measurements contains…
sea_level_20191012
```

This is similar to the `with open(...)` syntax we use to work with text files - it is another example of a context manager.

There are many more ways you can access a file with `h5py`. If you are interested, you can look at the quick-start guide from its documentation for an overview.

2.7.2 Other formats

HDF5 is used across various scientific fields to store data, but some disciplines tend to use other file formats. Examples of such formats (and the python libraries) that are popular in particular disciplines are DICOM (`pydicom`) for medical imaging, FITS (`astropy.io.fits`) in astronomy, and NetCDF (`netCDF4`) in the geosciences.

The overall points that we have made about HDF5 generally apply to these formats as well. They are binary files which require specific applications, but you can also use various libraries to interact with them programmatically. Some libraries even offer support for multiple related types of files, such as different image formats.

If you often need to work with a particular type of files, try finding a relevant library in your chosen language. If you have not used it before, are you able to read or write a file using it?

2.8 Plotting with Matplotlib

Plotting data is very common and useful in scientific work. Python does not include any plotting functionality in the language itself, but there are various frameworks available for producing plots and visualisations.

In this section, we will look at Matplotlib, one of those frameworks. As the name indicates, it was conceived to provide an interface similar to the MATLAB programming language, but no knowledge of MATLAB is required!

2.8.1 Importing Matplotlib

We import the `pyplot` object from Matplotlib, which provides us with an interface for making figures. We usually abbreviate it.

```
In [1]: from matplotlib import pyplot as plt
```

2.8.2 Notebook magics

When we write:

```
In [2]: %matplotlib inline
```

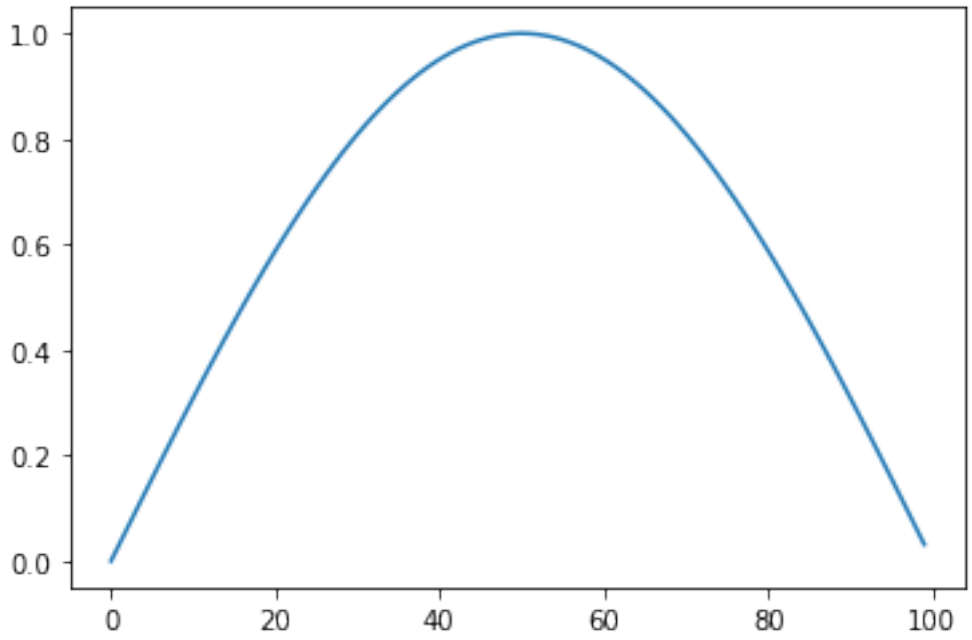
We tell the Jupyter notebook to show figures we generate alongside the code that created it, rather than in a separate window. Lines beginning with a single percent are not python code: they control how the notebook deals with python code.

Lines beginning with two percents are "cell magics", that tell Jupyter notebook how to interpret the particular cell; we've seen `%%writefile`, for example.

2.8.3 A basic plot

When we write:

```
In [3]: from math import sin, cos, pi
        my_fig = plt.plot([sin(pi * x / 100.0) for x in range(100)])
```

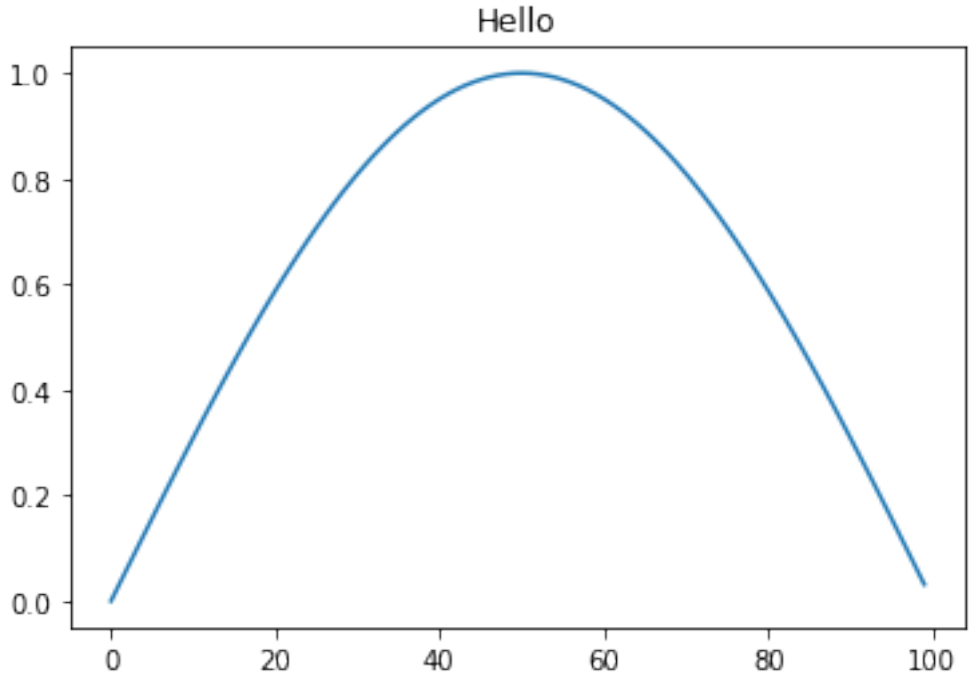


The plot command *returns* a figure, just like the return value of any function. The notebook then displays this.

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 [4]: plt.plot([sin(pi * x / 100.0) for x in range(100)])
        plt.title("Hello")
```

```
Out[4]: Text(0.5, 1.0, 'Hello')
```



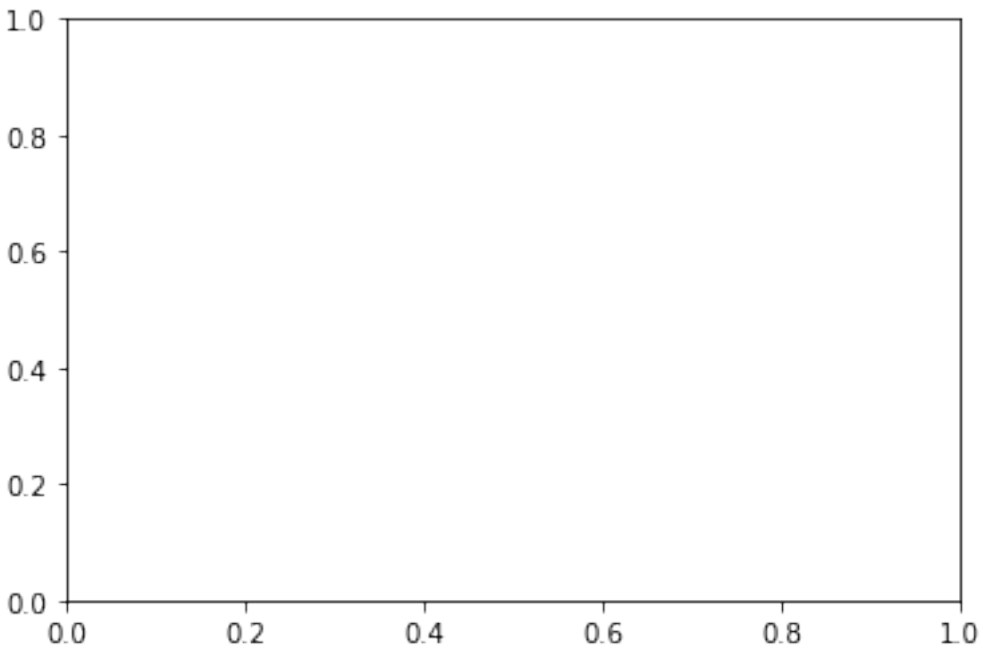
But this requires us to keep all our commands together in a single cell, and makes use of a "global" single "current plot", which, while convenient for quick exploratory sketches, is a bit cumbersome. To produce from our notebook proper plots to use in papers, the library defines some types we can use to treat individual figures as variables, and manipulate these.

2.8.4 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. The relevant function returns a (figure, axis) pair, which we can deal out with parallel assignment (unpacking).

```
In [5]: sine_graph, sine_graph_axes = plt.subplots()
```



Once we have some axes, we can plot a graph on them:

```
In [6]: sine_graph_axes.plot([sin(pi * x / 100.0) for x in range(100)], label='sin(x)')
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7f529e33fa10>]
```

We can add a title to a pair of axes:

```
In [7]: sine_graph_axes.set_title("My graph")
```

```
Out[7]: Text(0.5, 1, 'My graph')
```

```
In [8]: sine_graph_axes.set_ylabel("f(x)")
```

```
Out[8]: Text(3.200000000000003, 0.5, 'f(x)')
```

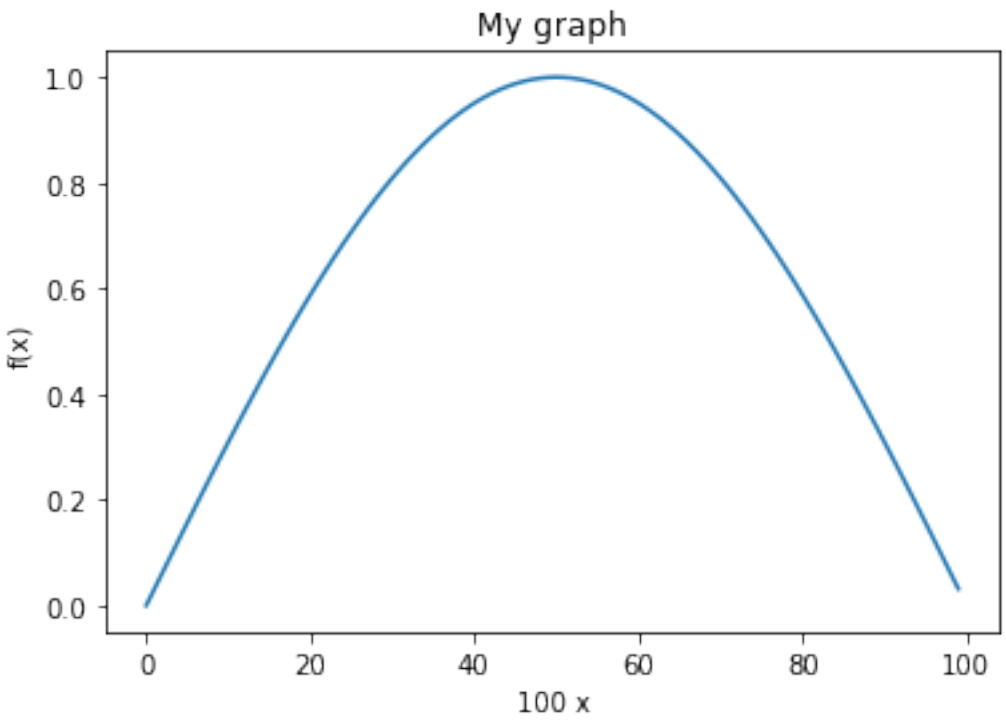
```
In [9]: sine_graph_axes.set_xlabel("100 x")
```

```
Out[9]: Text(0.5, 3.1999999999999993, '100 x')
```

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:

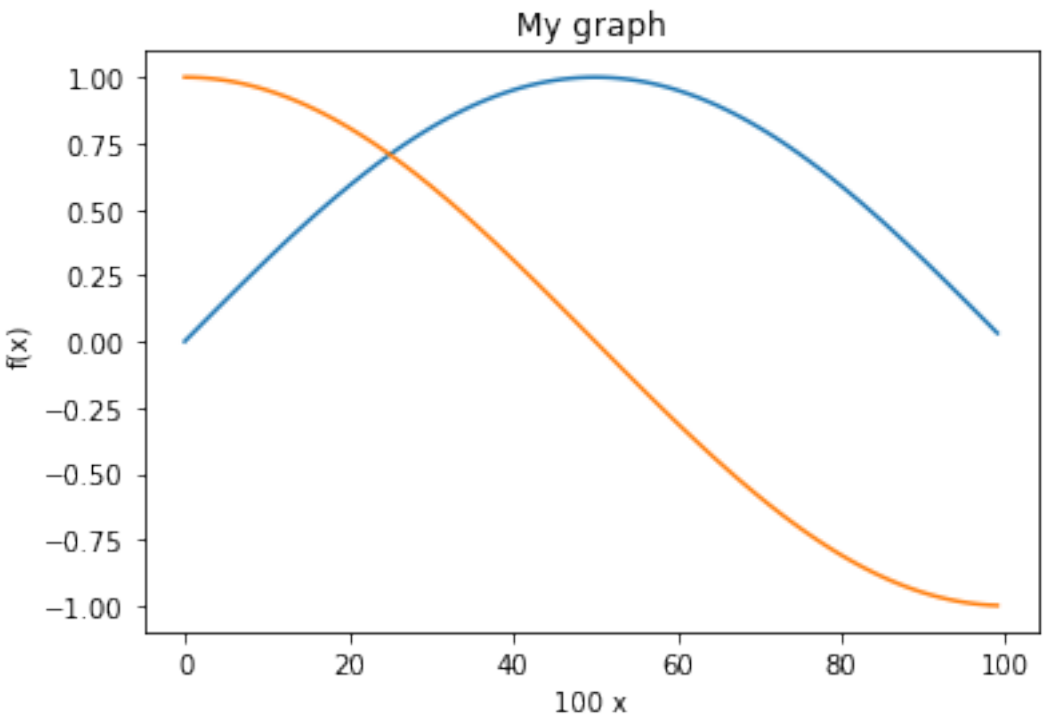
```
In [10]: sine_graph
```

```
Out[10]:
```



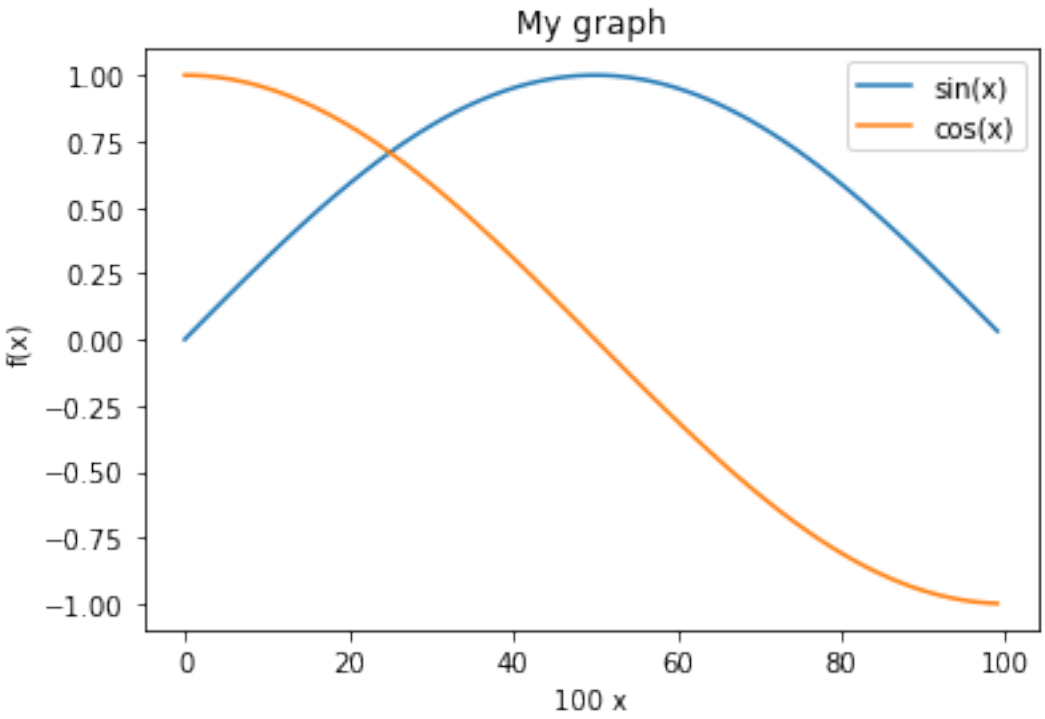
We can add another curve:

```
In [11]: sine_graph_axes.plot([cos(pi * x / 100.0) for x in range(100)], label='cos(x)')

Out[11]: [<matplotlib.lines.Line2D at 0x7f529e2e2650>]

In [12]: sine_graph

Out[12]:
```



A legend will help us distinguish the curves:

```
In [13]: sine_graph_axes.legend()

Out[13]: <matplotlib.legend.Legend at 0x7f529e2e2f10>

In [14]: sine_graph

Out[14]:
```



2.8.5 Saving figures

We must be able to save figures to disk, in order to use them in papers. This is really easy:

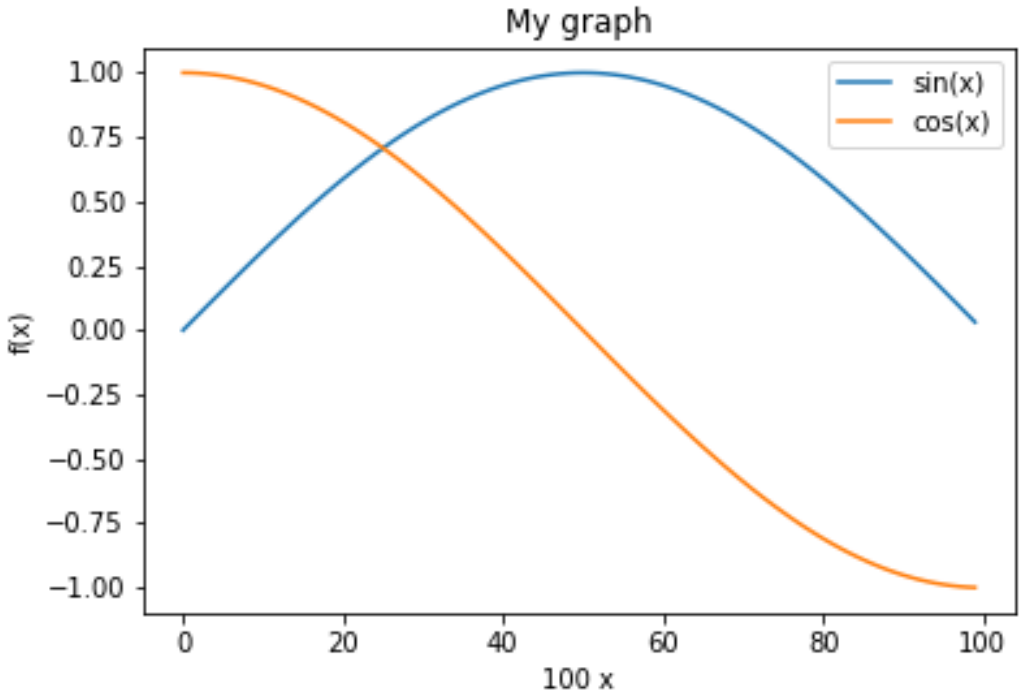
```
In [15]: sine_graph.savefig('my_graph.png')
```

In order to be able to check that it worked, we need to know how to display an arbitrary image in the notebook.

The programmatic way is like this:

```
In [16]: from IPython.display import Image # Get the notebook's own library for manipulating itself.
         Image(filename='my_graph.png')
```

```
Out[16]:
```



2.8.6 Subplots

We might have wanted the sin and cos graphs on separate axes:

```
In [17]: double_graph = plt.figure()
```

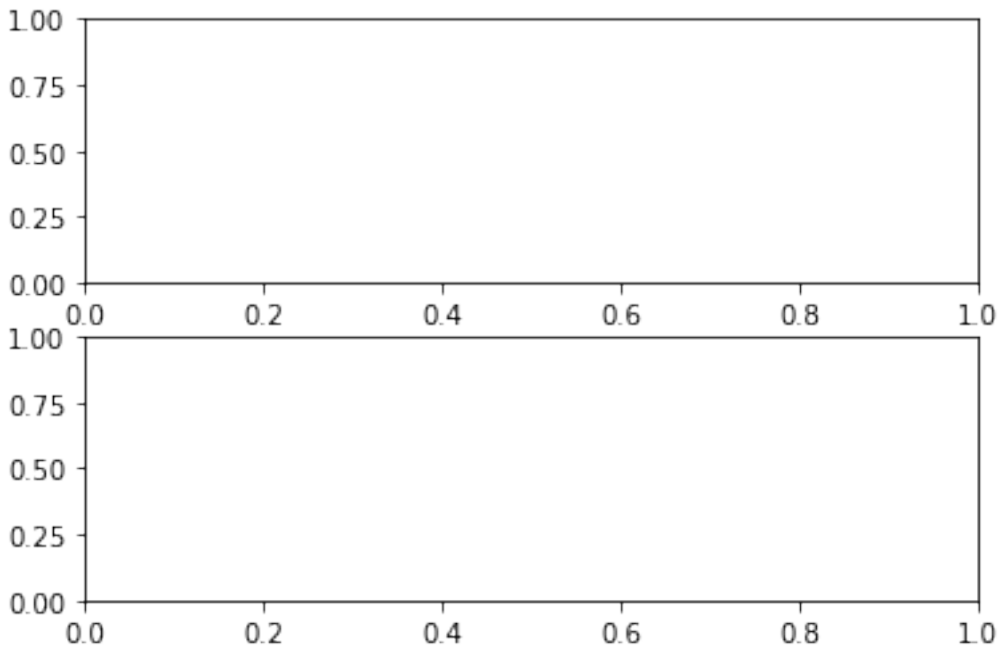
```
<Figure size 432x288 with 0 Axes>
```

```
In [18]: sin_axes = double_graph.add_subplot(2, 1, 1) # 2 rows, 1 column, 1st subplot
```

```
In [19]: cos_axes = double_graph.add_subplot(2, 1, 2) # 2 rows, 1 column, 2nd subplot
```

```
In [20]: double_graph
```

```
Out[20]:
```




```
In [21]: sin_axes.plot([sin(pi * x / 100.0) for x in range(100)])
```

```
Out[21]: [<matplotlib.lines.Line2D at 0x7f529e2f4b90>]
```

```
In [22]: sin_axes.set_ylabel("sin(x)")
```

```
Out[22]: Text(3.200000000000003, 0.5, 'sin(x)')
```

```
In [23]: cos_axes.plot([cos(pi * x / 100.0) for x in range(100)])
```

```
Out[23]: [<matplotlib.lines.Line2D at 0x7f529e23f490>]
```

```
In [24]: cos_axes.set_ylabel("cos(x)")
```

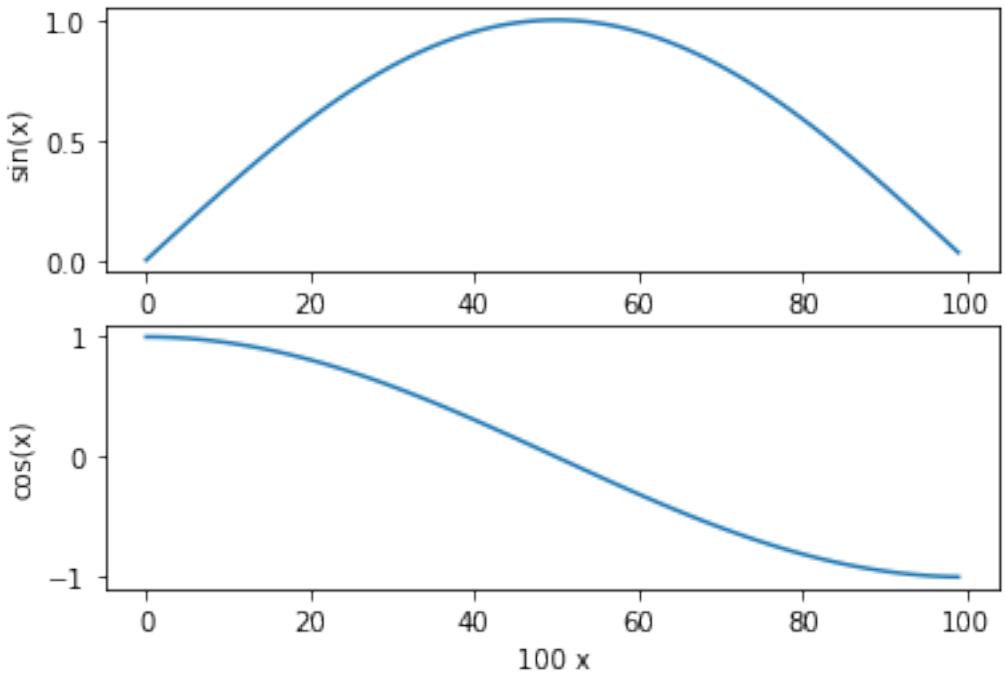
```
Out[24]: Text(3.200000000000003, 0.5, 'cos(x)')
```

```
In [25]: cos_axes.set_xlabel("100 x")
```

```
Out[25]: Text(0.5, 3.200000000000003, '100 x')
```

```
In [26]: double_graph
```

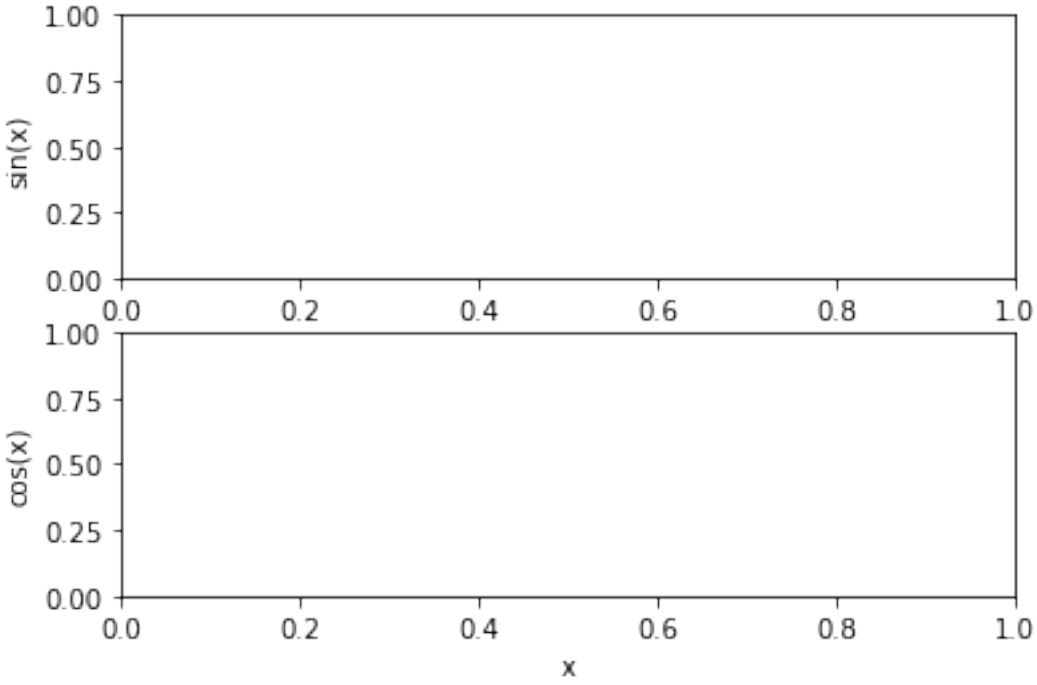
```
Out[26]:
```



2.8.7 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 [27]: 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[27]: Text(0.5, 0, 'x')
```

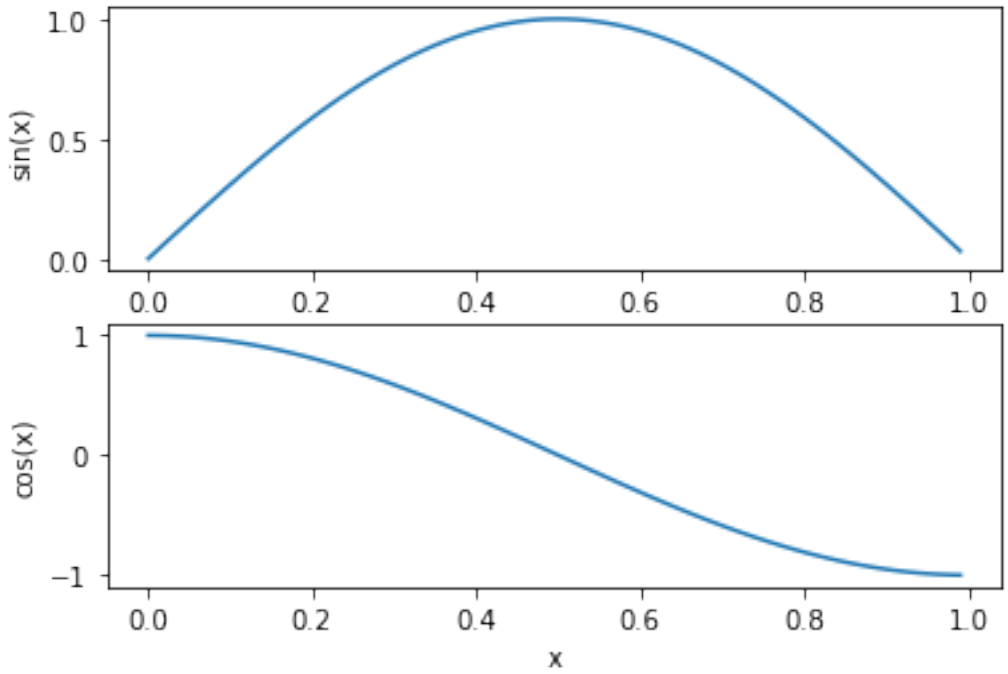


In [28]:
```
sin_axes.plot([x / 100.0 for x in range(100)], [sin(pi * x / 100.0) for x in range(100)])
cos_axes.plot([x / 100.0 for x in range(100)], [cos(pi * x / 100.0) for x in range(100)])
```

Out[28]: [<matplotlib.lines.Line2D at 0x7f529e1d7fd0>]

In [29]: double_graph

Out[29]:



2.8.8 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.

2.9 NumPy

2.9.1 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, Haskell)

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
- 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.

2.9.2 Limitations of Python Lists

The normal Python list is just one dimensional. To make a matrix, we have to nest Python lists:

```
In [1]: x = [list(range(5)) for N in range(5)]

In [2]: x

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

In [3]: x[2][2]

Out[3]: 2
```

Applying an operation to every element is a pain:

```
In [4]: x + 5

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

TypeError                                 Traceback (most recent call last)
```

```
    <ipython-input-4-9e8324a7b754> in <module>
----> 1 x + 5


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

```
In [5]: [[elem + 5 for elem in row] for row in x]
```

```
Out[5]: [[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.

2.9.3 The NumPy array

NumPy's array type represents a multidimensional matrix $M_{i,j,k...n}$

The NumPy array seems at first to be just like a list. For example, we can index it and iterate over it:

```
In [6]: import numpy as np
        my_array = np.array(range(5))
```

```
In [7]: my_array
```

```
Out[7]: array([0, 1, 2, 3, 4])
```

```
In [8]: my_array[2]
```

```
Out[8]: 2
```

```
In [9]: 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 [10]: my_array.append(4)
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-10-b12177763178> 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…

2.9.4 Elementwise Operations

Most operations can be applied element-wise automatically!

```
In [11]: my_array + 2
```

```
Out[11]: array([2, 3, 4, 5, 6])
```

These "vectorized" operations are very fast: (the `%%timeit` magic reports how long it takes to run a cell; there is more information available if interested)

```
In [12]: import numpy as np
         big_list = range(10000)
         big_array = np.arange(10000)
```

```
In [13]: %%timeit
         [x**2 for x in big_list]
```

```
2.99 ms ± 28.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [14]: %%timeit
         big_array**2
```

```
5.03 µs ± 18.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

2.9.5 arange and linspace

NumPy has two methods for quickly defining evenly-spaced arrays of (floating-point) numbers. These can be useful, for example, in plotting.

The first method is `arange`:

```
In [15]: x = np.arange(0, 10, 0.1)  # Start, stop, step size
```

This is similar to Python's `range`, although note that we can't use non-integer steps with the latter!

```
In [16]: y = list(range(0, 10, 0.1))
```

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

TypeError                                 Traceback (most recent call last)

<ipython-input-16-90c31a0aefc9> in <module>
----> 1 y = list(range(0, 10, 0.1))


TypeError: 'float' object cannot be interpreted as an integer
```

The second method is `linspace`:

```
In [17]: import math
         values = np.linspace(0, math.pi, 100)  # Start, stop, number of steps
```

```
In [18]: values
```

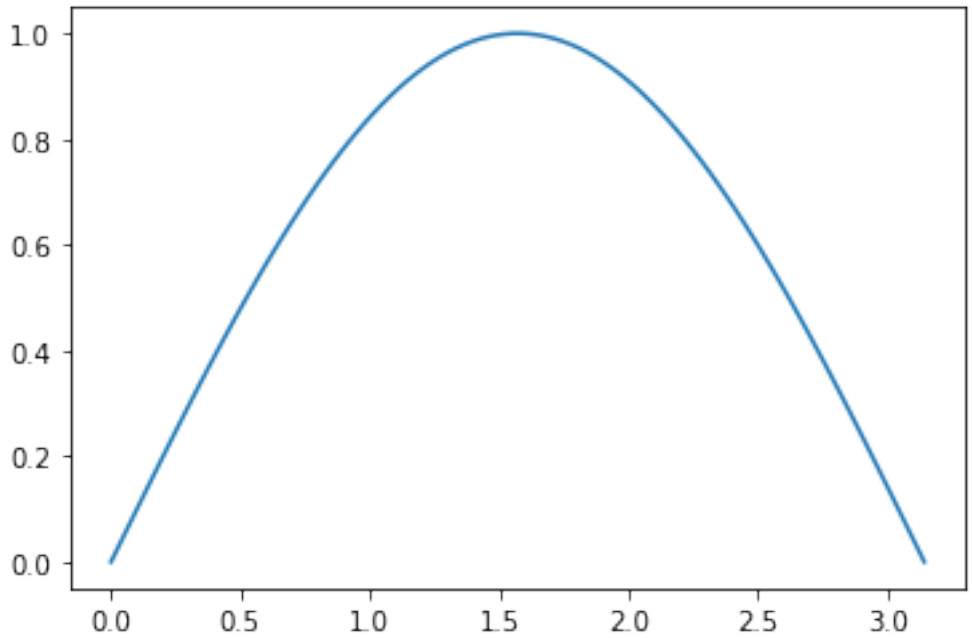
```
Out[18]: array([0.        , 0.03173326, 0.06346652, 0.09519978, 0.12693304,
                0.1586663 , 0.19039955, 0.22213281, 0.25386607, 0.28559933,
                0.31733259, 0.34906585, 0.38079911, 0.41253237, 0.44426563,
                0.47599889, 0.50773215, 0.53946541, 0.57119866, 0.60293192,
                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])
```

Regardless of the method used, the array of values that we get can be used in the same way.

In fact, NumPy comes with "vectorised" versions of common functions which work element-by-element when applied to arrays:

```
In [19]: %matplotlib inline

         from matplotlib import pyplot as plt
         plt.plot(values, np.sin(values))
```

```
Out[19]: [<matplotlib.lines.Line2D at 0x7fb34e88c910>]
```



So we don't have to use awkward list comprehensions when using these.

2.9.6 Multi-Dimensional Arrays

NumPy's true power comes from multi-dimensional arrays:

```
In [20]: np.zeros([3, 4, 2])  # 3 arrays with 4 rows and 2 columns each
```

```
Out[20]: 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 [21]: x = np.array(range(40))
         x
```

```
Out[21]: 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])
```

```
In [22]: y = x.reshape([4, 5, 2])
         y
```

```
Out[22]: 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 [23]: y[3, 2, 1]
```

```
Out[23]: 35
```

Including selecting on inner axes while taking all from the outermost:

```
In [24]: y[:, 2, 1]
```

```
Out[24]: array([ 5, 15, 25, 35])
```

And subselecting ranges:

```
In [25]: y[2:, :1, :]  # Last 2 axes, 1st row, all columns
```

```
Out[25]: array([[[20, 21]],

                [[30, 31]]])
```

And transpose arrays:

```
In [26]: y.transpose()
```

```
Out[26]: 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 [27]: y.shape
```

```
Out[27]: (4, 5, 2)
```

```
In [28]: y.transpose().shape
```

```
Out[28]: (2, 5, 4)
```

Some numpy functions apply by default to the whole array, but can be chosen to act only on certain axes:

```
In [29]: x = np.arange(12).reshape(4,3)
         x
```

```
Out[29]: array([[ 0,  1,  2],
                [ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])
```

```
In [30]: x.mean(1)  # Mean along the second axis, leaving the first.
```

```
Out[30]: array([ 1.,  4.,  7., 10.])
```

```
In [31]: x.mean(0)  # Mean along the first axis, leaving the second.
```

```
Out[31]: array([4.5, 5.5, 6.5])
```

```
In [32]: x.mean()  # mean of all axes
```

```
Out[32]: 5.5
```

2.9.7 Array Datatypes

A Python `list` can contain data of mixed type:

```
In [33]: x = ['hello', 2, 3.4]

In [34]: type(x[2])

Out[34]: float

In [35]: type(x[1])

Out[35]: int
```

A NumPy array always contains just one datatype:

```
In [36]: np.array(x)

Out[36]: array(['hello', '2', '3.4'], dtype='<U5')
```

NumPy will choose the least-generic-possible datatype that can contain the data:

```
In [37]: y = np.array([2, 3.4])

In [38]: y

Out[38]: array([2. , 3.4])
```

You can access the array's `dtype`, or check the type of individual elements:

```
In [39]: y.dtype

Out[39]: dtype('float64')

In [40]: type(y[0])

Out[40]: numpy.float64

In [41]: z = np.array([3, 4, 5])
         z

Out[41]: array([3, 4, 5])

In [42]: type(z[0])

Out[42]: numpy.int64
```

The results 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 [43]: x = [2, 3.4, 7.2, 0]

In [44]: int_array = np.array(x, dtype=int)

In [45]: float_array = np.array(x, dtype=float)

In [46]: int_array

Out[46]: array([2, 3, 7, 0])

In [47]: float_array

Out[47]: array([2. , 3.4, 7.2, 0. ])

In [48]: int_array.dtype

Out[48]: dtype('int64')

In [49]: float_array.dtype

Out[49]: dtype('float64')
```

2.9.8 Broadcasting

This is another really powerful feature of NumPy.

By default, array operations are element-by-element:

```
In [50]: np.arange(5) * np.arange(5)
```

```
Out[50]: array([ 0,  1,  4,  9, 16])
```

If we multiply arrays with non-matching shapes we get an error:

```
In [51]: np.arange(5) * np.arange(6)
```

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

ValueError                                Traceback (most recent call last)

<ipython-input-51-d87da4b8a218> in <module>
----> 1 np.arange(5) * np.arange(6)


ValueError: operands could not be broadcast together with shapes (5,) (6,)
```

```
In [52]: np.zeros([2,3]) * np.zeros([2,4])
```

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

ValueError                                Traceback (most recent call last)

<ipython-input-52-b6b30bdbcb53> 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 [53]: m1 = np.arange(100).reshape([10, 10])
```

```
In [54]: m2 = np.arange(100).reshape([10, 5, 2])
```

```
In [55]: m1 + m2
```

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

ValueError                                Traceback (most recent call last)

<ipython-input-55-92db99ada483> 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 [56]: np.ones([3, 3]) * np.ones([3, 3]) # Note elementwise multiply, *not* matrix multiply.
```

```
Out[56]: 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 [57]: col = np.arange(10).reshape([10, 1])
         col
```

```
Out[57]: array([[0],
                [1],
                [2],
                [3],
                [4],
                [5],
                [6],
                [7],
                [8],
                [9]])
```

```
In [58]: row = col.transpose()
         row
```

```
Out[58]: array([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

```
In [59]: col.shape # "Column Vector"
```

```
Out[59]: (10, 1)
```

```
In [60]: row.shape # "Row Vector"
```

```
Out[60]: (1, 10)
```

```
In [61]: row + col
```

```
Out[61]: 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 [62]: 10 * row + col
```

```
Out[62]: array([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
                [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
                [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
                [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
                [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
                [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
                [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
                [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
                [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
                [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]])
```

This works for arrays with more than one unit dimension.

2.9.9 Newaxis

Broadcasting is very powerful, and numpy allows indexing with `np.newaxis` to temporarily create new one-long dimensions on the fly.

```
In [63]: import numpy as np
         x = np.arange(10).reshape(2, 5)
         y = np.arange(8).reshape(2, 2, 2)

In [64]: x

Out[64]: array([[0, 1, 2, 3, 4],
                [5, 6, 7, 8, 9]])

In [65]: y

Out[65]: array([[[0, 1],
                 [2, 3]],

                [[4, 5],
                 [6, 7]]])

In [66]: x[:, :, np.newaxis, np.newaxis].shape

Out[66]: (2, 5, 1, 1)

In [67]: y[:, np.newaxis, :, :].shape

Out[67]: (2, 1, 2, 2)

In [68]: res = x[:, :, np.newaxis, np.newaxis] * y[:, np.newaxis, :, :]

In [69]: res.shape

Out[69]: (2, 5, 2, 2)

In [70]: np.sum(res)

Out[70]: 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 [71]: threebythree = np.arange(9).reshape(3, 3)
         threebythree

Out[71]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [72]: threebythree[:, np.newaxis, :]

Out[72]: array([[[0, 1, 2]],

                [[3, 4, 5]],

                [[6, 7, 8]]])
```

2.9.10 Dot Products

NumPy multiply is element-by-element, not a dot-product:

```
In [73]: a = np.arange(9).reshape(3, 3)
         a

Out[73]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])

In [74]: b = np.arange(3, 12).reshape(3, 3)
         b

Out[74]: array([[ 3,  4,  5],
                [ 6,  7,  8],
                [ 9, 10, 11]])

In [75]: a * b

Out[75]: array([[ 0,  4, 10],
                [18, 28, 40],
                [54, 70, 88]])
```

To get a dot-product, (matrix inner product) we can use a built in function:

```
In [76]: np.dot(a, b)

Out[76]: array([[ 24,  27,  30],
                [ 78,  90, 102],
                [132, 153, 174]])
```

Though it is possible to represent this in the algebra of broadcasting and newaxis:

```
In [77]: a[:, :, np.newaxis].shape

Out[77]: (3, 3, 1)

In [78]: b[np.newaxis, :, :].shape

Out[78]: (1, 3, 3)

In [79]: a[:, :, np.newaxis] * b[np.newaxis, :, :]

Out[79]: array([[[ 0,  0,  0],
                 [ 6,  7,  8],
                 [18, 20, 22]],

                [[ 9, 12, 15],
                 [24, 28, 32],
                 [45, 50, 55]],

                [[18, 24, 30],
                 [42, 49, 56],
                 [72, 80, 88]]])

In [80]: (a[:, :, np.newaxis] * b[np.newaxis, :, :]).sum(1)
```

```
Out[80]: array([[ 24,  27,  30],
                [ 78,  90, 102],
                [132, 153, 174]])
```

Or if you prefer:

```
In [81]: (a.reshape(3, 3, 1) * b.reshape(1, 3, 3)).sum(1)

Out[81]: array([[ 24,  27,  30],
                [ 78,  90, 102],
                [132, 153, 174]])
```

We use broadcasting to generate $A_{ij}B_{jk}$ as a 3-d matrix:

```
In [82]: a.reshape(3, 3, 1) * b.reshape(1, 3, 3)

Out[82]: array([[[ 0,  0,  0],
                 [ 6,  7,  8],
                 [18, 20, 22]],

                [[ 9, 12, 15],
                 [24, 28, 32],
                 [45, 50, 55]],

                [[18, 24, 30],
                 [42, 49, 56],
                 [72, 80, 88]]])
```

Then we sum over the middle, j axis, [which is the 1-axis of three axes numbered (0,1,2)] of this 3-d matrix. Thus we generate $\Sigma_j A_{ij}B_{jk}$.

We can see that the broadcasting concept gives us a powerful and efficient way to express many linear algebra operations computationally.

2.9.11 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 [83]: x = np.arange(50).reshape([10, 5])

In [84]: record_x = x.view(dtype={'names': ["col1", "col2", "another", "more", "last"],
                                  'formats': [int]*5 })

In [85]: record_x

Out[85]: 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'), ('last', '<
```

Record arrays can be addressed with field names like they were a dictionary:

```
In [86]: record_x['col1']

Out[86]: array([[ 0],
                [ 5],
                [10],
                [15],
                [20],
                [25],
                [30],
                [35],
                [40],
                [45]])
```

We've seen these already when we used NumPy's CSV parser.

2.9.12 Logical arrays, masking, and selection

Numpy defines operators like == and < to apply to arrays *element by element*:

```
In [87]: x = np.zeros([3, 4])
         x

Out[87]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])

In [88]: y = np.arange(-1, 2)[:, np.newaxis] * np.arange(-2, 2)[np.newaxis, :]
         y

Out[88]: array([[ 2,  1,  0, -1],
                [ 0,  0,  0,  0],
                [-2, -1,  0,  1]])

In [89]: iszero = x == y
         iszero

Out[89]: array([[False, False,  True, False],
                [ True,  True,  True,  True],
                [False, False,  True, False]])
```

A logical array can be used to select elements from an array:

```
In [90]: y[np.logical_not(iszero)]

Out[90]: 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 [91]: y[iszero] = 5

In [92]: y

Out[92]: array([[ 2,  1,  5, -1],
                [ 5,  5,  5,  5],
                [-2, -1,  5,  1]])
```

2.9.13 Numpy memory

Numpy memory management can be tricksy:

```
In [93]: x = np.arange(5)
         y = x[:]

In [94]: y[2] = 0
         x

Out[94]: array([0, 1, 0, 3, 4])
```

It does **not** behave like lists!

```
In [95]: x = list(range(5))
         y = x[:]

In [96]: y[2] = 0
         x

Out[96]: [0, 1, 2, 3, 4]
```

We must use `np.copy` to force separate memory. Otherwise NumPy tries its 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.

2.10 The Boids!

This section shows an example of using NumPy to encode a model of how a group of birds or other animals moves. It is based on a paper by Craig W. Reynolds. Reynolds calls the simulated creatures "bird-oids" or "boids", so that's what we'll be calling them here too.

2.10.1 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

The model includes three main behaviours which, together, give rise to "flocking". In the words of the 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

2.10.2 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 positions, 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 = 10
```

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, 2000])
```

```
In [4]: positions = np.random.rand(2, boid_count) * limits[:, np.newaxis]
        positions
```

```
Out[4]: array([[1474.93317535,  733.56082642,  603.89226756, 1068.09418858,
                1909.47295937, 1448.26453873, 1385.69037974, 1422.21900296,
                1237.3020266 , 1223.18088506],
               [ 531.71626086,  224.61963298, 1128.38236063, 1899.68060332,
                 218.33774488,  623.68293596,  408.41244217,  307.1306393 ,
                1653.22006796,   40.7833536 ]])
```

```
In [5]: positions.shape
```

```
Out[5]: (2, 10)
```

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.

```
In [6]: limits[:, np.newaxis]
```

```
Out[6]: array([[2000],
               [2000]])
```

```
In [7]: limits[:, np.newaxis].shape
```

```
Out[7]: (2, 1)
```

```
In [8]: np.random.rand(2, boid_count).shape
```

```
Out[8]: (2, 10)
```

So we multiply a 2×1 array by a 2×10 array – and get a 2×10 array.

Let's put that in a function:

```
In [9]: 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])
```

For example, let's assume that we want our initial positions to vary between 100 and 200 in the x axis, and 900 and 1100 in the y axis. We can generate random positions within these constraints with:

```
positions = new_flock(boid_count, np.array([100, 900]), np.array([200, 1100]))
```

But each bird will also need a starting velocity. Let's make these random too:

We can reuse the `new_flock` function defined above, since we're again essentially just generating random numbers from given limits. This saves us some code, but keep in mind that using a function for something other than what its name indicates can become confusing!

Here, we will let the initial x velocities range over $[0, 10]$ and the y velocities over $[-20, 20]$.

```
In [10]: velocities = new_flock(boid_count, np.array([0, -20]), np.array([10, 20]))
         velocities
```

```
Out[10]: array([[  3.1334097 ,   9.23996495,   4.85533178,   2.0087251 ,
                   3.09118637,   3.97648827,   2.47980135,   3.1137484 ,
                   4.52734415,   7.84662154],
                [-17.39973478,  -9.06848933, -13.95844304,  -5.97436454,
                  18.50265571, -10.07741974,   3.07803813,  -5.64727463,
                 -16.45097077,   4.74645941]])
```

2.10.3 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 [11]: positions += velocities
```

2.10.4 Matplotlib Animations

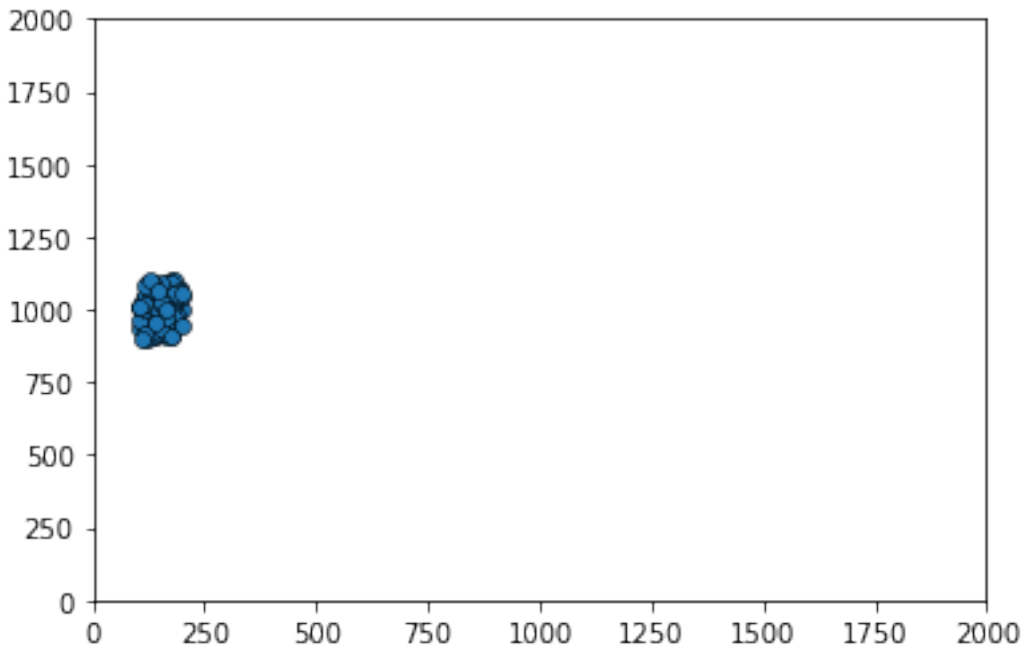
So now we can animate our Boids using the matplotlib animation tools. All we have to do is import the relevant libraries:

```
In [12]: from matplotlib import animation
         from matplotlib import pyplot as plt
         %matplotlib inline
```

Then, we make a static plot, showing our first frame:

```
In [13]: # create a simple plot
         # initial x position in [100, 200], initial y position in [900, 1100]
         # initial x velocity in [0, 10], initial y velocity in [-20, 20]
         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[1]))
         scatter = axes.scatter(positions[0, :], positions[1, :],
                                marker='o', edgecolor='k', lw=0.5)
         scatter
```

```
Out[13]: <matplotlib.collections.PathCollection at 0x7f33a8fb5550>
```



Then, we define a function which **updates** the figure for each timestep

```
In [14]: 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 [15]: anim = animation.FuncAnimation(figure, animate,
                                        frames=50, interval=50)
```

Save out the figure:

```
In [16]: 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.save('boids_1.mp4')
```

And download the saved animation.
You can even view the results directly in the notebook.

```
In [17]: from IPython.display import HTML
         HTML(anim.to_jshtml())

Out[17]: <IPython.core.display.HTML object>
```

2.10.5 Fly towards the middle

Boids try to fly towards the middle:

```
In [18]: 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 [19]: positions

Out[19]: array([[ 110.76403737,  114.58446977,  197.15148455,  166.00519123],
                [ 971.96519887,  970.71735729,  974.76485496, 1081.25840349]])

In [20]: velocities

Out[20]: array([[  3.42436633,   4.66224869,   7.83399023,   0.22552898],
                [-18.0218543 ,  12.55140088, -10.88869671,  10.53076395]])

In [21]: middle = np.mean(positions, 1)
         middle

Out[21]: array([147.12629573, 999.67645365])

In [22]: direction_to_middle = positions - middle[:, np.newaxis]
         direction_to_middle

Out[22]: array([[-36.36225836, -32.54182596,  50.02518882,  18.8788955 ],
                [-27.71125478, -28.95909636, -24.91159869,  81.58194984]])
```

This is easier and faster than:

```
for bird in birds:
    for dimension in [0, 1]:
        direction_to_middle[dimension][bird] = positions[dimension][bird] - middle[dimension]
```

```
In [23]: move_to_middle_strength = 0.01
         velocities = velocities - direction_to_middle * move_to_middle_strength
```

Let's update our function, and animate that:

```
In [24]: 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
             positions += velocities

In [25]: def animate(frame):
             update_boids(positions, velocities)
             scatter.set_offsets(positions.transpose())

In [26]: anim = animation.FuncAnimation(figure, animate,
                                        frames=50, interval=50)

In [27]: positions = new_flock(100, np.array([100, 900]), np.array([200, 1100]))
         velocities = new_flock(100, np.array([0, -20]), np.array([10, 20]))
         HTML(anim.to_jshtml())

Out[27]: <IPython.core.display.HTML object>
```

2.10.6 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 [28]: 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 [29]: xpos = positions[0, :]

In [30]: xsep_matrix = xpos[:, np.newaxis] - xpos[np.newaxis, :]

In [31]: xsep_matrix.shape

Out[31]: (4, 4)

In [32]: xsep_matrix

Out[32]: array([[  0.        ,  26.23838502,  18.13815663,  86.99320595],
                [-26.23838502,   0.        ,  -8.10022839,  60.75482093],
                [-18.13815663,   8.10022839,   0.        ,  68.85504932],
                [-86.99320595, -60.75482093, -68.85504932,   0.        ]])
```

But in NumPy we can be cleverer than that, and make a $2 \times N \times N$ matrix of separations:

```
In [33]: separations = positions[:, np.newaxis, :] - positions[:, :, np.newaxis]

In [34]: separations.shape

Out[34]: (2, 4, 4)
```

And then we can get the sum-of-squares $\delta_x^2 + \delta_y^2$ like this:

```
In [35]: squared_displacements = separations * separations

In [36]: square_distances = np.sum(squared_displacements, 0)

In [37]: square_distances

Out[37]: array([[    0.        ,  3055.58015707,  9056.13938541,  8086.08751597],
                [ 3055.58015707,     0.        , 20250.16097203,  4361.31591644],
                [ 9056.13938541, 20250.16097203,     0.        , 18239.91038846],
                [ 8086.08751597,  4361.31591644, 18239.91038846,     0.        ]])
```

Now we need to find birds that are too close:

```
In [38]: alert_distance = 2000
         close_birds = square_distances < alert_distance
         close_birds

Out[38]: array([[ True, False, False, False],
                [False,  True, False, False],
                [False, False,  True, False],
                [False, False, False,  True]])
```

Find the direction distances **only** to those birds which are too close:

```
In [39]: separations_if_close = np.copy(separations)
         far_away = np.logical_not(close_birds)
```

Set `x` and `y` values in `separations_if_close` to zero if they are far away:

```
In [40]: separations_if_close[0, :, :][far_away] = 0
         separations_if_close[1, :, :][far_away] = 0
         separations_if_close
```

```
Out[40]: 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.],
                 [0., 0., 0., 0.],
                 [0., 0., 0., 0.]]])
```

And fly away from them:

```
In [41]: np.sum(separations_if_close, 2)
```

```
Out[41]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.]])
```

```
In [42]: velocities = velocities + np.sum(separations_if_close, 2)
```

Now we can update our animation:

```
In [43]: 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 [44]: def animate(frame):
             update_boids(positions, velocities)
             scatter.set_offsets(positions.transpose())


         anim = animation.FuncAnimation(figure, animate,
                                        frames=50, interval=50)

         positions = new_flock(100, np.array([100, 900]), np.array([200, 1100]))
         velocities = new_flock(100, np.array([0, -20]), np.array([10, 20]))
         HTML(anim.to_jshtml())
```

```
Out[44]: <IPython.core.display.HTML object>
```

2.10.7 Match speed with nearby birds

This is pretty similar:

```
In [45]: 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_strength

             positions += velocities
```

```
In [46]: def animate(frame):
             update_boids(positions, velocities)
             scatter.set_offsets(positions.transpose())


         anim = animation.FuncAnimation(figure, animate,
                                        frames=200, interval=50)


         positions = new_flock(100, np.array([100, 900]), np.array([200, 1100]))
         velocities = new_flock(100, np.array([0, -20]), np.array([10, 20]))
         HTML(anim.to_jshtml())
```

```
Out[46]: <IPython.core.display.HTML object>
```

Hopefully the power of NumPy should be pretty clear now. This would be **enormously slower** and, I think, harder to understand using traditional lists.

2.11 Recap: Understanding the "Greengraph" Example

We now know enough to understand everything we did in the initial example chapter on the "Greengraph" (notebook). 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. Remember that it is generally a better idea to create files in an editor or integrated development environment (IDE) rather than through the notebook!

2.11.1 Classes for Greengraph

The original example was written as a collection of functions. Alternatively, we can rewrite it in an object-oriented style, using classes to group related functionality.

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.Yandex(lang="en_US")

    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
import imageio as img
import requests

class Map(object):
    def __init__(self, lat, long, satellite=True, zoom=10,
                 size=(400, 400), sensor=False):
        base = "https://static-maps.yandex.ru/1.x/?"

        params = dict(
```

```
                z=zoom,
                size=str(size[0]) + "," + str(size[1]),
                ll=str(long) + "," + str(lat),
                l="sat" if satellite else "map",
                lang="en_US"
            )

            self.image = requests.get(
                base, params=params).content  # Fetch our PNG image data
            content = BytesIO(self.image)
            self.pixels = img.imread(content) # Parse our PNG image as a numpy array

        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.imwrite(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
```

2.11.2 Invoking our code and making a plot

```
In [5]: from matplotlib import pyplot as plt
        from greengraph import Greengraph
        %matplotlib inline

        mygraph = Greengraph('New York', 'Chicago')
        data = mygraph.green_between(20)
```

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

        HTTPError                                 Traceback (most recent call last)

        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
        354         try:
    --> 355             page = requester(req, timeout=timeout, **kwargs)
```

```
    356         except Exception as error:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, 
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-5-a69e6d6508d4> in <module>
      4
      5 mygraph = Greengraph('New York', 'Chicago')
----> 6 data = mygraph.green_between(20)


    ~/build/UCL/rsd-engineeringcourse/ch01data/greengraph/graph.py in green_between(self, steps)
     21         return [Map(*location).count_green()
     22                 for location in self.location_sequence(
---> 23                     self.geolocate(self.start),
     24                     self.geolocate(self.end),
     25                     steps)]
```

```
~/build/UCL/rsd-engineeringcourse/ch01data/greengraph/graph.py in geolocate(self, place)
 11
 12     def geolocate(self, place):
---> 13         return self.geocoder.geocode(place, exactly_one=False)[0][1]
 14
 15     def location_sequence(self, start, end, steps):


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
117             exactly_one,
118         )


~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
371                                 exc_info=False)
372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
374                 except KeyError:
375                     raise GeocoderServiceError(message)


GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

In [6]:
```
plt.plot(data)
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-6-727d88478626> in <module>
----> 1 plt.plot(data)


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

2.12 Introduction

2.12.1 What's version control?

Version control is a tool for **managing changes** to a set of files.

There are many different **version control systems**:

- Git
- Mercurial (`hg`)
- CVS
- Subversion (`svn`)
- …

2.12.2 Why use version control?

- Better kind of **backup**.
- Review **history** ("When did I introduce this bug?").
- Restore older **code versions**.
- Ability to **undo mistakes**.
- Maintain **several versions** of the code at a time.

Git is also a **collaborative** tool:

- "How can I share my code?"
- "How can I submit a change to someone else's code?"
- "How can I merge my work with Sue's?"

2.12.3 Git != GitHub

- **Git**: version control system tool to manage source code history.
- **GitHub**: hosting service for Git repositories.

2.12.4 How do we use version control?

Do some programming, then commit our work:
`my_vcs commit`
Program some more.
Spot a mistake:
`my_vcs rollback`
Mistake is undone.

2.12.5 What is version control? (Team version)

Graham	Eric
`my_vcs commit`	…
…	Join the team
…	`my_vcs checkout`
…	Do some programming
…	`my_vcs commit`
`my_vcs update`	…
Do some programming	Do some programming
`my_vcs commit`	…
`my_vcs update`	…
`my_vcs merge`	…
`my_vcs commit`	…

2.12.6 Scope

This course will use the `git` version control system, but much of what you learn will be valid with other version control tools you may encounter, including subversion (`svn`) and mercurial (`hg`).

2.13 Practising with Git

2.13.1 Example Exercise

In this course, we will use, as an example, the development of a few text files containing a description of a topic of your choice.

This could be your research, a hobby, or something else. In the end, we will show you how to display the content of these files as a very simple website.

2.13.2 Programming and documents

The purpose of this exercise is to learn how to use Git to manage program code you write, not simple text website content, but we'll just use these text files instead of code for now, so as not to confuse matters with trying to learn version control while thinking about programming too.

In later parts of the course, you will use the version control tools you learn today with actual Python code.

2.13.3 Markdown

The text files we create will use a simple "wiki" markup style called markdown to show formatting. This is the convention used in this file, too.

You can view the content of this file in the way Markdown renders it by looking on the web, and compare the raw text.

2.13.4 Displaying Text in this Tutorial

This tutorial is based on use of the Git command line. So you'll be typing commands in the shell.

To make it easy for me to edit, I've built it using Jupyter notebook.

Commands you can type will look like this, using the %%bash "magic" for the notebook.

```
In [1]: %%bash
        echo some output
```

```
some output
```

with the results you should see below.

In this document, we will show the new content of an edited document like this:

```
In [2]: %%writefile somefile.md
        Some content here
```

```
Writing somefile.md
```

But if you are following along, you should edit the file using a text editor. On either Windows, Mac or Linux, we recommend VS Code.

2.13.5 Setting up somewhere to work

```
In [3]: %%bash
        rm -rf learning_git/git_example # Just in case it's left over from a previous class; you won't
        mkdir -p learning_git/git_example
        cd learning_git/git_example
```

I just need to move this Jupyter notebook's current directory as well:

```
In [4]: import os
        top_dir = os.getcwd()
        top_dir
```

```
Out[4]: '/home/travis/build/UCL/rsd-engineeringcourse/ch02git'
```

```
In [5]: git_dir = os.path.join(top_dir, 'learning_git')
        git_dir
```

```
Out[5]: '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git'
```

```
In [6]: working_dir=os.path.join(git_dir, 'git_example')
```

```
In [7]: os.chdir(working_dir)
```

2.14 Solo work

2.14.1 Configuring Git with your name and email

First, we should configure Git to know our name and email address:

```
In [8]: %%bash
        git config --global user.name "Lancelot the Brave"
        git config --global user.email "l.brave@spamalot.uk"
```

2.14.2 Initialising the repository

Now, we will tell Git to track the content of this folder as a git "repository".

```
In [9]: %%bash
        pwd # Note where we are standing-- MAKE SURE YOU INITIALISE THE RIGHT FOLDER
        git init
```

```
/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_example
Initialized empty Git repository in /home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/g
```

As yet, this repository contains no files:

```
In [10]: %%bash
         ls
```

```
In [11]: %%bash
         git status
```

```
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

2.15 Solo work with Git

So, we're in our git working directory:

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)
        working_dir
```

```
Out[1]: '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_example'
```

2.15.1 A first example file

So let's create an example file, and see how to start to manage a history of changes to it.

```
<my editor> index.md # Type some content into the file.
```

```
In [2]: %%writefile index.md
        Mountains in the UK
        ===================
        England is not very mountainous.
        But has some tall hills, and maybe a mountain or two depending on your definition.
```

```
Writing index.md
```

```
In [3]: cat index.md
```

```
Mountains in the UK
===================
England is not very mountainous.
But has some tall hills, and maybe a mountain or two depending on your definition.
```

2.15.2 Telling Git about the File

So, let's tell Git that `index.md` is a file which is important, and we would like to keep track of its history:

```
In [4]: %%bash
        git add index.md
```

Don't forget: Any files in repositories which you want to "track" need to be added with `git add` after you create them.

2.15.3 Our first commit

Now, we need to tell Git to record the first version of this file in the history of changes:

```
In [5]: %%bash
        git commit -m "First commit of discourse on UK topography"
```

```
[master (root-commit) 8db2c9c] First commit of discourse on UK topography
 1 file changed, 4 insertions(+)
 create mode 100644 index.md
```

And note the confirmation from Git.
There's a lot of output there you can ignore for now.

2.15.4 Configuring Git with your editor

If you don't type in the log message directly with -m "Some message", then an editor will pop up, to allow you to edit your message on the fly.

For this to work, you have to tell git where to find your editor.

```
In [6]: %%bash
        git config --global core.editor vim
```

You can find out what you currently have with:

```
In [7]: %%bash
        git config --get core.editor
```

```
vim
```

To configure Notepad++ on windows you'll need something like the below, ask a demonstrator to help for your machine.

```
$ git config --global core.editor "code --wait"
```

I'm going to be using `vim` as my editor, but you can use whatever editor you prefer. Find how to setup your favourite editor in the setup chapter of Software Carpentry's Git lesson.

2.15.5 Git log

Git now has one change in its history:

```
In [8]: %%bash
        git log
```

```
commit 8db2c9c5e612a7c5ca7eeab79121fbee6b8f0f6f
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:57:30 2020 +0000

    First commit of discourse on UK topography
```

You can see the commit message, author, and date…

2.15.6 Hash Codes

The commit "hash code", e.g.

`c438f1716b2515563e03e82231acbae7dd4f4656`

is a unique identifier of that particular revision.

(This is a really long code, but whenever you need to use it, you can just use the first few characters, however many characters is long enough to make it unique, `c438` for example.)

2.15.7 Nothing to see here

Note that git will now tell us that our "working directory" is up-to-date with the repository: there are no changes to the files that aren't recorded in the repository history:

```
In [9]: %%bash
        git status
```

```
On branch master
nothing to commit, working tree clean
```

Let's edit the file again:

```
vim index.md
```

```
In [10]: %%writefile index.md
         Mountains in the UK
         ===================
         England is not very mountainous.
         But has some tall hills, and maybe a mountain or two depending on your definition.

         Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

```
Overwriting index.md
```

```
In [11]: cat index.md
```

```
Mountains in the UK
===================
England is not very mountainous.
But has some tall hills, and maybe a mountain or two depending on your definition.

Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

2.15.8 Unstaged changes

```
In [12]: %%bash
         git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>…" to update what will be committed)
  (use "git restore <file>…" to discard changes in working directory)
        modified:   index.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We can now see that there is a change to "index.md" which is currently "not staged for commit". What does this mean?

If we do a `git commit` now *nothing will happen*.

Git will only commit changes to files that you choose to include in each commit.

This is a difference from other version control systems, where committing will affect all changed files.

We can see the differences in the file with:

```
In [13]: %%bash
         git diff
```

```
diff --git a/index.md b/index.md
index a1f85df..3a2f7b0 100644
--- a/index.md
```

```
+++ b/index.md
@@ -2,3 +2,5 @@ Mountains in the UK
 ====================
 England is not very mountainous.
 But has some tall hills, and maybe a mountain or two depending on your definition.
+
+Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

Deleted lines are prefixed with a minus, added lines prefixed with a plus.

2.15.9 Staging a file to be included in the next commit

To include the file in the next commit, we have a few choices. This is one of the things to be careful of with git: there are lots of ways to do similar things, and it can be hard to keep track of them all.

```
In [14]: %%bash
         git add --update
```

This says "include in the next commit, all files which have ever been included before".

Note that `git add` is the command we use to introduce git to a new file, but also the command we use to "stage" a file to be included in the next commit.

2.15.10 The staging area

The "staging area" or "index" is the git jargon for the place which contains the list of changes which will be included in the next commit.

You can include specific changes to specific files with `git add`, commit them, add some more files, and commit them. (You can even add specific changes within a file to be included in the index.)

2.15.11 Message Sequence Charts

In order to illustrate the behaviour of Git, it will be useful to be able to generate figures in Python of a "message sequence chart" flavour.

There's a nice online tool to do this, called "Web Sequence diagrams".

Instead of just showing you these diagrams, I'm showing you in this notebook how I make them. This is part of our "reproducible computing" approach; always generating all our figures from code.

Here's some quick code in the Notebook to download and display an MSC illustration, using the Web Sequence Diagrams API:

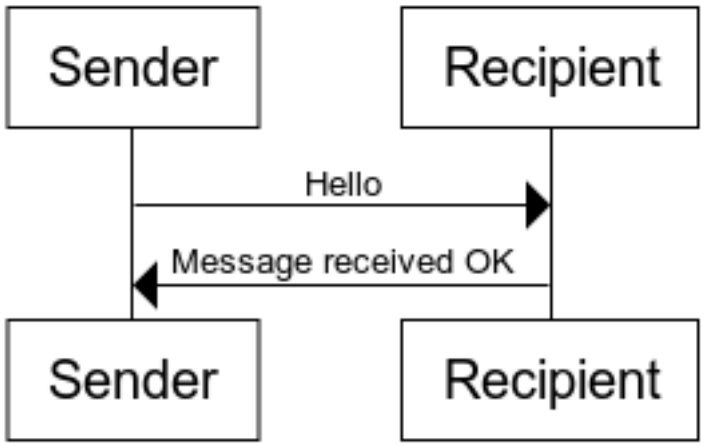
```
In [15]: %%writefile wsd.py
         import requests
         import re
         import IPython

         def wsd(code):
             response = requests.post("http://www.websequencediagrams.com/index.php", data={
                     'message': code,
                     'apiVersion': 1,
                 })
             expr = re.compile("(\?(img|pdf|png|svg)=[a-zA-Z0-9]+)")
             m = expr.search(response.text)
             if m == None:
                 print("Invalid response from server.")
                 return False
```

```
            image=requests.get("http://www.websequencediagrams.com/" + m.group(0))
            return IPython.core.display.Image(image.content)
```

```
Writing wsd.py
```

```
In [16]: from wsd import wsd
         %matplotlib inline
         wsd("Sender->Recipient: Hello\n Recipient->Sender: Message received OK")
```

Out[16]:

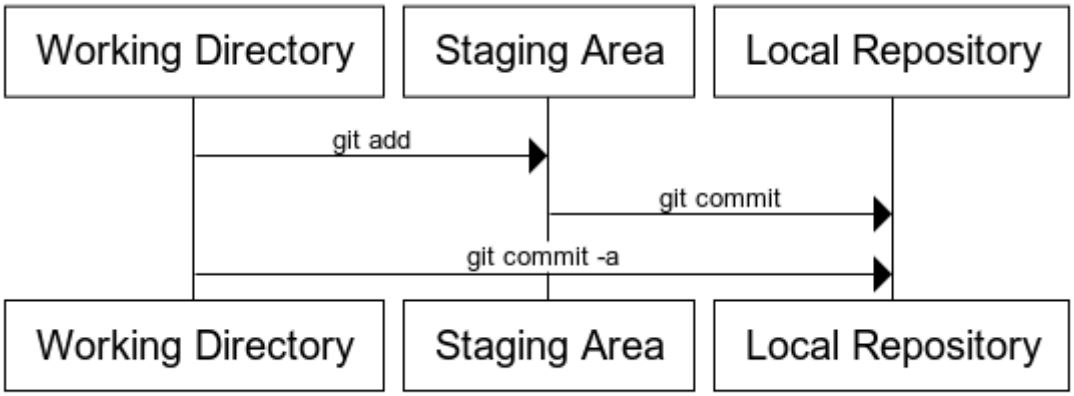


2.15.12 The Levels of Git

Let's make ourselves a sequence chart to show the different aspects of Git we've seen so far:

```
In [17]: message="""
         Working Directory -> Staging Area : git add
         Staging Area -> Local Repository : git commit
         Working Directory -> Local Repository : git commit -a
         """
         wsd(message)
```

Out[17]:



2.15.13 Review of status

```
In [18]: %%bash
         git status

On branch master
Changes to be committed:
  (use "git restore --staged <file>…" to unstage)
        modified:   index.md

Untracked files:
  (use "git add <file>…" to include in what will be committed)
        __pycache__/
        wsd.py



In [19]: %%bash
         git commit -m "Add a lie about a mountain"

[master 9aa861a] Add a lie about a mountain
 1 file changed, 2 insertions(+)


In [20]: %%bash
         git log

commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:57:31 2020 +0000

    Add a lie about a mountain

commit 8db2c9c5e612a7c5ca7eeab79121fbee6b8f0f6f
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:57:30 2020 +0000
```

```
    First commit of discourse on UK topography
```

Great, we now have a file which contains a mistake.

2.15.14 Carry on regardless

In a while, we'll use Git to roll back to the last correct version: this is one of the main reasons we wanted to use version control, after all! But for now, let's do just as we would if we were writing code, not notice our mistake and keep working…

```
vim index.md
```

```
In [21]: %%writefile index.md
         Mountains and Hills in the UK
         ==================
         England is not very mountainous.
         But has some tall hills, and maybe a mountain or two depending on your definition.

         Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

```
Overwriting index.md
```

```
In [22]: cat index.md
```

```
Mountains and Hills in the UK
==================
England is not very mountainous.
But has some tall hills, and maybe a mountain or two depending on your definition.

Mount Fictional, in Barsetshire, U.K. is the tallest mountain in the world.
```

2.15.15 Commit with a built-in-add

```
In [23]: %%bash
         git commit -am "Change title"
```

```
[master c9b2323] Change title
 1 file changed, 1 insertion(+), 1 deletion(-)
```

This last command, `git commit -a` automatically adds changes to all tracked files to the staging area, as part of the commit command. So, if you never want to just add changes to some tracked files but not others, you can just use this and forget about the staging area!

2.15.16 Review of changes

```
In [24]: %%bash
         git log | head
```

```
commit c9b2323938b782b0cf1231038ff2855097c3f7f1
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:57:31 2020 +0000
```

```
    Change title

commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:57:31 2020 +0000
```

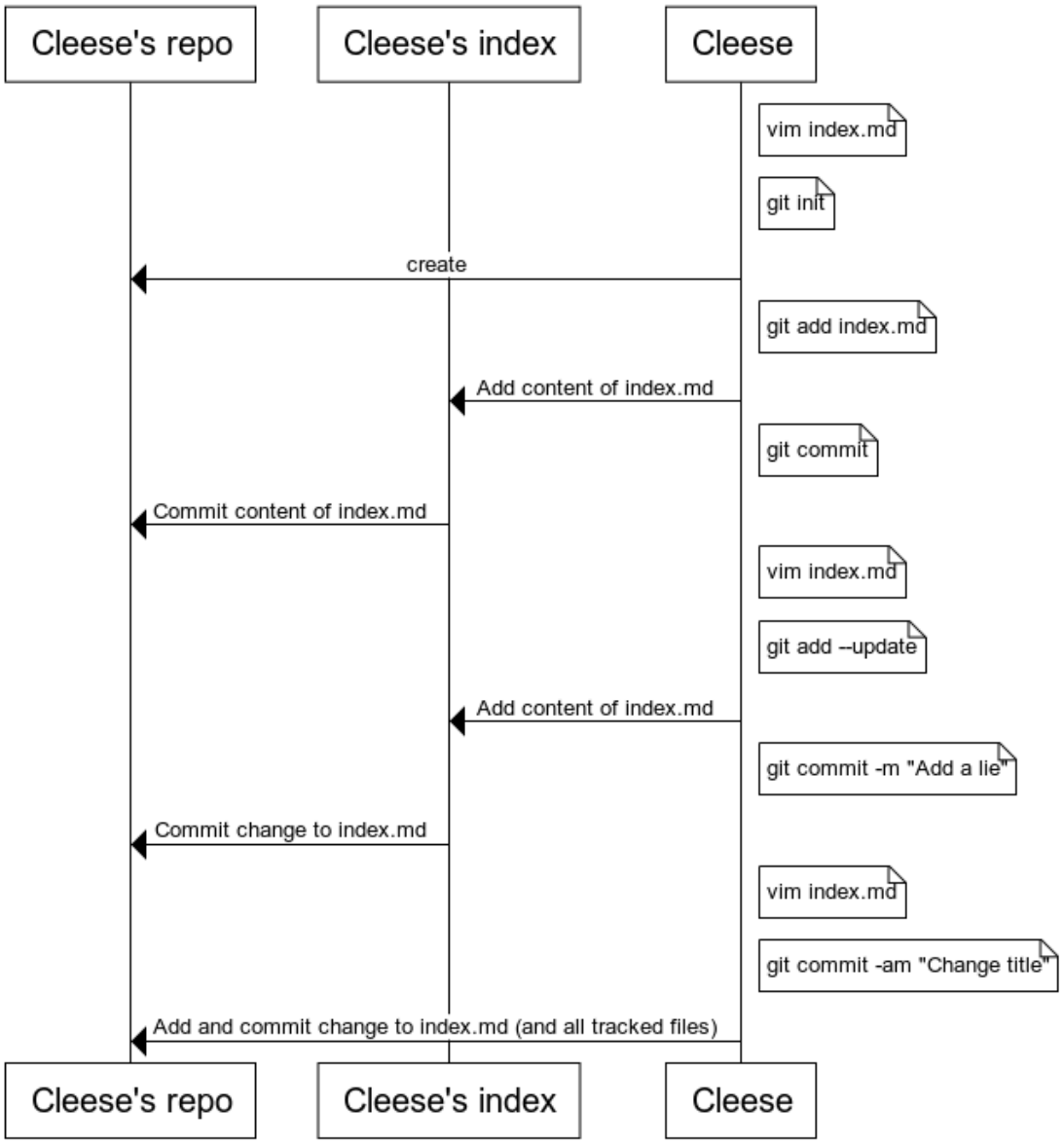
We now have three changes in the history:

```
In [25]: %%bash
         git log --oneline

c9b2323 Change title
9aa861a Add a lie about a mountain
8db2c9c First commit of discourse on UK topography
```

2.15.17 Git Solo Workflow

We can make a diagram that summarises the above story:

```
In [26]: message="""
         participant "Cleese's repo" as R
         participant "Cleese's index" as I
         participant Cleese as C

         note right of C: vim index.md

         note right of C: git init
         C->R: create

         note right of C: git add index.md

         C->I: Add content of index.md

         note right of C: git commit
         I->R: Commit content of index.md

         note right of C:  vim index.md

         note right of C: git add --update
         C->I: Add content of index.md
         note right of C: git commit -m "Add a lie"
         I->R: Commit change to index.md

         note right of C:  vim index.md
         note right of C: git commit -am "Change title"
         C->R: Add and commit change to index.md (and all tracked files)
         """
         wsd(message)

Out[26]:
```



2.16 Fixing mistakes

We're still in our git working directory:

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)
        working_dir
```

```
Out[1]: '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_example'
```

2.16.1 Referring to changes with HEAD and ^

The commit we want to revert to is the one before the latest.

`HEAD` refers to the latest commit. That is, we want to go back to the change before the current `HEAD`.

We could use the hash code (e.g. 73fbeaf) to reference this, but you can also refer to the commit before the `HEAD` as `HEAD^`, the one before that as `HEAD^^`, the one before that as `HEAD~3`.

2.16.2 Reverting

Ok, so now we'd like to undo the nasty commit with the lie about Mount Fictional.

```
In [2]: %%bash
        git revert HEAD^

Auto-merging index.md
[master c124739] Revert "Add a lie about a mountain"
 Date: Fri Jan 17 18:57:34 2020 +0000
 1 file changed, 2 deletions(-)
```

An editor may pop up, with some default text which you can accept and save.

2.16.3 Conflicted reverts

You may, depending on the changes you've tried to make, get an error message here.

If this happens, it is because git could not automagically decide how to combine the change you made after the change you want to revert, with the attempt to revert the change: this could happen, for example, if they both touch the same line.

If that happens, you need to manually edit the file to fix the problem. Skip ahead to the section on resolving conflicts, or ask a demonstrator to help.

2.16.4 Review of changes

The file should now contain the change to the title, but not the extra line with the lie. Note the log:

```
In [3]: %%bash
        git log --date=short

commit c1247395a9617120445afe96dc1cb74cbaed57d6
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Revert "Add a lie about a mountain"

    This reverts commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd.

commit c9b2323938b782b0cf1231038ff2855097c3f7f1
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Change title

commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17
```

```
    Add a lie about a mountain

commit 8db2c9c5e612a7c5ca7eeab79121fbee6b8f0f6f
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    First commit of discourse on UK topography
```

2.16.5 Antipatch

Notice how the mistake has stayed in the history.

There is a new commit which undoes the change: this is colloquially called an "antipatch". This is nice: you have a record of the full story, including the mistake and its correction.

2.16.6 Rewriting history

It is possible, in git, to remove the most recent change altogether, "rewriting history". Let's make another bad change, and see how to do this.

2.16.7 A new lie

```
In [4]: %%writefile index.md
        Mountains and Hills in the UK
        ====================
        Engerland is not very mountainous.
        But has some tall hills, and maybe a
        mountain or two depending on your definition.

Overwriting index.md


In [5]: %%bash
        cat index.md

Mountains and Hills in the UK
====================
Engerland is not very mountainous.
But has some tall hills, and maybe a
mountain or two depending on your definition.


In [6]: %%bash
        git diff

diff --git a/index.md b/index.md
index dd5cf9c..4801c98 100644
--- a/index.md
+++ b/index.md
@@ -1,4 +1,5 @@
 Mountains and Hills in the UK
 ====================
-England is not very mountainous.
-But has some tall hills, and maybe a mountain or two depending on your definition.
+Engerland is not very mountainous.
+But has some tall hills, and maybe a
```

```
+mountain or two depending on your definition.
```

```
In [7]: %%bash
        git commit -am "Add a silly spelling"
```

```
[master ffaeb55] Add a silly spelling
 1 file changed, 3 insertions(+), 2 deletions(-)
```

```
In [8]: %%bash
        git log --date=short
```

```
commit ffaeb556011d90ae3bc9f74bd591e0c34fb37984
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Add a silly spelling

commit c1247395a9617120445afe96dc1cb74cbaed57d6
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Revert "Add a lie about a mountain"

    This reverts commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd.

commit c9b2323938b782b0cf1231038ff2855097c3f7f1
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Change title

commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Add a lie about a mountain

commit 8db2c9c5e612a7c5ca7eeab79121fbee6b8f0f6f
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    First commit of discourse on UK topography
```

2.16.8 Using reset to rewrite history

```
In [9]: %%bash
        git reset HEAD^
```

```
Unstaged changes after reset:
M       index.md
```

```
In [10]: %%bash
         git log --date=short
```

```
commit c1247395a9617120445afe96dc1cb74cbaed57d6
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Revert "Add a lie about a mountain"

    This reverts commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd.

commit c9b2323938b782b0cf1231038ff2855097c3f7f1
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Change title

commit 9aa861a3e3fc30830515a1673f565dd7c8b988dd
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    Add a lie about a mountain

commit 8db2c9c5e612a7c5ca7eeab79121fbee6b8f0f6f
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   2020-01-17

    First commit of discourse on UK topography
```

2.16.9 Covering your tracks

The silly spelling *is no longer in the log.* This approach to fixing mistakes, "rewriting history" with `reset`, instead of adding an antipatch with `revert`, is dangerous, and we don't recommend it. But you may want to do it for small silly mistakes, such as to correct a commit message.

2.16.10 Resetting the working area

When `git reset` removes commits, it leaves your working directory unchanged – so you can keep the work in the bad change if you want.

```
In [11]: %%bash
         cat index.md

Mountains and Hills in the UK
==========
Engerland is not very mountainous.
But has some tall hills, and maybe a
mountain or two depending on your definition.
```

If you want to lose the change from the working directory as well, you can do `git reset --hard`.

I'm going to get rid of the silly spelling, and I didn't do `--hard`, so I'll reset the file from the working directory to be the same as in the index:

```
In [12]: %%bash
         git checkout index.md
```

```
Updated 1 path from the index
```

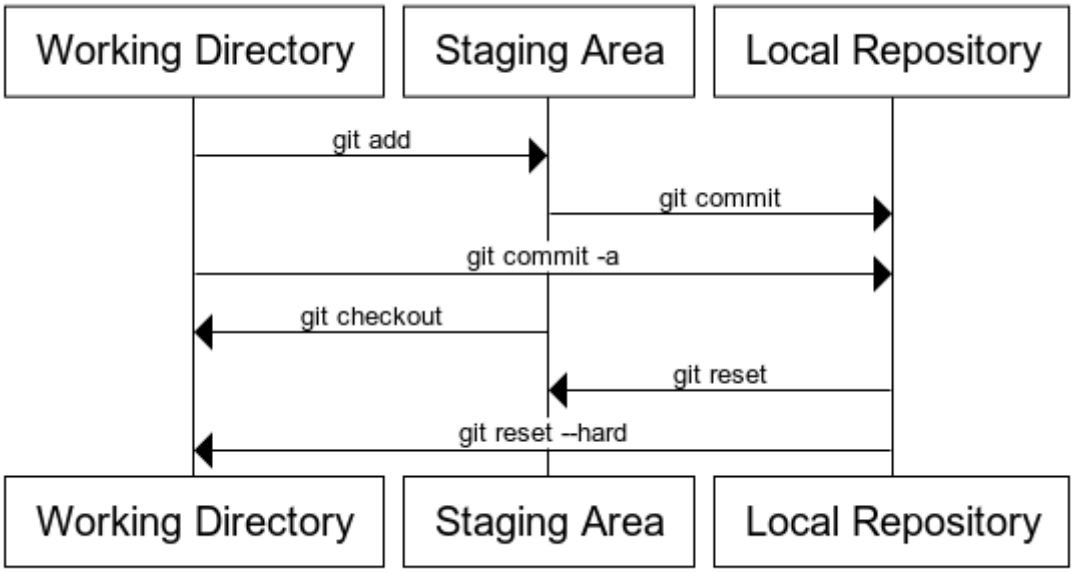
```
In [13]: %%bash
         cat index.md
```

```
Mountains and Hills in the UK
==================
England is not very mountainous.
But has some tall hills, and maybe a mountain or two depending on your definition.
```

We can add this to our diagram:

```
In [14]: message="""
         Working Directory -> Staging Area : git add
         Staging Area -> Local Repository : git commit
         Working Directory -> Local Repository : git commit -a
         Staging Area -> Working Directory : git checkout
         Local Repository -> Staging Area : git reset
         Local Repository -> Working Directory: git reset --hard
         """
         from wsd import wsd
         %matplotlib inline
         wsd(message)
```

```
Out[14]:
```

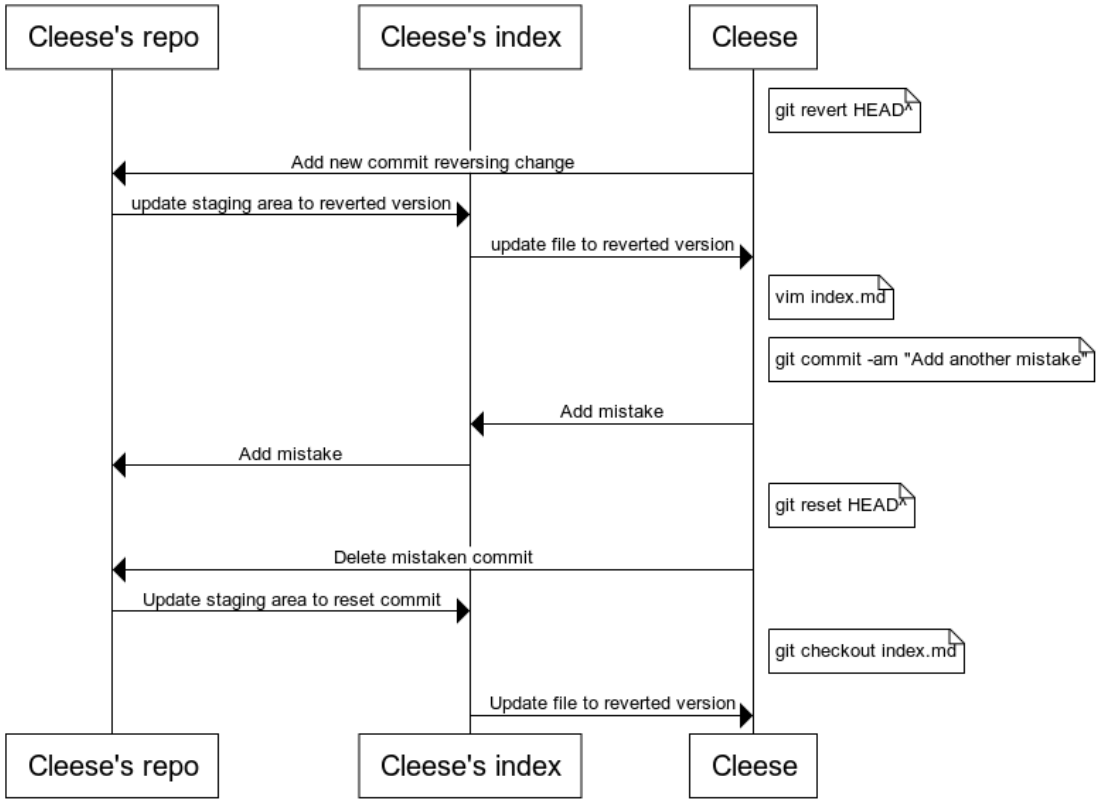


We can add it to Cleese's story:

```
In [15]: message="""
         participant "Cleese's repo" as R
         participant "Cleese's index" as I
```

```
        participant Cleese as C

        note right of C: git revert HEAD^

        C->R: Add new commit reversing change
        R->I: update staging area to reverted version
        I->C: update file to reverted version



        note right of C: vim index.md
        note right of C: git commit -am "Add another mistake"
        C->I: Add mistake
        I->R: Add mistake

        note right of C: git reset HEAD^

        C->R: Delete mistaken commit
        R->I: Update staging area to reset commit

        note right of C: git checkout index.md

        I->C: Update file to reverted version


        """
        wsd(message)
```

Out[15]:



2.17 Publishing

We're still in our working directory:

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)
        working_dir
```

```
Out[1]: '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_example'
```

2.17.1 Sharing your work

So far, all our work has been on our own computer. But a big part of the point of version control is keeping your work safe, on remote servers. Another part is making it easy to share your work with the world In this example, we'll be using the "GitHub" cloud repository to store and publish our work.

If you have not done so already, you should create an account on GitHub: go to GitHub's website, fill in a username and password, and click on "sign up for GitHub".

2.17.2 Creating a repository

Ok, let's create a repository to store our work. Hit "new repository" on the right of the github home screen.

Fill in a short name, and a description. Choose a "public" repository. Don't choose to initialize the repository with a README. That will create a repository with content and we only want a placeholder where to upload what we've created locally.

2.17.3 Paying for GitHub

For this course, you should use public repositories in your personal account for your example work: it's good to share! GitHub is free for open source, but in general, charges a fee if you want to keep your work private.

In the future, you might want to keep your work on GitHub private.

Students can get free private repositories on GitHub, by going to GitHub Education and filling in a form (look for the Student Developer Pack).

UCL pays for private GitHub repositories for UCL research groups: you can find the service details on the Research Software Development Group's website.

2.17.4 Adding a new remote to your repository

Instructions will appear, once you've created the repository, as to how to add this new "remote" server to your repository, in the lower box on the screen. Mine say:

```
In [2]: %%bash
        git remote add origin git@github.com:UCL/github-example.git
```

```
In [3]: %%bash
        git push -uf origin master # I have an extra `f` switch here.
              #You should copy the instructions from YOUR repository.
```

```
Branch 'master' set up to track remote branch 'master' from 'origin'.


Warning: Permanently added the RSA host key for IP address '140.82.113.4' to the list of known hosts.
To github.com:UCL/github-example.git
 + 7fd37ad…c124739 master -> master (forced update)
```

2.17.5 Remotes

The first command sets up the server as a new `remote`, called `origin`.

Git, unlike some earlier version control systems is a "distributed" version control system, which means you can work with multiple remote servers.

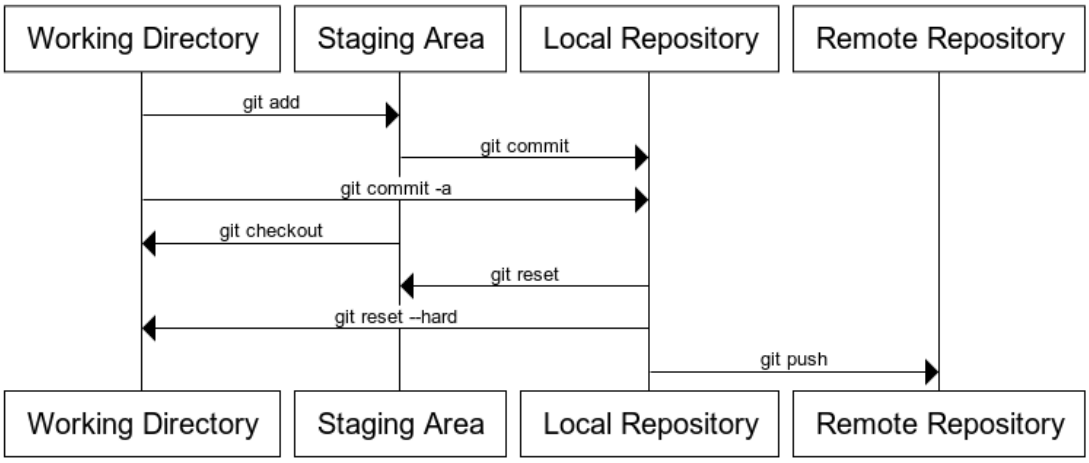
Usually, commands that work with remotes allow you to specify the remote to use, but assume the `origin` remote if you don't.

Here, `git push` will push your whole history onto the server, and now you'll be able to see it on the internet! Refresh your web browser where the instructions were, and you'll see your repository!

Let's add these commands to our diagram:

```
In [4]: message="""
        Working Directory -> Staging Area : git add
        Staging Area -> Local Repository : git commit
        Working Directory -> Local Repository : git commit -a
        Staging Area -> Working Directory : git checkout
        Local Repository -> Staging Area : git reset
        Local Repository -> Working Directory: git reset --hard
        Local Repository -> Remote Repository : git push
        """
        from wsd import wsd
        %matplotlib inline
        wsd(message)
```

Out[4]:



2.17.6 Playing with GitHub

Take a few moments to click around and work your way through the GitHub interface. Try clicking on 'index.md' to see the content of the file: notice how the markdown renders prettily.

Click on "commits" near the top of the screen, to see all the changes you've made. Click on the commit number next to the right of a change, to see what changes it includes: removals are shown in red, and additions in green.

2.18 Working with multiple files

2.18.1 Some new content

So far, we've only worked with one file. Let's add another:

```
vim lakeland.md
```

```
In [5]: %%writefile lakeland.md
        Lakeland
        ========

        Cumbria has some pretty hills, and lakes too.
```

```
Writing lakeland.md
```

```
In [6]: cat lakeland.md
```

```
Lakeland
========

Cumbria has some pretty hills, and lakes too.
```

2.18.2 Git will not by default commit your new file

```
In [7]: %%bash --no-raise-error
        git commit -am "Try to add Lakeland"
```

```
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
        __pycache__/
        lakeland.md
        wsd.py

nothing added to commit but untracked files present
```

This didn't do anything, because we've not told git to track the new file yet.

2.18.3 Tell git about the new file

```
In [8]: %%bash
        git add lakeland.md
        git commit -am "Add lakeland"
```

```
[master 28202e2] Add lakeland
 1 file changed, 4 insertions(+)
 create mode 100644 lakeland.md
```

Ok, now we have added the change about Cumbria to the file. Let's publish it to the origin repository.

```
In [9]: %%bash
        git push
```

```
To github.com:UCL/github-example.git
   c124739..28202e2  master -> master
```

Visit GitHub, and notice this change is on your repository on the server. We could have said `git push origin` to specify the remote to use, but origin is the default.

2.19 Changing two files at once

What if we change both files?

```
In [10]: %%writefile lakeland.md
         Lakeland
         ========

         Cumbria has some pretty hills, and lakes too

         Mountains:
         * Helvellyn
```

```
Overwriting lakeland.md
```

```
In [11]: %%writefile index.md
         Mountains and Lakes in the UK
         ==========
         Engerland is not very mountainous.
         But has some tall hills, and maybe a
         mountain or two depending on your definition.

Overwriting index.md
```

```
In [12]: %%bash
         git status

On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>…" to update what will be committed)
  (use "git restore <file>…" to discard changes in working directory)
        modified:   index.md
        modified:   lakeland.md

Untracked files:
  (use "git add <file>…" to include in what will be committed)
        __pycache__/
        wsd.py

no changes added to commit (use "git add" and/or "git commit -a")
```

These changes should really be separate commits. We can do this with careful use of git add, to **stage** first one commit, then the other.

```
In [13]: %%bash
         git add index.md
         git commit -m "Include lakes in the scope"

[master f033da8] Include lakes in the scope
 1 file changed, 4 insertions(+), 3 deletions(-)
```

Because we "staged" only index.md, the changes to lakeland.md were not included in that commit.

```
In [14]: %%bash
         git commit -am "Add Helvellyn"

[master e899c3c] Add Helvellyn
 1 file changed, 4 insertions(+), 1 deletion(-)
```

```
In [15]: %%bash
         git log --oneline

e899c3c Add Helvellyn
f033da8 Include lakes in the scope
28202e2 Add lakeland
```

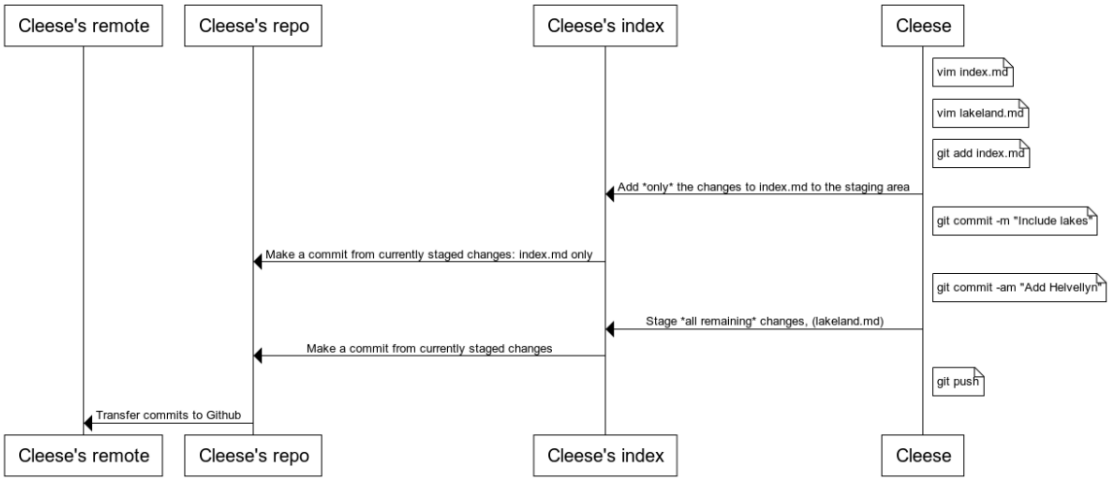
```
c124739 Revert "Add a lie about a mountain"
c9b2323 Change title
9aa861a Add a lie about a mountain
8db2c9c First commit of discourse on UK topography
```

```
In [16]: %%bash
         git push
```

```
To github.com:UCL/github-example.git
   28202e2..e899c3c  master -> master
```

```
In [17]: message="""
         participant "Cleese's remote" as M
         participant "Cleese's repo" as R
         participant "Cleese's index" as I
         participant Cleese as C

         note right of C: vim index.md
         note right of C: vim lakeland.md

         note right of C: git add index.md
         C->I: Add *only* the changes to index.md to the staging area

         note right of C: git commit -m "Include lakes"
         I->R: Make a commit from currently staged changes: index.md only

         note right of C: git commit -am "Add Helvellyn"
         C->I: Stage *all remaining* changes, (lakeland.md)
         I->R: Make a commit from currently staged changes

         note right of C: git push
         R->M: Transfer commits to Github
         """
         wsd(message)
```

Out[17]:



2.20 Collaboration

2.20.1 Form a team

Now we're going to get to the most important question of all with Git and GitHub: working with others.

Organise into pairs. You're going to be working on the website of one of the two of you, together, so decide who is going to be the leader, and who the collaborator.

2.20.2 Giving permission

The leader needs to let the collaborator have the right to make changes to his code.

In GitHub, go to `Settings` on the right, then `Collaborators & teams` on the left.

Add the user name of your collaborator to the box. They now have the right to push to your repository.

2.20.3 Obtaining a colleague's code

Next, the collaborator needs to get a copy of the leader's code. For this example notebook, I'm going to be collaborating with myself, swapping between my two repositories. Make yourself a space to put it your work. (I will have two)

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(git_dir)
```

```
In [2]: %%bash
        pwd
        rm -rf github-example # cleanup after previous example
        rm -rf partner_repo # cleanup after previous example
```

```
/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git
```

Next, the collaborator needs to find out the URL of the repository: they should go to the leader's repository's GitHub page, and note the URL on the top of the screen. Make sure the "ssh" button is pushed, the URL should begin with `git@github.com`.

Copy the URL into your clipboard by clicking on the icon to the right of the URL, and then:

```
In [3]: %%bash
        pwd
        git clone git@github.com:UCL/github-example.git partner_repo
```

```
/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git


Cloning into 'partner_repo'…
```

```
In [4]: partner_dir = os.path.join(git_dir, 'partner_repo')
        os.chdir(partner_dir)
```

```
In [5]: %%bash
        pwd
        ls
```

```
/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/partner_repo
index.md
lakeland.md
```

Note that your partner's files are now present on your disk:

```
In [6]: %%bash
        cat lakeland.md
```

```
Lakeland
========

Cumbria has some pretty hills, and lakes too

Mountains:
* Helvellyn
```

2.20.4 Nonconflicting changes

Now, both of you should make some changes. To start with, make changes to *different* files. This will mean your work doesn't "conflict". Later, we'll see how to deal with changes to a shared file.

Both of you should commit, but not push, your changes to your respective files:

E.g., the leader:

```
In [7]: os.chdir(working_dir)
```

```
In [8]: %%writefile Wales.md
        Mountains In Wales
        ==================

        * Tryfan
        * Yr Wyddfa
```

```
Writing Wales.md
```

```
In [9]: %%bash
        ls
```

```
index.md
lakeland.md
__pycache__
Wales.md
wsd.py
```

```
In [10]: %%bash
         git add Wales.md
         git commit -m "Add wales"
```

```
[master cd12c03] Add wales
 1 file changed, 5 insertions(+)
 create mode 100644 Wales.md
```

And the partner:

```
In [11]: os.chdir(partner_dir)
```

```
In [12]: %%writefile Scotland.md
         Mountains In Scotland
         ==================

         * Ben Eighe
         * Cairngorm
```

```
Writing Scotland.md
```

```
In [13]: %%bash
         ls
```

```
index.md
lakeland.md
Scotland.md
```

```
In [14]: %%bash
         git add Scotland.md
         git commit -m "Add Scotland"
```

```
[master 95afb78] Add Scotland
 1 file changed, 5 insertions(+)
 create mode 100644 Scotland.md
```

One of you should now push with `git push`:

```
In [15]: %%bash
         git push
```

```
To github.com:UCL/github-example.git
   e899c3c..95afb78  master -> master
```

2.20.5 Rejected push

The other should then push, but should receive an error message:

```
In [16]: os.chdir(working_dir)
```

```
In [17]: %%bash --no-raise-error
         git push
```

```
To github.com:UCL/github-example.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:UCL/github-example.git'
```

```
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull …') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Do as it suggests:

```
In [18]: %%bash
         git pull
```

```
Merge made by the 'recursive' strategy.
 Scotland.md | 5 +++++
 1 file changed, 5 insertions(+)
 create mode 100644 Scotland.md


From github.com:UCL/github-example
   e899c3c..95afb78  master     -> origin/master
 * [new branch]      gh-pages   -> origin/gh-pages
```

2.20.6 Merge commits

A window may pop up with a suggested default commit message. This commit is special: it is a *merge* commit. It is a commit which combines your collaborator's work with your own.

Now, push again with `git push`. This time it works. If you look on GitHub, you'll now see that it contains both sets of changes.

```
In [19]: %%bash
         git push
```

```
To github.com:UCL/github-example.git
   95afb78..dbc9d65  master -> master
```

The partner now needs to pull down that commit:

```
In [20]: os.chdir(partner_dir)
```

```
In [21]: %%bash
         git pull
```

```
Updating 95afb78..dbc9d65
Fast-forward
 Wales.md | 5 +++++
 1 file changed, 5 insertions(+)
 create mode 100644 Wales.md


From github.com:UCL/github-example
   95afb78..dbc9d65  master     -> origin/master
```

```
In [22]: %%bash
         ls
```

```
index.md
lakeland.md
Scotland.md
Wales.md
```

2.20.7 Nonconflicted commits to the same file

Go through the whole process again, but this time, both of you should make changes to a single file, but make sure that you don't touch the same *line*. Again, the merge should work as before:

```
In [23]: %%writefile Wales.md
         Mountains In Wales
         ==================

         * Tryfan
         * Snowdon
```

```
Overwriting Wales.md
```

```
In [24]: %%bash
         git diff
```

```
diff --git a/Wales.md b/Wales.md
index f3e88b4..90f23ec 100644
--- a/Wales.md
+++ b/Wales.md
@@ -2,4 +2,4 @@ Mountains In Wales
 ==================

 * Tryfan
-* Yr Wyddfa
+* Snowdon
```

```
In [25]: %%bash
         git commit -am "Translating from the Welsh"
```

```
[master 37d36fa] Translating from the Welsh
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
In [26]: %%bash
         git log --oneline
```

```
37d36fa Translating from the Welsh
dbc9d65 Merge branch 'master' of github.com:UCL/github-example
cd12c03 Add wales
95afb78 Add Scotland
e899c3c Add Helvellyn
f033da8 Include lakes in the scope
28202e2 Add lakeland
c124739 Revert "Add a lie about a mountain"
c9b2323 Change title
9aa861a Add a lie about a mountain
8db2c9c First commit of discourse on UK topography
```

```
In [27]: os.chdir(working_dir)

In [28]: %%writefile Wales.md
         Mountains In Wales
         ==================

         * Pen y Fan
         * Tryfan
         * Snowdon

Overwriting Wales.md


In [29]: %%bash
         git commit -am "Add a beacon"

[master fc30222] Add a beacon
 1 file changed, 2 insertions(+), 1 deletion(-)


In [30]: %%bash
         git log --oneline

fc30222 Add a beacon
dbc9d65 Merge branch 'master' of github.com:UCL/github-example
cd12c03 Add wales
95afb78 Add Scotland
e899c3c Add Helvellyn
f033da8 Include lakes in the scope
28202e2 Add lakeland
c124739 Revert "Add a lie about a mountain"
c9b2323 Change title
9aa861a Add a lie about a mountain
8db2c9c First commit of discourse on UK topography


In [31]: %%bash
         git push

To github.com:UCL/github-example.git
   dbc9d65..fc30222  master -> master
```

Switching back to the other partner…

```
In [32]: os.chdir(partner_dir)

In [33]: %%bash --no-raise-error
         git push

To github.com:UCL/github-example.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:UCL/github-example.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull …') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

```
In [34]: %%bash
         git pull
```

```
Auto-merging Wales.md
Merge made by the 'recursive' strategy.
 Wales.md | 1 +
 1 file changed, 1 insertion(+)


From github.com:UCL/github-example
   dbc9d65..fc30222  master     -> origin/master
```

```
In [35]: %%bash
         git push
```

```
To github.com:UCL/github-example.git
   fc30222..7cae13a  master -> master
```

```
In [36]: %%bash
         git log --oneline --graph
```

```
*   7cae13a Merge branch 'master' of github.com:UCL/github-example
|\
| * fc30222 Add a beacon
* | 37d36fa Translating from the Welsh
|/
*   dbc9d65 Merge branch 'master' of github.com:UCL/github-example
|\
| * 95afb78 Add Scotland
* | cd12c03 Add wales
|/
* e899c3c Add Helvellyn
* f033da8 Include lakes in the scope
* 28202e2 Add lakeland
* c124739 Revert "Add a lie about a mountain"
* c9b2323 Change title
* 9aa861a Add a lie about a mountain
* 8db2c9c First commit of discourse on UK topography
```

```
In [37]: os.chdir(working_dir)
```

```
In [38]: %%bash
         git pull
```

```
Updating fc30222..7cae13a
Fast-forward


From github.com:UCL/github-example
   fc30222..7cae13a  master     -> origin/master
```

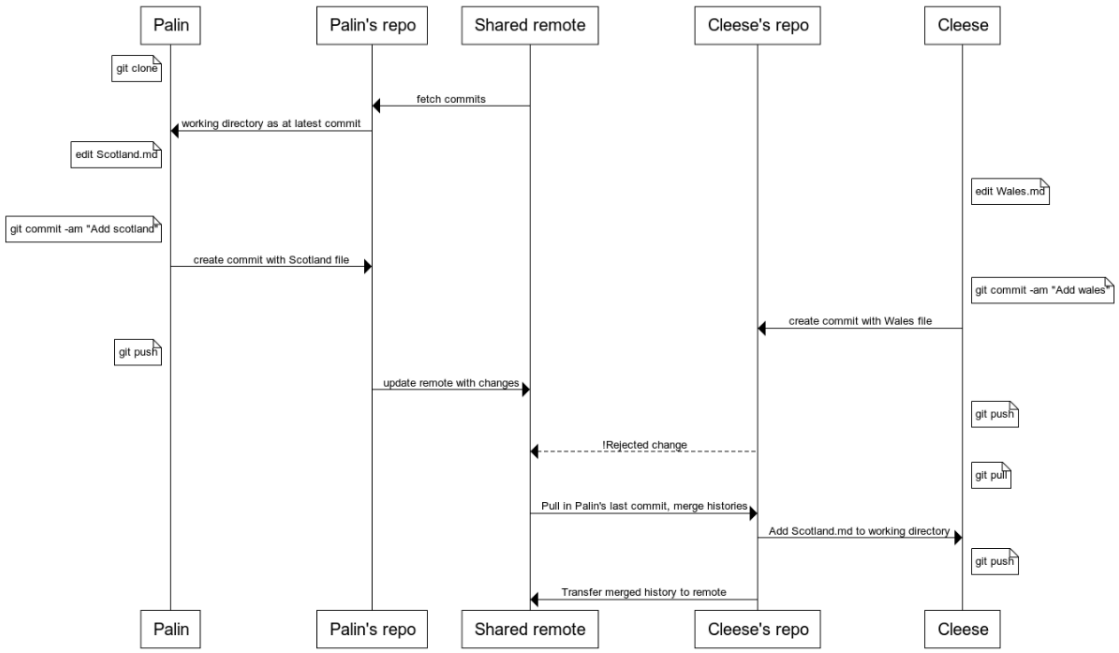
```
In [39]: %%bash
         git log --graph --oneline
```

```
*   7cae13a Merge branch 'master' of github.com:UCL/github-example
|\
| * fc30222 Add a beacon
* | 37d36fa Translating from the Welsh
|/
*   dbc9d65 Merge branch 'master' of github.com:UCL/github-example
|\
| * 95afb78 Add Scotland
* | cd12c03 Add wales
|/
* e899c3c Add Helvellyn
* f033da8 Include lakes in the scope
* 28202e2 Add lakeland
* c124739 Revert "Add a lie about a mountain"
* c9b2323 Change title
* 9aa861a Add a lie about a mountain
* 8db2c9c First commit of discourse on UK topography
```

```
In [40]: message="""
         participant Palin as P
         participant "Palin's repo" as PR
         participant "Shared remote" as M
         participant "Cleese's repo" as CR
         participant Cleese as C

         note left of P: git clone
         M->PR: fetch commits
         PR->P: working directory as at latest commit

         note left of P: edit Scotland.md
         note right of C: edit Wales.md

         note left of P: git commit -am "Add scotland"
         P->PR: create commit with Scotland file

         note right of C: git commit -am "Add wales"
         C->CR: create commit with Wales file

         note left of P: git push
         PR->M: update remote with changes

         note right of C: git push
         CR-->M: !Rejected change

         note right of C: git pull
         M->CR: Pull in Palin's last commit, merge histories
         CR->C: Add Scotland.md to working directory

         note right of C: git push
         CR->M: Transfer merged history to remote

         """
         from wsd import wsd
```

```
%matplotlib inline
wsd(message)
```

Out[40]:



2.20.8 Conflicting commits

Finally, go through the process again, but this time, make changes which touch the same line.

```
In [41]: %%writefile Wales.md
         Mountains In Wales
         ==================

         * Pen y Fan
         * Tryfan
         * Snowdon
         * Fan y Big
```

```
Overwriting Wales.md
```

```
In [42]: %%bash
         git commit -am "Add another Beacon"
         git push
```

```
[master 41e49f6] Add another Beacon
 1 file changed, 1 insertion(+)


To github.com:UCL/github-example.git
   7cae13a..41e49f6  master -> master
```

```
In [43]: os.chdir(partner_dir)

In [44]: %%writefile Wales.md
         Mountains In Wales
         ==================

         * Pen y Fan
         * Tryfan
         * Snowdon
         * Glyder Fawr

Overwriting Wales.md


In [45]: %%bash --no-raise-error
         git commit -am "Add Glyder"
         git push

[master 0dff108] Add Glyder
 1 file changed, 1 insertion(+)


To github.com:UCL/github-example.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'git@github.com:UCL/github-example.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull …') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

When you pull, instead of offering an automatic merge commit message, it says:

```
In [46]: %%bash --no-raise-error
         git pull

Auto-merging Wales.md
CONFLICT (content): Merge conflict in Wales.md
Automatic merge failed; fix conflicts and then commit the result.


From github.com:UCL/github-example
   7cae13a..41e49f6  master     -> origin/master
```

2.20.9 Resolving conflicts

Git couldn't work out how to merge the two different sets of changes.

You now need to manually resolve the conflict.

It has marked the conflicted area:

```
In [47]: %%bash
         cat Wales.md
```

```
Mountains In Wales
==================

* Pen y Fan
* Tryfan
* Snowdon
<<<<<<< HEAD
* Glyder Fawr
=======
* Fan y Big
>>>>>>> 41e49f613a7c00f2c735a4ce06f64b8755abf7c4
```

Manually edit the file, to combine the changes as seems sensible and get rid of the symbols:

```
In [48]: %%writefile Wales.md
         Mountains In Wales
         ==================

         * Pen y Fan
         * Tryfan
         * Snowdon
         * Glyder Fawr
         * Fan y Big

Overwriting Wales.md
```

2.20.10 Commit the resolved file

Now commit the merged result:

```
In [49]: %%bash
         git commit -a --no-edit # I added a No-edit for this non-interactive session. You can edit the

[master d61e07e] Merge branch 'master' of github.com:UCL/github-example


In [50]: %%bash
         git push

To github.com:UCL/github-example.git
   41e49f6..d61e07e  master -> master


In [51]: os.chdir(working_dir)

In [52]: %%bash
         git pull

Updating 41e49f6..d61e07e
Fast-forward
 Wales.md | 1 +
 1 file changed, 1 insertion(+)
```

```
From github.com:UCL/github-example
   41e49f6..d61e07e  master     -> origin/master
```

```
In [53]: %%bash
         cat Wales.md
```

```
Mountains In Wales
==================

* Pen y Fan
* Tryfan
* Snowdon
* Glyder Fawr
* Fan y Big
```

```
In [54]: %%bash
         git log --oneline --graph
```

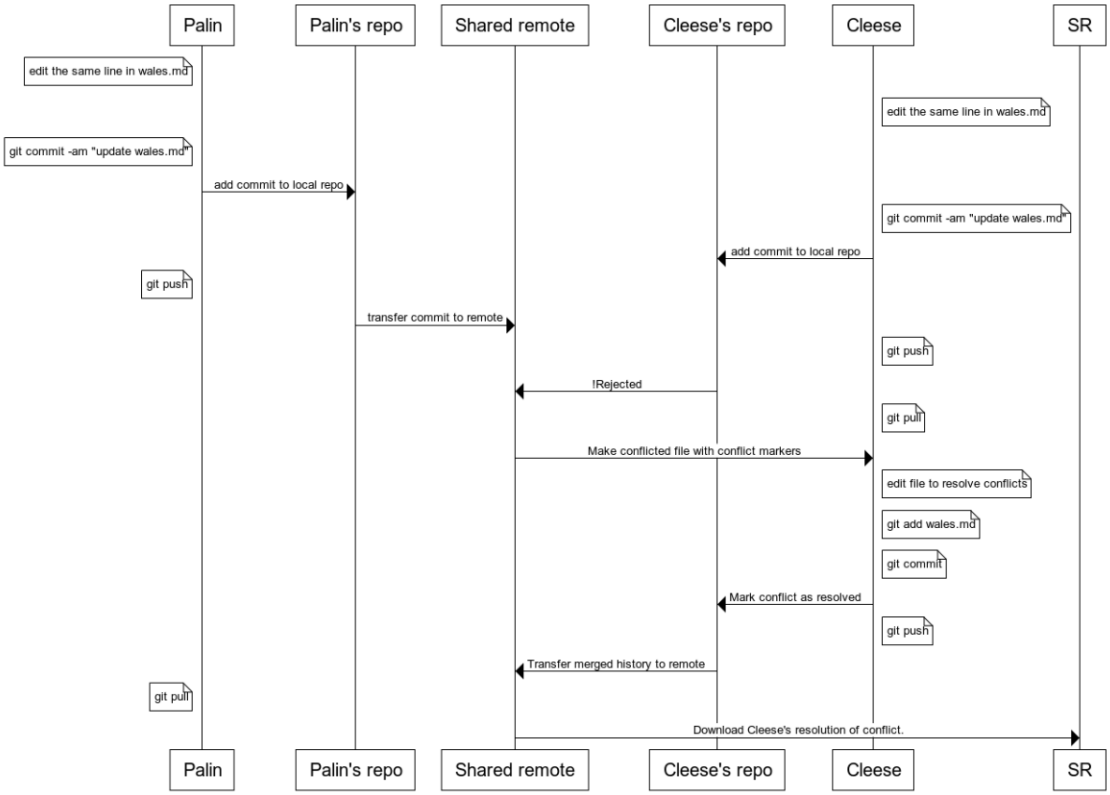
```
*   d61e07e Merge branch 'master' of github.com:UCL/github-example
|\
| * 41e49f6 Add another Beacon
* | 0dff108 Add Glyder
|/
*   7cae13a Merge branch 'master' of github.com:UCL/github-example
|\
| * fc30222 Add a beacon
* | 37d36fa Translating from the Welsh
|/
*   dbc9d65 Merge branch 'master' of github.com:UCL/github-example
|\
| * 95afb78 Add Scotland
* | cd12c03 Add wales
|/
* e899c3c Add Helvellyn
* f033da8 Include lakes in the scope
* 28202e2 Add lakeland
* c124739 Revert "Add a lie about a mountain"
* c9b2323 Change title
* 9aa861a Add a lie about a mountain
* 8db2c9c First commit of discourse on UK topography
```

2.20.11 Distributed VCS in teams with conflicts

```
In [55]: message="""
         participant Palin as P
         participant "Palin's repo" as PR
         participant "Shared remote" as M
         participant "Cleese's repo" as CR
         participant Cleese as C

         note left of P: edit the same line in wales.md
         note right of C: edit the same line in wales.md
```

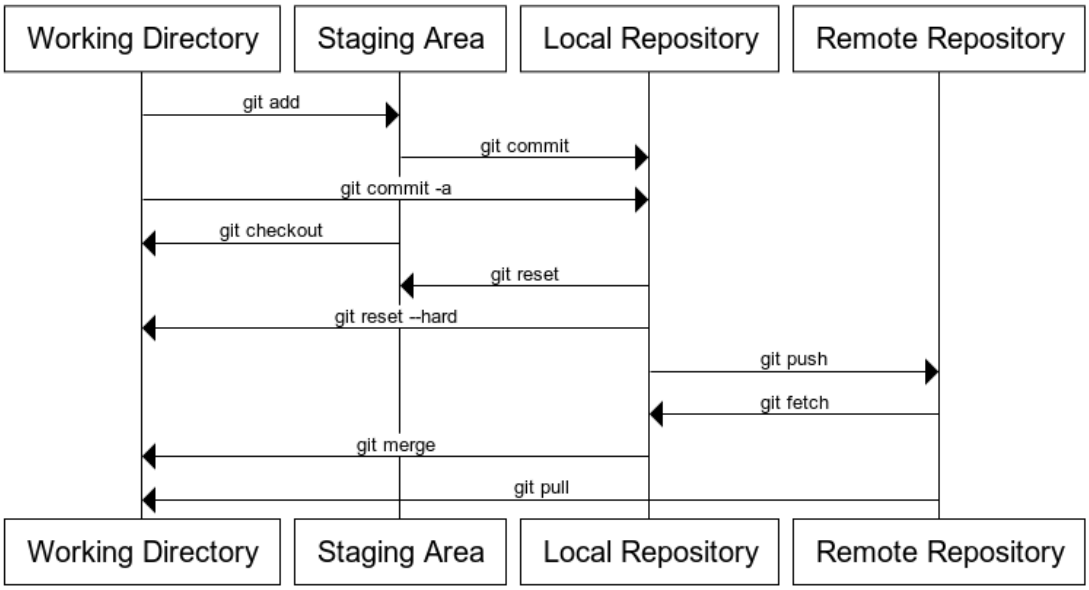
```
        note left of P: git commit -am "update wales.md"
        P->PR: add commit to local repo

        note right of C: git commit -am "update wales.md"
        C->CR: add commit to local repo

        note left of P: git push
        PR->M: transfer commit to remote

        note right of C: git push
        CR->M: !Rejected

        note right of C: git pull
        M->C: Make conflicted file with conflict markers

        note right of C: edit file to resolve conflicts
        note right of C: git add wales.md
        note right of C: git commit
        C->CR: Mark conflict as resolved

        note right of C: git push
        CR->M: Transfer merged history to remote

        note left of P: git pull
        M->SR: Download Cleese's resolution of conflict.

        """

        wsd(message)
```

```
Out[55]:
```



2.20.12 The Levels of Git

```
In [56]: message="""
         Working Directory -> Staging Area : git add
         Staging Area -> Local Repository : git commit
         Working Directory -> Local Repository : git commit -a
         Staging Area -> Working Directory : git checkout
         Local Repository -> Staging Area : git reset
         Local Repository -> Working Directory: git reset --hard
         Local Repository -> Remote Repository : git push
         Remote Repository -> Local Repository : git fetch
         Local Repository -> Working Directory : git merge
         Remote Repository -> Working Directory: git pull
         """

         wsd(message)

Out[56]:
```



2.21 Editing directly on GitHub

2.21.1 Editing directly on GitHub

Note that you can also make changes in the GitHub website itself. Visit one of your files, and hit "edit".

Make a change in the edit window, and add an appropriate commit message.

That change now appears on the website, but not in your local copy. (Verify this).

Now pull, and check the change is now present on your local version.

2.22 Social Coding

2.22.1 GitHub as a social network

In addition to being a repository for code, and a way to publish code, GitHub is a social network.

You can follow the public work of other coders: go to the profile of your collaborator in your browser, and hit the "follow" button.

Check out the profiles of Linus Torvalds - creator of git (first git commit ever) and Linux - , Guido van Rossum - creator of Python -, or James Hetherington - the creator of these course notes.

Using GitHub to build up a good public profile of software projects you've worked on is great for your CV!

2.23 Fork and Pull

2.23.1 Different ways of collaborating

We have just seen how we can work with others on GitHub: we add them as collaborators on our repositories and give them permissions to push changes.

Let's talk now about some other type of collaboration.

Imagine you are a user of an Open Source project like Numpy and find a bug in one of their methods.

You can inspect and clone Numpy's code in GitHub, play around a bit and find how to fix the bug.

Numpy has done so much for you asking nothing in return, that you really want to contribute back by fixing the bug for them.

You make all of the changes but you can't push it back to Numpy's repository because you don't have permissions.

The right way to do this is **forking Numpy's repository**.

2.23.2 Forking a repository on GitHub

By forking a repository, all you do is make a copy of it in your GitHub account, where you will have write permissions as well.

If you fork Numpy's repository, you will find a new repository in your GitHub account that is an exact copy of Numpy. You can then clone it to your computer, work locally on fixing the bug and push the changes to your *fork* of Numpy.

Once you are happy with with the changes, GitHub also offers you a way to notify Numpy's developers of this changes so that they can include them in the official Numpy repository via starting a **Pull Request**.

2.23.3 Pull Request

You can create a Pull Request and select those changes that you think can be useful for fixing Numpy's bug.

Numpy's developers will review your code and make comments and suggestions on your fix. Then, you can commit more improvements in the pull request for them to review and so on.

Once Numpy's developers are happy with your changes, they'll accept your Pull Request and merge the changes into their original repository, for everyone to use.

2.23.4 Practical example - Team up!

We will be working in the same repository with one of you being the leader and the other being the collaborator.

Collaborators need to go to the leader's GitHub profile and find the repository we created for that lesson. Mine is in https://github.com/jamespjh/github-example

1. Fork repository

You will see on the top right of the page a `Fork` button with an accompanying number indicating how many GitHub users have forked that repository.

Collaborators need to navigate to the leader's repository and click the `Fork` button.

Collaborators: note how GitHub has redirected you to your own GitHub page and you are now looking at an exact copy of the team leader's repository.

2. Clone your forked repo

Collaborators: go to your terminal and clone the newly created fork.

```
git clone git@github.com:jamespjh/github-example.git
```

3. Create a feature branch

It's a good practice to create a new branch that'll contain the changes we want. We'll learn more about branches later on. For now, just think of this as a separate area where our changes will be kept not to interfere with other people's work.

```
git checkout -b southwest
```

4. Make, commit and push changes to new branch

For example, let's create a new file called `SouthWest.md` and edit it to add this text:

```
* Exmoor
* Dartmoor
* Bodmin Moor
```

Save it, and push this changes to your fork's new branch:

```
git add SouthWest.md
git commit -m "The South West is also hilly."
git push origin southwest
```

5. Create Pull Request

Go back to the collaborator's GitHub site and reload the fork. GitHub has noticed there is a new branch and is presenting us with a green button to `Compare & pull request`. Fantastic! Click that button.

Fill in the form with additional information about your change, as you consider necesary to make the team leader understand what this is all about.

Take some time to inspect the commits and the changes you are submitting for review. When you are ready, click on the `Create Pull Request` button.

Now, the leader needs to go to their GitHub site. They have been notified there is a pull request in their repo awaiting revision.

6. Feedback from team leader

Leaders can see the list of pull requests in the vertical menu of the repo, on the right hand side of the screen. Select the pull request the collaborator has done, and inspect the changes.

There are three tabs: in one you can start a conversation with the collaborator about their changes, and in the others you can have a look at the commits and changes made.

Go to the tab labeled as "Files Changed". When you hover over the changes, a small `+` button appears. Select one line you want to make a comment on. For example, the line that contains "Exmoor".

GitHub allows you to add a comment about that specific part of the change. Your collaborator has forgotten to add a title at the beginning of the file right before "Exmoor", so tell them so in the form presented after clicking the `+` button.

7. Fixes by collaborator

Collaborators will be notified of this comment by email and also in their profiles page. Click the link accompanying this notification to read the comment from the team leader.

Go back to your local repository, make the changes suggested and push them to the new branch.

Add this at the beginning of your file:

```
Hills in the South West:
========================
```

Then push the change to your fork:

```
git add .
git commit -m "Titles added as requested."
git push origin southwest
```

This change will automatically be added to the pull request you started.

8. Leader accepts pull request

The team leader will be notified of the new changes that can be reviewed in the same fashion as earlier.

Let's assume the team leader is now happy with the changes.

Leaders can see in the "Conversation" tab of the pull request a green button labelled `Merge pull request`. Click it and confirm the decision.

The collaborator's pull request has been accepted and appears now in the original repository owned by the team leader.

Fork and Pull Request done!

2.23.5 Some Considerations

- Fork and Pull Request are things happening only on the repository's server side (GitHub in our case). Consequently, you can't do things like `git fork` or `git pull-request` from the local copy of a repository.
- You don't always need to fork repositories with the intention of contributing. You can fork a library you use, install it manually on your computer, and add more functionality or customise the existing one, so that it is more useful for you and your team.
- Numpy's example is only illustrative. Normally, Open Source projects have in their documentation (sometimes in the form of a wiki) a set of instructions you need to follow if you want to contribute to their software.
- Pull Requests can also be done for merging branches in a non-forked repository. It's typically used in teams to merge code from a branch into the master branch and ask team colleagues for code reviews before merging.
- It's a good practice before starting a fork and a pull request to have a look at existing forks and pull requests. On GitHub, you can find the list of pull requests on the horizontal menu on the top of the page. Try to also find the network graph displaying all existing forks of a repo, e.g., NumpyDoc repo's network graph.

2.24 Branches

Branches are incredibly important to why `git` is cool and powerful.

They are an easy and cheap way of making a second version of your software, which you work on in parallel, and pull in your changes when you are ready.

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)

In [2]: %%bash
        git branch # Tell me what branches exist
```

```
* master
```

```
In [3]: %%bash
        git checkout -b experiment # Make a new branch
```

```
Switched to a new branch 'experiment'
```

```
In [4]: %%bash
        git branch
```

```
* experiment
  master
```

```
In [5]: %%writefile Wales.md
        Mountains In Wales
        ==================

        * Pen y Fan
        * Tryfan
        * Snowdon
        * Glyder Fawr
        * Fan y Big
        * Cadair Idris
```

```
Overwriting Wales.md
```

```
In [6]: %%bash
        git commit -am "Add Cadair Idris"
```

```
[experiment dc38d6f] Add Cadair Idris
 1 file changed, 1 insertion(+)
```

```
In [7]: %%bash
        git checkout master # Switch to an existing branch
```

```
Your branch is up to date with 'origin/master'.


Switched to branch 'master'
```

```
In [8]: %%bash
        cat Wales.md
```

```
Mountains In Wales
==================

* Pen y Fan
* Tryfan
* Snowdon
* Glyder Fawr
* Fan y Big
```

```
In [9]: %%bash
        git checkout experiment
```

```
Switched to branch 'experiment'
```

```
In [10]: cat Wales.md
```

```
Mountains In Wales
==================

* Pen y Fan
* Tryfan
* Snowdon
* Glyder Fawr
* Fan y Big
* Cadair Idris
```

2.24.1 Publishing branches

To let the server know there's a new branch use:

```
In [11]: %%bash
         git push -u origin experiment
```

```
Branch 'experiment' set up to track remote branch 'experiment' from 'origin'.


Warning: Permanently added the RSA host key for IP address '140.82.114.4' to the list of known hosts.
remote:
remote: Create a pull request for 'experiment' on GitHub by visiting:
remote:      https://github.com/UCL/github-example/pull/new/experiment
remote:
To github.com:UCL/github-example.git
 * [new branch]      experiment -> experiment
```

We use `--set-upstream origin` (Abbreviation `-u`) to tell git that this branch should be pushed to and pulled from origin per default.

If you are following along, you should be able to see your branch in the list of branches in GitHub.

Once you've used `git push -u` once, you can push new changes to the branch with just a git push.

If others checkout your repository, they will be able to do `git checkout experiment` to see your branch content, and collaborate with you **in the branch**.

```
In [12]: %%bash
         git branch -r
```

```
  origin/experiment
  origin/gh-pages
  origin/master
```

Local branches can be, but do not have to be, connected to remote branches They are said to "track" remote branches. `push -u` sets up the tracking relationship. You can see the remote branch for each of your local branches if you ask for "verbose" output from `git branch`:

```
In [13]: %%bash
         git branch -vv
```

```
* experiment dc38d6f [origin/experiment] Add Cadair Idris
  master     d61e07e [origin/master] Merge branch 'master' of github.com:UCL/github-example
```

2.24.2 Find out what is on a branch

In addition to using `git diff` to compare to the state of a branch, you can use `git log` to look at lists of commits which are in a branch and haven't been merged yet.

```
In [14]: %%bash
         git log master..experiment
```

```
commit dc38d6fc18f743da76c59defadb59559e26a073c
Author: Lancelot the Brave <l.brave@spamalot.uk>
Date:   Fri Jan 17 18:58:22 2020 +0000

    Add Cadair Idris
```

Git uses various symbols to refer to sets of commits. The double dot `A..B` means "ancestor of B and not ancestor of A"

So in a purely linear sequence, it does what you'd expect.

```
In [15]: %%bash
         git log --graph --oneline HEAD~9..HEAD~5
```

```
*   dbc9d65 Merge branch 'master' of github.com:UCL/github-example
|\
| * 95afb78 Add Scotland
* | cd12c03 Add wales
|/
* e899c3c Add Helvellyn
* f033da8 Include lakes in the scope
```

But in cases where a history has branches, the definition in terms of ancestors is important.

```
In [16]: %%bash
         git log --graph --oneline HEAD~5..HEAD
```

```
* dc38d6f Add Cadair Idris
*   d61e07e Merge branch 'master' of github.com:UCL/github-example
|\
| * 41e49f6 Add another Beacon
* | 0dff108 Add Glyder
|/
*   7cae13a Merge branch 'master' of github.com:UCL/github-example
|\
| * fc30222 Add a beacon
* 37d36fa Translating from the Welsh
```

If there are changes on both sides, like this:

```
In [17]: %%bash
         git checkout master
```

```
Your branch is up to date with 'origin/master'.


Switched to branch 'master'
```

```
In [18]: %%writefile Scotland.md
         Mountains In Scotland
         ==================

         * Ben Eighe
         * Cairngorm
         * Aonach Eagach
```

```
Overwriting Scotland.md
```

```
In [19]: %%bash
         git diff Scotland.md
```

```
diff --git a/Scotland.md b/Scotland.md
index 9613dda..bf5c643 100644
--- a/Scotland.md
+++ b/Scotland.md
@@ -3,3 +3,4 @@ Mountains In Scotland

 * Ben Eighe
 * Cairngorm
+* Aonach Eagach
```

```
In [20]: %%bash
         git commit -am "Commit Aonach onto master branch"
```

```
[master aa571b4] Commit Aonach onto master branch
 1 file changed, 1 insertion(+)
```

Then this notation is useful to show the content of what's on what branch:

```
In [21]: %%bash
         git log --left-right --oneline master...experiment
```

```
< aa571b4 Commit Aonach onto master branch
> dc38d6f Add Cadair Idris
```

Three dots means "everything which is not a common ancestor" of the two commits, i.e. the differences between them.

2.24.3 Merging branches

We can merge branches, and just as we would pull in remote changes, there may or may not be conflicts.

```
In [22]: %%bash
         git branch
         git merge experiment
```

```
  experiment
* master
Merge made by the 'recursive' strategy.
 Wales.md | 1 +
 1 file changed, 1 insertion(+)
```

```
In [23]: %%bash
         git log --graph --oneline HEAD~3..HEAD
```

```
*   215eafd Merge branch 'experiment'
|\
| * dc38d6f Add Cadair Idris
* | aa571b4 Commit Aonach onto master branch
|/
* d61e07e Merge branch 'master' of github.com:UCL/github-example
* 41e49f6 Add another Beacon
```

2.24.4 Cleaning up after a branch

```
In [24]: %%bash
         git branch
```

```
  experiment
* master
```

```
In [25]: %%bash
         git branch -d experiment
```

```
Deleted branch experiment (was dc38d6f).
```

```
In [26]: %%bash
         git branch
```

```
* master
```

```
In [27]: %%bash
         git branch --remote
```

```
  origin/experiment
  origin/gh-pages
  origin/master
```

```
In [28]: %%bash
         git push --delete origin experiment
         # Remove remote branch
         # - also can use github interface
```

```
To github.com:UCL/github-example.git
 - [deleted]         experiment
```

```
In [29]: %%bash
         git branch --remote
```

```
  origin/gh-pages
  origin/master
```

2.24.5 A good branch strategy

- A `production` branch: code used for active work
- A `develop` branch: for general new code
- `feature` branches: for specific new ideas
- `release` branches: when you share code with others
- Useful for isolated bug fixes

2.24.6 Grab changes from a branch

Make some changes on one branch, switch back to another, and use:

```
git checkout <branch> <path>
```

to quickly grab a file from one branch into another. This will create a copy of the file as it exists in `<branch>` into your current branch, overwriting it if it already existed. For example, if you have been experimenting in a new branch but want to undo all your changes to a particular file (that is, restore the file to its version in the `master` branch), you can do that with:

```
git checkout master test_file
```

Using `git checkout` with a path takes the content of files. To grab the content of a specific *commit* from another branch, and apply it as a patch to your branch, use:

```
git cherry-pick <commit>
```

2.25 Git Stash

Before you can `git pull`, you need to have committed any changes you have made. If you find you want to pull, but you're not ready to commit, you have to temporarily "put aside" your uncommitted changes. For this, you can use the `git stash` command, like in the following example:

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)

In [2]: %%writefile Wales.md
        Mountains In Wales
        ==================

        * Pen y Fan
        * Tryfan
        * Snowdon
        * Glyder Fawr
        * Fan y Big
        * Cadair Idris

Overwriting Wales.md


In [3]: %%bash
        git stash
        git pull
```

```
No local changes to save
Already up to date.
```

By stashing your work first, your repository becomes clean, allowing you to pull. To restore your changes, use `git stash apply`.

```
In [4]: %%bash --no-raise-error
        git stash apply
```

```
No stash entries found.
```

The "Stash" is a way of temporarily saving your working area, and can help out in a pinch.

2.26 Tagging

Tags are easy to read labels for revisions, and can be used anywhere we would name a commit.

Produce real results *only* with tagged revisions

```
In [5]: %%bash
        git tag -a v1.0 -m "Release 1.0"
        git push --tags
```

```
To github.com:UCL/github-example.git
 ! [rejected]        v1.0 -> v1.0 (already exists)
error: failed to push some refs to 'git@github.com:UCL/github-example.git'
hint: Updates were rejected because the tag already exists in the remote.
```

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

        CalledProcessError                        Traceback (most recent call last)

        <ipython-input-5-30c586933bd0> in <module>
    ----> 1 get_ipython().run_cell_magic('bash', '', 'git tag -a v1.0 -m "Release 1.0"\ngit push --tags'


        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/core/interactiveshell.py in run_cel
       2350             with self.builtin_trap:
       2351                 args = (magic_arg_s, cell)
    -> 2352                 result = fn(*args, **kwargs)
       2353             return result
       2354


        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/core/magics/script.py in named_scri
        140             else:
        141                 line = script
    --> 142             return self.shebang(line, cell)
        143
        144         # write a basic docstring:
```

```
        </home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/decorator.py:decorator-gen-110

        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/core/magic.py in <lambda>(f, *a, *
        185     # but it's overkill for just that one bit of state.
        186     def magic_deco(arg):
    --> 187         call = lambda f, *a, **k: f(*a, **k)
        188
        189         if callable(arg):

        ~/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/core/magics/script.py in shebang(s
        243             sys.stderr.flush()
        244         if args.raise_error and p.returncode!=0:
    --> 245             raise CalledProcessError(p.returncode, cell, output=out, stderr=err)
        246
        247     def _run_script(self, p, cell, to_close):

        CalledProcessError: Command 'b'git tag -a v1.0 -m "Release 1.0"\ngit push --tags\n'' returned n
```

```
In [6]: %%writefile Pennines.md

        Mountains In the Pennines
        =========================

        * Cross Fell
```

```
Writing Pennines.md
```

```
In [7]: %%bash
        git add Pennines.md
        git commit -am "Add Pennines"
```

```
[master 4f09edc] Add Pennines
 1 file changed, 5 insertions(+)
 create mode 100644 Pennines.md
```

You can also use tag names in the place of commmit hashes, such as to list the history between particular commits:

```
In [8]: %%bash
        git log v1.0.. --graph --oneline
```

```
* 4f09edc Add Pennines
```

If .. is used without a following commit name, HEAD is assumed.

2.27 Working with generated files: gitignore

We often end up with files that are generated by our program. It is bad practice to keep these in Git; just keep the sources.

Examples include `.o` and `.x` files for compiled languages, `.pyc` files in Python.
In our example, we might want to make our .md files into a PDF with pandoc:

```
In [9]: %%writefile Makefile

        MDS=$(wildcard *.md)
        PDFS=$(MDS:.md=.pdf)

        default: $(PDFS)

        %.pdf: %.md
                pandoc $< -o $@

Writing Makefile
```

```
In [10]: %%bash
         make

make[1]: Entering directory '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_exam
pandoc Scotland.md -o Scotland.pdf
pandoc lakeland.md -o lakeland.pdf
pandoc Pennines.md -o Pennines.pdf
pandoc index.md -o index.pdf
pandoc Wales.md -o Wales.pdf
make[1]: Leaving directory '/home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/git_examp
```

We now have a bunch of output .pdf files corresponding to each Markdown file.
But we don't want those to show up in git:

```
In [11]: %%bash
         git status

On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>…" to include in what will be committed)
        Makefile
        Pennines.pdf
        Scotland.pdf
        Wales.pdf
        __pycache__/
        index.pdf
        lakeland.pdf
        wsd.py

nothing added to commit but untracked files present (use "git add" to track)
```

Use .gitignore files to tell Git not to pay attention to files with certain paths:

```
In [12]: %%writefile .gitignore
         *.pdf
```

```
Writing .gitignore
```

```
In [13]: %%bash
         git status
```

```
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>…" to include in what will be committed)
        .gitignore
        Makefile
        __pycache__/
        wsd.py

nothing added to commit but untracked files present (use "git add" to track)
```

```
In [14]: %%bash
         git add Makefile
         git add .gitignore
         git commit -am "Add a makefile and ignore generated files"
         git push
```

```
[master afac5fd] Add a makefile and ignore generated files
 2 files changed, 9 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 Makefile


To github.com:UCL/github-example.git
   d61e07e..afac5fd  master -> master
```

2.28 Git clean

Sometimes you end up creating various files that you do not want to include in version control. An easy way of deleting them (if that is what you want) is the `git clean` command, which will remove the files that git is not tracking.

```
In [15]: %%bash
         git clean -fX
```

```
Removing Pennines.pdf
Removing Scotland.pdf
Removing Wales.pdf
Removing index.pdf
Removing lakeland.pdf
```

```
In [16]: %%bash
         ls
```

```
index.md
lakeland.md
Makefile
Pennines.md
__pycache__
Scotland.md
Wales.md
wsd.py
```

- With -f: don't prompt
- with -d: remove directories
- with -x: Also remote .gitignored files
- with -X: Only remove .gitignore files

2.29 Hunks

2.29.1 Git Hunks

A "Hunk" is one git change. This changeset has three hunks:

```
+import matplotlib
+import numpy as np

 from matplotlib import pylab
 from matplotlib.backends.backend_pdf import PdfPages

+def increment_or_add(key,hash,weight=1):
+       if key not in hash:
+               hash[key]=0
+       hash[key]+=weight
+
 data_path=os.path.join(os.path.dirname(
                        os.path.abspath(__file__)),
-regenerate=False
+regenerate=True
```

2.29.2 Interactive add

`git add` and `git reset` can be used to stage/unstage a whole file, but you can use interactive mode to stage by hunk, choosing yes or no for each hunk.

```
git add -p myfile.py
```

```
+import matplotlib
+import numpy as np
#Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

2.30 GitHub pages

2.30.1 Yaml Frontmatter

GitHub will publish repositories containing markdown as web pages, automatically.

You'll need to add this content:

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

A pair of lines with three dashes, to the top of each markdown file. This is how GitHub knows which markdown files to make into web pages. Here's why for the curious.

```
In [17]: %%writefile index.md
         ---
         title: Github Pages Example
         ---
         Mountains and Lakes in the UK
         ==================

         Engerland is not very mountainous.
         But has some tall hills, and maybe a mountain or two depending on your definition.
```

```
Overwriting index.md
```

```
In [18]: %%bash
         git commit -am "Add github pages YAML frontmatter"
```

```
[master 136e57a] Add github pages YAML frontmatter
 1 file changed, 7 insertions(+), 4 deletions(-)
```

2.30.2 The gh-pages branch

GitHub creates github pages when you use a special named branch.

This is best used to create documentation for a program you write, but you can use it for anything.

```
In [19]: os.chdir(working_dir)
```

```
In [20]: %%bash

         git checkout -b gh-pages
         git push -uf origin gh-pages
```

```
Branch 'gh-pages' set up to track remote branch 'gh-pages' from 'origin'.
```

```
Switched to a new branch 'gh-pages'
To github.com:UCL/github-example.git
 + 59d92ab…136e57a gh-pages -> gh-pages (forced update)
```

The first time you do this, GitHub takes a few minutes to generate your pages.

The website will appear at `http://username.github.io/repositoryname`, for example: http://UCL.github.io/github-example/

2.30.3 UCL layout for GitHub pages

You can use GitHub pages to make HTML layouts, here's an example of how to do it, and how it looks. We won't go into the detail of this now, but after the class, you might want to try this.

2.31 Working with multiple remotes

2.31.1 Distributed versus centralised

Older version control systems (cvs, svn) were "centralised"; the history was kept only on a server, and all commits required an internet.

Centralised	Distributed
Server has history	Every user has full history
Your computer has one snapshot	Many local branches
To access history, need internet	History always available
You commit to remote server	Users synchronise histories
cvs, subversion(svn)	git, mercurial (hg), bazaar (bzr)

With modern distributed systems, we can add a second remote. This might be a personal *fork* on github:

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        working_dir = os.path.join(git_dir, 'git_example')
        os.chdir(working_dir)
```

```
In [2]: %%bash
        git checkout master
        git remote add rits git@github.com:ucl-rits/github-example.git
        git remote -v
```

```
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
origin        git@github.com:UCL/github-example.git (fetch)
origin        git@github.com:UCL/github-example.git (push)
rits        git@github.com:ucl-rits/github-example.git (fetch)
rits        git@github.com:ucl-rits/github-example.git (push)


Switched to branch 'master'
```

We can push to a named remote:

```
In [3]: %%writefile Pennines.md

        Mountains In the Pennines
        =========================

        * Cross Fell
        * Whernside
```

```
Overwriting Pennines.md
```

```
In [4]: %%bash
        git commit -am "Add Whernside"
```

```
[master f8b480d] Add Whernside
 1 file changed, 1 insertion(+)
```

```
In [5]: %%bash
        git push -uf rits master
```

```
Branch 'master' set up to track remote branch 'master' from 'rits'.


remote: This repository moved. Please use the new location:
remote:   git@github.com:UCL-RITS/github-example.git
To github.com:ucl-rits/github-example.git
 + f214a57…f8b480d master -> master (forced update)
```

2.31.2 Referencing remotes

You can always refer to commits on a remote like this:

```
In [6]: %%bash
        git fetch
        git log --oneline --left-right rits/master...origin/master
```

```
< f8b480d Add Whernside
< 136e57a Add github pages YAML frontmatter


From github.com:ucl-rits/github-example
 * [new branch]      gh-pages   -> rits/gh-pages
```

To see the differences between remotes, for example.
To see what files you have changed that aren't updated on a particular remote, for example:

```
In [7]: %%bash
        git diff --name-only origin/master
```

```
Pennines.md
index.md
```

When you reference remotes like this, you're working with a cached copy of the last time you interacted with the remote. You can do `git fetch` to update local data with the remotes without actually pulling. You can also get useful information about whether tracking branches are ahead or behind the remote branches they track:

```
In [8]: %%bash
        git branch -vv
```

```
  gh-pages 136e57a [origin/gh-pages] Add github pages YAML frontmatter
* master   f8b480d [rits/master] Add Whernside
```

2.32 Hosting Servers

2.32.1 Hosting a local server

- Any repository can be a remote for pulls
- Can pull/push over shared folders or ssh

- Pushing to someone's working copy is dangerous
- Use `git init --bare` to make a copy for pushing
- You don't need to create a "server" as such, any 'bare' git repo will do.

```
In [9]: bare_dir = os.path.join(git_dir, 'bare_repo')
        os.chdir(git_dir)

In [10]: %%bash
         mkdir -p bare_repo
         cd bare_repo
         git init --bare

Initialized empty Git repository in /home/travis/build/UCL/rsd-engineeringcourse/ch02git/learning_git/ba


In [11]: os.chdir(working_dir)

In [12]: %%bash
         git remote add local_bare ../bare_repo
         git push -u local_bare master

Branch 'master' set up to track remote branch 'master' from 'local_bare'.


To ../bare_repo
 * [new branch]      master -> master


In [13]: %%bash
         git remote -v

local_bare        ../bare_repo (fetch)
local_bare        ../bare_repo (push)
origin        git@github.com:UCL/github-example.git (fetch)
origin        git@github.com:UCL/github-example.git (push)
rits        git@github.com:ucl-rits/github-example.git (fetch)
rits        git@github.com:ucl-rits/github-example.git (push)
```

You can now work with this local repository, just as with any other git server. If you have a colleague on a shared file system, you can use this approach to collaborate through that file system.

2.32.2 Home-made SSH servers

Classroom exercise: Try creating a server for yourself using a machine you can SSH to:

```
ssh <mymachine>
mkdir mygitserver
cd mygitserver
git init --bare
exit
git remote add <somename> ssh://user@host/mygitserver
git push -u <somename> master
```

2.33 SSH keys and GitHub

Classroom exercise: If you haven't already, you should set things up so that you don't have to keep typing in your password whenever you interact with GitHub via the command line.

You can do this with an "ssh keypair". You may have created a keypair in the Software Carpentry shell training. Go to the ssh settings page on GitHub and upload your public key by copying the content from your computer. (Probably at .ssh/id_rsa.pub)

If you have difficulties, the instructions for this are on the GitHub website.

2.34 Rebasing

2.34.1 Rebase vs merge

A git *merge* is only one of two ways to get someone else's work into yours. The other is called a rebase.

In a merge, a revision is added, which brings the branches together. Both histories are retained. In a rebase, git tries to work out

> What would you need to have done, to make your changes, if your colleague had already made theirs?

Git will invent some new revisions, and the result will be a repository with an apparently linear history. This can be useful if you want a cleaner, non-branching history, but it has the risk of creating inconsistencies, since you are, in a way, "rewriting" history.

2.34.2 An example rebase

We've built a repository to help visualise the difference between a merge and a rebase, at https://github.com/UCL-RITS/wocky_rebase/blob/master/wocky.md .

The initial state of both collaborators is a text file, wocky.md:

```
It was clear and cold,
and the slimy monsters
```

On the master branch, a second commit ('Dancing') has been added:

```
It was clear and cold,
and the slimy monsters
danced and spun in the waves
```

On the "Carollian" branch, a commit has been added translating the initial state into Lewis Caroll's language:

```
'Twas brillig,
and the slithy toves
```

So the logs look like this:

```
git log --oneline --graph master

* 2a74d89 Dancing
* 6a4834d Initial state

git log --oneline --graph carollian

* 2232bf3 Translate into Caroll's language
* 6a4834d Initial state
```

If we now **merge** carollian into master, the final state will include both changes:

```
'Twas brillig,
and the slithy toves
danced and spun in the waves
```

But the graph shows a divergence and then a convergence:

```
git log --oneline --graph
```

```
*   b41f869 Merge branch 'carollian' into master_merge_carollian
|\
| * 2232bf3 Translate into Caroll's language
* | 2a74d89 Dancing
|/
* 6a4834d Initial state
```

But if we **rebase**, the final content of the file is still the same, but the graph is different:

```
git log --oneline --graph master_rebase_carollian
```

```
* df618e0 Dancing
* 2232bf3 Translate into Caroll's language
* 6a4834d Initial state
```

We have essentially created a new history, in which our changes come after the ones in the carollian branch. Note that, in this case, the hash for our "Dancing" commit has changed (from `2a74d89` to `df618e0`)!

To trigger the rebase, we did:

```
git checkout master
git rebase carollian
```

If this had been a remote, we would merge it with:

```
git pull --rebase
```

2.34.3 Fast Forwards

If we want to continue with the translation, and now want to merge the rebased branch into the carollian branch, we get:

```
git checkout carollian
git merge master
```

```
Updating 2232bf3..df618e0
Fast-forward
 wocky.md | 1 +
 1 file changed, 1 insertion(+)
```

The master branch was already **rebased on** the carollian branch, so this merge was just a question of updating *metadata* (moving the label for the carollian branch so that it points to the same commit master does): a "fast forward".

2.34.4 Rebasing pros and cons

Some people like the clean, apparently linear history that rebase provides.

But *rebase rewrites history.*

If you've already pushed, or anyone else has got your changes, things will get screwed up.

If you know your changes are still secret, it might be better to rebase to keep the history clean. If in doubt, just merge.

2.35 Squashing

A second way to use the `git rebase` command is to rebase your work on top of one of *your own* earlier commits, in interactive mode (`-i`). A common use of this is to "squash" several commits that should really be one, i.e. combine them into a single commit that contains all their changes:

```
git log
```

```
ea15 Some good work
ll54 Fix another typo
de73 Fix a typo
ab11 A great piece of work
cd27 Initial commit
```

2.35.1 Using rebase to squash

If we type

```
git rebase -i ab11 #OR HEAD^^
```

an edit window pops up with:

```
pick cd27 Initial commit
pick ab11 A great piece of work
pick de73 Fix a typo
pick ll54 Fix another typo
pick ea15 Some good work

# Rebase 60709da..30e0ccb onto 60709da
#
# Commands:
#  p, pick = use commit
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
```

We can rewrite select commits to be merged, so that the history is neater before we push. This is a great idea if you have lots of trivial typo commits.

```
pick cd27 Initial commit
pick ab11 A great piece of work
squash de73 Fix a typo
squash ll54 Fix another typo
pick ea15 Some good work
```

save the interactive rebase config file, and rebase will build a new history:

```
git log
```

```
de82 Some good work
fc52 A great piece of work
cd27 Initial commit
```

Note the commit hash codes for 'Some good work' and 'A great piece of work' have changed, as the change they represent has changed.

2.36 Debugging With Git Bisect

You can use

```
git bisect
```

to find out which commit caused a bug.

2.36.1 An example repository

In a nice open source example, I found an arbitrary exemplar on github

```
In [1]: import os
        top_dir = os.getcwd()
        git_dir = os.path.join(top_dir, 'learning_git')
        os.chdir(git_dir)

In [2]: %%bash
        rm -rf bisectdemo
        git clone git@github.com:shawnsi/bisectdemo.git
```

```
Cloning into 'bisectdemo'…
```

```
In [3]: bisect_dir=os.path.join(git_dir,'bisectdemo')
        os.chdir(bisect_dir)

In [4]: %%bash
        python squares.py 2 # 4
```

```
4
```

This has been set up to break itself at a random commit, and leave you to use bisect to work out where it has broken:

```
In [5]: %%bash
        ./breakme.sh > break_output
```

```
error: branch 'buggy' not found.
Switched to a new branch 'buggy'
```

Which will make a bunch of commits, of which one is broken, and leave you in the broken final state

```
In [6]: python squares.py 2 # Error message
```

```
  File "<ipython-input-6-8e2377cd54bf>", line 1
    python squares.py 2 # Error message
                ^
SyntaxError: invalid syntax
```

2.36.2 Bisecting manually

```
In [7]: %%bash
        git bisect start
        git bisect bad # We know the current state is broken
        git checkout master
        git bisect good # We know the master branch state is OK
```

```
Your branch is up to date with 'origin/master'.
Bisecting: 500 revisions left to test after this (roughly 9 steps)
[506a15c8e30778254808357b84bbd7ba9aa63346] Comment 500


Switched to branch 'master'
```

Bisect needs one known good and one known bad commit to get started

2.36.3 Solving Manually

```
python squares.py 2 # 4
git bisect good
python squares.py 2 # 4
git bisect good
python squares.py 2 # 4
git bisect good
python squares.py 2 # Crash
git bisect bad
python squares.py 2 # Crash
git bisect bad
python squares.py 2 # Crash
git bisect bad
python squares.py 2 #Crash
git bisect bad
python squares.py 2 # 4
git bisect good
python squares.py 2 # 4
git bisect good
python squares.py 2 # 4
git bisect good
```

And eventually:

```
git bisect good
    Bisecting: 0 revisions left to test after this (roughly 0 steps)

python squares.py 2
    4

git bisect good
2777975a2334c2396ccb9faf98ab149824ec465b is the first bad commit
commit 2777975a2334c2396ccb9faf98ab149824ec465b
Author: Shawn Siefkas <shawn.siefkas@meredith.com>
Date:   Thu Nov 14 09:23:55 2013 -0600

    Breaking argument type

git bisect end
```

2.36.4 Solving automatically

If we have an appropriate unit test, we can do all this automatically:

```
In [8]: %%bash
        git bisect start
        git bisect bad HEAD # We know the current state is broken
        git bisect good master # We know master is good
        git bisect run python squares.py 2
```

```
Bisecting: 500 revisions left to test after this (roughly 9 steps)
[506a15c8e30778254808357b84bbd7ba9aa63346] Comment 500
running python squares.py 2
4
Bisecting: 250 revisions left to test after this (roughly 8 steps)
[09854dcad82b8ec61647fadf034cab39c844cdae] Comment 749
running python squares.py 2
Bisecting: 124 revisions left to test after this (roughly 7 steps)
[5caba758a742eed77ebb77ef06c5801891718fb2] Comment 624
running python squares.py 2
Bisecting: 62 revisions left to test after this (roughly 6 steps)
[d624fbee4f4ac036b1ec5b56df48707843d8524d] Comment 561
running python squares.py 2
Bisecting: 30 revisions left to test after this (roughly 5 steps)
[d96c67fe3c86fff4096b80cab5c823c752f25c53] Comment 531
running python squares.py 2
4
Bisecting: 15 revisions left to test after this (roughly 4 steps)
[c46edadf25043cdc340e5027d718f1ed10491153] Comment 546
running python squares.py 2
4
Bisecting: 7 revisions left to test after this (roughly 3 steps)
[904cd2be98edbcde00ab0dbea3bf40e048b91c18] Comment 554
running python squares.py 2
4
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[9dffd0ade41d66d36884aca53aa3890860e0359a] Breaking argument type
running python squares.py 2
Bisecting: 1 revision left to test after this (roughly 1 step)
[2db7d4a5a1f8320ff576dc9594dc127d2a61c2e9] Comment 556
running python squares.py 2
4
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[250785c190bf080461dbecef6c88347a7d914a68] Comment 557
running python squares.py 2
4
9dffd0ade41d66d36884aca53aa3890860e0359a is the first bad commit
commit 9dffd0ade41d66d36884aca53aa3890860e0359a
Author: Shawn Siefkas <shawn.siefkas@meredith.com>
Date:   Thu Nov 14 09:23:55 2013 -0600

    Breaking argument type

 squares.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

```
bisect run success


Previous HEAD position was 506a15c Comment 500
Switched to branch 'buggy'
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    print(integer**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    print(integer**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    print(integer**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    print(integer**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Boom!

Chapter 3

Testing

3.1 Introduction

When programming, it is very important to know that the code we have written does what it was intended. Unfortunately, this step is often skipped in scientific programming, especially when developing code for our own personal work.

Researchers sometimes check that their code behaves correctly by manually running it on some sample data and inspecting the results. However, it is much better and safer to automate this process, so the tests can be run often – perhaps even after each new commit! This not only reassures us that the code behaves as it should at any given moment, it also gives us more flexibility to change it, because we have a way of knowing when we have broken something by accident.

In this chapter, we will mostly look at how to write **unit tests**, which check the behaviour of small parts of our code. We will work with a particular framework for Python code, but the principles we discuss are general. We will also look at how to use a debugger to locate problems in our code, and services that simplify the automated running of tests.

3.1.1 A few reasons not to do testing

Sensibility	Sense
It's boring	*Maybe*
Code is just a one off throwaway	*As with most research codes*
No time for it	*A bit more code, a lot less debugging*
Tests can be buggy too	*See above*
Not a professional programmer	*See above*
Will do it later	*See above*

3.1.2 A few reasons to do testing

- **laziness**: testing saves time
- **peace of mind**: tests (should) ensure code is correct
- **runnable specification**: best way to let others know what a function should do and not do
- **reproducible debugging**: debugging that happened and is saved for later reuse
- **code structure / modularity**: since we may have to call parts of the code independently during the tests
- **ease of modification**: since results can be tested

3.1.3 Not a panacea

> Trying to improve the quality of software by doing more testing is like trying to lose weight by weighting yourself more often. - Steve McConnell

- Testing won't corrrect a buggy code
- Testing will tell you were the bugs are…
- … if the test cases *cover* the bugs

3.1.4 Tests at different scales

Level of test	Area covered by test
Unit testing	smallest logical block of work (often < 10 lines of code)
Component testing	several logical blocks of work together
Integration testing	all components together / whole program

Always start at the smallest scale!

If a unit test is too complicated, go smaller.

3.1.5 Legacy code hardening

- Very difficult to create unit-tests for existing code
- Instead we make a **regression test**
- Run program as a black box:

```
setup input
run program
read output
check output against expected result
```

- Does not test correctness of code
- Checks code is a similarly wrong on day N as day 0

3.1.6 Testing vocabulary

- **fixture**: input data
- **action**: function that is being tested
- **expected result**: the output that should be obtained
- **actual result**: the output that is obtained
- **coverage**: proportion of all possible paths in the code that the tests take

3.1.7 Branch coverage:

```
if energy > 0:
    ! Do this
else:
    ! Do that
```

Is there a test for both `energy > 0` and `energy <= 0`?

3.2 How to Test

3.2.1 Equivalence partitioning

Think hard about the different cases the code will run under: this is science, not coding!

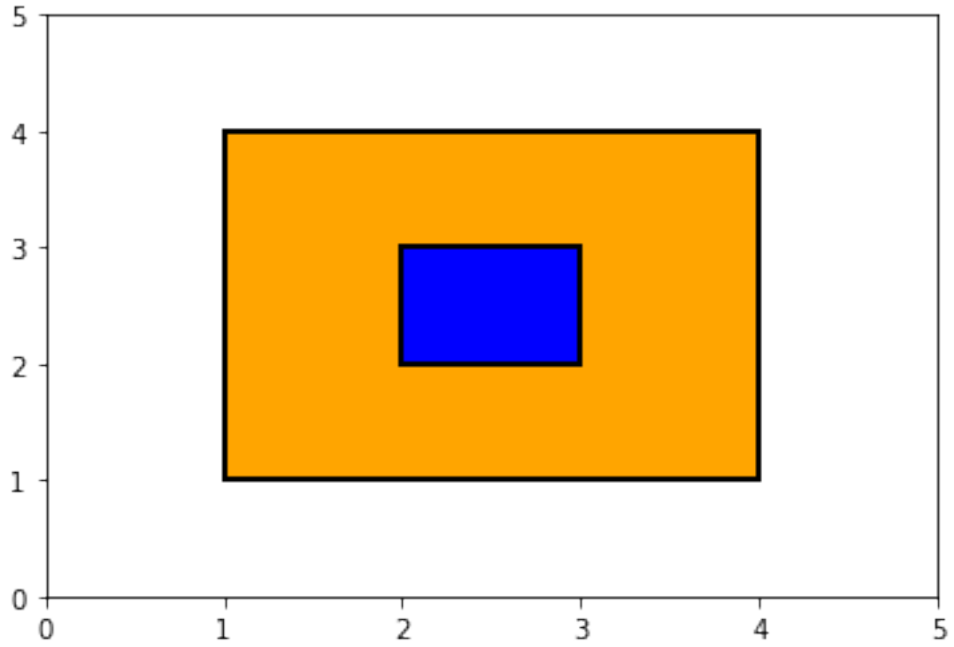
We can't write a test for every possible input: this is an infinite amount of work.
We need to write tests to rule out different bugs. There's no need to separately test *equivalent* inputs.
Let's look at an example of this question outside of coding:

- Research Project : Evolution of agricultural fields in Saskatchewan from aerial photography
- In silico translation : Compute overlap of two rectangles

```
In [1]: import matplotlib.pyplot as plt
        from matplotlib.path import Path
        import matplotlib.patches as patches
        %matplotlib inline
```

Let's make a little fragment of matplotlib code to visualise a pair of fields.

```
In [2]: def show_fields(field1, field2):
            def vertices(left, bottom, right, top):
                verts = [(left, bottom),
                         (left, top),
                         (right, top),
                         (right, bottom),
                         (left, bottom)]
                return verts

            codes = [Path.MOVETO,
                     Path.LINETO,
                     Path.LINETO,
                     Path.LINETO,
                     Path.CLOSEPOLY]
            path1 = Path(vertices(*field1), codes)
            path2 = Path(vertices(*field2), codes)
            fig = plt.figure()
            ax = fig.add_subplot(111)
            patch1 = patches.PathPatch(path1, facecolor='orange', lw=2)
            patch2 = patches.PathPatch(path2, facecolor='blue', lw=2)
            ax.add_patch(patch1)
            ax.add_patch(patch2)
            ax.set_xlim(0,5)
            ax.set_ylim(0,5)

        show_fields((1.,1.,4.,4.), (2.,2.,3.,3.))
```



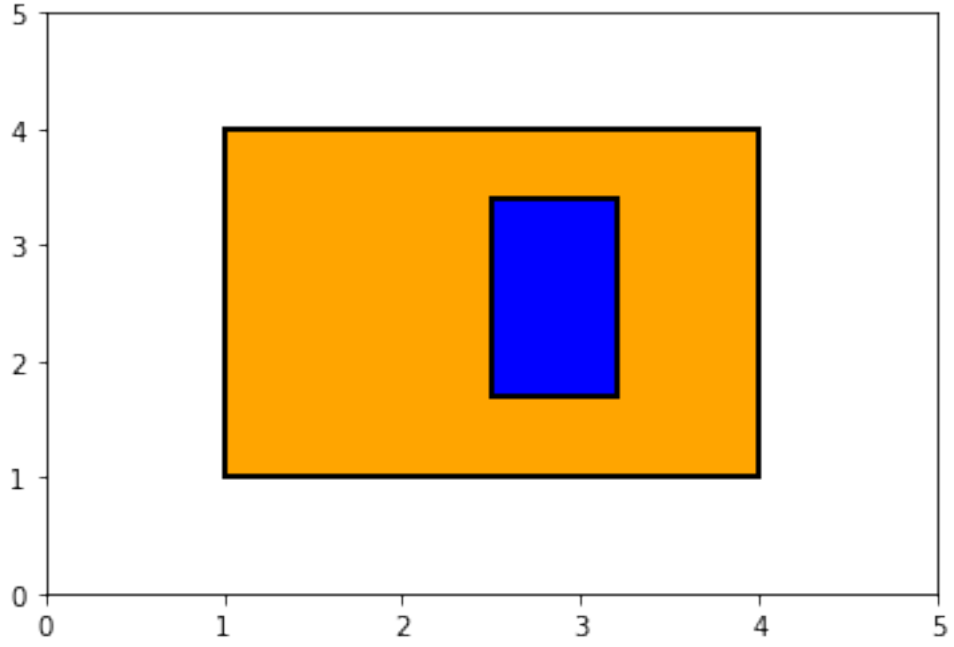
Here, we can see that the area of overlap, is the same as the smaller field, with area 1.

We could now go ahead and write a subroutine to calculate that, and also write some test cases for our answer.

But first, let's just consider that question abstractly, what other cases, *not equivalent to this* might there be?

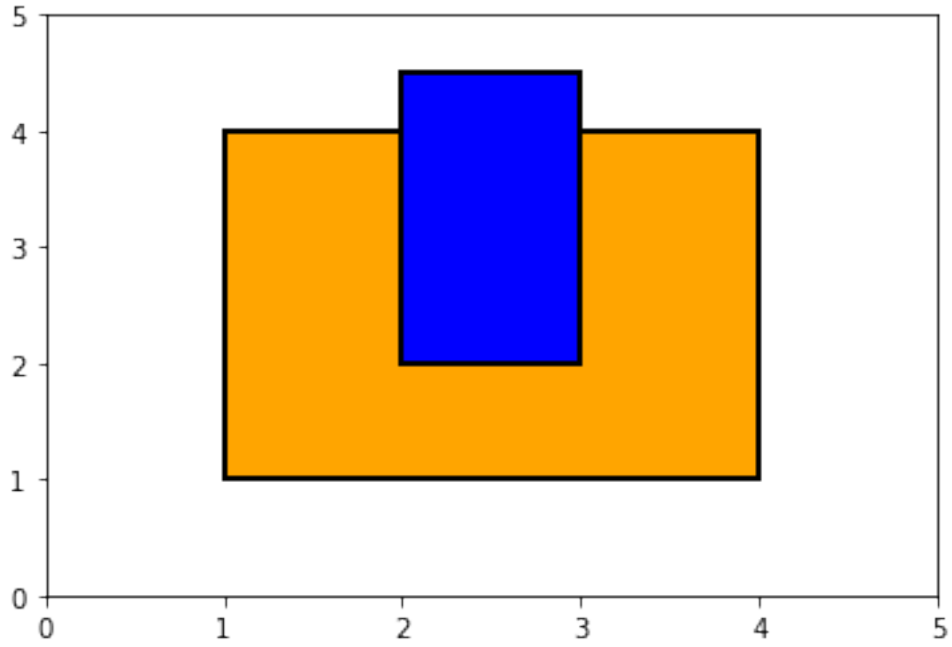
For example, this case, is still just a full overlap, and is sufficiently equivalent that it's not worth another test:

```
In [3]: show_fields((1.,1.,4.,4.),(2.5,1.7,3.2,3.4))
```



But this case is no longer a full overlap, and should be tested separately:

```
In [4]: show_fields((1.,1.,4.,4.),(2.,2.,3.,4.5))
```



On a piece of paper, sketch now the other cases you think should be treated as non-equivalent. Some answers are below:

```
In [5]: for _ in range(10):
            print("\n\n\nSpoiler space\n\n\n")
```

```
Spoiler space
```

```
Spoiler space
```

```
Spoiler space
```

Spoiler space

Spoiler space

Spoiler space

Spoiler space

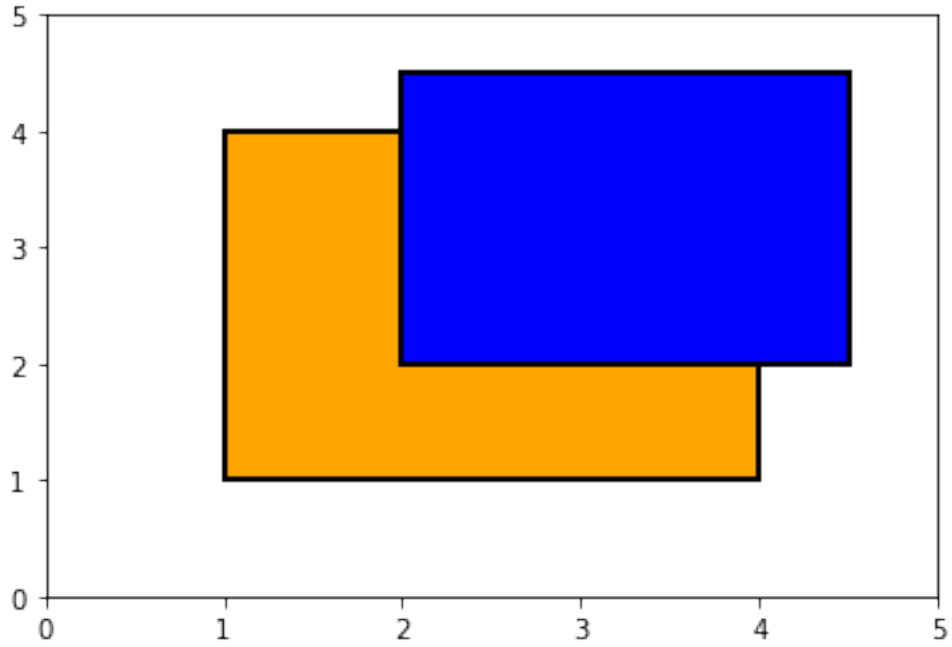
Spoiler space

Spoiler space

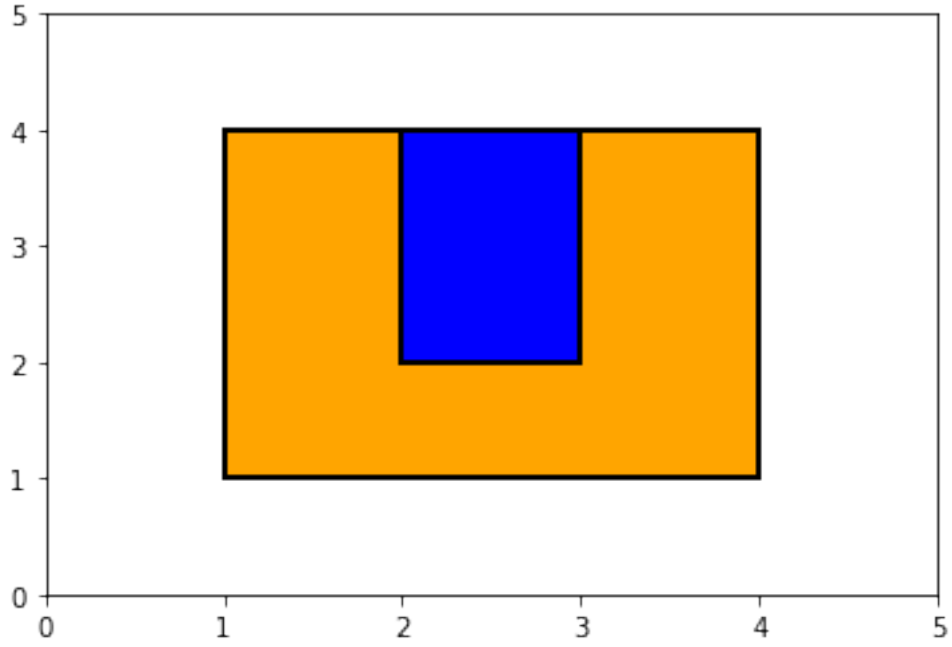
Spoiler space

Spoiler space

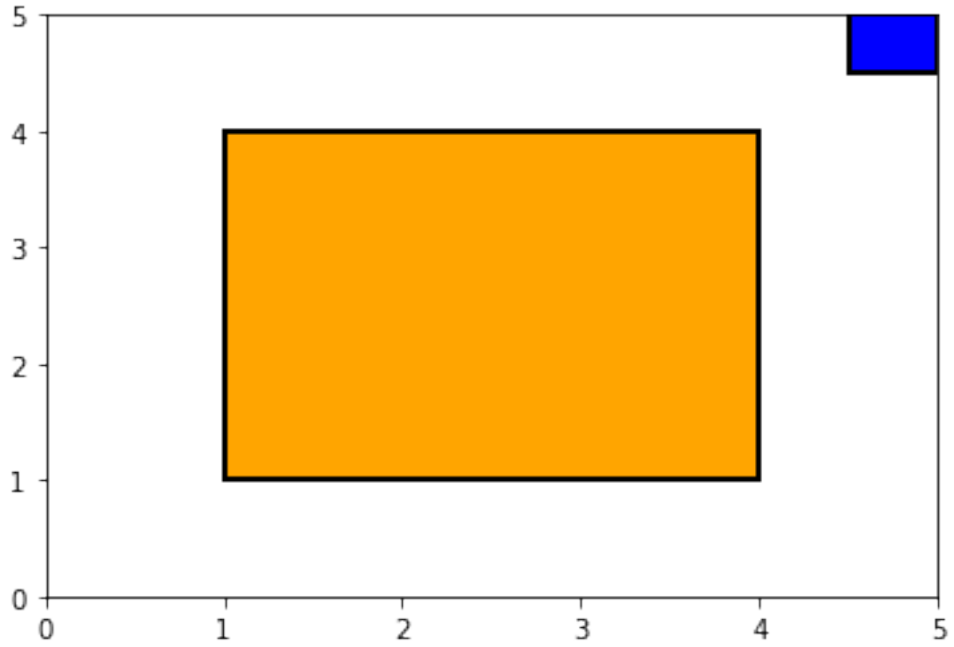
```
In [6]: show_fields((1.,1.,4.,4.),(2,2,4.5,4.5)) # Overlap corner
```



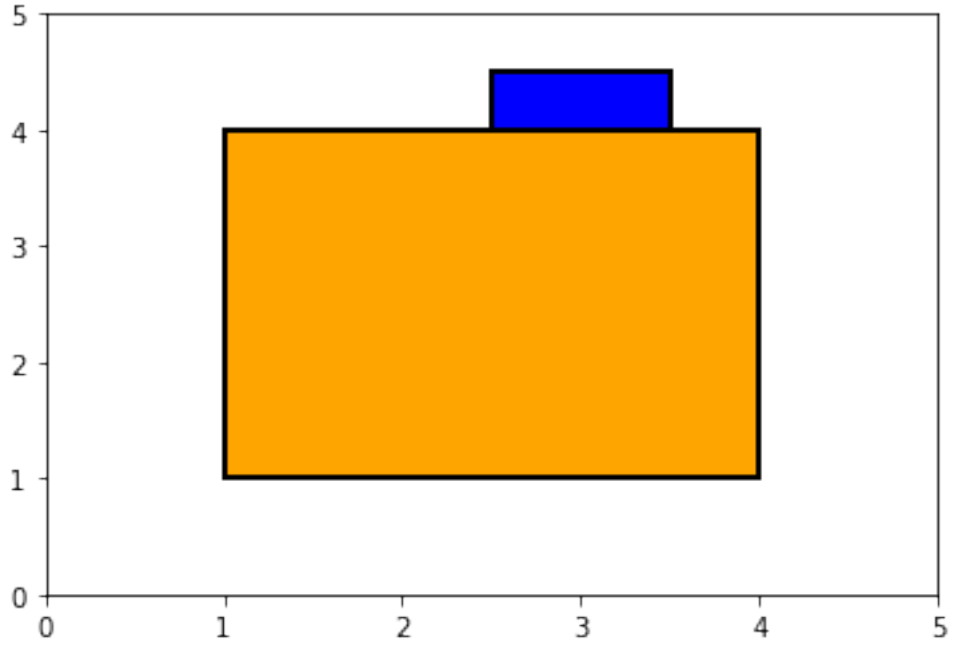

```
In [7]: show_fields((1.,1.,4.,4.),(2.,2.,3.,4.)) # Just touching
```



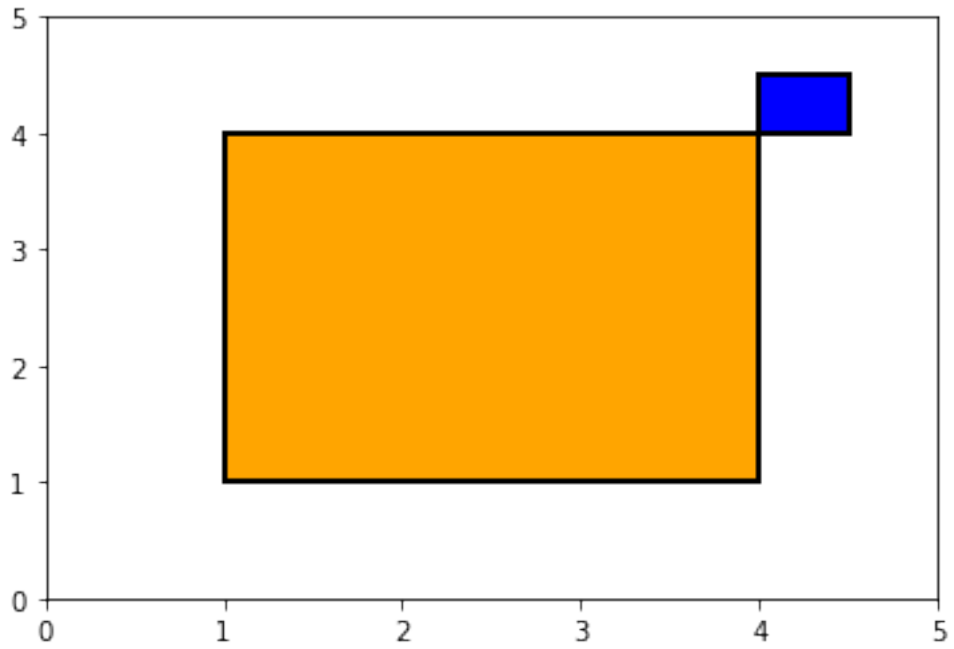

```
In [8]: show_fields((1.,1.,4.,4.),(4.5,4.5,5,5)) # No overlap
```




```
In [9]: show_fields((1.,1.,4.,4.),(2.5,4,3.5,4.5)) # Just touching from outside
```




```
In [10]: show_fields((1.,1.,4.,4.),(4,4,4.5,4.5)) # Touching corner
```



3.2.2 Using our tests

OK, so how might our tests be useful?

Here's some code that **might** correctly calculate the area of overlap:

```
In [11]: def overlap(field1, field2):
             left1, bottom1, top1, right1 = field1
             left2, bottom2, top2, right2 = field2
             overlap_left = max(left1, left2)
             overlap_bottom = max(bottom1, bottom2)
             overlap_right = min(right1, right2)
             overlap_top = min(top1, top2)
             overlap_height = (overlap_top-overlap_bottom)
             overlap_width = (overlap_right-overlap_left)
             return overlap_height * overlap_width
```

So how do we check our code?

The manual approach would be to look at some cases, and, once, run it and check:

```
In [12]: overlap((1.,1.,4.,4.),(2.,2.,3.,3.))
```

```
Out[12]: 1.0
```

That looks OK.

But we can do better, we can write code which **raises an error** if it gets an unexpected answer:

```
In [13]: assert overlap((1.,1.,4.,4.),(2.,2.,3.,3.)) == 1.0
```

```
In [14]: assert overlap((1.,1.,4.,4.),(2.,2.,3.,4.5)) == 2.0
```

```
In [15]: assert overlap((1.,1.,4.,4.),(2.,2.,4.5,4.5)) == 4.0
```

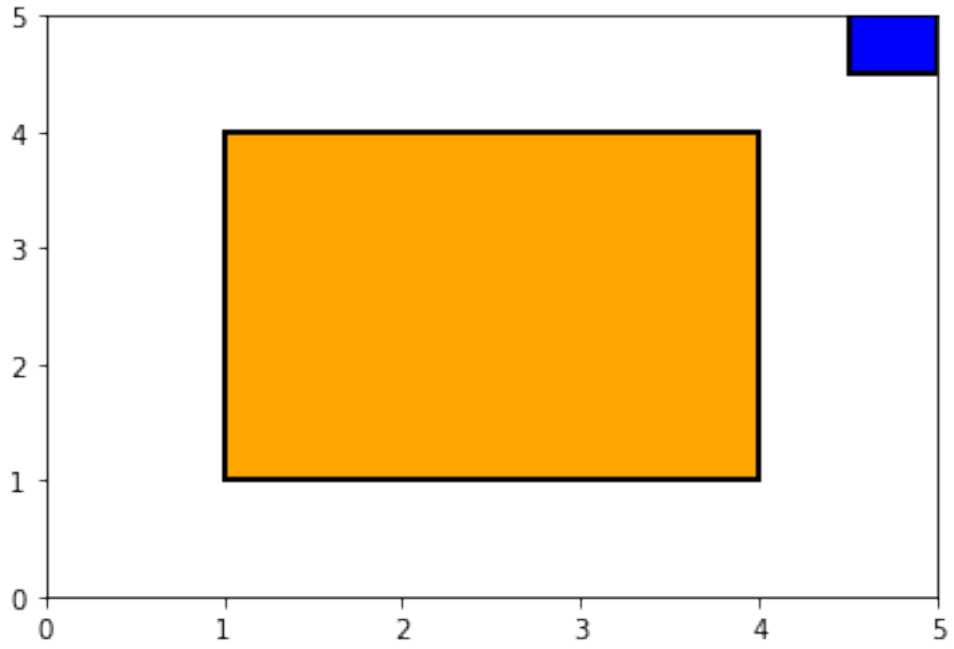
```
In [16]: assert overlap((1.,1.,4.,4.),(4.5,4.5,5,5)) == 0.0
```

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

<ipython-input-16-21bafdf6842e> in <module>
----> 1 assert overlap((1.,1.,4.,4.),(4.5,4.5,5,5)) == 0.0


AssertionError:
```

```
In [17]: print(overlap((1.,1.,4.,4.),(4.5,4.5,5,5)))
```

```
0.25
```

```
In [18]: show_fields((1.,1.,4.,4.),(4.5,4.5,5,5))
```



What? Why is this wrong?
In our calculation, we are actually getting:

```
In [19]: overlap_left = 4.5
         overlap_right = 4
         overlap_width = -0.5
         overlap_height = -0.5
```

Both width and height are negative, resulting in a positive area. The above code didn't take into account the non-overlap correctly.
It should be:

```
In [20]: def overlap(field1, field2):
             left1, bottom1, top1, right1 = field1
             left2, bottom2, top2, right2 = field2

             overlap_left = max(left1, left2)
             overlap_bottom = max(bottom1, bottom2)
             overlap_right = min(right1, right2)
             overlap_top = min(top1, top2)

             overlap_height = max(0, (overlap_top-overlap_bottom))
             overlap_width = max(0, (overlap_right-overlap_left))

             return overlap_height*overlap_width

In [21]: assert overlap((1,1,4,4), (2,2,3,3)) == 1.0
         assert overlap((1,1,4,4), (2,2,3,4.5)) == 2.0
         assert overlap((1,1,4,4), (2,2,4.5,4.5)) == 4.0
         assert overlap((1,1,4,4), (4.5,4.5,5,5)) == 0.0
         assert overlap((1,1,4,4), (2.5,4,3.5,4.5)) == 0.0
         assert overlap((1,1,4,4), (4,4,4.5,4.5)) == 0.0
```

Note, we reran our other tests, to check our fix didn't break something else. (We call that "fallout")

3.2.3 Boundary cases

"Boundary cases" are an important area to test:

- Limit between two equivalence classes: edge and corner sharing fields
- Wherever indices appear, check values at `0`, `N`, `N+1`
- Empty arrays:

```
atoms = [read_input_atom(input_atom) for input_atom in input_file]
energy = force_field(atoms)
```

- What happens if `atoms` is an empty list?
- What happens when a matrix/data-frame reaches one row, or one column?

3.2.4 Positive *and* negative tests

- **Positive tests**: code should give correct answer with various inputs
- **Negative tests**: code should crash as expected given invalid inputs, rather than lying

Bad input should be expected and should fail early and explicitly.
Testing should ensure that explicit failures do indeed happen.

3.2.5 Raising exceptions

In Python, we can signal an error state by raising an error:

```
In [22]: def I_only_accept_positive_numbers(number):
             # Check input
             if number < 0:
                 raise ValueError("Input {} is negative".format(number))

             # Do something
```

```
In [23]: I_only_accept_positive_numbers(5)

In [24]: I_only_accept_positive_numbers(-5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-ac3b0fd3c476> in <module>
----> 1 I_only_accept_positive_numbers(-5)

<ipython-input-22-198af6344050> in I_only_accept_positive_numbers(number)
      2     # Check input
      3     if number < 0:
----> 4         raise ValueError("Input {} is negative".format(number))
      5
      6     # Do something

ValueError: Input -5 is negative
```

There are standard "Exception" types, like `ValueError` we can `raise`
We would like to be able to write tests like this:

```
In [25]: assert I_only_accept_positive_numbers(-5) == # Gives a value error
```

```
  File "<ipython-input-25-55b8782568ca>", line 1
    assert I_only_accept_positive_numbers(-5) == # Gives a value error
                                                 ^
SyntaxError: invalid syntax
```

But to do that, we need to learn about more sophisticated testing tools, called "test frameworks".

3.3 Testing frameworks

3.3.1 Why use testing frameworks?

Frameworks should simplify our lives:

- Should be easy to add simple test
- Should be possible to create complex test:
  - Fixtures
  - Setup/Tear down
  - Parameterized tests (same test, mostly same input)
- Find all our tests in a complicated code-base
- Run all our tests with a quick command
- Run only some tests, e.g. `test --only "tests about fields"`
- **Report failing tests**
- Additional goodies, such as code coverage

3.3.2 Common testing frameworks

- Language agnostic: CTest
- Test runner for executables, bash scripts, etc…
- Great for legacy code hardening
- C unit-tests:
  - all c++ frameworks,
  - Check,
  - CUnit
- C++ unit-tests:
  - CppTest,
  - Boost::Test,
  - google-test,
  - Catch (best)
- Python unit-tests:
  - nose includes test discovery, coverage, etc
  - unittest comes with standard python library
  - pytest, branched off of nose
- R unit-tests:
  - RUnit,
  - svUnit
  - (works with SciViews GUI)
- Fortran unit-tests:
  - funit,
  - pfunit(works with MPI)

3.3.3 pytest framework: usage

pytest is a recommended python testing framework.

We can use its tools in the notebook for on-the-fly tests in the notebook. This, happily, includes the negative-tests example we were looking for a moment ago.

```
In [1]: def I_only_accept_positive_numbers(number):
            # Check input
            if number < 0:
                raise ValueError("Input {} is negative".format(number))

            # Do something

In [2]: from pytest import raises

In [3]: with raises(ValueError):
            I_only_accept_positive_numbers(-5)
```

but the real power comes when we write a test file alongside our code files in our homemade packages:

```
In [4]: %%bash
        mkdir -p saskatchewan
        touch saskatchewan/__init__.py
```

```
In [5]: %%writefile saskatchewan/overlap.py
        def overlap(field1, field2):
            left1, bottom1, top1, right1 = field1
            left2, bottom2, top2, right2 = field2

            overlap_left = max(left1, left2)
            overlap_bottom = max(bottom1, bottom2)
            overlap_right = min(right1, right2)
            overlap_top = min(top1, top2)
            # Here's our wrong code again
            overlap_height = (overlap_top - overlap_bottom)
            overlap_width = (overlap_right - overlap_left)

            return overlap_height * overlap_width
```

```
Writing saskatchewan/overlap.py
```

```
In [6]: %%writefile saskatchewan/test_overlap.py
        from .overlap import overlap

        def test_full_overlap():
            assert overlap((1.,1.,4.,4.), (2.,2.,3.,3.)) == 1.0

        def test_partial_overlap():
            assert overlap((1,1,4,4), (2,2,3,4.5)) == 2.0

        def test_no_overlap():
            assert overlap((1,1,4,4), (4.5,4.5,5,5)) == 0.0
```

```
Writing saskatchewan/test_overlap.py
```

```
In [7]: %%bash --no-raise-error
        cd saskatchewan
        pytest
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/saskatchewan
plugins: cov-2.8.1
collected 3 items

test_overlap.py ..F                                                      [100%]

=================================== FAILURES ===================================
_____________________________ test_no_overlap ______________________________

    def test_no_overlap():
>       assert overlap((1,1,4,4), (4.5,4.5,5,5)) == 0.0
E       assert 0.25 == 0.0
```

```
E        +  where 0.25 = overlap((1, 1, 4, 4), (4.5, 4.5, 5, 5))

test_overlap.py:10: AssertionError
======================== 1 failed, 2 passed in 0.03s ========================
```

Note that it reported **which** test had failed, how many tests ran, and how many failed.
The symbol `..F` means there were three tests, of which the third one failed.
Pytest will:

- automagically finds files `test_*.py`
- collects all subroutines called `test_*`
- runs tests and reports results

Some options:

- help: `pytest --help`
- run only tests for a given feature: `pytest -k foo` # tests with 'foo' in the test name

3.4 Testing with floating points

3.4.1 Floating points are not reals

Floating points are inaccurate representations of real numbers:
`1.0 == 0.9999999999999999` is true to the last bit.
This can lead to numerical errors during calculations: $1000(a-b) \neq 1000a - 1000b$

```
In [8]: 1000.0 * 1.0 - 1000.0 * 0.9999999999999998

Out[8]: 2.2737367544323206e-13

In [9]: 1000.0 * (1.0 - 0.9999999999999998)

Out[9]: 2.220446049250313e-13
```

Both results are wrong: `2e-13` is the correct answer.
The size of the error will depend on the magnitude of the floating points:

```
In [10]: 1000.0 * 1e5 - 1000.0 * 0.9999999999999998e5

Out[10]: 1.4901161193847656e-08
```

The result should be `2e-8`.

3.4.2 Comparing floating points

Use the "approx", for a default of a relative tolerance of 10^{-6}

```
In [11]: from pytest import approx
         assert  0.7 == approx(0.7 + 1e-7)
```

Or be more explicit:

```
In [12]: magnitude = 0.7
         assert 0.7 == approx(0.701 , rel=0.1, abs=0.1)
```

Choosing tolerances is a big area of debate.

3.4.3 Comparing vectors of floating points

Numerical vectors are best represented using numpy.

```
In [13]: from numpy import array, pi

         vector_of_reals = array([0.1, 0.2, 0.3, 0.4]) * pi
```

Numpy ships with a number of assertions (in `numpy.testing`) to make comparison easy:

```
In [14]: from numpy import array, pi
         from numpy.testing import assert_allclose
         expected = array([0.1, 0.2, 0.3, 0.4, 1e-12]) * pi
         actual = array([0.1, 0.2, 0.3, 0.4, 2e-12]) * pi
         actual[:-1] += 1e-6

         assert_allclose(actual, expected, rtol=1e-5, atol=1e-8)
```

It compares the difference between `actual` and `expected` to `atol + rtol * abs(expected)`.

3.5 Classroom exercise: energy calculation

3.5.1 Diffusion model in 1D

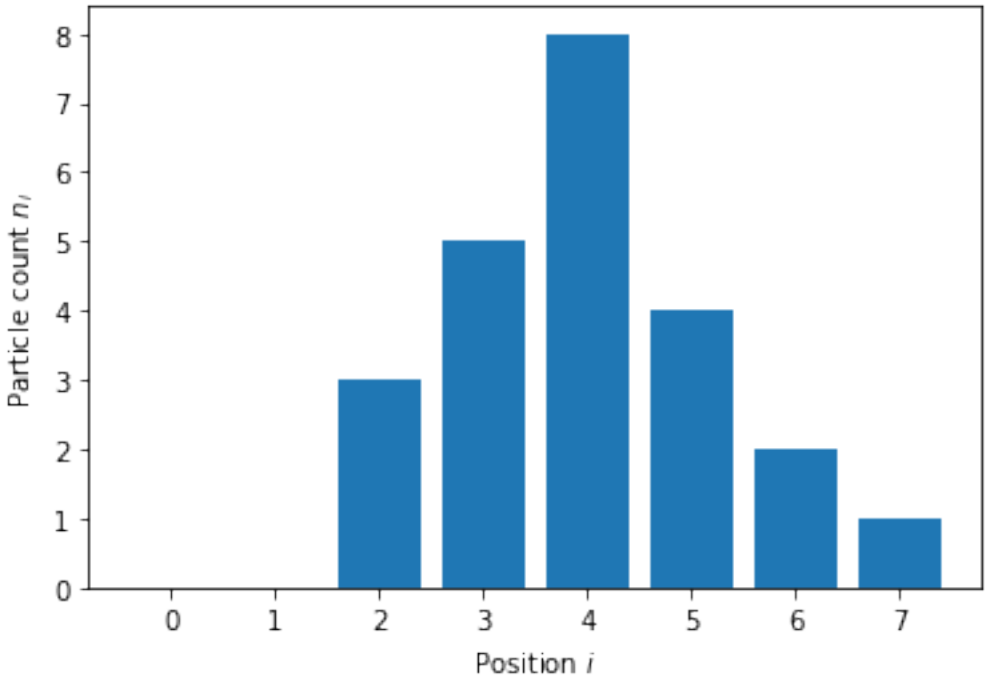
Description: A one-dimensional diffusion model. (Could be a gas of particles, or a bunch of crowded people in a corridor, or animals in a valley habitat…)

- Agents are on a 1d axis
- Agents do not want to be where there are other agents
- This is represented as an 'energy': the higher the energy, the more unhappy the agents.

Implementation:

- Given a vector n of positive integers, and of arbitrary length
- Compute the energy, $E(n) = \sum_i n_i(n_i - 1)$
- Later, we will have the likelyhood of an agent moving depend on the change in energy.

```
In [1]: import numpy as np
        from matplotlib import pyplot as plt
        %matplotlib inline

        density =  np.array([0, 0, 3, 5, 8, 4, 2, 1])
        fig, ax = plt.subplots()
        ax.bar(np.arange(len(density)), density)
        ax.xrange=[-0.5, len(density)-0.5]
        ax.set_ylabel("Particle count $n_i$")
        ax.set_xlabel("Position $i$")

Out[1]: Text(0.5, 0, 'Position $i$')
```



Here, the total energy due to position 2 is $3(3-1) = 6$, and due to column 7 is $1(1-1) = 0$. We need to sum these to get the total energy.

3.5.2 Starting point

Create a Python module:

```
In [2]: %%bash
        mkdir -p diffusion
        touch diffusion/__init__.py
```

- Implementation file: diffusion_model.py

```
In [3]: %%writefile diffusion/model.py
        def energy(density, coeff=1.0):
            """
            Energy associated with the diffusion model

            Parameters
            ----------

            density: array of positive integers
                Number of particles at each position i in the array
            coeff: float
                Diffusion coefficient.
            """
          # implementation goes here
```

```
Writing diffusion/model.py
```

- Testing file: test_diffusion_model.py

```
In [4]: %%writefile diffusion/test_model.py
        from .model import energy
        def test_energy():
            """ Optional description for reporting """
            # Test something
```

```
Writing diffusion/test_model.py
```

Invoke the tests:

```
In [5]: %%bash
        cd diffusion
        pytest
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/diffusion
plugins: cov-2.8.1
collected 1 item

test_model.py .                                                          [100%]

============================== 1 passed in 0.01s ===============================
```

Now, write your code (in `model.py`), and tests (in `test_model.py`), testing as you do.

3.5.3 Solution

Don't look until after you've tried!

```
In [6]: %%writefile diffusion/model.py
        """  Simplistic 1-dimensional diffusion model """

        def energy(density):
            """
            Energy associated with the diffusion model

            :Parameters:

            density: array of positive integers
              Number of particles at each position i in the array/geometry
            """
            from numpy import array, any, sum

            # Make sure input is an numpy array
            density = array(density)

            # ...of the right kind (integer). Unless it is zero length,
            #    in which case type does not matter.

            if density.dtype.kind != 'i' and len(density) > 0:
                raise TypeError("Density should be a array of *integers*.")
```

```
            # and the right values (positive or null)
            if any(density < 0):
                raise ValueError("Density should be an array of *positive* integers.")
            if density.ndim != 1:
                raise ValueError("Density should be an a *1-dimensional*" +
                                 "array of positive integers.")

            return sum(density * (density - 1))
```

```
Overwriting diffusion/model.py
```

```
In [7]: %%writefile diffusion/test_model.py
        """ Unit tests for a diffusion model """

        from pytest import raises
        from .model import energy

        def test_energy_fails_on_non_integer_density():
            with raises(TypeError) as exception:
                energy([1.0, 2, 3])

        def test_energy_fails_on_negative_density():
            with raises(ValueError) as exception: energy(
                    [-1, 2, 3])

        def test_energy_fails_ndimensional_density():
            with raises(ValueError) as exception: energy(
                    [[1, 2, 3], [3, 4, 5]])

        def test_zero_energy_cases():
            # Zero energy at zero density
            densities = [ [], [0], [0, 0, 0] ]
            for density in densities:
                assert energy(density) == 0

        def test_derivative():
            from numpy.random import randint

            # Loop over vectors of different sizes (but not empty)
            for vector_size in randint(1, 1000, size=30):

                # Create random density of size N
                density = randint(50, size=vector_size)

                # will do derivative at this index
                element_index = randint(vector_size)

                # modified densities
                density_plus_one = density.copy()
                density_plus_one[element_index] += 1

                # Compute and check result
                # d(n^2-1)/dn = 2n
                expected = (2.0 * density[element_index]
```

```
                            if density[element_index] > 0
                            else 0 )
                actual = energy(density_plus_one) - energy(density)
                assert expected == actual

        def test_derivative_no_self_energy():
            """ If particle is alone, then its participation to energy is zero """
            from numpy import array

            density = array([1, 0, 1, 10, 15, 0])
            density_plus_one = density.copy()
            density[1] += 1

            expected = 0
            actual = energy(density_plus_one) - energy(density)
            assert expected == actual
```

```
Overwriting diffusion/test_model.py
```

```
In [8]: %%bash
        cd diffusion
        pytest
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/diffusion
plugins: cov-2.8.1
collected 6 items

test_model.py …                                                          [100%]

============================== 6 passed in 0.16s ===============================
```

3.5.4 Coverage

With py.test, you can use the "pytest-cov" plugin to measure test coverage

```
In [9]: %%bash
        cd diffusion
        pytest --cov="diffusion"
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/diffusion
plugins: cov-2.8.1
collected 6 items

test_model.py …                                                          [100%]

----------- coverage: platform linux, python 3.7.5-final-0 -----------
Name            Stmts   Miss  Cover
---------------------------------
__init__.py         0      0   100%
```

```
model.py            10      0   100%
test_model.py       31      0   100%
-----------------------------------
TOTAL               41      0   100%


============================== 6 passed in 0.20s ===============================
```

Or an html report:

```
In [10]: %%bash
         cd diffusion
         pytest --cov="diffusion" --cov-report html
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/diffusion
plugins: cov-2.8.1
collected 6 items

test_model.py …                                                      [100%]

----------- coverage: platform linux, python 3.7.5-final-0 -----------
Coverage HTML written to dir htmlcov


============================== 6 passed in 0.21s ===============================
```

Look at the coverage results

```
In [ ]:
```

3.6 Mocking

3.6.1 Definition

Mock: *verb*,

1. to tease or laugh at in a scornful or contemptuous manner
2. to make a replica or imitation of something

Mocking

- Replace a real object with a pretend object, which records how it is called, and can assert if it is called wrong

3.6.2 Mocking frameworks

- C: CMocka
- C++: googletest
- Python: unittest.mock

3.6.3 Recording calls with mock

Mock objects record the calls made to them:

```
In [1]: from unittest.mock import Mock
        function = Mock(name="myroutine", return_value=2)

In [2]: function(1)

Out[2]: 2

In [3]: function(5, "hello", a=True)

Out[3]: 2

In [4]: function.mock_calls

Out[4]: [call(1), call(5, 'hello', a=True)]
```

The arguments of each call can be recovered

```
In [5]: name, args, kwargs = function.mock_calls[1]
        args, kwargs

Out[5]: ((5, 'hello'), {'a': True})
```

Mock objects can return different values for each call

```
In [6]: function = Mock(name="myroutine", side_effect=[2, "xyz"])

In [7]: function(1)

Out[7]: 2

In [8]: function(1, "hello", {'a': True})

Out[8]: 'xyz'
```

We expect an error if there are no return values left in the list:

```
In [9]: function()


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

        StopIteration                             Traceback (most recent call last)

        <ipython-input-9-30ca0b4348da> in <module>
    ----> 1 function()


        /opt/python/3.7.5/lib/python3.7/unittest/mock.py in __call__(_mock_self, *args, **kwargs)
       1009         # in the signature
       1010         _mock_self._mock_check_sig(*args, **kwargs)
    -> 1011         return _mock_self._mock_call(*args, **kwargs)
       1012
       1013
```

```
/opt/python/3.7.5/lib/python3.7/unittest/mock.py in _mock_call(_mock_self, *args, **kwargs)
   1071                 raise effect
   1072             elif not _callable(effect):
-> 1073                 result = next(effect)
   1074                 if _is_exception(result):
   1075                     raise result


StopIteration:
```

3.6.4 Using mocks to model test resources

Often we want to write tests for code which interacts with remote resources. (E.g. databases, the internet, or data files.)

We don't want to have our tests *actually* interact with the remote resource, as this would mean our tests failed due to lost internet connections, for example.

Instead, we can use mocks to assert that our code does the right thing in terms of the *messages it sends*: the parameters of the function calls it makes to the remote resource.

For example, consider the following code that downloads a map from the internet:

```
In [10]: import requests

         def map_at(lat, long, satellite=False, zoom=12,
                    size=(400, 400)):

             base = "https://static-maps.yandex.ru/1.x/?"

             params = dict(
                 z = zoom,
                 size = ",".join(map(str,size)),
                 ll = ",".join(map(str,(long,lat))),
                 lang = "en_US")

             if satellite:
                 params["l"] = "sat"
             else:
                 params["l"] = "map"

             return requests.get(base, params=params)

In [11]: london_map = map_at(51.5073509, -0.1277583)
         from IPython.display import Image

In [12]: %matplotlib inline
         Image(london_map.content)

Out[12]:
```

We would like to test that it is building the parameters correctly. We can do this by **mocking** the requests object. We need to temporarily replace a method in the library with a mock. We can use "patch" to do this:

```
In [13]: from unittest.mock import patch
         with patch.object(requests,'get') as mock_get:
             london_map = map_at(51.5073509, -0.1277583)
             print(mock_get.mock_calls)
```

```
[call('https://static-maps.yandex.ru/1.x/?', params={'z': 12, 'size': '400,400', 'll': '-0.1277583,51.5
```

Our tests then look like:

```
In [14]: def test_build_default_params():
             with patch.object(requests,'get') as mock_get:
                 default_map = map_at(51.0, 0.0)
                 mock_get.assert_called_with(
                 "https://static-maps.yandex.ru/1.x/?",
```

```
            params={
                'z':12,
                'size':'400,400',
                'll':'0.0,51.0',
                'lang':'en_US',
                'l': 'map'
            }
        )
    test_build_default_params()
```

That was quiet, so it passed. When I'm writing tests, I usually modify one of the expectations, to something 'wrong', just to check it's not passing "by accident", run the tests, then change it back!

3.6.5 Testing functions that call other functions

```
In [15]: def partial_derivative(function, at, direction, delta=1.0):
             f_x = function(at)
             x_plus_delta = at[:]
             x_plus_delta[direction] += delta
             f_x_plus_delta = function(x_plus_delta)
             return (f_x_plus_delta - f_x) / delta
```

We want to test that the above function does the right thing. It is supposed to compute the derivative of a function of a vector in a particular direction.

E.g.:

```
In [16]: partial_derivative(sum, [0,0,0], 1)
```

```
Out[16]: 1.0
```

How do we assert that it is doing the right thing? With tests like this:

```
In [17]: from unittest.mock import MagicMock

         def test_derivative_2d_y_direction():
             func = MagicMock()
             partial_derivative(func, [0,0], 1)
             func.assert_any_call([0, 1.0])
             func.assert_any_call([0, 0])


         test_derivative_2d_y_direction()
```

We made our mock a "Magic Mock" because otherwise, the mock results `f_x_plus_delta` and `f_x` can't be subtracted:

```
In [18]: MagicMock() - MagicMock()
```

```
Out[18]: <MagicMock name='mock.__sub__()' id='139833411952848'>
```

```
In [19]: Mock() - Mock()
```

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

TypeError                                 Traceback (most recent call last)
```

```
    <ipython-input-19-ef96ecbf0feb> in <module>
----> 1 Mock() - Mock()


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

3.7 Using a debugger

3.7.1 Stepping through the code

Debuggers are programs that can be used to test other programs. They allow programmers to suspend execution of the target program and inspect variables at that point.

- Mac - compiled languages: Xcode
- Windows - compiled languages: Visual Studio
- Linux: DDD
- all platforms: eclipse, gdb (DDD and eclipse are GUIs for gdb)
- python: spyder,
- `[pdb](https://docs.python.org/3.6/library/pdb.html)`
- R: RStudio, debug, browser

3.7.2 Using the python debugger

Unfortunately this doesn't work nicely in the notebook. But from the command line, you can run a python program with:

```
python -m pdb my_program.py
```

3.7.3 Basic navigation:

Basic command to navigate the code and the python debugger:

- `help`: prints the help
- `help n`: prints help about command `n`
- `n`(ext): executes one line of code. Executes and steps **over** functions.
- `s`(tep): step into current function in line of code
- `l`(ist): list program around current position
- `w`(where): prints current stack (where we are in code)
- `[enter]`: repeats last command
- `anypythonvariable`: print the value of that variable

The python debugger is **a python shell**: it can print and compute values, and even change the values of the variables at that point in the program.

3.7.4 Breakpoints

Break points tell debugger where and when to stop We say * `b somefunctionname`

```
In [1]: %%writefile solutions/diffusionmodel/energy_example.py
        from diffusion_model import energy
        print(energy([5, 6, 7, 8, 0, 1]))
```

```
Writing solutions/diffusionmodel/energy_example.py
```

The debugger is, of course, most used interactively, but here I'm showing a prewritten debugger script:

```
In [2]: %%writefile commands
        restart  # restart session
        n
        b energy # program will stop when entering energy
        c        # continue program until break point is reached
        print(density) # We are now "inside" the energy function and can print any variable.
```

```
Writing commands
```

```
In [3]: %%bash
        python -m pdb solutions/diffusionmodel/energy_example.py < commands
```

```
> /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/solutions/diffusionmodel/energy_example.py(1)<m
-> from diffusion_model import energy
(Pdb) Restarting solutions/diffusionmodel/energy_example.py with arguments:
        solutions/diffusionmodel/energy_example.py
> /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/solutions/diffusionmodel/energy_example.py(1)<m
-> from diffusion_model import energy
(Pdb) > /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/solutions/diffusionmodel/energy_example.p
-> print(energy([5, 6, 7, 8, 0, 1]))
(Pdb) Breakpoint 1 at /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/solutions/diffusionmodel/di
(Pdb) > /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/solutions/diffusionmodel/diffusion_model
-> from numpy import array, any, sum
(Pdb) [5, 6, 7, 8, 0, 1]
(Pdb)
```

Alternatively, break-points can be set on files: `b file.py:20` will stop on line 20 of `file.py`.

3.7.5 Post-mortem

Debugging when something goes wrong:

1. Have a crash somewhere in the code
2. run `python -m pdb file.py` or run the cell with `%pdb on`

The program should stop where the exception was raised

1. use `w` and `l` for position in code and in call stack
2. use `up` and `down` to navigate up and down the call stack
3. inspect variables along the way to understand failure

This **does** work in the notebook.

```
%pdb on
from diffusion.model import energy
partial_derivative(energy,[5,6,7,8,0,1],5)
```

3.8 Continuous Integration

3.8.1 Test servers

Goal:

1. run tests nightly
2. run tests after each commit to github (or other)
3. run tests on different platforms

Various groups run servers that can be used to do this automatically.
RITS run a university-wide one.

3.8.2 Memory and profiling

For compiled languages (C, C++, Fortran): * Checking for memory leaks with valgrind: `valgrind --leak-check=full program` * Checking cache hits and cache misses with cachegrind: `valgrind --tool=cachegrind program` * Profiling the code with callgrind: `valgrind --tool=callgrind program`

- Python: profile with the standard library or runsnake
- R: Rprof

3.9 Recap example: Monte-Carlo

3.9.1 Problem: Implement and test a simple Monte-Carlo algorithm

Given an input function (energy) and starting point (density) and a temperature T:

1. Compute energy at current density.
2. Move randomly chosen agent randomly left or right.
3. Compute second energy.
4. Compare the two energies:
5. If second energy is lower, accept move.
6. β is a parameter which determines how likely the simulation is to move from a 'less favourable' situation to a 'more favourable' one.
7. Compute $P_0 = e^{-\beta(E_1 - E_0)}$ and P_1 a random number between 0 and 1,
8. If $P_0 > P_1$, do the move anyway.
9. Repeat.

- the algorithm should work for (m)any energy function(s).
- there should be separate tests for separate steps! What constitutes a step?
- tests for the Monte-Carlo should not depend on other parts of code.
- Use matplotlib to plot density at each iteration, and make an animation

3.9.2 Solution

We need to break our problem down into pieces:

1. A function to generate a random change: `random_agent()`, `random_direction()`
2. A function to compute the energy before the change and after it: `energy()`
3. A function to determine the probability of a change given the energy difference (1 if decreases, otherwise based on exponential): `change_density()`
4. A function to determine whether to execute a change or not by drawing a random number`accept_change()`
5. A method to iterate the above procedure: `step()`

Next Step: Think about the possible unit tests

1. Input insanity: e.g. density should non-negative integer; testing by giving negative values etc.
2. `change_density()`: density is change by a particle hopping left or right? Do all positions have an equal chance of moving?
3. `accept_change()` will move be accepted when second energy is lower?
4. Make a small test case for the main algorithm. (Hint: by using mocking, we can pre-set who to move where.)

In [1]:
```
%%bash
mkdir -p DiffusionExample
```

In [2]:
```
%%writefile DiffusionExample/MonteCarlo.py
import matplotlib.pyplot as plt
from numpy import sum, array
from numpy.random import randint, choice



class MonteCarlo(object):
    """ A simple Monte Carlo implementation """

    def __init__(self, energy, density, temperature=1, itermax=1000):
        from numpy import any, array
        density = array(density)
        self.itermax = itermax

        if temperature == 0:
            raise NotImplementedError(
                "Zero temperature not implemented")
        if temperature < 0e0:
            raise ValueError(
                "Negative temperature makes no sense")

        if len(density) < 2:
            raise ValueError("Density is too short")
        # of the right kind (integer). Unless it is zero length,
        # in which case type does not matter.
        if density.dtype.kind != 'i' and len(density) > 0:
            raise TypeError("Density should be an array of *integers*.")
        # and the right values (positive or null)
        if any(density < 0):
            raise ValueError("Density should be an array of" +
                             "*positive* integers.")
        if density.ndim != 1:
            raise ValueError("Density should be an a *1-dimensional*" +
                             "array of positive integers.")
        if sum(density) == 0:
            raise ValueError("Density is empty.")

        self.current_energy = energy(density)
        self.temperature = temperature
        self.density = density

    def random_direction(self): return choice([-1, 1])
```

```
def random_agent(self, density):
    # Particle index
    particle = randint(sum(density))
    current = 0
    for location, n in enumerate(density):
        current += n
        if current > particle:
            break
    return location

def change_density(self, density):
    """ Move one particle left or right. """

    location = self.random_agent(density)

    # Move direction
    if(density[location]-1 < 0):
        return array(density)
    if location == 0:
        direction = 1
    elif location == len(density) - 1:
        direction = -1
    else:
        direction = self.random_direction()

    # Now make change
    result = array(density)
    result[location] -= 1
    result[location + direction] += 1
    return result

def accept_change(self, prior, successor):
    """ Returns true if should accept change. """
    from numpy import exp
    from numpy.random import uniform
    if successor <= prior:
        return True
    else:
        return exp(-(successor - prior) / self.temperature) > uniform()

def step(self):
    iteration = 0
    while iteration < self.itermax:
        new_density = self.change_density(self.density)
        new_energy = energy(new_density)

        accept = self.accept_change(self.current_energy, new_energy)
        if accept:
            self.density, self.current_energy = new_density, new_energy
        iteration += 1

    return self.current_energy, self.density
```

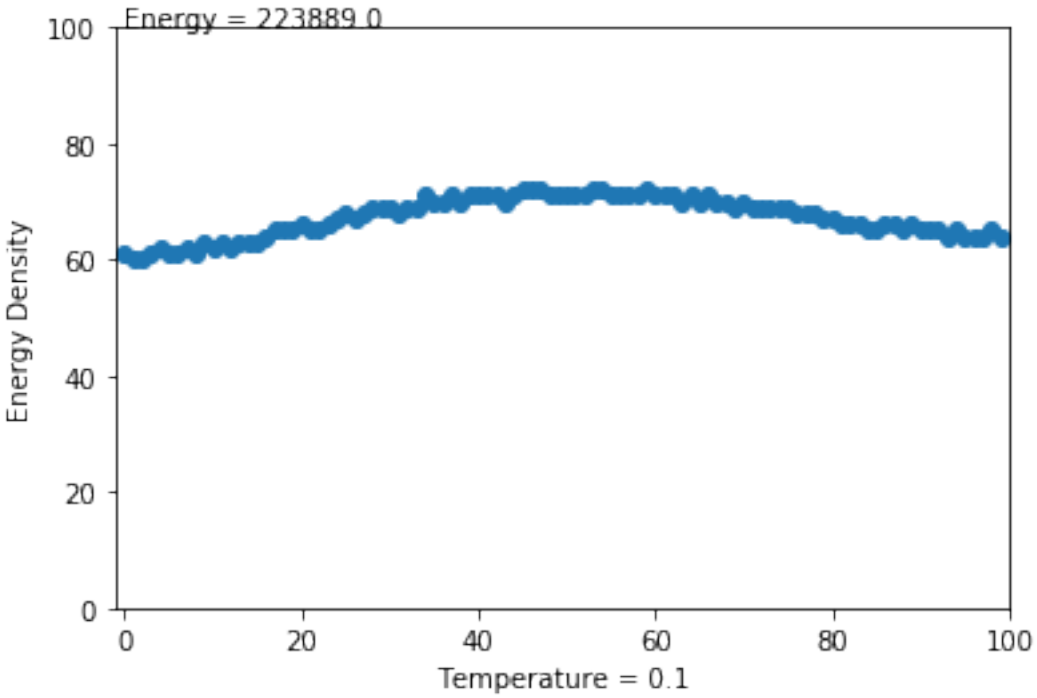
```
        def energy(density, coefficient=1):
            """ Energy associated with the diffusion model
                :Parameters:
                density: array of positive integers
                Number of particles at each position i in the array/geometry
            """
            from numpy import array, any, sum

            # Make sure input is an array
            density = array(density)

            # of the right kind (integer). Unless it is zero length, in which case type does not matter
            if density.dtype.kind != 'i' and len(density) > 0:
                raise TypeError("Density should be an array of *integers*.")
            # and the right values (positive or null)
            if any(density < 0):
                raise ValueError("Density should be an array" +
                                 "of *positive* integers.")
            if density.ndim != 1:
                raise ValueError("Density should be an a *1-dimensional*" +
                                 "array of positive integers.")

            return coefficient * 0.5 * sum(density * (density - 1))
```

```
Writing DiffusionExample/MonteCarlo.py
```

In [3]:
```
        import sys
        sys.path.append('DiffusionExample')
        from MonteCarlo import MonteCarlo, energy
        import numpy as np
        import numpy.random as random
        from matplotlib import animation
        from matplotlib import pyplot as plt
        from IPython.display import HTML


        Temperature = 0.1
        density = [np.sin(i) for i in np.linspace(0.1, 3, 100)]
        density = np.array(density)*100
        density = density.astype(int)

        fig = plt.figure()
        ax = plt.axes(xlim=(-1, len(density)), ylim=(0, np.max(density)+1))
        image = ax.scatter(range(len(density)), density)

        txt_energy = plt.text(0, 100, 'Energy = 0')
        plt.xlabel('Temperature = 0.1')
        plt.ylabel('Energy Density')


        mc = MonteCarlo(energy, density, temperature=Temperature)
```

```
def simulate(step):
    energy, density = mc.step()
    image.set_offsets(np.vstack((range(len(density)), density)).T)
    txt_energy.set_text('Energy = {}'.format(energy))


anim = animation.FuncAnimation(fig, simulate, frames=200,
                               interval=50)
HTML(anim.to_jshtml())
```

Out[3]: <IPython.core.display.HTML object>



In [4]:
```
%%writefile DiffusionExample/test_model.py
from MonteCarlo import MonteCarlo
from unittest.mock import MagicMock
from pytest import raises, approx


def test_input_sanity():
    """ Check incorrect input do fail """
    energy = MagicMock()

    with raises(NotImplementedError) as exception:
        MonteCarlo(sum, [1, 1, 1], 0e0)
    with raises(ValueError) as exception:
        MonteCarlo(energy, [1, 1, 1], temperature=-1e0)

    with raises(TypeError) as exception:
```

```
        MonteCarlo(energy, [1.0, 2, 3])
    with raises(ValueError) as exception:
        MonteCarlo(energy, [-1, 2, 3])
    with raises(ValueError) as exception:
        MonteCarlo(energy, [[1, 2, 3], [3, 4, 5]])
    with raises(ValueError) as exception:
        MonteCarlo(energy, [3])
    with raises(ValueError) as exception:
        MonteCarlo(energy, [0, 0])


def test_move_particle_one_over():
    """ Check density is change by a particle hopping left or right. """
    from numpy import nonzero, multiply
    from numpy.random import randint

    energy = MagicMock()

    for i in range(100):
        # Do this n times, to avoid
        # issues with random numbers
        # Create density

        density = randint(50, size=randint(2, 6))
        mc = MonteCarlo(energy, density)
        # Change it
        new_density = mc.change_density(density)

        # Make sure any movement is by one
        indices = nonzero(density - new_density)[0]
        assert len(indices) == 2, "densities differ in two places"
        assert \
            multiply.reduce((density - new_density)[indices]) == -1, \
            "densities differ by + and - 1"


def test_equal_probability():
    """ Check particles have equal probability of movement. """
    from numpy import array, sqrt, count_nonzero

    energy = MagicMock()

    density = array([1, 0, 99])
    mc = MonteCarlo(energy, density)
    changes_at_zero = [
        (density - mc.change_density(density))[0] != 0 for i in range(10000)]
    assert count_nonzero(changes_at_zero) \
            == approx(0.01 * len(changes_at_zero), 0.5 * sqrt(len(changes_at_zero)))


def test_accept_change():
    """ Check that move is accepted if second energy is lower """
    from numpy import sqrt, count_nonzero, exp
```

```
            energy = MagicMock
            mc = MonteCarlo(energy, [1, 1, 1], temperature=100.0)
            # Should always be true.
            # But do more than one draw,
            # in case randomness incorrectly crept into
            # implementation
            for i in range(10):
                assert mc.accept_change(0.5, 0.4)
                assert mc.accept_change(0.5, 0.5)

            # This should be accepted only part of the time,
            # depending on exponential distribution
            prior, successor = 0.4, 0.5
            accepted = [mc.accept_change(prior, successor) for i in range(10000)]
            assert count_nonzero(accepted) / float(len(accepted)) \
                == approx(exp(-(successor - prior) / mc.temperature), 3e0 / sqrt(len(accepted)))



        def test_main_algorithm():
            import numpy as np
            from numpy import testing
            from unittest.mock import Mock

            density=[1, 1, 1, 1, 1]
            energy=MagicMock()
            mc=MonteCarlo(energy, density, itermax = 5)

            acceptance=[True, True, True, True, True]
            mc.accept_change=Mock(side_effect = acceptance)
            mc.random_agent=Mock(side_effect = [0, 1, 2, 3, 4])
            mc.random_direction=Mock(side_effect = [1, 1, 1, 1, -1])
            np.testing.assert_equal(mc.step()[1], [0, 1, 1, 2, 1])
```

```
Writing DiffusionExample/test_model.py
```

```
In [5]: %%bash
        cd DiffusionExample
        py.test
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch03tests/DiffusionExample
plugins: cov-2.8.1
collected 5 items

test_model.py …                                                          [100%]

============================== warnings summary ================================
/home/travis/virtualenv/python3.7.5/lib/python3.7/distutils/__init__.py:4
  /home/travis/virtualenv/python3.7.5/lib/python3.7/distutils/__init__.py:4: DeprecationWarning: the imp
    import imp

-- Docs: https://docs.pytest.org/en/latest/warnings.html
```

```
======================== 5 passed, 1 warning in 0.86s ========================
```

Chapter 4

Packaging your code

4.1 Installing Libraries

We've seen that there are lots of python libraries. But how do we install them?

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**.

4.1.1 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 -> Anaconda3 -> Anaconda Prompt
- On mac, start *Terminal*.
- On linux, open a bash shell.

Into this shell, type:

```
pip install geopy
```

The computer will install the package and its dependencies automatically from PyPI (a repository of packages, which we'll talk about later).

Now, close the Jupyter notebook if you have it open, and reopen it. Check your new library is installed with:

```
In [1]: import geopy
        geocoder = geopy.geocoders.Yandex(lang="en_US")

In [2]: geocoder.geocode('Cambridge', exactly_one=False)
```

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

HTTPError                                 Traceback (most recent call last)

~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(
```

```
    354         try:
--> 355             page = requester(req, timeout=timeout, **kwargs)
    356         except Exception as error:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in open(self, fullurl, data, timeout)
    530             meth = getattr(processor, meth_name)
--> 531             response = meth(req, response)
    532


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_response(self, request, response)
    640             response = self.parent.error(
--> 641                 'http', request, response, code, msg, hdrs)
    642


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in error(self, proto, *args)
    568             args = (dict, 'default', 'http_error_default') + orig_args
--> 569             return self._call_chain(*args)
    570


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in _call_chain(self, chain, kind, meth_name, *
    502             func = getattr(handler, meth_name)
--> 503             result = func(*args)
    504             if result is not None:


    /opt/python/3.7.5/lib/python3.7/urllib/request.py in http_error_default(self, req, fp, code, msg
    648     def http_error_default(self, req, fp, code, msg, hdrs):
--> 649         raise HTTPError(req.full_url, code, msg, hdrs, fp)
    650


    HTTPError: HTTP Error 403: Forbidden


During handling of the above exception, another exception occurred:


    GeocoderInsufficientPrivileges            Traceback (most recent call last)

    <ipython-input-2-ca3d5ea40875> in <module>
----> 1 geocoder.geocode('Cambridge', exactly_one=False)


    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/yandex.py in geocode(self,
    114         logger.debug("%s.geocode: %s", self.__class__.__name__, url)
    115         return self._parse_json(
--> 116             self._call_geocoder(url, timeout=timeout),
    117             exactly_one,
    118         )
```

```
    ~/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy/geocoders/base.py in _call_geocoder(s
    371                                     exc_info=False)
    372                 try:
--> 373                     raise ERROR_CODE_MAP[code](message)
    374                 except KeyError:
    375                     raise GeocoderServiceError(message)


    GeocoderInsufficientPrivileges: HTTP Error 403: Forbidden
```

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:

- `pip install --user geopy`

If you get a complaint like: 'pip is not recognized as an internal or external command', try the following:

- `conda install pip` (if you are using Anaconda - though it should be already available)
- or follow the official instructions otherwise.

4.1.2 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 `conda search`. If it is there you can `conda install` it, you'll likely have less problems. But Anaconda doesn't package everything, so you'll need to `pip install` from time to time.

The maintainers of packages may have also provided releases of their software via conda-forge, a community-driven project that provides a collection of packages for the anaconda environment. In such case you can add conda-forge to your anaconda installation and use `search` and `install` as explained above.

4.1.3 Where do these libraries go?

```
In [3]: geopy.__path__
```

```
Out[3]: ['/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/geopy']
```

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` (`%PYTHONPATH%` in Windows). It will try each of these places in turn, until it finds a matching library name.

```
In [4]: import sys
        sys.path

Out[4]: ['/home/travis/build/UCL/rsd-engineeringcourse/ch04packaging',
         '/home/travis/virtualenv/python3.7.5/lib/python37.zip',
         '/home/travis/virtualenv/python3.7.5/lib/python3.7',
         '/home/travis/virtualenv/python3.7.5/lib/python3.7/lib-dynload',
         '/opt/python/3.7.5/lib/python3.7',
         '',
         '/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages',
         '/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/extensions',
         '/home/travis/.ipython']
```

You can add (`append`) more paths to this list, and so allow libraries to be load from there. Thought this is not a recommended practice, let's do it once to understand how the import works.

1. Create a new directory (*e.g.*, `myexemplar`),
2. create a file inside that directory (`exemplar.py`),
3. write a function inside such file (`exemplar_works`),
4. open python, import `sys` and add the path of `myexemplar` to `sys.path`,
5. import your new file, and
6. run the function.

4.1.4 Libraries not in PyPI

Sometimes 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:

- `pip install --user .`

This is all pretty awkward, but it is worth practising this stuff, as most of the power of using programming for research resides in all the libraries that are out there.

4.1.5 Python virtual environments

Sometimes you need to have different versions of a package installed, or you would like to install a set of libraries that you don't want to affect the rest of the installation in your system. In such cases you can create environments that are isolated from the rest.

There are multiple solutions to this, only for python or for anaconda. Find more information on how to create and use the virtual enviroments.

4.2 Libraries

4.2.1 Libraries are awesome

The strength of a language lies as much in the set of libraries available, as it does in the language itself.

A great set of libraries allows for a very powerful programming style:

- Write minimal code yourself
- Choose the right libraries
- Plug them together
- Create impressive results

Not only is this efficient with your programming time, it's also more efficient with computer time.

The chances are any algorithm you might want to use has already been programmed better by someone else.

4.2.2 Drawbacks of libraries.

- Sometimes, libraries are not looked after by their creator: code that is not maintained *rots*:
  - It no longer works with later versions of *upstream* libraries.
  - It doesn't work on newer platforms or systems.
  - Features that are needed now, because the field has moved on, are not added
- Sometimes, libraries are hard to get working:
  - For libraries in pure python, this is almost never a problem
  - But many libraries involve *compiled components*: these can be hard to install.

4.2.3 Contribute, don't duplicate

- You have a duty to the ecosystem of scholarly software:
  - If there's a tool or algorithm you need, find a project which provides it.
  - If there are features missing, or problems with it, fix them, don't create your own library.

4.2.4 How to choose a library

- Is the code on an open version control tool like GitHub?
  - When was the last commit?
  - How often are there commits?
- Can you find the lead contributor on the internet?
- Do they respond when approached:
  - emails to developer list
  - personal emails
  - tweets
  - irc/gitter/slack/[matrix]
  - issues raised on GitHub
- Are there contributors other than the lead contributor?
- Is there discussion of the library on Stack Exchange?
- Is it on standard package repositories? (PyPI, apt/yum/brew)
- Are there any tests?
- Download it. Can you build it? Do the tests pass?
- Is there an open test dashboard? (Travis/Jenkins/CDash)
- What dependencies does the library itself have? Do they pass this list?
- Are different versions of the library clearly labeled with version numbers?
- Is there a changelog?

4.2.5 Sensible Version Numbering

The best approach to version numbers clearly distinguishes kinds of change:

Given a version number MAJOR.MINOR.PATCH, e.g. 2.11.14 increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

This is called Semantic Versioning.

4.2.6 The Python Standard Library

Python comes with a powerful standard library.

Learning python is as much about learning this library as learning the language itself.

You've already seen a few packages in this library: `math`, `pdb`, `datetime`.

4.2.7 The Python Package Index

Python's real power, however, comes with the Python Package Index: PyPI. This is a huge array of libraries, with all kinds of capabilities, all easily installable from the command line or through your Python distribution.

4.3 Python not in the Notebook

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.

4.3.1 Writing Python in Text Files

You can use a text editor like VS Code or Spyder. 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, though this is not encouraged!

Libraries are usually structured with multiple files, one for each class.

We will be turning the code we have written for the maze into a library, so that other code can reuse it.

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 # Python 3 relative import
```

```
Writing mazetool/__init__.py
```

In this case we are making it easier to import `Maze` as we are making it available one level above.

4.3.2 Loading Our Package

We just wrote the files, there is no "Maze" class in this notebook yet:

```
In [7]: myhouse = Maze('My New House')
```

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-7-3bc371b39bcd> in <module>
    ----> 1 myhouse = Maze('My New House')


        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]: import mazetool
```

Let's see how we can access the files we created:

```
In [9]: mazetool.exit.Exit
```

```
Out[9]: mazetool.exit.Exit
```

```
In [10]: from mazetool import Maze
```

```
In [11]: 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 [12]: import os
```

```
In [13]: os.listdir(os.path.join(os.getcwd(), 'mazetool') )
```

```
Out[13]: ['room.py', 'person.py', '__pycache__', '__init__.py', 'maze.py', 'exit.py']
```

You may get also `.pyc` files. Those 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. They may appear inside the `__pycache__` directory.

4.3.3 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.

The best way to do this is to learn how to make our code into a proper module that we can install. We'll see more on that in a few lectures' time (notebook).

Alternatively, we can add a folder to the "`PYTHONPATH`", where python searches for modules:

```
In [14]: import sys
         print('\n'.join(sys.path[-3:]))
```

```
/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages
/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/IPython/extensions
/home/travis/.ipython
```

```
In [15]: from pathlib import Path
         sys.path.append(os.path.join(Path.home(), 'devel', 'libraries', 'python'))

In [16]: print(sys.path[-1])
```

```
/home/travis/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.

4.4 Argparse

This is the standard library for building programs with a command-line interface. Here we show a short introduction to it, but we recommend to read the official tutorial.

```
In [1]: %%writefile greeter.py
        #!/usr/bin/env python
        from argparse import ArgumentParser

        if __name__ == "__main__":
            parser = ArgumentParser(description="Generate appropriate greetings")
            parser.add_argument('--title', '-t')
            parser.add_argument('--polite','-p', action="store_true")
            parser.add_argument('personal')
            parser.add_argument('family')
            arguments= parser.parse_args()

            greeting = "How do you do, " if arguments.polite else "Hey, "
            if arguments.title:
                greeting += f"{arguments.title} "
            greeting += f"{arguments.personal} {arguments.family}."
            print(greeting)
```

```
Writing greeter.py
```

If you are using MacOS or Linux, you do the following to create an executable:

```
In [2]: %%bash
        chmod u+x greeter.py
```

and then running it as:

```
In [3]: %%bash
        ./greeter.py --help
```

```
usage: greeter.py [-h] [--title TITLE] [--polite] personal family

Generate appropriate greetings
```

```
positional arguments:
  personal
  family

optional arguments:
  -h, --help            show this help message and exit
  --title TITLE, -t TITLE
  --polite, -p
```

if you are using Windows, change `bash` by `cmd`, and prepend the commands by `python`

```
%%cmd
python greeter.py John Cleese
```

```
In [4]: %%bash
        ./greeter.py John Cleese
```

```
Hey, John Cleese.
```

```
In [5]: %%bash
        ./greeter.py --polite John Cleese
```

```
How do you do, John Cleese.
```

```
In [6]: %%bash
        ./greeter.py John Cleese --title Dr
```

```
Hey, Dr John Cleese.
```

Yes, he is!

4.5 Packaging

Once we've made a working program, we'd like to be able to share it with others.

A good cross-platform build tool is the most important thing: you can always have collaborators build from source.

4.5.1 Distribution tools

Distribution tools allow one to obtain a working copy of someone else's package.

- Language-specific tools:
- python: PyPI,
- ruby: Ruby Gems,
- perl: CPAN,
- R: CRAN
- Platform specific packagers e.g.:
- `brew` for MacOS,
- `apt`/`yum` for Linux or
- `choco` for Windows.

4.5.2 Laying out a project

When planning to package a project for distribution, defining a suitable project layout is essential.

```
In [1]: %%bash
        tree --charset ascii greetings -I "doc|build|Greetings.egg-info|dist|*.pyc"
```

```
greetings
|-- CITATION.md
|-- conf.py
|-- greetings
|   |-- command.py
|   |-- greeter.py
|   |-- __init__.py
|   `-- test
|       |-- fixtures
|       |   `-- samples.yaml
|       |-- __init__.py
|       `-- test_greeter.py
|-- index.rst
|-- LICENSE.md
|-- README.md
`-- setup.py

3 directories, 12 files
```

We can start by making our directory structure. You can create many nested directories at once using the `-p` switch on `mkdir`.

```
In [2]: %%bash
        mkdir -p greetings/greetings/test/fixtures
        mkdir -p greetings/scripts
```

4.5.3 Using setuptools

To make python code into a package, we have to write a `setup.py` file:

```
from setuptools import setup, find_packages

setup(
    name="Greetings",
    version="0.1.0",
    packages=find_packages(exclude=['*test']),
)
```

We can now install this code with

```
pip install .
```

And the package will be then available to use everywhere on the system.

```
In [3]: from greetings.greeter import greet
        greet("Terry","Gilliam")
```

```
Out[3]: 'Hey, Terry Gilliam.'
```

4.5.4 Convert the script to a module

Of course, there's more to do when taking code from a quick script and turning it into a proper module:
We need to add docstrings to our functions, so people can know how to use them.

```
In [4]: from IPython.display import Code
        Code("greetings/greetings/greeter.py")
```

Out[4]:

```
def greet(personal, family, title="", polite=False):
    """ Generate a greeting string for a person.

    Parameters
    ----------
    personal: str
        A given name, such as Will or Jean-Luc
    family: str
        A family name, such as Riker or Picard
    title: str
        An optional title, such as Captain or Reverend
    polite: bool
        True for a formal greeting, False for informal.

    Returns
    -------
    string
        An appropriate greeting

    Examples
    --------
    >>> from greetings.greeter import greet
    >>> greet("Terry", "Jones")
    'Hey, Terry Jones.
    """

    greeting = "How do you do, " if polite else "Hey, "
    if title:
        greeting += f"{title} "

    greeting += f"{personal} {family}."
    return greeting
```

```
In [5]: import greetings
        help(greetings.greeter.greet)
```

```
Help on function greet in module greetings.greeter:

greet(personal, family, title='', polite=False)
    Generate a greeting string for a person.

    Parameters
    ----------
    personal: str
        A given name, such as Will or Jean-Luc
    family: str
```

```
        A family name, such as Riker or Picard
    title: str
        An optional title, such as Captain or Reverend
    polite: bool
        True for a formal greeting, False for informal.

    Returns
    -------
    string
        An appropriate greeting

    Examples
    --------
    >>> from greetings.greeter import greet
    >>> greet("Terry", "Jones")
    'Hey, Terry Jones.
```

The documentation string explains how to use the function; don't worry about this for now, we'll consider this on the next section (notebook version).

4.5.5 Write an executable script

In [6]: `Code("greetings/greetings/command.py")`

Out[6]:

```
from argparse import ArgumentParser
from .greeter import greet # Note python 3 relative import

def process():
   parser = ArgumentParser(description="Generate appropriate greetings")

   parser.add_argument('--title', '-t')
   parser.add_argument('--polite', '-p', action="store_true")
   parser.add_argument('personal')
   parser.add_argument('family')

   arguments = parser.parse_args()

   print(greet(arguments.personal, arguments.family,
               arguments.title, arguments.polite))

if __name__ == "__main__":
    process()
```

4.5.6 Specify dependencies

We use the setup.py file to specify the packages we depend on:

```
setup(
    name="Greetings",
    version="0.1.0",
    packages=find_packages(exclude=['*test']),
    install_requires=['numpy', 'pyyaml'] # NOTE: this is an example to ilustrate how to add dependencie
)                                        #       Greetings doesn't have any external dependency.
```

4.5.7 Specify entry point

This allows us to create a command to execute part of our library. In this case when we execute `greet` on the terminal, we will be calling the `process` function under `greetings/command.py`.

```
In [7]: Code("greetings/setup.py")
```

```
Out[7]:
```

```
from setuptools import setup, find_packages

setup(
    name="Greetings",
    version="0.1.0",
    packages=find_packages(exclude=['*test']),
    entry_points={
        'console_scripts': [
            'greet = greetings.command:process'
        ]})
```

And the scripts are now available as command line commands:

```
In [8]: %%bash
        greet --help
```

```
usage: greet [-h] [--title TITLE] [--polite] personal family

Generate appropriate greetings

positional arguments:
  personal
  family

optional arguments:
  -h, --help            show this help message and exit
  --title TITLE, -t TITLE
  --polite, -p
```

```
In [9]: %%bash
        greet Terry Gilliam
        greet --polite Terry Gilliam
        greet Terry Gilliam --title Cartoonist
```

```
Hey, Terry Gilliam.
How do you do, Terry Gilliam.
Hey, Cartoonist Terry Gilliam.
```

4.5.8 Installing from GitHub

We could now submit "greeter" to PyPI for approval, so everyone could `pip install` it.

However, when using git, we don't even need to do that: we can install directly from any git URL:

```
pip install git+git://github.com/ucl-rits/greeter
```

```
In [10]: %%bash
         greet Lancelot the-Brave --title Sir
```

```
Hey, Sir Lancelot the-Brave.
```

4.5.9 Write a readme file

e.g.:

```
In [11]: Code("greetings/README.md")
```

```
Out[11]:
```

```
Greetings!
==========

This is a very simple example package used as part of the UCL
[Research Software Engineering with Python](development.rc.ucl.ac.uk/training/engineering) course.

Usage:

Invoke the tool with `greet <FirstName> <Secondname>`
```

4.5.10 Write a license file

e.g.:

```
In [12]: Code("greetings/LICENSE.md")
```

```
Out[12]:
```

```
(C) University College London 2014

This "greetings" example package is granted into the public domain.
```

4.5.11 Write a citation file

e.g.:

```
In [13]: Code("greetings/CITATION.md")
```

```
Out[13]:
```

```
If you wish to refer to this course, please cite the URL
http://github-pages.ucl.ac.uk/rsd-engineeringcourse/

Portions of the material are taken from [Software Carpentry](http://software-carpentry.org/)
```

You may well want to formalise this using the codemeta.json standard or the citation file format - these don't have wide adoption yet, but we recommend it.

4.5.12 Define packages and executables

```
In [14]: %%bash
         touch greetings/greetings/test/__init__.py
         touch greetings/greetings/__init__.py
```

4.5.13 Write some unit tests

Separating the script from the logical module made this possible:

```
In [15]: Code("greetings/greetings/test/test_greeter.py")
```

```
Out[15]:
```

```
import yaml
import os
from ..greeter import greet

def test_greeter():
    with open(os.path.join(os.path.dirname(__file__),
                           'fixtures',
                           'samples.yaml')) as fixtures_file:
        fixtures = yaml.safe_load(fixtures_file)
        for fixture in fixtures:
            answer = fixture.pop('answer')
            assert greet(**fixture) == answer
```

Add a fixtures file:

```
In [16]: Code("greetings/greetings/test/fixtures/samples.yaml")
```

```
Out[16]:
```

```
- personal: Eric
  family: Idle
  answer: "Hey, Eric Idle."
- personal: Graham
  family: Chapman
  polite: True
  answer: "How do you do, Graahm Chapman."
- personal: Michael
  family: Palin
  title: CBE
  answer: "Hey, CBE Mike Palin."
```

```
In [17]: %%bash --no-raise-error
         pytest
```

```
============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch04packaging
plugins: cov-2.8.1
collected 1 item

greetings/greetings/test/test_greeter.py F                               [100%]

=================================== FAILURES ===================================
_______________________________ test_greeter ________________________________

    def test_greeter():
        with open(os.path.join(os.path.dirname(__file__),
                               'fixtures',
```

```
                            'samples.yaml')) as fixtures_file:
            fixtures = yaml.safe_load(fixtures_file)
            for fixture in fixtures:
                answer = fixture.pop('answer')
>               assert greet(**fixture) == answer
E               AssertionError: assert 'How do you d…aham Chapman.' == 'How do you d…aahm Chapman.'
E                 - How do you do, Graham Chapman.
E                 ?                    -
E                 + How do you do, Graahm Chapman.
E                 ?                   +

greetings/greetings/test/test_greeter.py:12: AssertionError
============================== 1 failed in 0.06s ===============================
```

However, this hasn't told us that also the third test is wrong! A better aproach is to parametrize the test as follows:

```
In [18]: %%writefile greetings/greetings/test/test_greeter.py
         import yaml
         import os
         import pytest
         from ..greeter import greet

         def read_fixture():
             with open(os.path.join(os.path.dirname(__file__),
                                    'fixtures',
                                    'samples.yaml')) as fixtures_file:
                 fixtures = yaml.safe_load(fixtures_file)
             return fixtures

         @pytest.mark.parametrize("fixture", read_fixture())
         def test_greeter(fixture):
             answer = fixture.pop('answer')
             assert greet(**fixture) == answer

Overwriting greetings/greetings/test/test_greeter.py
```

Now when we run `pytest`, we get a failure per element in our fixture and we know all that fails.

```
In [19]: %%bash --no-raise-error
         pytest

============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch04packaging
plugins: cov-2.8.1
collected 3 items

greetings/greetings/test/test_greeter.py .FF                             [100%]

=================================== FAILURES ===================================
_____________________________ test_greeter[fixture1] ______________________________
```

```
fixture = {'family': 'Chapman', 'personal': 'Graham', 'polite': True}

    @pytest.mark.parametrize("fixture", read_fixture())
    def test_greeter(fixture):
        answer = fixture.pop('answer')
>       assert greet(**fixture) == answer
E       AssertionError: assert 'How do you d…aham Chapman.' == 'How do you d…aahm Chapman.'
E         - How do you do, Graham Chapman.
E         ?                    -
E         + How do you do, Graahm Chapman.
E         ?                   +

greetings/greetings/test/test_greeter.py:16: AssertionError
_____________________________ test_greeter[fixture2] _____________________________

fixture = {'family': 'Palin', 'personal': 'Michael', 'title': 'CBE'}

    @pytest.mark.parametrize("fixture", read_fixture())
    def test_greeter(fixture):
        answer = fixture.pop('answer')
>       assert greet(**fixture) == answer
E       AssertionError: assert 'Hey, CBE Michael Palin.' == 'Hey, CBE Mike Palin.'
E         - Hey, CBE Michael Palin.
E         ?            ^^^ -
E         + Hey, CBE Mike Palin.
E         ?            ^

greetings/greetings/test/test_greeter.py:16: AssertionError
======================== 2 failed, 1 passed in 0.07s ========================
```

We can also make pytest to check whether the docstrings are correct by adding the `--doctest-modules` flag:

```
In [20]: %%bash --no-raise-error
         pytest --doctest-modules

============================= test session starts ==============================
platform linux -- Python 3.7.5, pytest-5.3.2, py-1.8.0, pluggy-0.13.0
rootdir: /home/travis/build/UCL/rsd-engineeringcourse/ch04packaging
plugins: cov-2.8.1
collected 4 items

greetings/greetings/greeter.py F                                         [ 25%]
greetings/greetings/test/test_greeter.py .FF                             [100%]

=================================== FAILURES ===================================
______________________ [doctest] greetings.greeter.greet _______________________
014
015     Returns
016     -------
017     string
018         An appropriate greeting
019
020     Examples
```

```
021     --------
022     >>> from greetings.greeter import greet
023     >>> greet("Terry", "Jones")
Expected:
    'Hey, Terry Jones.
Got:
    'Hey, Terry Jones.'

/home/travis/build/UCL/rsd-engineeringcourse/ch04packaging/greetings/greetings/greeter.py:23: DocTestFa
_____________________________ test_greeter[fixture1] _____________________________

fixture = {'family': 'Chapman', 'personal': 'Graham', 'polite': True}

    @pytest.mark.parametrize("fixture", read_fixture())
    def test_greeter(fixture):
        answer = fixture.pop('answer')
>       assert greet(**fixture) == answer
E       AssertionError: assert 'How do you d…aham Chapman.' == 'How do you d…aahm Chapman.'
E         - How do you do, Graham Chapman.
E         ?                    -
E         + How do you do, Graahm Chapman.
E         ?                   +

greetings/greetings/test/test_greeter.py:16: AssertionError
_____________________________ test_greeter[fixture2] _____________________________

fixture = {'family': 'Palin', 'personal': 'Michael', 'title': 'CBE'}

    @pytest.mark.parametrize("fixture", read_fixture())
    def test_greeter(fixture):
        answer = fixture.pop('answer')
>       assert greet(**fixture) == answer
E       AssertionError: assert 'Hey, CBE Michael Palin.' == 'Hey, CBE Mike Palin.'
E         - Hey, CBE Michael Palin.
E         ?            ^^^ -
E         + Hey, CBE Mike Palin.
E         ?            ^

greetings/greetings/test/test_greeter.py:16: AssertionError
======================== 3 failed, 1 passed in 0.10s =========================
```

4.5.14 Developer Install

If you modify your source files, you would now find it appeared as if the program doesn't change.

That's because pip install **copies** the files.

If you want to install a package, but keep working on it, you can do:

```
pip install --editable .
```

4.5.15 Distributing compiled code

If you're working in C++ or Fortran, there is no language specific repository. You'll need to write platform installers for as many platforms as you want to support.

Typically:

- `dpkg` for `apt-get` on Ubuntu and Debian
- `rpm` for `yum/dnf` on Redhat and Fedora
- `homebrew` on OSX (Possibly `macports` as well)
- An executable `msi` installer for Windows.

Homebrew

Homebrew: A ruby DSL, you host off your own webpage
See an installer for the cppcourse example
If you're on OSX, do:

```
brew tap jamespjh/homebrew-reactor
brew install reactor
```

4.6 Documentation

4.6.1 Documentation is hard

- Good documentation is hard, and very expensive.
- Bad documentation is detrimental.
- Good documentation quickly becomes bad if not kept up-to-date with code changes.
- Professional companies pay large teams of documentation writers.

4.6.2 Prefer readable code with tests and vignettes

If you don't have the capacity to maintain great documentation, focus on:

- Readable code
- Automated tests
- Small code samples demonstrating how to use the api

4.6.3 Comment-based Documentation tools

Documentation tools can produce extensive documentation about your code by pulling out comments near the beginning of functions, together with the signature, into a web page.

The most popular is Doxygen. Have a look at an example of some Doxygen output.

Sphinx is nice for Python, and works with C++ as well. Here's some Sphinx-generated output and the corresponding source code. Breathe can be used to make Sphinx and Doxygen work together.

Roxygen is good for R.

4.7 Example of using Sphinx

4.7.1 Write some docstrings

We're going to document our "greeter" example using docstrings with Sphinx.

There are various conventions for how to write docstrings, but the native sphinx one doesn't look nice when used with the built in `help` system.

In writing Greeter, we used the docstring conventions from NumPy. So we use the numpydoc sphinx extension to support these.

```
"""
Generate a greeting string for a person.

Parameters
----------
```

```
personal: str
    A given name, such as Will or Jean-Luc

family: str
    A family name, such as Riker or Picard

title: str
    An optional title, such as Captain or Reverend

polite: bool
    True for a formal greeting, False for informal.

Returns
-------
string
    An appropriate greeting
"""
```

4.7.2 Set up sphinx

Invoke the sphinx-quickstart command to build Sphinx's configuration file automatically based on questions at the command line:

```
sphinx-quickstart
```

Which responds:

```
Welcome to the Sphinx 1.8.0 quickstart utility.

Please enter avalues for the following settings (just press Enter to
accept a default value, if one is given in brackets).

Enter the root path for documentation.
> Root path for the documentation [.]:
```

and then look at and adapt the generated config, a file called conf.py in the root of the project. This contains the project's Sphinx configuration, as Python variables:

```
#Add any Sphinx extension module names here, as strings. They can be
#extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
# ones.
extensions = [
    'sphinx.ext.autodoc',  # Support automatic documentation
    'sphinx.ext.coverage', # Automatically check if functions are documented
    'sphinx.ext.mathjax',  # Allow support for algebra
    'sphinx.ext.viewcode', # Include the source code in documentation
    'numpydoc'             # Support NumPy style docstrings
]
```

To proceed with the example, we'll copy a finished conf.py into our folder, though normally you'll always use `sphinx-quickstart`

```
In [1]: %%writefile greetings/conf.py

        import sys
        import os
```

```
        extensions = [
            'sphinx.ext.autodoc',  # Support automatic documentation
            'sphinx.ext.coverage', # Automatically check if functions are documented
            'sphinx.ext.mathjax',  # Allow support for algebra
            'sphinx.ext.viewcode', # Include the source code in documentation
            'numpydoc'             # Support NumPy style docstrings
        ]
        templates_path = ['_templates']
        source_suffix = '.rst'
        master_doc = 'index'
        project = u'Greetings'
        copyright = u'2014, James Hetherington'
        version = '0.1'
        release = '0.1'
        exclude_patterns = ['_build']
        pygments_style = 'sphinx'
        htmlhelp_basename = 'Greetingsdoc'
        latex_elements = {
        }

        latex_documents = [
          ('index', 'Greetings.tex', u'Greetings Documentation',
           u'James Hetherington', 'manual'),
        ]

        man_pages = [
            ('index', 'greetings', u'Greetings Documentation',
             [u'James Hetherington'], 1)
        ]

        texinfo_documents = [
          ('index', 'Greetings', u'Greetings Documentation',
           u'James Hetherington', 'Greetings', 'One line description of project.',
           'Miscellaneous'),
        ]
```

```
Overwriting greetings/conf.py
```

4.7.3 Define the root documentation page

Sphinx uses RestructuredText another wiki markup format similar to Markdown.

You define an "index.rst" file to contain any preamble text you want. The rest is autogenerated by `sphinx-quickstart`

```
In [2]: %%writefile greetings/index.rst
        Welcome to Greetings's documentation!
        =====================================

        Simple "Hello, James" module developed to teach research software engineering.

        .. autofunction:: greetings.greeter.greet
```

```
Overwriting greetings/index.rst
```

4.7.4 Run sphinx

We can run Sphinx using:

```
In [3]: %%bash
        cd greetings/
        sphinx-build . doc
```

```
Running Sphinx v2.3.1
making output directory… done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 1 source files that are out of date
updating environment: [new config] 1 added, 0 changed, 0 removed
reading sources… [100%] index

looking for now-outdated files… none found
pickling environment… done
checking consistency… done
preparing documents… done
writing output… [100%] index

generating indices…  genindexdone
highlighting module code… [100%] greetings.greeter

writing additional pages…  searchdone
copying static files… … done
copying extra files… done
dumping search index in English (code: en)… done
dumping object inventory… done
build succeeded.

The HTML pages are in doc.
```

4.7.5 Sphinx output

Sphinx's output is html. We just created a simple single function's documentation, but Sphinx will create multiple nested pages of documentation automatically for many functions.

4.8 Doctest - testing your documentation is up to date

`doctest` is a module included in the standard library. It runs all the code within the docstrings and checks whether the output is what it's claimed on the documentation.

Let's add an example to our greeting function and check it with `doctest`. We are leaving the output with a small typo to see what's the type of output we get from `doctest`.

```
In [4]: %%writefile greetings/greetings/greeter.py
        def greet(personal, family, title="", polite=False):
            """ Generate a greeting string for a person.

            Parameters
            ----------
            personal: str
                A given name, such as Will or Jean-Luc
            family: str
```

```
            A family name, such as Riker or Picard
        title: str
            An optional title, such as Captain or Reverend
        polite: bool
            True for a formal greeting, False for informal.

        Returns
        -------
        string
            An appropriate greeting

        Examples
        --------
        >>> from greetings.greeter import greet
        >>> greet("Terry", "Jones")
        'Hey, Terry Jones.
        """

        greeting= "How do you do, " if polite else "Hey, "
        if title:
            greeting += f"{title} "

        greeting += f"{personal} {family}."
        return greeting
```

```
Overwriting greetings/greetings/greeter.py
```

```
In [5]: %%bash --no-raise-error
        python -m doctest greetings/greetings/greeter.py
```

```
**********************************************************************
File "greetings/greetings/greeter.py", line 23, in greeter.greet
Failed example:
    greet("Terry", "Jones")
Expected:
    'Hey, Terry Jones.
Got:
    'Hey, Terry Jones.'
**********************************************************************
1 items had failures:
   1 of   2 in greeter.greet
***Test Failed*** 1 failures.
```

which clearly identifies a tiny error in our example.
pytest can run the doctest too if you call it as:
`pytest  --doctest-modules`

4.9 Software Project Management

4.9.1 Software Engineering Stages

- Requirements
- Functional Design

- Architectural Design
- Implementation
- Integration

4.9.2 Requirements Engineering

Requirements capture obviously means describing the things the software needs to be able to do.

A common approach is to write down lots of "user stories", describing how the software helps the user achieve something:

> As a clinician, when I finish an analysis, I want a report to be created on the test results, so that I can send it to the patient.

As a *role*, when *condition or circumstance applies* I want *a goal or desire* so that *benefits occur*.

These are easy to map into the Gherkin behaviour driven design test language.

4.9.3 Functional and architectural design

Engineers try to separate the functional design, how the software appears to and is used by the user, from the architectural design, how the software achieves that functionality.

Changes to functional design require users to adapt, and are thus often more costly than changes to architectural design.

4.9.4 Waterfall

The *Waterfall* design philosophy argues that the elements of design should occur in order: first requirements capture, then functional design, then architectural design. This approach is based on the idea that if a mistake is made in the design, then programming effort is wasted, so significant effort is spent in trying to ensure that requirements are well understood and that the design is correct before programming starts.

4.9.5 Why Waterfall?

Without a design approach, programmers resort to designing as we go, typing in code, trying what works, and making it up as we go along. When trying to collaborate to make software with others this can result in lots of wasted time, software that only the author understands, components built by colleagues that don't work together, or code that the programmer thinks is nice but that doesn't meet the user's requirements.

4.9.6 Problems with Waterfall

Waterfall results in a contractual approach to development, building an us-and-them relationship between users, business types, designers, and programmers.

> I built what the design said, so I did my job.

Waterfall results in a paperwork culture, where people spend a long time designing standard forms to document each stage of the design, with less time actually spent *making things*.

Waterfall results in excessive adherence to a plan, even when mistakes in the design are obvious to people doing the work.

4.9.7 Software is not made of bricks

The waterfall approach to software engineering comes from the engineering tradition applied to building physical objects, where Architects and Engineers design buildings, and builders build them according to the design.

Software is intrinsically different:

4.9.8 Software is not made of bricks

> Software is not the same 'stuff' as that from which physical systems are constructed. Software systems differ in material respects from physical systems. Much of this has been rehearsed by Fred Brooks in his classic 'No Silver Bullet' paper. First, complexity and scale are different in the case of software systems: relatively functionally simple software systems comprise more independent parts, placed in relation to each other, than do physical systems of equivalent functional value. Second, and clearly linked to this, we do not have well developed components and composition mechanisms from which to build software systems (though clearly we are working hard on providing these) nor do we have a straightforward mathematical account that permits us to reason about the effects of composition.

4.9.9 Software is not made of bricks

> Third, software systems operate in a domain determined principally by arbitrary rules about information and symbolic communication whilst the operation of physical systems is governed by the laws of physics. Finally, software is readily changeable and thus is changed, it is used in settings where our uncertainty leads us to anticipate the need to change.

– Prof. Anthony Finkelstein, UCL Dean of Engineering, and Professor of Software Systems Engineering

4.9.10 The Agile Manifesto

In 2001, authors including Martin Fowler, Ward Cunningham and Kent Beck met in a Utah ski resort, and published the following manifesto.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions* over processes and tools
- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

4.9.11 Agile is not absence of process

> The Agile movement is not anti-methodology, in fact, many of us want to restore credibility to the word methodology. We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment. Those who would brand proponents of XP or SCRUM or any of the other Agile Methodologies as "hackers" are ignorant of both the methodologies and the original definition of the term hacker

– Jim Highsmith.

4.9.12 Elements of an Agile Process

- Continuous delivery
- Self-organising teams
- Iterative development
- Ongoing design

4.9.13 Ongoing Design

Agile development doesn't eschew design. Design documents should still be written, but treated as living documents, updated as more insight is gained into the task, as work is done, and as requirements change.

Use of a Wiki or version control repository to store design documents thus works much better than using Word documents!

Test-driven design and refactoring are essential techniques to ensure that lack of "Big Design Up Front" doesn't produce badly constructed spaghetti software which doesn't meet requirements. By continously scouring our code for smells, and stopping to refactor, we evolve towards a well-structured design with weakly interacting units. By starting with tests which describe how our code should behave, we create executable specifications, giving us confidence that the code does what it is supposed to.

4.9.14 Iterative Development

Agile development maintains a backlog of features to be completed and bugs to be fixed. In each iteration, we start with a meeting where we decide which backlog tasks will be attempted during the development cycle, estimating how long each will take, and selecting an achievable set of goals for the "sprint". At the end of each cycle, we review the goals completed and missed, and consider what went well, what went badly, and what could be improved.

We try not to add work to a cycle mid-sprint. New tasks that emerge are added to the backlog, and considered in the next planning meeting. This reduces stress and distraction.

4.9.15 Continuous Delivery

In agile development, we try to get as quickly as possible to code that can be *demonstrated* to clients. A regular demo of progress to clients at the end of each development iteration says so much more than sharing a design document. "Release early, release often" is a common slogan. Most bugs are found by people *using* code – so exposing code to users as early as possible will help find bugs quickly.

4.9.16 Self-organising teams

Code is created by people. People work best when they feel ownership and pride in their work. Division of responsiblities into designers and programmers results in a "Code Monkey" role, where the craftspersonship and sense of responsibility for code quality is lost. Agile approaches encourage programmers, designers, clients, and businesspeople to see themselves as one team, working together, with fluid roles. Programmers grab issues from the backlog according to interest, aptitude, and community spirit.

4.9.17 Agile in Research

Agile approaches, where we try to turn the instincts and practices which emerge naturally when smart programmers get together into well-formulated best practices, have emerged as antidotes to both the chaotic free-form typing in of code, and the rigid paperwork-driven approaches of Waterfall.

If these approaches have turned out to be better even in industrial contexts, where requirements for code can be well understood, they are even more appropriate in a research context, where we are working in poorly understood fields with even less well captured requirements.

4.9.18 Conclusion

- Don't ignore design
- See if there's a known design pattern that will help
- Do try to think about how your code will work before you start typing
- Do use design tools like UML to think about your design without coding straight away
- Do try to write down some user stories
- Do maintain design documents.

BUT

- Do change your design as you work, updating the documents if you have them
- Don't go dark – never do more than a couple of weeks programming without showing what you've done to colleagues
- Don't get isolated from the reasons for your code's existence, stay involved in the research, don't be a Code Monkey.
- Do keep a list of all the things your code needs, estimate and prioritise tasks carefully.

4.10 Managing software issues

4.10.1 Issues

Code has *bugs*. It also has *features*, things it should do.

A good project has an organised way of managing these. Generally you should use an issue tracker.

4.10.2 Some Issue Trackers

There are lots of good issue trackers.

The most commonly used open source ones are Trac and Redmine.

Cloud based issue trackers include Lighthouse and GitHub.

Commercial solutions include Jira.

4.10.3 Anatomy of an issue

- Reporter
- Description
- Owner
- Type [Bug, Feature]
- Component
- Status
- Severity

4.10.4 Reporting a Bug

The description should make the bug reproducible:

- Version
- Steps

If possible, submit a minimal reproducing code fragment - look at this detailed answer about how to create a minimal example for *LaTeX*.

4.10.5 Owning an issue

- Whoever the issue is assigned to works next.
- If an issue needs someone else's work, assign it to them.

4.10.6 Status

- Submitted
- Accepted
- Underway
- Blocked

4.10.7 Resolutions

- Resolved
- Will Not Fix
- Not reproducible
- Not a bug (working as intended)

4.10.8 Bug triage

Some organisations use a severity matrix based on:

- Severity [Wrong answer, crash, unusable, workaround, cosmetic…]
- Frequency [All users, most users, some users…]

4.10.9 The backlog

The list of all the bugs that need to be fixed or features that have been requested is called the "backlog".

4.10.10 Development cycles

Development goes in *cycles*.

Cycles range in length from a week to three months.

In a given cycle:

- Decide which features should be implemented
- Decide which bugs should be fixed
- Move these issues from the Backlog into the current cycle. (Aka Sprint)

4.10.11 GitHub issues

GitHub doesn't have separate fields for status, component, severity etc. Instead, it just has labels, which you can create and delete.

See for example Jupyter.

4.11 Software Licensing

4.11.1 Reuse

4.11.2 Disclaimer

Here we attempt to give some basic advice on choosing a licence for your software. But:

- we are NOT lawyers (IANAL),
- opinions differ (and flamewars are boring),
- this training does NOT constitute legal advice.

For an in-depth discussion of software licences, read the O'Reilly book Understanding Open Source and Free Software Licensing.

Your department, or UCL, may have policies about applying licences to code you create while a UCL employee or student. This training doesn't address this issue, and does not represent UCL policy – seek advice from your supervisor or manager if concerned.

4.11.3 Choose a licence

It is important to choose a licence and to create a *license file* to tell people what it is.

The licence lets people know whether they can reuse your code and under what terms. This course has one, for example.

Your licence file should typically be called LICENSE.txt or similar. GitHub will offer to create a licence file automatically when you create a new repository.

4.11.4 Open source doesn't stop you making money

A common misconception about open source software is the thought that open source means you can't make any money. This is *wrong*.

Plenty of people open source their software and profit from:

- The software under a different licence e.g. Saxon
- Consulting. For example: Anaconda who help maintain NumPy.
- Manuals. For example: VTK.
- Add-ons. For example: Puppet.
- Server software, which open source client software interacts with. For example: GitHub API clients.

4.11.5 Plagiarism vs promotion

Many researchers worry about people stealing their work if they open source their code. But often the biggest problem is not theft, but the fact no one is aware of your work.

Open source is a way to increase the probability that someone else on the planet will care enough about your work to cite you.

So when thinking about whether to open source your code, think about whether you're more worried about anonymity or theft.

4.11.6 Your code *is* good enough

New coders worry that they'll be laughed at if they put their code online. Don't worry. Everyone, including people who've been coding for decades, writes shoddy code that is full of bugs.

The only thing that will make your code better, is *other people reading it*.

For small scripts that no one but you will ever use, my recommendation is to use an open repository anyway. Find a buddy, and get them to comment on it.

4.11.7 Worry about licence compatibility and proliferation

Not all open source code can be used in all projects. Some licences are legally incompatible.

This is a huge and annoying problem. As an author, you might not care, but you can't anticipate the exciting uses people might find by mixing your code with someone else's.

Use a standard licence from the small list that are well-used. Then people will understand. *Don't make up your own*.

When you're about to use a licence, see if there's a more common one which is recommended, e.g.: using the opensource.org proliferation report.

4.11.8 Academic licence proliferation

Academics often write their own licence terms for their software.

For example:

> XXXX NON-COMMERCIAL EDUCATIONAL LICENSE Copyright (c) 2013 Prof. Foo. All rights reserved.

> You may use and modify this software for any non-commercial purpose within your educational institution. Teaching, academic research, and personal experimentation are examples of purpose which can be non-commercial.
>
> You may redistribute the software and modifications to the software for non-commercial purposes, but only to eligible users of the software (for example, to another university student or faculty to support joint academic research).

Please don't do this. Your desire to slightly tweak the terms is harmful to the future software ecosystem. Also, *Unless you are a lawyer, you cannot do this safely!*

4.11.9 Licences for code, content, and data.

Licences designed for code should not be used to license data or prose.

Don't use Creative Commons for software, or GPL for a book.

4.11.10 Licensing issues

- Permissive vs share-alike
- Non-commercial and academic Use Only
- Patents
- Use as a web service

4.11.11 Permissive vs share-alike

Some licences require all derived software to be licensed under terms that are similarly free. Such licences are called "Share Alike" or "Copyleft".

- Licences in this class include the GPL.

Those that don't are called "Permissive"

- These include Apache, BSD, and MIT licences.

If you want your code to be maximally reusable, use a permissive licence If you want to force other people using your code to make derivatives open source, use a copyleft licence.

If you want to use code that has a permissive licence, it's safe to use it and keep your code secret. If you want to use code that has a copyleft licence, you'll have to release your code under such a licence.

4.11.12 Academic use only

Some researchers want to make their code free for 'academic use only'. None of the standard licences state this, and this is a reason why academic bespoke licences proliferate.

However, there is no need for this, in our opinion.

Use of a standard Copyleft licence precludes derived software from being sold without also publishing the source

So use of a Copyleft licence precludes commercial use.

This is a very common way of making a business from open source code: offer the code under GPL for free but offer the code under more permissive terms, allowing for commercial use, for a fee.

4.11.13 Patents

Intellectual property law distinguishes copyright from patents. This is a complex field, which I am far from qualified to teach!

People who think carefully about intellectual property law distinguish software licences based on how they address patents. Very roughly, if you want to ensure that contributors to your project can't then go off and patent their contribution, some licences, such as the Apache licence, protect you from this.

4.11.14 Use as a web service

If I take copyleft code, and use it to host a web service, I have not sold the software.

Therefore, under some licences, I do not have to release any derivative software. This "loophole" in the GPL is closed by the AGPL ("Affero GPL")

4.11.15 Library linking

If I use your code just as a library, without modifying it or including it directly in my own code, does the copyleft term of the GPL apply?

Yes

If you don't want it to, use the LGPL. ("Lesser GPL"). This has an exception for linking libraries.

4.11.16 Citing software

Almost all software licences require people to credit you for what they used ("attribution").

In an academic context, it is useful to offer a statement as to how best to do this, citing *which paper to cite in all papers which use the software.*

This is best done with a CITATION file in your repository.

> To cite ggplot2 in publications, please use:
>
> H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2016.
>
> A BibTeX entry for LaTeX users is
>
> @Book{, author = {Hadley Wickham}, title = {ggplot2: Elegant Graphics for Data Analysis}, publisher = {Springer-Verlag New York}, year = {2016}, isbn = {978-3-319-24277-4}, url = {https://ggplot2.tidyverse.org}, }

4.11.17 Referencing the licence in every file

Some licences require that you include licence information in every file. Others do not.

Typically, every file should contain something like:

```
# (C) University College London 2010-2014
# This software is licenced under the terms of the <foo licence>
# See <somewhere> for the licence details.
```

Check your licence at opensource.org for details of how to apply it to your software. For example, for the GPL.

4.11.18 Choose a licence

See GitHub's advice on how to choose a licence.

4.11.19 Open source does not equal free maintenance

One common misunderstanding of open source software is that you'll automatically get loads of contributors from around the internets. This is wrong. Most open source projects get no commits from anyone else.

Open source does *not* guarantee your software will live on with people adding to it after you stop working on it.

Learn more about these issues from the website of the Software Sustainability Institute.

Chapter 5

Construction

5.1 Construction

Software *design* gets a lot of press (Object orientation, UML, design patterns).

In this session we're going to look at advice on software *construction*.

5.1.1 Construction vs Design

For a given piece of code, there exist several different ways one could write it:

- Choice of variable names
- Choice of comments
- Choice of layout

The consideration of these questions is the area of Software Construction.

5.1.2 Low-level design decisions

We will also look at some of the lower-level software design decisions in the context of this section:

- Division of code into subroutines
- Subroutine access signatures
- Choice of data structures for readability

5.1.3 Algorithms and structures

We will not, in discussing construction, be looking at decisions as to how design questions impact performance:

- Choice of algorithms
- Choice of data structures for performance
- Choice of memory layout

We will consider these in a future discussion of performance programming.

5.1.4 Architectural design

We will not, in this session, be looking at the large-scale questions of how program components interact, the stategic choices that govern how software behaves at the large scale:

- Where do objects get made?
- Which objects own or access other objects?
- How can I hide complexity in one part of the code from other parts of the code?

We will consider these in a future session.

5.1.5 Construction

So, we've excluded most of the exciting topics. What's left is the bricks and mortar of software: how letters and symbols are used to build code which is readable.

5.1.6 Literate programming

In literature, books are enjoyable for different reasons:

- The beauty of stories
- The beauty of plots
- The beauty of characters
- The beauty of paragraphs
- The beauty of sentences
- The beauty of words

Software has beauty at these levels too: stories and characters correspond to architecture and object design, plots corresponds to algorithms, but the rhythm of sentences and the choice of words corresponds to software construction.

5.1.7 Programming for humans

- Remember you're programming for humans as well as computers
- A program is the best, most rigorous way to describe an algorithm
- Code should be pleasant to read, a form of scholarly communication

Read Steve McConnell's Code Complete [UCL library].

5.1.8 Setup

This notebook is based on a number of fragments of code, with an implicit context. We've made a library to set up the context so the examples work.

```
In [1]: %%writefile context.py
        from unittest.mock import Mock, MagicMock
        class CompMock(Mock):
            def __sub__(self, b):
                return CompMock()
            def __lt__(self,b):
                return True
            def __abs__(self):
                return CompMock()
        array=[]
        agt=[]
        ws=[]
        agents=[]
        counter=0
        x=MagicMock()
        y=None
        agent=MagicMock()
        value=0
        bird_types=["Starling", "Hawk"]
        import numpy as np
        average=np.mean
        hawk=CompMock()
        starling=CompMock()
```

```
        sEntry="2.0"
        entry ="2.0"
        iOffset=1
        offset =1
        anothervariable=1
        flag1=True
        variable=1
        flag2=False
        def do_something(): pass
        chromosome=None
        start_codon=None
        subsequence=MagicMock()
        transcribe=MagicMock()
        ribe=MagicMock()
        find=MagicMock()
        can_see=MagicMock()
        my_name=""
        your_name=""
        flag1=False
        flag2=False
        start=0.0
        end=1.0
        step=0.1
        birds=[MagicMock()]*2
        resolution=100
        pi=3.141
        result= [0]*resolution
        import numpy as np
        import math
        data= [math.sin(y) for y in np.arange(0,pi,pi/resolution)]
        import yaml
        import os
```

```
Writing context.py
```

5.2 Coding Conventions

Let's import first the context for this chapter.

```
In [1]: from context import *
```

5.2.1 One code, many layouts:

Consider the following fragment of python:

```
In [2]: import species
        def AddToReaction(name, reaction):
            reaction.append(species.Species(name))
```

this could also have been written:

```
In [3]: from species import Species

        def add_to_reaction(a_name,
```

```
                        a_reaction):
            l_species = Species(a_name)
            a_reaction.append( l_species )
```

5.2.2 So many choices

- Layout
- Naming
- Syntax choices

5.2.3 Layout

```
In [4]: reaction = {
            "reactants": ["H", "H", "O"],
            "products": ["H2O"]
        }
```

```
In [5]: reaction2=(
        {
          "reactants":
          [
            "H",
            "H",
            "O"
          ],
          "products":
          [
            "H2O"
          ]
        }
        )
```

5.2.4 Layout choices

- Brace style
- Line length
- Indentation
- Whitespace/Tabs

Inconsistency will produce a mess in your code! Some choices will make your code harder to read, whereas others may affect the code. For example, if you copy/paste code with tabs in a place that's using spaces, they may appear OK in your screen but it will fail when running it.

5.2.5 Naming Conventions

Camel case is used in the following example, where class name is in UpperCamel, functions in lowerCamel and underscore_separation for variables names. This convention is used broadly in the python community.

```
In [6]: class ClassName:
            def methodName(variable_name):
                instance_variable = variable_name
```

This other example uses underscore_separation for all the names.

```
In [7]: class class_name:
            def method_name(a_variable):
                m_instance_variable = a_variable
```

5.2.6 Hungarian Notation

Prefix denotes *type*:

```
In [8]: fNumber = float(sEntry) + iOffset
```

So in the example above we know that we are creating a `f`loat number as a composition of a `s`tring entry and an `i`nteger offset.

People may find this useful in languages like Python where the type is intrisic in the variable.

```
In [9]: number = float(entry) + offset
```

5.2.7 Newlines

- Newlines make code easier to read
- Newlines make less code fit on a screen

Use newlines to describe your code's *rhythm*.

5.2.8 Syntax Choices

The following two snippets do the same, but the second is separated into more steps, making it more readable.

```
In [10]: anothervariable += 1
         if ((variable == anothervariable) and flag1 or flag2): do_something()
```

```
In [11]: anothervariable = anothervariable + 1
         variable_equality = (variable == anothervariable)
         if ((variable_equality and flag1) or flag2):
             do_something()
```

We create extra variables as an intermediate step. Don't worry about the performance now, the compiler will do the right thing.

What about operator precedence? Being explicit helps to remind yourself what you are doing.

5.2.9 Syntax choices

- Explicit operator precedence
- Compound expressions
- Package import choices

5.2.10 Coding Conventions

You should try to have an agreed policy for your team for these matters.

If your language sponsor has a standard policy, use that. For example:

- **Python**: PEP8
- **R**: Google's guide for R, tidyverse style guide
- **C++**: Google's style guide, Mozilla's
- **Julia**: Official style guide

5.2.11 Lint

There are automated tools which enforce coding conventions and check for common mistakes.

These are called **linters**. A popular one is pycodestyle:

E.g. `pip install pycodestyle`

```
In [12]: %%bash --no-raise-error
         pycodestyle species.py
```

```
species.py:2:6: E111 indentation is not a multiple of four
species.py:2:6: E117 over-indented
```

It is a good idea to run a linter before every commit, or include it in your CI tests.

There are other tools that help with linting that are worth mentioning. With pylint you can also get other useful information about the quality of your code:

`pip install pylint`

```
In [13]: %%bash --no-raise-error
         pylint species.py
```

```
************* Module species
species.py:2:0: W0311: Bad indentation. Found 5 spaces, expected 4 (bad-indentation)
species.py:1:0: C0114: Missing module docstring (missing-module-docstring)
species.py:1:0: C0115: Missing class docstring (missing-class-docstring)
species.py:1:0: R0205: Class 'Species' inherits from object, can be safely removed from bases in python3
species.py:1:0: R0903: Too few public methods (0/2) (too-few-public-methods)

-------------------------------------
Your code has been rated at -15.00/10
```

and with black you can fix all the errors at once.

```
black species.py
```

These linters can be configured to choose which points to flag and which to ignore.

Do not blindly believe all these automated tools! Style guides are **guides** not **rules**.

Finally, there are tools like editorconfig to help sharing the conventions used within a project, where each contributor uses different IDEs and tools. There are also bots like pep8speaks that comments on contributors' pull requests suggesting what to change to follow the conventions for the project.

5.3 Comments

Let's import first the context for this chapter.

```
In [1]: from context import *
```

5.3.1 Why comment?

- You're writing code for people, as well as computers.
- Comments can help you build code, by representing your design
- Comments explain subtleties in the code which are not obvious from the syntax
- Comments explain *why* you wrote the code the way you did

5.3.2 Bad Comments

"I write good code, you can tell by the number of comments."

This is wrong.

5.3.3 Comments which are obvious

```
In [2]: counter = counter + 1 # Increment the counter
        for element in array: # Loop over elements
            pass
```

5.3.4 Comments which could be replaced by better style

The following piece of code could be a part of a game to move a turtle in a certain direction, with a particular angular velocity and step size.

```
In [3]: for i in range(len(agt)): #for each agent
            agt[i].theta += ws[i]     # Increment the angle of each agent
                                      #by its angular velocity
            agt[i].x += r * sin(agt[i].theta) #Move the agent by the step-size
            agt[i].y += r * cos(agt[i].theta) #r in the direction indicated
```

we have used comments to make the code readable.

Why not make the code readable instead?

```
In [4]: for agent in agents:
            agent.turn()
            agent.move()

        class Agent:
            def turn(self):
                 self.direction += self.angular_velocity;
            def move(self):
                self.x += Agent.step_length * sin(self.direction)
                self.y += Agent.step_length * cos(self.direction)
```

This is probably better. We are using the name of the functions (*i.e.*, `turn`, `move`) instead of comments. Therefore, we've got *self-documenting* code.

5.3.5 Comments vs expressive code

> The proper use of comments is to compensate for our failure to express yourself in code. Note that I used the word failure. I meant it. Comments are always failures.

– Robert Martin, Clean Code [UCL library].

I wouldn't disagree, but still, writing "self-documenting" code is very hard, so do comment if you're unsure!

5.3.6 Comments which belong in an issue tracker

```
In [5]: x.clear() # Code crashes here sometimes
        class Agent(object):
            pass
            # TODO: Implement pretty-printer method
```

BUT comments that reference issues in the tracker can be good.

E.g.

```
In [6]: if x.safe_to_clear(): # Guard added as temporary workaround for #32
            x.clear()
```

is OK. And platforms like GitHub will create a link to it when browsing the code.

5.3.7 Comments which only make sense to the author today

```
In [7]: agent.turn() # Turtle Power!
        agent.move()
        agents[:]=[]# Shredder!
```

5.3.8 Comments which are unpublishable

```
In [8]: # Stupid supervisor made me write this code
        # So I did it while very very drunk.
```

5.3.9 Good commenting: pedagogical comments

Code that *is* good style, but you're not familiar with, or that colleagues might not be familiar with

```
In [9]: # This is how you define a decorator in python
        # See https://wiki.python.org/moin/PythonDecorators
        def double(decorated_function):
            # Here, the result function forms a closure over
            # the decorated function
            def result_function(entry):
                return decorated_function(decorated_function(entry))
            # The returned result is a function
            return result_function

        @double
        def try_me_twice():
            pass
```

5.3.10 Good commenting: reasons and definitions

Comments which explain coding definitions or reasons for programming choices.

```
In [10]: def __init__(self):
             self.angle = 0 # clockwise from +ve y-axis
             nonzero_indices = [] # Use sparse model as memory constrained
```

5.4 Refactoring

Let's import first the context for this chapter.

```
In [1]: from context import *
```

Let's put ourselves in an scenario - that you've probably been in before. Imagine you are changing a large piece of legacy code that's not well structured, introducing many changes at once, trying to keep in your head all the bits and pieces that need to be modified to make it all work again. And suddenly, your officemate comes and ask you to go for coffee… and you've lost all track of what you had in your head and need to start again.

Instead of doing so, we could use a more robust approach to go from nasty ugly code to clean code in a safer way.

5.4.1 Refactoring

To refactor is to:

- Make a change to the design of some software
- Which improves the structure or readability
- But which leaves the actual behaviour of the program completely unchanged.

5.4.2 A word from the Master

> Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which "too small to be worth doing". However the cumulative effect of each of these transformations is quite significant. By doing them in small steps you reduce the risk of introducing errors. You also avoid having the system broken while you are carrying out the restructuring - which allows you to gradually refactor a system over an extended period of time.

– Martin Fowler Refactoring [UCL library].

5.4.3 List of known refactorings

The next few sections will present some known refactorings.

We'll show before and after code, present any new coding techniques needed to do the refactoring, and describe *code smells*: how you know you need to refactor.

5.4.4 Replace magic numbers with constants

Smell: Raw numbers appear in your code

Before:

```
In [2]: data = [math.sin(x) for x in np.arange(0,3.141,3.141/100)]
        result = [0]*100
        for i in range(100):
            for j in range(i+1, 100):
                result[j] += data[i] * data[i-j] / 100
```

after:

```
In [3]: resolution = 100
        pi = 3.141
        data = [math.sin(x) for x in np.arange(0, pi, pi/resolution)]
        result = [0] * resolution
        for i in range(resolution):
            for j in range(i + 1, resolution):
                result[j] += data[i] * data[i-j] / resolution
```

5.4.5 Replace repeated code with a function

Smell: Fragments of repeated code appear.

Fragment of model where some birds are chasing each other: if the angle of view of one can see the prey, then start hunting, and if the other see the predator, then start running away.

Before:

```
In [4]: if abs(hawk.facing - starling.facing) < hawk.viewport:
            hawk.hunting()

        if abs(starling.facing - hawk.facing) < starling.viewport:
            starling.flee()
```

After:

```
In [5]: def can_see(source, target):
            return (source.facing - target.facing) < source.viewport

        if can_see(hawk, starling):
            hawk.hunting()

        if can_see(starling, hawk):
            starling.flee()
```

5.4.6 Change of variable name

Smell: Code needs a comment to explain what it is for.
Before:

```
In [6]: z = find(x,y)
        if z:
            ribe(x)
```

After:

```
In [7]: gene = subsequence(chromosome, start_codon)
        if gene:
            transcribe(gene)
```

5.4.7 Separate a complex expression into a local variable

Smell: An expression becomes long.

```
In [8]: if ((my_name == your_name) and flag1 or flag2): do_something()
```

vs

```
In [9]: same_names = (my_name == your_name)
        flags_OK = flag1 or flag2
        if same_names and flags_OK:
            do_something()
```

5.4.8 Replace loop with iterator

Smell: Loop variable is an integer from 1 to something.
Before:

```
In [10]: sum = 0
         for i in range(resolution):
             sum += data[i]
```

After:

```
In [11]: sum = 0
         for value in data:
             sum += value
```

5.4.9 Replace hand-written code with library code

Smell: It feels like surely someone else must have done this at some point.
Before:

```
In [12]: xcoords = [start + i * step for i in range(int((end - start) / step))]
```

After:

```
In [13]: import numpy as np
         xcoords = np.arange(start, end, step)
```

See Numpy, Pandas.

5.4.10 Replace set of arrays with array of structures

Smell: A function needs to work corresponding indices of several arrays:
Before:

```
In [14]: def can_see(i, source_angles, target_angles, source_viewports):
             return abs(source_angles[i] - target_angles[i]) < source_viewports[i]
```

After:

```
In [15]: def can_see(source, target):
             return (source["facing"] - target["facing"]) < source["viewport"]
```

Warning: this refactoring greatly improves readability but can make code slower, depending on memory layout. Be careful.

5.4.11 Replace constants with a configuration file

Smell: You need to change your code file to explore different research scenarios.
Before:

```
In [16]: flight_speed = 2.0 # mph
         bounds = [0, 0, 100, 100]
         turning_circle = 3.0 # m
         bird_counts = {"hawk": 5, "starling": 500}
```

After:

```
In [17]: %%writefile config.yaml
         bounds: [0, 0, 100, 100]
         counts:
             hawk: 5
             starling: 500
         speed: 2.0
         turning_circle: 3.0

Writing config.yaml


In [18]: config = yaml.safe_load(open("config.yaml"))
```

See YAML and PyYaml, and Python's os module.

5.4.12 Replace global variables with function arguments

Smell: A global variable is assigned and then used inside a called function:

```
In [19]: viewport = pi/4

         if hawk.can_see(starling):
             hawk.hunt(starling)

         class Hawk(object):
             def can_see(self, target):
                 return (self.facing - target.facing) < viewport
```

Becomes:

```
In [20]: viewport = pi/4
         if hawk.can_see(starling, viewport):
             hawk.hunt(starling)

         class Hawk(object):
             def can_see(self, target, viewport):
                 return (self.facing - target.facing) < viewport
```

5.4.13 Merge neighbouring loops

Smell: Two neighbouring loops have the same for statement

```
In [21]: for bird in birds:
             bird.build_nest()

         for bird in birds:
             bird.lay_eggs()
```

Becomes:

```
In [22]: for bird in birds:
             bird.build_nest()
             bird.lay_eggs()
```

Though there may be a case where all the nests need to be built before the birds can start laying eggs.

5.4.14 Break a large function into smaller units

- Smell: A function or subroutine no longer fits on a page in your editor.
- Smell: A line of code is indented more than three levels.
- Smell: A piece of code interacts with the surrounding code through just a few variables.

Before:

```
In [23]: def do_calculation():
             for predator in predators:
                 for prey in preys:
                     if predator.can_see(prey):
                         predator.hunt(prey)
                     if predator.can_reach(prey):
                         predator.eat(prey)
```

After:

```
In [24]: def do_calculation():
             for predator in predators:
                 for prey in preys:
                     predate(predator, prey)

         def predate(predator, prey):
             if predator.can_see(prey):
                 predator.hunt(prey)
             if predator.can_reach(prey):
                 predator.eat(prey)
```

5.4.15 Separate code concepts into files or modules

Smell: You find it hard to locate a piece of code.

Smell: You get a lot of version control conflicts.

Before:

```
In [25]: class One(object):
             pass

         class Two(object):
             def __init__():
                 self.child = One()
```

After:

```
In [26]: %%writefile anotherfile.py
         class One(object):
             pass
```

```
Writing anotherfile.py
```

```
In [27]: from anotherfile import One

         class Two(object):
             def __init__():
                 self.child = One()
```

5.4.16 Refactoring is a safe way to improve code

You may think you can see how to rewrite a whole codebase to be better.

However, you may well get lost halfway through the exercise.

By making the changes as small, reversible, incremental steps, you can reach your target design more reliably.

5.4.17 Tests and Refactoring

Badly structured code cannot be unit tested. There are no "units".

Before refactoring, ensure you have a robust regression test.

This will allow you to *Refactor with confidence*.

As you refactor, if you create any new units (functions, modules, classes), add new tests for them.

5.4.18 Refactoring Summary

- Replace magic numbers with constants
- Replace repeated code with a function
- Change of variable/function/class name
- Replace loop with iterator
- Replace hand-written code with library code
- Replace set of arrays with array of structures
- Replace constants with a configuration file
- Replace global variables with function arguments
- Break a large function into smaller units
- Separate code concepts into files or modules

And many more…
Read The Refactoring Book.

Chapter 6

Design

Let's import first the context for this chapter.

```
In [1]: from context import *
```

6.1 Object-Oriented Design

In this session, we will finally discuss the thing most people think of when they refer to "Software Engineering": the deliberate *design* of software. We will discuss processes and methodologies for planned development of large-scale software projects: *Software Architecture*.

The software engineering community has, in large part, focused on an object-oriented approach to the design and development of large scale software systems. The basic concepts of object orientation are necessary to follow much of the software engineering conversation.

6.1.1 Design processes

In addition to object-oriented architecture, software engineers have focused on the development of processes for robust, reliable software development. These codified ways of working hope to enable organisations to repeatably and reliably complete complex software projects in a way that minimises both development and maintainance costs, and meets user requirements.

6.1.2 Design and research

Software engineering theory has largely been developed in the context of commercial software companies.

The extent to which the practices and processes developed for commercial software are applicable in a research context is itself an active area of research.

6.2 Recap of Object-Orientation

6.2.1 Classes: User defined types

```
In [2]: class Person:
            def __init__(self, name, age):
                self.name = name
                self.age = age
            def grow_up(self):
                self.age += 1

        terry = Person("Terry", 76)
        terry.home = "Colwyn Bay"
```

Notice, that in Python, you can add properties to an object once it's been defined. Just because you can doesn't mean you should!

6.2.2 Declaring a class

Class: A user-defined type

```
In [3]: class MyClass:
            pass
```

6.2.3 Object instances

Instance: A particular object *instantiated* from a class.

```
In [4]: my_object = MyClass()
```

6.2.4 Method

Method: A function which is "built in" to a class

```
In [5]: class MyClass:
            def someMethod(self, argument):
                pass

        my_object = MyClass()
        my_object.someMethod(value)
```

6.2.5 Constructor

Constructor: A special method called when instantiating a new object

```
In [6]: class MyClass:
            def __init__(self, argument):
                pass

        my_object = MyClass(value)
```

6.2.6 Member Variable

Member variable: a value stored inside an instance of a class.

```
In [7]: class MyClass:
            def __init__(self):
                self.member = "Value"

        my_object = MyClass()
        assert(my_object.member == "Value")
```

6.3 Object refactorings

6.3.1 Replace add-hoc structure with user defined classes

Smell: A data structure made of nested arrays and dictionaries becomes unwieldy.
Before:

```
In [8]: from random import random
        birds = [{"position": random(),
                  "velocity": random(),
                  "type": kind} for kind in bird_types]

        average_position = average([bird["position"] for bird in birds])
```

After:

```
In [9]: class Bird:
            def __init__(self, kind):
                from random import random
                self.type = type
                self.position = random()
                self.velocity = random()

        birds = [Bird(kind) for kind in bird_types]
        average_position = average([bird.position for bird in birds])
```

6.3.2 Replace function with a method

Smell: A function is always called with the same kind of thing

Before:

```
In [10]: def can_see(source, target):
             return (source.facing - target.facing) < source.viewport

         if can_see(hawk, starling):
             hawk.hunt()
```

After:

```
In [11]: class Bird:
             def can_see(self, target):
                 return (self.facing - target.facing) < self.viewport

         if hawk.can_see(starling):
             hawk.hunt()
```

6.3.3 Replace method arguments with class members

Smell: A variable is nearly always used in arguments to a class.

```
In [12]: class Person:
             def __init__(self, genes):
                 self.genes = genes
             def reproduce_probability(self, age): pass
             def death_probability(self, age): pass
             def emigrate_probability(self, age): pass
```

After:

```
In [13]: class Person:
             def __init__(self, genes, age):
                 self.age = age
                 self.genes = genes
             def reproduce_probability(self): pass
             def death_probability(self): pass
             def emigrate_probability(self): pass
```

6.3.4 Replace global variable with class and member

Smell: A global variable is referenced by a few functions

```
In [14]: name = "Terry Jones"
         birthday = [1, 2, 1942]
         today = [22, 11]

         if today == birthday[0:2]:
             print(f"Happy Birthday, {name}")
         else:
             print("No birthday for you today.")
```

```
No birthday for you today.
```

```
In [15]: class Person(object):
             def __init__(self, birthday, name):
                 self.birth_day = birthday[0]
                 self.birth_month = birthday[1]
                 self.birth_year = birthday[2]
                 self.name = name
             def check_birthday(self, today_day, today_month):
                 if not self.birth_day == today_day:
                     return False
                 if not self.birth_month == today_month:
                     return False
                 return True
             def greet_appropriately(self, today):
                 if self.check_birthday(*today):
                     print(f"Happy Birthday, {self.name}")
                 else:
                     print("No birthday for you.")

         john = Person([5, 5, 1943], "Michael Palin")
         john.greet_appropriately(today)
```

```
No birthday for you.
```

6.3.5 Object Oriented Refactoring Summary

- Replace ad-hoc structure with a class
- Replace function with a method
- Replace method argument with class member
- Replace global variable with class data

6.4 Class design

The concepts we have introduced are common between different object oriented languages. Thus, when we design our program using these concepts, we can think at an architectural level, independent of language syntax.

In Python:

```
In [1]: class Particle:
            def __init__(self, position, velocity):
                self.position = position
                self.velocity = velocity
            def move(self, delta_t):
                self.position += self.velocity * delta_t
```

In C++:

```
class Particle {
    std::vector<double> position;
    std::vector<double> velocity;
    Particle(std::vector<double> position, std::vector<double> velocity);
    void move(double delta_t);
}
```

In Fortran:

```
type particle
    real :: position
    real :: velocity
  contains
    procedure :: init
    procedure :: move
end type particle
```

6.4.1 UML

UML is a conventional diagrammatic notation used to describe "class structures" and other higher level aspects of software design.

Computer scientists get worked up about formal correctness of UML diagrams and learning the conventions precisely. Working programmers can still benefit from using UML to describe their designs.

6.4.2 YUML

We can see a YUML model for a Particle class with `position` and `velocity` data and a `move()` method using the YUML online UML drawing tool (example).

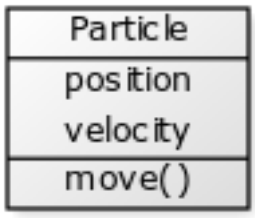
```
http://yuml.me/diagram/boring/class/[Particle|position;velocity|move%28%29]
```

Here's how we can use Python code to get an image back from YUML:

```
In [2]: import requests
        from IPython.display import Image

        def yuml(model):
            result = requests.get("http://yuml.me/diagram/boring/class/" + model)
            return Image(result.content)
```

```
In [3]: yuml("[Particle|position;velocity|move()]")
```

Out[3]:



The representation of the `Particle` class defined above in UML is done with a box with three sections. The name of the class goes on the top, then the name of the member variables in the middle, and the name of the methods on the bottom. We will see later why this is useful.

6.4.3 Information Hiding

Sometimes, our design for a program would be broken if users start messing around with variables we don't want them to change.

Robust class design requires consideration of which subroutines are intended for users to use, and which are internal. Languages provide features to implement this: access control.

In python, we use leading underscores to control whether member variables and methods can be accessed from outside the class: - single leading underscore (_) is used to document it's private but people could use it if wanted (thought they shouldn't); - double leading underscore (__) raises errors if called.

```
In [4]: class MyClass:
            def __init__(self):
                self.__private_data = 0
                self._private_data = 0
                self.public_data = 0

            def __private_method(self): pass

            def _private_method(self): pass

            def public_method(self): pass

            def called_inside(self):
                self.__private_method()
                self._private_method()
                self.__private_data = 1
                self._private_data = 1


        MyClass().called_inside()

In [5]: MyClass()._private_method() # Works, but forbidden by convention

In [6]: MyClass().public_method() # OK

        print(MyClass()._private_data)

0


In [7]: print(MyClass().public_data)

0


In [8]: MyClass().__private_method() # Generates error

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

```
        AttributeError                            Traceback (most recent call last)

        <ipython-input-8-e4355512aeb6> in <module>
    ----> 1 MyClass().__private_method() # Generates error


        AttributeError: 'MyClass' object has no attribute '__private_method'
```

```
In [9]: print(MyClass().__private_data) # Generates error
```

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

        AttributeError                            Traceback (most recent call last)

        <ipython-input-9-6c81459189e2> in <module>
    ----> 1 print(MyClass().__private_data) # Generates error


        AttributeError: 'MyClass' object has no attribute '__private_data'
```

6.4.4 Property accessors

Python provides a mechanism to make functions appear to be variables. This can be used if you want to change the way a class is implemented without changing the interface:

```
In [10]: class Person:
             def __init__(self):
                 self.name = "Graham Chapman"

         assert(Person().name == "Graham Chapman")
```

becomes:

```
In [11]: class Person(object):
             def __init__(self):
                 self._first = "Graham"
                 self._second = "Chapman"


             @property
             def name(self):
                 return f"{self._first self._second}"

         assert(Person().name == "Graham Chapman")
```

```
          File "<fstring>", line 1
        (self._first self._second)
                       ^
    SyntaxError: invalid syntax
```

Making the same external code work as before.

Note that the code behaves the same way to the outside user. The implementation detail is hidden by private variables. In languages without this feature, such as C++, it is best to always make data private, and always access data through functions:

```
In [12]: class Person(object):
             def __init__(self):
                 self._name = "Graham Chapman"

             def name(self):  # an access function
                 return self._name

         assert(Person().name() == "Graham Chapman")
```

But in Python this is unnecessary because the `@property` capability.

Another way could be to create a member variable `name` which holds the full name. However, this could lead to inconsistent data. If we create a `get_married` function, then the name of the person won't change!

```
In [13]: class Person(object):
             def __init__(self, first, second):
                 self._first = first
                 self._second = second
                 self.name = f"{self._first} {self._second}"

             def get_married(self, to):
                 self._second = to._second

         graham = Person("Graham", "Chapman")
         david = Person("David", "Sherlock")
         assert(graham.name == "Graham Chapman")
         graham.get_married(david)
         assert(graham.name == "Graham Sherlock")


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

         AssertionError                            Traceback (most recent call last)

         <ipython-input-13-bb03ebf6e67c> in <module>
          12 assert(graham.name == "Graham Chapman")
          13 graham.get_married(david)
     ---> 14 assert(graham.name == "Graham Sherlock")


         AssertionError:
```

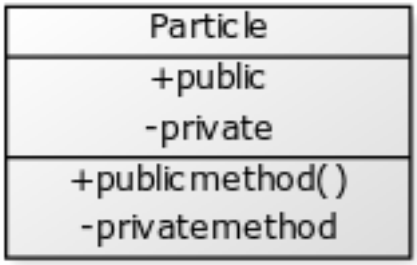
This type of situation could makes that the object data structure gets inconsistent with itself. Making variables being out of sync with other variables. Each piece of information should only be stored in once place! In this case, `name` should be calculated each time it's required as previously shown. In database design, this is called Normalisation.

UML for private/public

We prepend a `+/-` on public/private member variables and methods:

```
In [14]: yuml("[Particle|+public;-private|+publicmethod();-privatemethod]")
```

Out[14]:



6.4.5 Class Members

Class, or *static* members, belong to the class as a whole, and are shared between instances.

This is an object that keeps a count on how many have been created of it.

```
In [15]: class Counted:
             number_created = 0

             def __init__(self):
                 Counted.number_created += 1

             @classmethod
             def howMany(cls):
                 return cls.number_created

         Counted.howMany()  # 0
         x = Counted()
         Counted.howMany()  # 1
         z = [Counted() for x in range(5)]
         Counted.howMany()  # 6
```

Out[15]: 6

The data is shared among all the objects instantiated from that class. Note that in `__init__` we are not using `self.number_created` but the name of the class. The `howMany` function is not a method of a particular object. It's called on the class, not on the object. This is possible by using the `@classmethod` decorator.

6.5 Inheritance and Polymorphism

6.5.1 Object-based vs Object-Oriented

So far we have seen only object-based programming, not object-oriented programming.

Using Objects doesn't mean your code is object-oriented.

To understand object-oriented programming, we need to introduce **polymorphism** and **inheritance**.

6.5.2 Inheritance

- Inheritance is a mechanism that allows related classes to share code.
- Inheritance allows a program to reflect the *ontology* of kinds of thing in a program.

6.5.3 Ontology and inheritance

- A bird is a kind of animal
- An eagle is a kind of bird
- A starling is also a kind of bird
- All animals can be born and die
- Only birds can fly (Ish.)
- Only eagles hunt
- Only starlings flock

6.5.4 Inheritance in python

```
In [16]: class Animal:
             def beBorn(self):
                 print("I exist")
             def die(self):
                 print("Argh!")

         class Bird(Animal):
             def fly(self):
                 print("Whee!")

         class Eagle(Bird):
             def hunt(self):
                 print("I'm gonna eatcha!")

         class Starling(Bird):
             def flew(self):
                 print("I'm flying away!")

         Eagle().beBorn()
         Eagle().hunt()
```

```
I exist
I'm gonna eatcha!
```

6.5.5 Inheritance terminology

Here are two equivalents definition, one coming from C++ and another from Java: * A *derived class derives* from a *base class*. * A *subclass inherits* from a *superclass*.

These are different terms for the same thing. So, we can say:

- Eagle is a subclass of the Animal superclass.
- Animal is the base class of the Eagle derived class.

Another equivalent definition is using the synonym *child* / *parent* for *derived* / *base* class: * A *child class* extends a *parent class*.

6.5.6 Inheritance and constructors

To use implicitly constructors from a *superclass*, we can use `super` as shown below.

```
In [17]: class Animal:
             def __init__(self, age):
                 self.age = age
```

```
class Person(Animal):
    def __init__(self, age, name):
        super().__init__(age)
        self.name = name
```

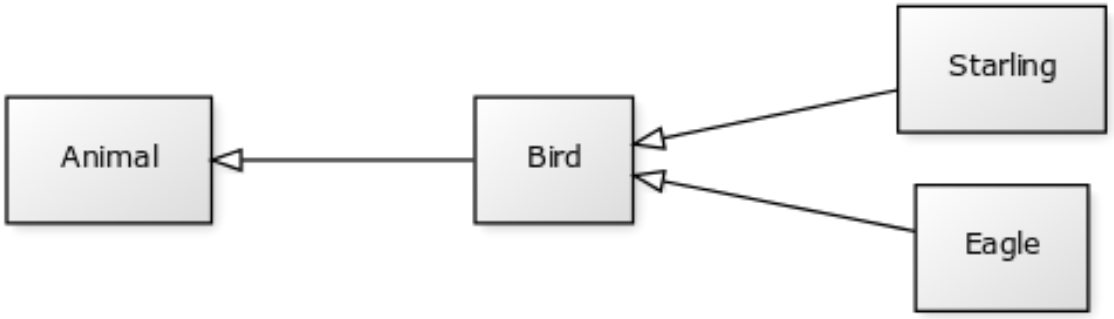
Read Raymond Hettinger's article about `super` to see various real examples.

6.5.7 Inheritance UML diagrams

UML shows inheritance with an open triangular arrow pointing from subclass to superclass.

```
In [18]: yuml("[Animal]^-[Bird],[Bird]^-[Eagle],[Bird]^-[Starling]%")
```

Out[18]:



6.5.8 Aggregation vs Inheritance

If one object *has* or *owns* one or more objects, this is *not* inheritance.

For example, the boids example we saw few weeks ago, could be organised as an overall Model, which it owns several Boids, and each Boid owns two 2-vectors, one for position and one for velocity.

Aggregation in UML

The Boids situation can be represented thus:

```
In [19]: yuml("[Model]<>-*>[Boid],[Boid]position++->[Vector],[Boid]velocity++->[Vector]%")
```

Out[19]:

The open diamond indicates **Aggregation**, the closed diamond **composition**. (A given boid might belong to multiple models, a given position vector is forever part of the corresponding Boid.)

The asterisk represents cardinality, a model may contain multiple Boids. This is a one to many relationship. Many to many relationship is shown with * on both sides.

Refactoring to inheritance

Smell: Repeated code between two classes which are both ontologically subtypes of something
Before:

```
In [20]: class Person:
             def __init__(self, age, job):
                 self.age = age
                 self.job = job
             def birthday(self):
                 self.age += 1

         class Pet:
             def __init__(self, age, owner):
                 self.age = age
                 self.owner = owner
             def birthday(self):
                 self.age += 1
```

After:

```
In [21]: class Animal:
             def __init__(self, age):
                 self.age = age
             def birthday(self):
                 self.age += 1

         class Person(Animal):
             def __init__(self, age, job):
                 self.job = job
                 super().__init__(age)

         class Pet(Animal):
             def __init__(self, age, owner):
                 self.owner = owner
                 super().__init__(age)
```

6.5.9 Polymorphism

```
In [22]: class Dog:
             def noise(self):
                 return "Bark"

         class Cat:
             def noise(self):
                 return "Miaow"

         class Pig:
             def noise(self):
                 return "Oink"

         class Cow:
             def noise(self):
                 return "Moo"
```

```
        animals = [Dog(), Dog(), Cat(), Pig(), Cow(), Cat()]
        for animal in animals:
            print(animal.noise())
```

```
Bark
Bark
Miaow
Oink
Moo
Miaow
```

This will print "Bark Bark Miaow Oink Moo Miaow"

If two classes support the same method, but it does different things for the two classes, then if an object is of an unknown class, calling the method will invoke the version for whatever class the instance is an instance of.

6.5.10 Polymorphism and Inheritance

Often, polymorphism uses multiple derived classes with a common base class. However, duck typing in Python means that all that is required is that the types support a common **Concept** (Such as iterable, or container, or, in this case, the Noisy concept.)

A common base class is used where there is a likely **default** that you want several of the derived classes to have.

```
In [23]: class Animal:
             def noise(self):
                 return "I don't make a noise."

         class Dog(Animal):
             def noise(self):
                 return "Bark"

         class Worm(Animal):
             pass

         class Poodle(Dog):
             pass

         animals = [Dog(), Worm(), Pig(), Cow(), Poodle()]
         for animal in animals:
             print(animal.noise())
```

```
Bark
I don't make a noise.
Oink
Moo
Bark
```

6.5.11 Undefined Functions and Polymorphism

In the above example, we put in a dummy noise for Animals that don't know what type they are.

Instead, we can explicitly deliberately leave this undefined, and we get a crash if we access an undefined method.

```
In [24]: class Animal:
             pass

         class Worm(Animal):
             pass

In [25]: Worm().noise() # Generates error
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-25-9a56606e40c2> in <module>
----> 1 Worm().noise() # Generates error


AttributeError: 'Worm' object has no attribute 'noise'
```

6.5.12 Refactoring to Polymorphism

Smell: a function uses a big set of `if` statements or a `case` statement to decide what to do:
Before:

```
In [26]: class Animal:
             def __init__(self, animal_kind):
                 self.animal_kind = animal_kind

             def noise(self):
                 if self.animal_kind == "Dog":
                     return "Bark"
                 elif self.animal_kind == "Cat":
                     return "Miaow"
                 elif self.animal_kind == "Cow":
                     return "Moo"
```

which is better replaced by the code above.

6.5.13 Interfaces and concepts

In C++, it is common to define classes which declare dummy methods, called "virtual" methods, which specify the methods which derived classes must implement. Classes which define these methods, but which cannot be instantiated into actual objects, are called "abstract base" classes or "interfaces".

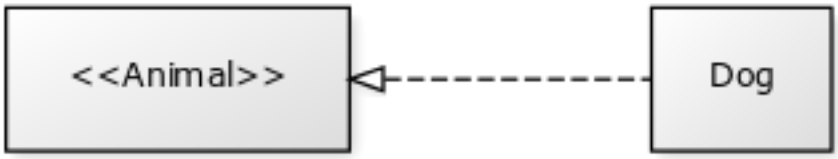
Python's Duck Typing approach means explicitly declaring these is unnesssary: any class concept which implements appropriately named methods will do. These as user-defined **concepts**, just as "iterable" or "container" are built-in Python concepts. A class is said to "implement an interface" or "satisfy a concept".

6.5.14 Interfaces in UML

Interfaces implementation (a common ancestor that doesn't do anything but defines methods to share) in UML is indicated thus:

```
In [27]: yuml("[<<Animal>>]^-.-[Dog]")
```

Out[27]:



6.5.15 Further UML

UML is a much larger diagram language than the aspects we've shown here.

- Message sequence charts show signals passing back and forth between objects (Web Sequence Diagrams).
- Entity Relationship Diagrams can be used to show more general relationships between things in a system.

Read more about UML on Martin Fowler's book about the topic.

6.6 Patterns

6.6.1 Class Complexity

We've seen that using object orientation can produce quite complex class structures, with classes owning each other, instantiating each other, and inheriting from each other.

There are lots of different ways to design things, and decisions to make.

- Should I inherit from this class, or own it as a member variable? ("is a" vs "has a")
- How much flexibility should I allow in this class's inner workings?
- Should I split this related functionality into multiple classes or keep it in one?

6.6.2 Design Patterns

Programmers have noticed that there are certain ways of arranging classes that work better than others.

These are called "design patterns".

They were first collected on one of the world's first Wikis, as the Portland Pattern Repository.

6.6.3 Reading a pattern

A description of a pattern in a book such as the Gang Of Four book (UCL Library) usually includes:

- **Intent** - what's the purpose
- **Motivation** - why you want to use it
- **Applicability** - when do you want to use it
- **Structure** - what does it look like (e.g., UML diagram)
- **Participants** - What are the different classes in it
- **Collaborations** - how they work together
- **Consequences** - What are the results and trade-offs
- **Implementation** - How is it implemented
- **Sample Code** - In practice.

6.6.4 Introducing Some Patterns

There are lots and lots of design patterns, and it's a great literature to get into to read about design questions in programming and learn from other people's experience.

We'll just show a few in this session:

- Factory Method
- Builder
- Strategy
- Model-View-Controller

6.6.5 Supporting code

```
In [1]: %matplotlib inline
        from unittest.mock import Mock
        import requests
        from IPython.display import Image, HTML

        def yuml(model):
            result=requests.get("http://yuml.me/diagram/boring/class/" + model)
            return Image(result.content)
```

6.7 Factory Pattern

Here's what the Gang of Four Book says about Factory Method:

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Applicability: Use the Factory method pattern when:

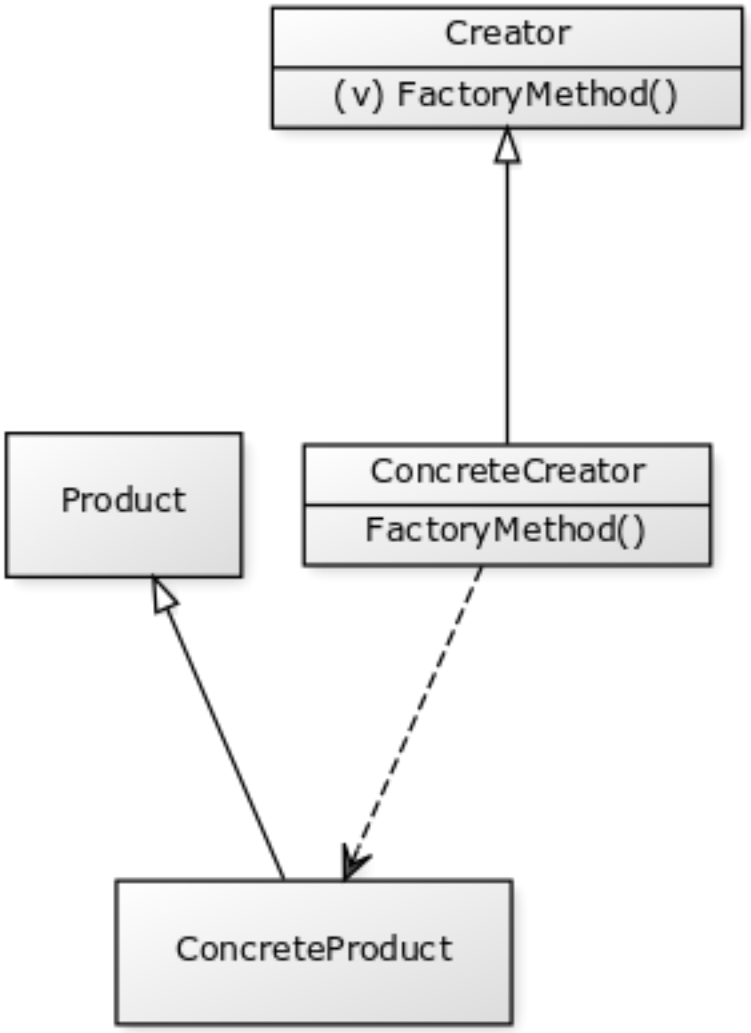
- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates

This is pretty hard to understand, so let's look at an example.

6.7.1 Factory UML

```
In [2]: yuml("[Product]^-[ConcreteProduct], "+
             "[Creator| (v) FactoryMethod()]^-[ConcreteCreator| FactoryMethod()], "+
             "[ConcreteCreator]-.->[ConcreteProduct]")
```

```
Out[2]:
```



6.7.2 Factory Example

An "agent based model" is one like the Boids model from last week: agents act and interact under certain rules. Complex phenomena can be described by simple agent behaviours.

```
In [3]: class AgentModel:
            def simulate(self):
                for agent in agents:
                    for target in agents:
                        agent.interact(target)
                agent.simulate()
```

6.7.3 Agent model constructor

This logic is common to many kinds of Agent based model (ABM), so we can imagine a common class for agent based models: the constructor could parse a configuration specifying how many agents of each type to create, their initial conditions and so on.

However, this common constructor doesn't know what kind of agent to create; as a common base, it could be a model of boids, or the agents could be remote agents on foreign servers, or they could even be physical hardware robots connected to the driving model over Wifi!

We need to defer the construction of the agents. We can do this with polymorphism: each derived class of the ABM can have an appropriate method to create its agents:

```
In [4]: class AgentModel:
            def __init__(self, config):
                self.agents = []
                for agent_config in config:
                    self.agents.append(self.create(**agent_config))
```

This is the *factory method* pattern: a common design solution to the need to defer the construction of daughter objects to a derived class. `self.create` is not defined here, but in each of the agents that inherits from `AgentModel`. Using polimorphism to get deffered behaviour on what you want to create.

6.7.4 Agent derived classes

The type that is created is different in the different derived classes:

```
In [5]: class BirdModel(AgentModel):
            def create(self, agent_config):
                return Boid(agent_config)
```

Agents are the base product, boids or robots are a ConcreteProduct.

```
In [6]: class WebAgentFactory(AgentModel):
            def __init__(self, url):
                self.url = url
                self.connection = AmazonCompute.connect(url)
                AgentModel.__init__(self)
            def create(self, agent_config):
                return OnlineAgent(agent_config, self.connection)
```

There is no need to define an explicit base interface for the "Agent" concept in Python: anything that responds to "simulate" and "interact" methods will do: this is our Agent concept.

6.7.5 Refactoring to Patterns

I personally have got into a terrible tangle trying to make base classes which somehow "promote" themselves into a derived class based on some code in the base class.

This is an example of an "Antipattern": like a Smell, this is a recognised Wrong Way of doing things.

What I should have written was a Creator with a FactoryMethod.

Consider the following code:

```
In [7]: class AgentModel:
            def simulate(self):
                for agent in agents:
                    for target in agents:
                        agent.interact(target)
                    agent.simulate()

        class BirdModel(AgentModel):
            def __init__(self, config):
                self.boids = []
                for boid_config in config:
```

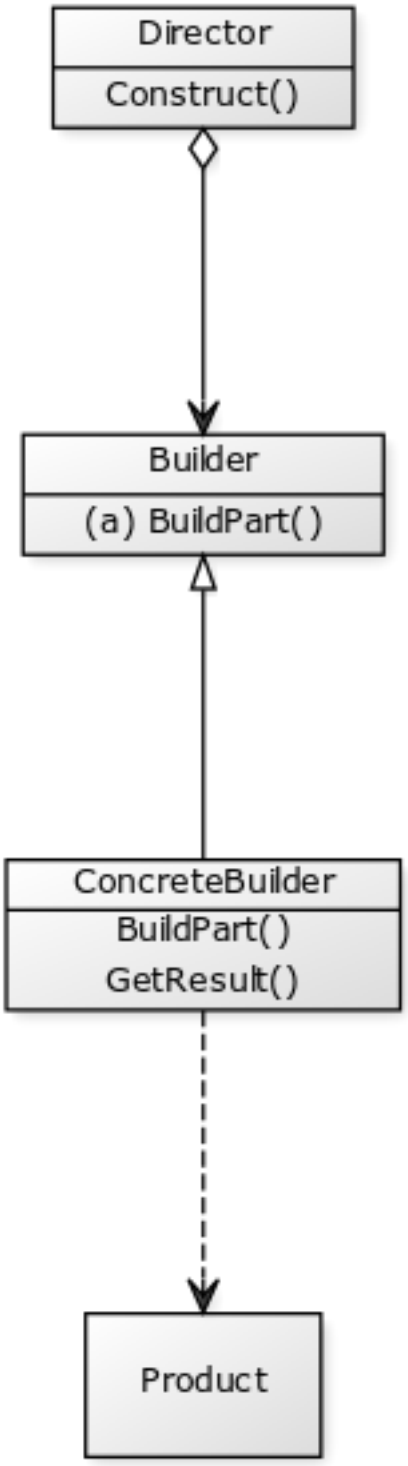
```
                self.boids.append(Boid(**boid_config))

        class WebAgentFactory(AgentModel):
            def __init__(self, url, config):
                self.url = url
                connection = AmazonCompute.connect(url)
                AgentModel.__init__(self)
                self.web_agents = []
                for agent_config in config:
                    self.web_agents.append(OnlineAgent(agent_config, connection))
```

The agent creation loop is almost identical in the two classes; so we can be sure we need to refactor it away; but the **type** that is created is different in the two cases, so this is the smell that we need a factory pattern.

6.8 Builder Pattern

Intent: Separate the steps for constructing a complex object from its final representation.

```
In [8]: yuml("[Director|Construct()]<>->[Builder| (a) BuildPart()],"+
             " [Builder]^-[ConcreteBuilder| BuildPart();GetResult() ],"+
             "[ConcreteBuilder]-.->[Product]")

Out[8]:
```



6.8.1 Builder example

Let's continue our Agent Based modelling example.

There's a lot more to defining a model than just adding agents of different kinds: we need to define boundary conditions, specify wind speed or light conditions.

We could define all of this for an imagined advanced Model with a very very long constructor, with lots of optional arguments:

```
In [9]: class Model:
            def __init__(self, xsize, ysize,
                         agent_count, wind_speed,
                         agent_sight_range, eagle_start_location):
                pass
```

6.8.2 Builder preferred to complex constructor

However, long constructors easily become very complicated. Instead, it can be cleaner to define a Builder for models. A builder is like a deferred factory: each step of the construction process is implemented as an individual method call, and the completed object is returned when the model is ready.

```
In [10]: Model = Mock() # Create a temporary mock so the example works!
```

```
In [11]: class ModelBuilder:
             def start_model(self):
                 self.model = Model()
                 self.model.xlim = None
                 self.model.ylim = None

             def set_bounds(self, xlim, ylim):
                 self.model.xlim = xlim
                 self.model.ylim = ylim

             def add_agent(self, xpos, ypos):
                 pass # Implementation here

             def finish(self):
                 self.validate()
                 return self.model

             def validate(self):
                 assert(self.model.xlim is not None)
                 # Check that the all the
                 # parameters that need to be set
                 # have indeed been set.
```

Inheritance of an Abstract Builder for multiple concrete builders could be used where there might be multiple ways to build models with the same set of calls to the builder: for example a version of the model builder yielding models which can be executed in parallel on a remote cluster.

6.8.3 Using a builder

```
In [12]: builder = ModelBuilder()
         builder.start_model()

         builder.set_bounds(500, 500)
         builder.add_agent(40, 40)
         builder.add_agent(400, 100)
```

```
model = builder.finish()
model.simulate()
```

```
Out[12]: <Mock name='mock().simulate()' id='139680497290320'>
```

6.8.4 Avoid staged construction without a builder.

We could, of course, just add all the building methods to the model itself, rather than having the model be yielded from a separate builder.

This is an antipattern that is often seen: a class whose `__init__` constructor alone is insufficient for it to be ready to use. A series of methods must be called, in the right order, in order for it to be ready to use.

This results in very fragile code: its hard to keep track of whether an object instance is "ready" or not. Use the builder pattern to keep deferred construction in control.

We might ask why we couldn't just use a validator in all of the methods that must follow the deferred constructors; to check they have been called. But we'd need to put these in *every* method of the class, whereas with a builder, we can validate only in the `finish` method.

6.9 Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

6.9.1 Strategy pattern example: sunspots

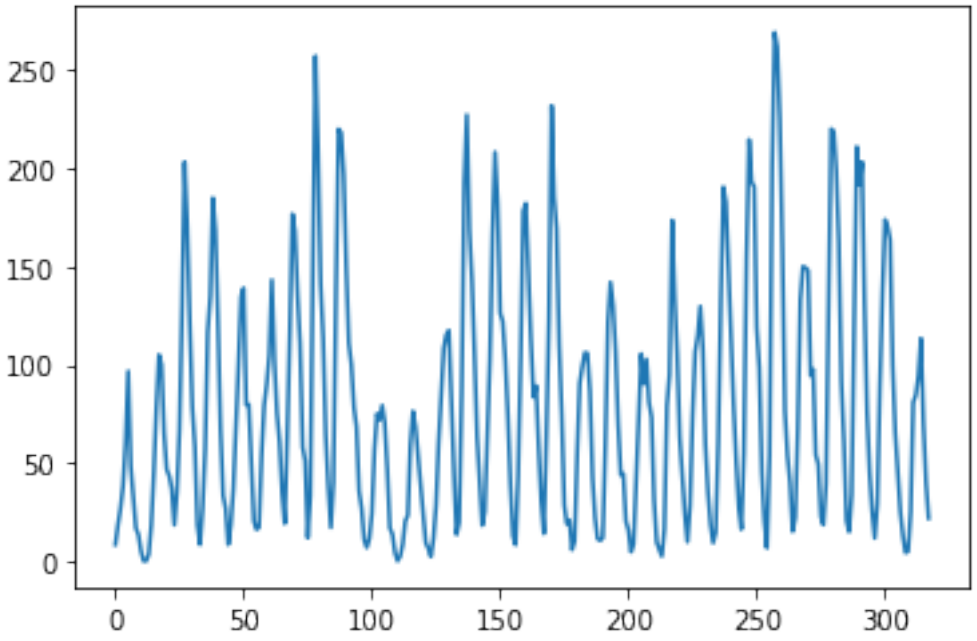
```
In [13]: import csv
         from datetime import datetime
         from io import StringIO
         import math

         import matplotlib.pyplot as plt
         from numpy import linspace,exp,log,sqrt, array
         from numpy.fft import rfft,fft,fftfreq
         from scipy.interpolate import UnivariateSpline
         from scipy.signal import lombscargle
         from scipy.integrate import cumtrapz
         import requests
```

Consider the sequence of sunspot observations:

```
In [14]: def load_sunspots():
             url_base = "http://www.quandl.com/api/v1/datasets/SIDC/SUNSPOTS_A.csv"
             x = requests.get(url_base,params={'trim_start':'1700-12-31',
                                               'trim_end':'2018-01-01',
                                               'sort_order':'asc'})
             data = csv.reader(StringIO(x.text))  # Convert requests
                                                  # result to look
                                                  # like a file buffer before
                                                  # reading with CSV
             next(data)  # Skip header row
             return [float(row[1]) for row in data]
```

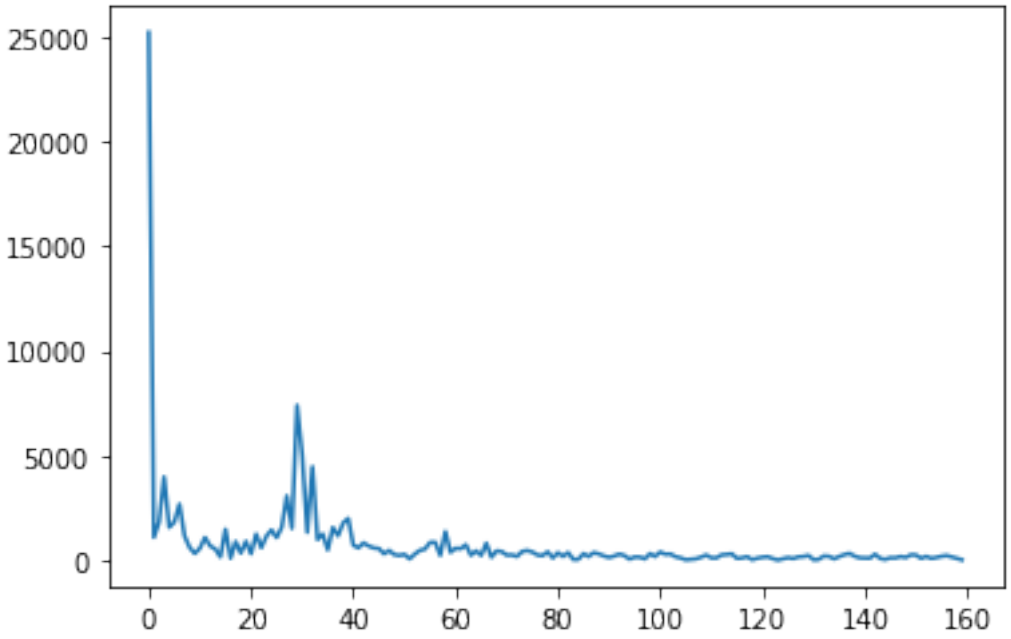
```
In [15]: spots = load_sunspots()
         plt.plot(spots)
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x7f09d5893e10>]
```



6.9.2 Sunspot cycle has periodicity

In [16]:
```
spectrum = rfft(spots)

plt.figure()
plt.plot(abs(spectrum))
plt.savefig('fixed.png')
```



6.9.3 Years are not constant length

There's a potential problem with this analysis however:

- Years are not constant length
- Leap years exist
- But, the Fast Fourier Transform assumes evenly spaced intervals

6.9.4 Strategy Pattern for Algorithms

6.9.5 Uneven time series

The Fast Fourier Transform cannot be applied to uneven time series.
We could:

- Ignore this problem, and assume the effect is small;
- Interpolate and resample to even times;
- Use a method which is robust to unevenly sampled series, such as LSSA;

We also want to find the period of the strongest periodic signal in the data, there are various different methods we could use for this also, such as integrating the fourier series by quadrature to find the mean frequency, or choosing the largest single value.

6.9.6 Too many classes!

We could implement a base class for our common code between the different approaches, and define derived classes for each different algorithmic approach. However, this has drawbacks:

- The constructors for each derived class will need arguments for all the numerical method's control parameters, such as the degree of spline for the interpolation method, the order of quadrature for integrators, and so on.
- Where we have multiple algorithmic choices to make (interpolator, periodogram, peak finder…) the number of derived classes would explode: `class SunspotAnalyzerSplineFFTTrapeziumNearMode` is a bit unwieldy.
- The algorithmic choices are not then available for other projects
- This design doesn't fit with a clean Ontology of "kinds of things": there's no Abstract Base for spectrogram generators…

6.9.7 Apply the strategy pattern:

- We implement each algorithm for generating a spectrum as its own Strategy class.
- They all implement a common interface
- Arguments to strategy constructor specify parameters of algorithms, such as spline degree
- One strategy instance for each algorithm is passed to the constructor for the overall analysis

First, we'll define a helper class for our time series.

```
In [17]: class Series:
             """Enhance NumPy N-d array with some helper functions for clarity"""
             def __init__(self, data):
                 self.data = array(data)
                 self.count = self.data.shape[0]
                 self.start = self.data[0, 0]
```

```
            self.end = self.data[-1, 0]
            self.range = self.end - self.start
            self.step = self.range / self.count
            self.times = self.data[:, 0]
            self.values = self.data[:, 1]
            self.plot_data = [self.times, self.values]
            self.inverse_plot_data = [1.0 / self.times[20:], self.values[20:]]
```

Then, our class which contains the analysis code, *except* the numerical methods

```
In [18]: class AnalyseSunspotData(object):
            def format_date(self, date):
                date_format="%Y-%m-%d"
                return datetime.strptime(date, date_format)

            def load_data(self):
                start_date_str = '1700-12-31'
                end_date_str = '2014-01-01'
                self.start_date = self.format_date(start_date_str)
                end_date = self.format_date(end_date_str)
                url_base = ("http://www.quandl.com/api/v1/datasets/" +
                            "SIDC/SUNSPOTS_A.csv")
                x = requests.get(url_base,params={'trim_start': start_date_str,
                                                  'trim_end': end_date_str,
                                                  'sort_order': 'asc'})
                secs_per_year = (datetime(2014, 1, 1) - datetime(2013, 1, 1)
                                 ).total_seconds()
                data = csv.reader(StringIO(x.text))  # Convert requests
                                                     # result to look
                                                     # like a file buffer before
                                                     # reading with CSV
                next(data) # Skip header row
                self.series = Series([[
                        (self.format_date(row[0]) - self.start_date
                        ).total_seconds()/secs_per_year,
                        float(row[1])] for row in data])

            def __init__(self, frequency_strategy):
                self.load_data()
                self.frequency_strategy = frequency_strategy

            def frequency_data(self):
                return self.frequency_strategy.transform(self.series)
```

Our existing simple fourier strategy

```
In [19]: class FourierNearestFrequencyStrategy:
            def transform(self, series):
                transformed = fft(series.values)[0:series.count//2]
                frequencies = fftfreq(series.count, series.step)[0:series.count//2]
                return Series(list(zip(frequencies, abs(transformed)/series.count)))
```

A strategy based on interpolation to a spline

```
In [20]: class FourierSplineFrequencyStrategy:
            def next_power_of_two(self, value):
```

```
            "Return the next power of 2 above value"
            return 2**(1 + int(log(value) / log(2)))

        def transform(self, series):
            spline = UnivariateSpline(series.times, series.values)
            # Linspace will give us *evenly* spaced points in the series
            fft_count = self.next_power_of_two(series.count)
            points = linspace(series.start,series.end,fft_count)
            regular_xs = [spline(point) for point in points]
            transformed = fft(regular_xs)[0:fft_count//2]
            frequencies = fftfreq(fft_count,
                                  series.range/fft_count)[0:fft_count//2]
            return Series(list(zip(frequencies, abs(transformed)/fft_count)))
```

A strategy using the Lomb-Scargle Periodogram

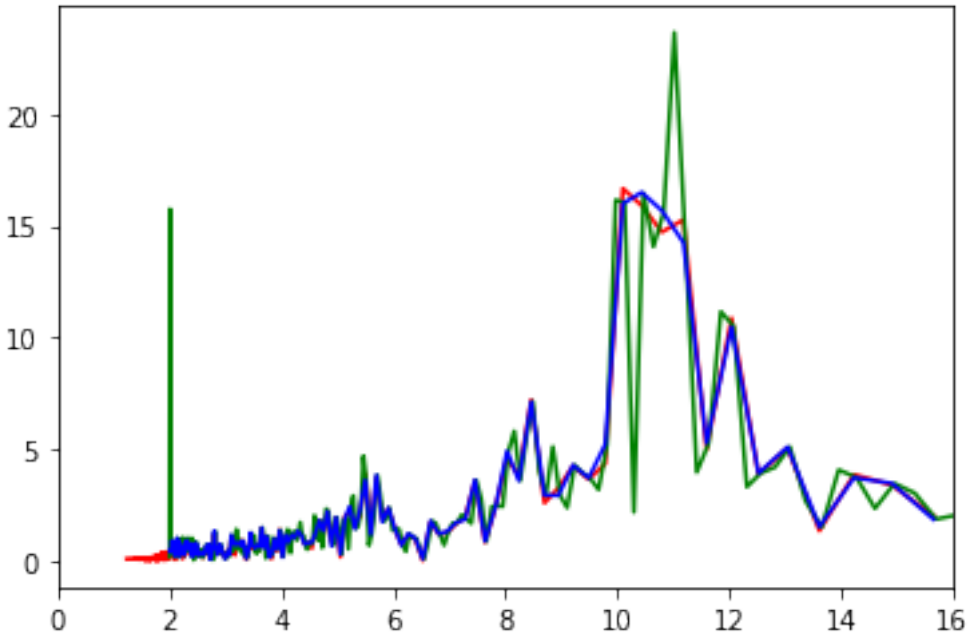
```
In [21]: class LombFrequencyStrategy:
             def transform(self, series):
                 frequencies = array(linspace(1.0 / series.range,
                                              0.5 / series.step,
                                              series.count))
                 result = lombscargle(series.times,
                                      series.values,
                                      2.0 * math.pi * frequencies)
                 return Series(list(zip(frequencies, sqrt(result / series.count))))
```

Define our concrete solutions with particular strategies

```
In [22]: fourier_model = AnalyseSunspotData(FourierSplineFrequencyStrategy())
         lomb_model = AnalyseSunspotData(LombFrequencyStrategy())
         nearest_model = AnalyseSunspotData(FourierNearestFrequencyStrategy())
```

Use these new tools to compare solutions

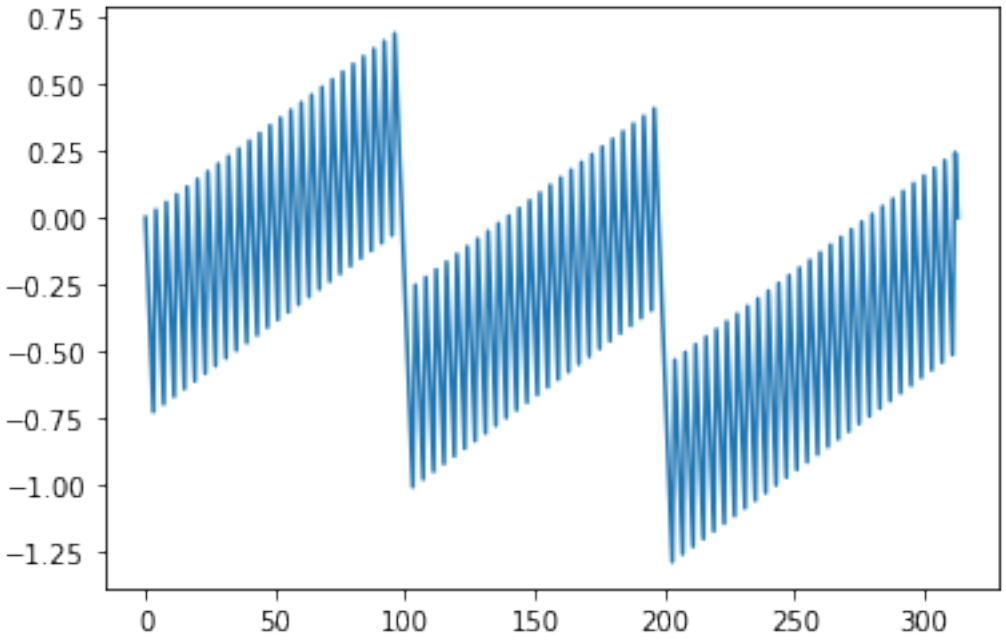
```
In [23]: comparison = fourier_model.frequency_data().inverse_plot_data + ['r']
         comparison += lomb_model.frequency_data().inverse_plot_data + ['g']
         comparison += nearest_model.frequency_data().inverse_plot_data + ['b']

In [24]: deviation = 365 * (fourier_model.series.times-linspace(
             fourier_model.series.start,
             fourier_model.series.end,
             fourier_model.series.count))

In [25]: plt.plot(*comparison)
         plt.xlim(0, 16)

Out[25]: (0, 16)
```



6.9.8 Results: Deviation of year length from average

```
In [26]: plt.plot(deviation)
```

```
Out[26]: [<matplotlib.lines.Line2D at 0x7f09d50df4d0>]
```



6.10 Model-View-Controller

6.10.1 Separate graphics from science!

Whenever we are coding a simulation or model we want to:

- Implement the maths of the model
- Visualise, plot, or print out what is going on.

We often see scientific programs where the code which is used to display what is happening is mixed up with the mathematics of the analysis. This is hard to understand.

We can do better by separating the `Model` from the `View`, and using a "`Controller`" to manage them.

6.10.2 Model

This is where we describe our internal logic, rules, etc.

```
In [27]: import numpy as np

         class Model:
             def __init__(self):
                 self.positions = np.random.rand(100, 2)
                 self.speeds = (np.random.rand(100, 2) +
                                np.array([-0.5, -0.5])[np.newaxis, :])
                 self.deltat = 0.01

             def simulation_step(self):
                 self.positions += self.speeds * self.deltat

             def agent_locations(self):
                 return self.positions
```

6.10.3 View

This is where we describe what the user sees of our Model, what's displayed. You may have different type of visualisation (*e.g.*, on one type of projection, a 3D view, a surface view, …) which can be implemented in different *view* classes.

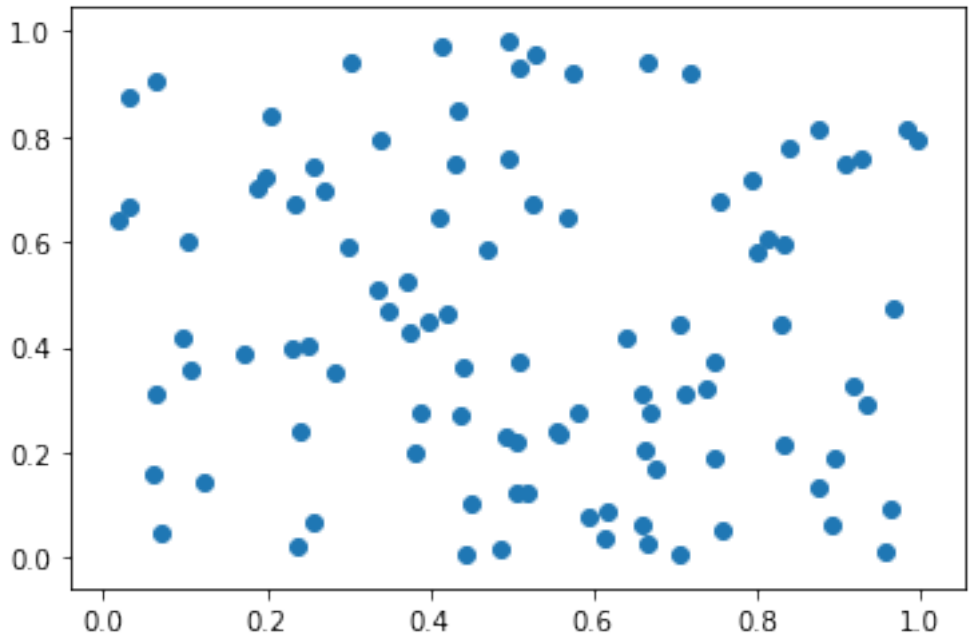
```
In [28]: class View:
             def __init__(self, model):
                 from matplotlib import pyplot as plt
                 self.figure = plt.figure()
                 axes = plt.axes()
                 self.model = model
                 self.scatter = axes.scatter(
                         model.agent_locations()[:, 0],
                         model.agent_locations()[:, 1])

             def update(self):
                 self.scatter.set_offsets(
                     self.model.agent_locations())
```

6.10.4 Controller

This is the class that tells the view that the models has changed and updates the model with any change the user has input through the view.

```
In [29]: class Controller:
             def __init__(self):
                 self.model = Model() # Or use Builder
                 self.view = View(self.model)
                 def animate(frame_number):
                     self.model.simulation_step()
                     self.view.update()
                 self.animator = animate

             def go(self):
                 from matplotlib import animation
                 anim = animation.FuncAnimation(self.view.figure,
                                                self.animator,
                                                frames=200,
                                                interval=50)
                 return anim.to_jshtml()
```

```
In [30]: contl = Controller()
```




```
In [31]: HTML(contl.go())
```

```
Out[31]: <IPython.core.display.HTML object>
```

6.10.5 Other resources

- Course on design patterns and Advanced design patterns with Python at Lynda.com.
- A collection of design patterns and idioms in Python.
- Head First Desssign Patterns (Available online at UCL) - based on Java (with online course at Lynda.com).
- Design Pattern for Dummies.

6.11 Exercise: Refactoring The Bad Boids

6.11.1 Bad_Boids

We have written some *very bad* code implementing our Boids flocking example.

Here's the Github link.

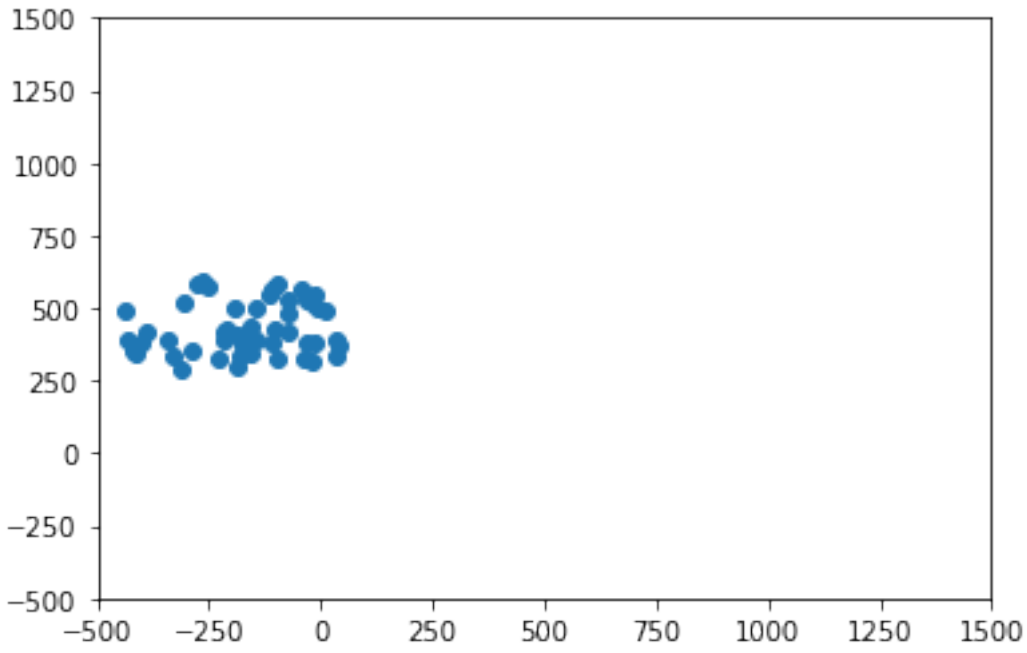
Please fork it on GitHub, and clone your fork.

```
git clone      git@github.com:yourname/bad-boids.git
# OR git clone https://github.com/yourname/bad-boids.git
```

For the Exercise, you should start from the GitHub repository, but here's our terrible code:

```
In [1]: """
        A deliberately bad implementation of
        [Boids](http://dl.acm.org/citation.cfm?doid=37401.37406)
        for use as an exercise on refactoring.
        """

        from matplotlib import pyplot as plt
        from matplotlib import animation

        import random

        # Deliberately terrible code for teaching purposes

        boids_x=[random.uniform(-450,50.0) for x in range(50)]
        boids_y=[random.uniform(300.0,600.0) for x in range(50)]
        boid_x_velocities=[random.uniform(0,10.0) for x in range(50)]
        boid_y_velocities=[random.uniform(-20.0,20.0) for x in range(50)]
        boids=(boids_x,boids_y,boid_x_velocities,boid_y_velocities)

        def update_boids(boids):
            xs,ys,xvs,yvs=boids
            # Fly towards the middle
            for i in range(len(xs)):
                for j in range(len(xs)):
                    xvs[i]=xvs[i]+(xs[j]-xs[i])*0.01/len(xs)
            for i in range(len(xs)):
                for j in range(len(xs)):
                    yvs[i]=yvs[i]+(ys[j]-ys[i])*0.01/len(xs)
            # Fly away from nearby boids
            for i in range(len(xs)):
                for j in range(len(xs)):
                    if (xs[j]-xs[i])**2 + (ys[j]-ys[i])**2 < 100:
                        xvs[i]=xvs[i]+(xs[i]-xs[j])
                        yvs[i]=yvs[i]+(ys[i]-ys[j])
            # Try to match speed with nearby boids
            for i in range(len(xs)):
                for j in range(len(xs)):
                    if (xs[j]-xs[i])**2 + (ys[j]-ys[i])**2 < 10000:
                        xvs[i]=xvs[i]+(xvs[j]-xvs[i])*0.125/len(xs)
                        yvs[i]=yvs[i]+(yvs[j]-yvs[i])*0.125/len(xs)
            # Move according to velocities
            for i in range(len(xs)):
```

```
            xs[i]=xs[i]+xvs[i]
            ys[i]=ys[i]+yvs[i]


figure=plt.figure()
axes=plt.axes(xlim=(-500,1500), ylim=(-500,1500))
scatter=axes.scatter(boids[0],boids[1])

def animate(frame):
    update_boids(boids)
    scatter.set_offsets(list(zip(boids[0],boids[1])))


anim = animation.FuncAnimation(figure, animate,
                               frames=200, interval=50)
```



If you go into your folder and run the code:

```
cd bad_boids
python boids.py
```

You should be able to see some birds flying around, and then disappearing as they leave the window.

```
In [2]: from IPython.display import HTML
        HTML(anim.to_jshtml())
```

```
Out[2]: <IPython.core.display.HTML object>
```

6.11.2 Your Task

Transform bad_boids **gradually** into better code, while making sure it still works, using a Refactoring approach.

6.11.3 A regression test

First, have a look at the regression test we made.

To create it, we saved out the before and after state for one iteration of some boids, using ipython:

```
import yaml
import boids
from copy import deepcopy

before = deepcopy(boids.boids)
boids.update_boids(boids.boids)
after = boids.boids
fixture = {"before": before, "after": after}
fixture_file = open("fixture.yml", 'w')
fixture_file.write(yaml.dump(fixture))
fixture_file.close()
```

6.11.4 Invoking the test

Then, I used the fixture file to define the test:

```
from boids import update_boids
from nose.tools import assert_almost_equal
import os
import yaml

def test_bad_boids_regression():
    regression_data = yaml.safe_load(open(os.path.join(os.path.dirname(__file__),'fixture.yml')))
    boid_data = regression_data["before"]
    update_boids(boid_data)
    for after, before in zip(regression_data["after"], boid_data):
        for after_value, before_value in zip(after, before):
            assert_almost_equal(after_value, before_value, delta=0.01)
```

6.11.5 Make the regression test fail

Check the tests pass:

```
pytest
```

Edit the file to make the test fail, see the fail, then reset it:

```
git checkout boids.py
```

6.11.6 Start Refactoring

Look at the code, consider the list of refactorings, and make changes.

Each time, do a git commit on your fork, and write a commit message explaining the refactoring you did.

Try to keep the changes as small as possible.

If your refactoring creates any units, (functions, modules, or classes) **write a unit test** for the unit: it is a good idea to get away from regression testing as soon as you can.

Chapter 7

Advanced Python Programming

… or, how to avoid repeating yourself.

7.1 Avoid Boiler-Plate

Code can often be annoyingly full of "boiler-plate" code: characters you don't really want to have to type.

Not only is this tedious, it's also time-consuming and dangerous: unnecessary code is an unnecessary potential place for mistakes.

There are two important phrases in software design that we've spoken of before in this context:

> Once And Only Once
>
> Don't Repeat Yourself (DRY)

All concepts, ideas, or instructions should be in the program in just one place. Every line in the program should say something useful and important.

We refer to code that respects this principle as DRY code.

In this chapter, we'll look at some techniques that can enable us to refactor away repetitive code.

Since in many of these places, the techniques will involve working with functions as if they were variables, we'll learn some **functional** programming. We'll also learn more about the innards of how Python implements classes.

We'll also think about how to write programs that *generate* the more verbose, repetitive program we could otherwise write. We call this **metaprogramming**.

7.2 Functional programming

We have previously seen the object-oriented style of programming, and how to organise our code according to it using objects, classes and inheritance. While widely-adopted and very useful, this is not the only way of writing code. The *functional paradigm*, as the name suggests, emphasises functions as building blocks of programs.

Understanding to think in a functional programming style is almost as important as object orientation for building DRY, clear scientific software, and is just as conceptually difficult. However, being aware of different paradigms and styles gives you access to more techniques that you can use to write, structure and reason about your code.

7.2.1 Functions within functions

Programs are composed of functions: they take data in (which we call *parameters* or *arguments*) and send data out (through `return` statements).

A conceptual trick which is often used by computer scientists to teach the core idea of functional programming is this: to write a program, in theory, you only ever need functions with **one** argument, even when you think you need two or more. Why?

Let's define a program to add two numbers:

```
In [1]: def add(a, b):
            return a + b

        add(5, 6)
```

```
Out[1]: 11
```

How could we do this, in a fictional version of Python which only defined functions of one argument? In order to understand this, we'll have to understand several of the concepts of functional programming. Let's start with a program which just adds five to something:

```
In [2]: def add_five(a):
            return a + 5

        add_five(6)
```

```
Out[2]: 11
```

OK, we could define lots of these, one for each number we want to add. But that would be infinitely repetitive. So, let's try to metaprogram that: we want a function which returns these add_N() functions.

Let's start with the easy case: a function which returns a function which adds 5 to something:

```
In [3]: def generate_five_adder():
            def _five_adder(a):
                return a + 5
            return _five_adder

        coolfunction = generate_five_adder()
        coolfunction(7)
```

```
Out[3]: 12
```

OK, so what happened there? Well, we defined a function **inside** the other function. We can always do that:

```
In [4]: def thirty_function():
            def times_three(a):
                return a * 3
            def add_seven(a):
                return a + 7
            return times_three(add_seven(3))

        thirty_function()
```

```
Out[4]: 30
```

When we do this, the functions enclosed inside the outer function are **local** functions, and can't be seen outside:

```
In [5]: add_seven
```

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

NameError                                 Traceback (most recent call last)

<ipython-input-5-6fa1bcd39365> in <module>
----> 1 add_seven


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

There's not really much of a difference between functions and other variables in python. A function is just a variable which can have () put after it to call the code!

```
In [6]: print(thirty_function)
```

```
<function thirty_function at 0x7f96601e50e0>
```

```
In [7]: x = [thirty_function, add_five, add]
```

```
In [8]: for fun in x:
            print(fun)
```

```
<function thirty_function at 0x7f96601e50e0>
<function add_five at 0x7f9660230c20>
<function add at 0x7f9660230a70>
```

And we know that one of the things we can do with a variable is `return` it. So we can return a function, and then call it outside:

```
In [9]: def deferred_greeting():
            def greet():
                print("Hello")
            return greet

        friendlyfunction = deferred_greeting()
```

```
In [10]: # Do something else
         print("Just passing the time...")
```

```
Just passing the time…
```

```
In [11]: # OK, Go!
         friendlyfunction()
```

```
Hello
```

So now, to finish this, we just need to return a function to add an arbitrary amount:

```
In [12]: def generate_adder(increment):
             def _adder(a):
                 return a + increment
             return _adder

         add_3 = generate_adder(3)
```

```
In [13]: add_3(9)
```

```
Out[13]: 12
```

We can make this even prettier: let's make another variable pointing to our define_adder() function:

```
In [14]: add = generate_adder
```

And now we can do the real magic:

```
In [15]: add(8)(5)
```

```
Out[15]: 13
```

In summary, we have started with a function that takes two arguments (`add(a, b)`) and replaced it with a new function (`add(a)(b)`). This new function takes a single argument, and returns a function that itself takes the second argument.

This may seem like an overly complicated process - and, in some cases, it is! However, this pattern of functions that return functions (or even take them as arguments!) can be very useful. In fact, it is the basis of decorators, a Python feature that we will discuss more in this chapter [notebook].

7.2.2 Closures

You may have noticed something a bit weird:

In the definition of `generate_adder`, `increment` is a local variable. It should have gone out of scope and died at the end of the definition. How can the amount the returned adder function is adding still be kept?

This is called a **closure**. In Python, whenever a function definition references a variable in the surrounding scope, it is preserved within the function definition.

You can close over global module variables as well:

```
In [16]: name = "Eric"

         def greet():
             print("Hello, ", name)

         greet()
```

```
Hello,  Eric
```

And note that the closure stores a reference to the variable in the surrounding scope: ("Late Binding")

```
In [17]: name = "John"

         greet()
```

```
Hello,  John
```

7.2.3 Map and Reduce

We often want to apply a function to each variable in an array, to return a new array. We can do this with a list comprehension:

```
In [18]: numbers = range(10)

         [add_five(i) for i in numbers]
```

```
Out[18]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

But this is sufficiently common that there's a quick built-in:

```
In [19]: list(map(add_five, numbers))

Out[19]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

This **map** operation is really important conceptually when understanding efficient parallel programming: different computers can apply the *mapped* function to their input at the same time. We call this Single Program, Multiple Data (SPMD). **map** is half of the **map-reduce** functional programming paradigm which is key to the efficient operation of much of today's "data science" explosion.

Let's continue our functional programming mind-stretch by looking at **reduce** operations.

We very often want to loop with some kind of accumulator (an intermediate result that we update), such as when finding a mean:

```
In [20]: def summer(data):
             total = 0.0

             for x in data:
                 total += x

             return total

In [21]: summer(range(10))

Out[21]: 45.0
```

or finding a maximum:

```
In [22]: import sys

         def my_max(data):
             # Start with the smallest possible number
             highest = sys.float_info.min

             for x in data:
                 if x > highest:
                     highest = x

             return highest

In [23]: my_max([2, 5, 10, -11, -5])

Out[23]: 10
```

These operations, where we have some variable which is building up a result, and the result is updated with some operation, can be gathered together as a functional program, taking in (as an argument) the operation to be used to combine results:

```
In [24]: def accumulate(initial, operation, data):
             accumulator = initial
             for x in data:
                 accumulator = operation(accumulator, x)
             return accumulator

         def my_sum(data):
             def _add(a, b):
                 return a + b
             return accumulate(0, _add, data)
```

```
In [25]: my_sum(range(5))

Out[25]: 10

In [26]: def bigger(a, b):
             if b > a:
                 return b
             return a

         def my_max(data):
             return accumulate(sys.float_info.min, bigger, data)

         my_max([2, 5, 10, -11, -5])

Out[26]: 10
```

Anyway, this accumulate-under-an-operation process is so fundamental to computing that it's usually in standard libraries for languages which allow functional programming:

```
In [27]: from functools import reduce

         def my_max(data):
             return reduce(bigger, data, sys.float_info.min)

         my_max([2, 5, 10, -11, -5])

Out[27]: 10
```

Efficient map-reduce

Now, because these operations, `bigger` and `_add`, are such that e.g. (a+b)+c = a+(b+c) , i.e. they are **associative**, we could apply our accumulation to the left half and the right half of the array, each on a different computer, and then combine the two halves:

$1 + 2 + 3 + 4 = (1 + 2) + (3 + 4)$

Indeed, with a bigger array, we can divide-and-conquer more times:

$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = ((1 + 2) + (3 + 4)) + ((5 + 6) + (7 + 8))$

So with enough parallel computers, we could do this operation on eight numbers in three steps: first, we use four computers to do one each of the pairwise adds.

Then, we use two computers to add the four totals.

Then, we use one of the computers to do the final add of the two last numbers.

You might be able to do the maths to see that with an N element list, the number of such steps is proportional to the logarithm of N.

We say that with enough computers, reduction operations are O(ln N)

This course isn't an introduction to algorithms, but we'll talk more about this O() notation when we think about programming for performance.

7.2.4 Lambda Functions

When doing functional programming, we often want to be able to define a function on the fly:

```
In [28]: def most_Cs_in_any_sequence(sequences):

             def count_Cs(sequence):
                 return sequence.count('C')

             counts = map(count_Cs, sequences)
```

```
        return max(counts)


    def most_Gs_in_any_sequence(sequences):
        return max(map(lambda sequence: sequence.count('G'), sequences))


    data = [
        "CGTA",
        "CGGGTAAACG",
        "GATTACA"
    ]

    most_Gs_in_any_sequence(data)
```

```
Out[28]: 4
```

The syntax here means that these two definitions are identical:

```
In [29]: func_name = lambda a, b, c: a + b + c

         def func_name(a, b, c):
             return a + b + c
```

The **lambda** keyword defines an "anonymous" function.

```
In [30]: def most_of_given_base_in_any_sequence(sequences, base):
             return max(map(lambda sequence: sequence.count(base), sequences))

         most_of_given_base_in_any_sequence(data, 'A')
```

```
Out[30]: 3
```

The above fragment defined a lambda function as a **closure** over `base`. If you understood that, you've got it!

To double all elements in an array:

```
In [31]: data = range(10)
         list(map(lambda x: 2*x, data))
```

```
Out[31]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
In [32]: [2*x for x in data]
```

```
Out[32]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Similarly, to find the maximum value in a sequence:

```
In [33]: def my_max(data):
             return reduce(lambda a, b: a if a > b else b, data, sys.float_info.min)

         my_max([2, 5, 10, -11, -5])
```

```
Out[33]: 10
```

7.2.5 Using functional programming for numerical methods

Probably the most common use in research computing for functional programming is the application of a numerical method to a function.

Consider this example which uses the `newton` function from SciPy, a root-finding function implementing the Newton-Raphson method. The arguments we pass to `newton` are the function whose roots we want to find, and a starting point to search from.

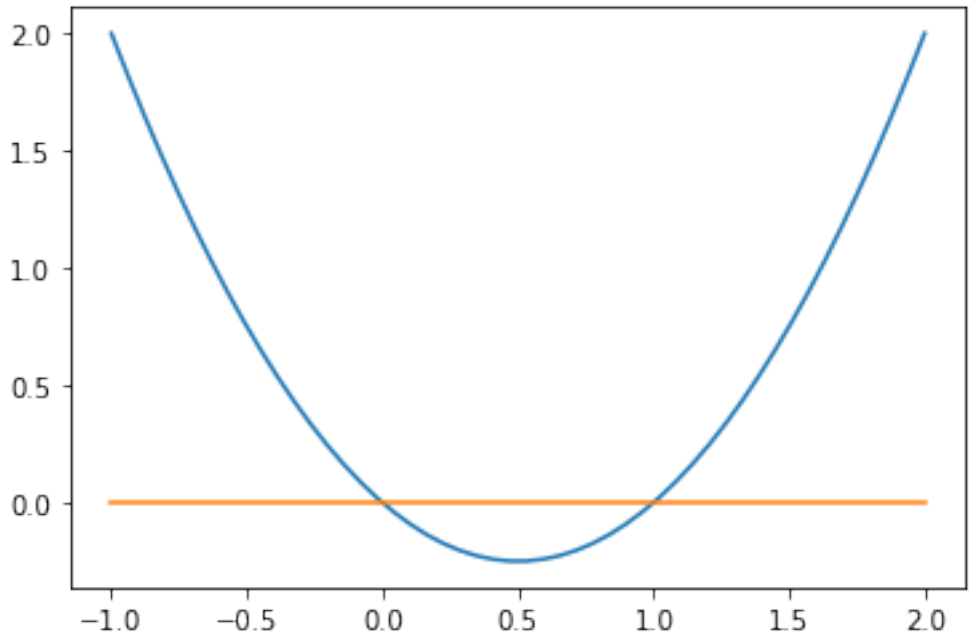
We will be using this to find the roots of the function $f(x) = x^2 - x$.

```
In [34]: %matplotlib inline
```

```
In [35]: from scipy.optimize import newton
         from numpy import linspace, zeros
         from matplotlib import pyplot as plt

         solve_me = lambda x: x**2 - x

         for x0 in [2, 0.2]:
             answer = newton(solve_me, x0)
             print("Starting from {}, the root I found is {}".format(x0, answer))

         xs = linspace(-1, 2, 50)
         solved = [xs, list(map(solve_me, xs)), xs, zeros(len(xs))]

         plt.plot(*solved)
```

```
Starting from 2, the root I found is 1.0
Starting from 0.2, the root I found is -3.441905100203782e-21
```

```
Out[35]: [<matplotlib.lines.Line2D at 0x7f9634d63f50>,
          <matplotlib.lines.Line2D at 0x7f9634d75210>]
```



Sometimes such tools return another function, for example the derivative of their input function. This is what a naive implementation of that could look like:

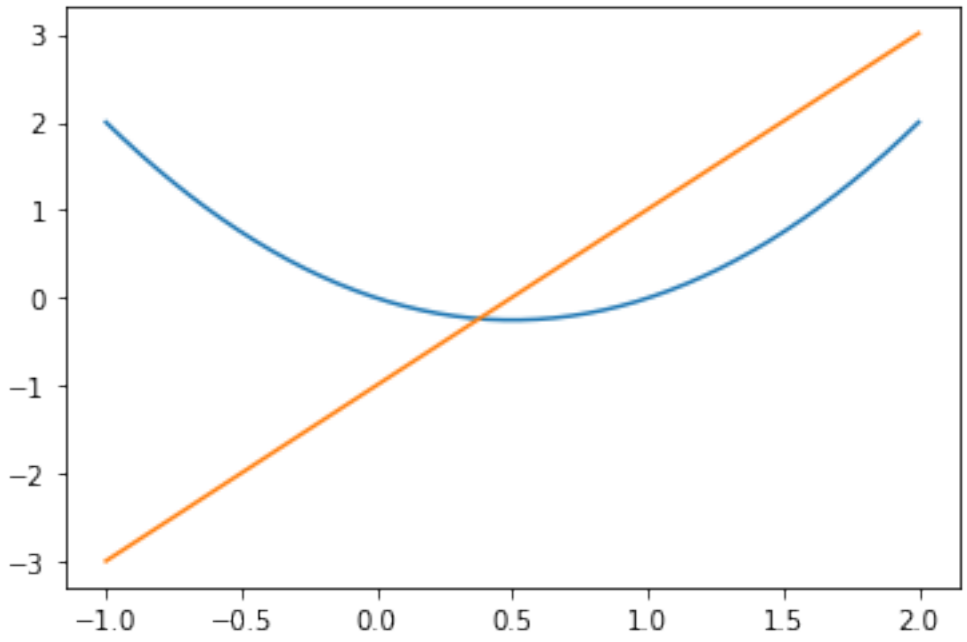
```
In [36]: def derivative_simple(func, eps, at):
             return (func(at + eps) - func(at)) / eps

In [37]: def derivative(func, eps):

             def _func_derived(x):
                 return (func(x + eps) - func(x)) / eps

             return _func_derived

         straight = derivative(solve_me, 0.01)
```

The derivative of `solve_me` is $f'(x) = 2x - 1$, which represents a straight line. We can verify that our computations are correct, i.e. that the returned function `straight` matches $f'(x)$, by checking the value of `straight` at some x:

```
In [38]: straight(3)

Out[38]: 5.00999999999987
```

or by plotting it:

```
In [39]: derived = (
             xs, list(map(solve_me, xs)),
             xs, list(map(derivative(solve_me, 0.01), xs))
         )
         plt.plot(*derived)
         print(newton(derivative(solve_me, 0.01), 0))

0.495000000000001
```



Of course, coding your own numerical methods is bad, because the implementations you develop are likely to be less efficient, less accurate and more error-prone than what you can find in existing established libraries.

For example, the above definition could be replaced by:

```
In [40]: import scipy.misc

         def derivative(func):
             def _func_derived(x):
                 return scipy.misc.derivative(func, x)
             return _func_derived

         newton(derivative(solve_me), 0)
```

```
Out[40]: 0.5
```

If you've done a moderate amount of calculus, then you'll find similarities between functional programming in computer science and Functionals in the calculus of variations.

7.3 Iterators and Generators

In Python, anything which can be iterated over is called an iterable:

```
In [1]: bowl = {
             "apple": 5,
             "banana": 3,
             "orange": 7
         }

         for fruit in bowl:
             print(fruit.upper())
```

```
APPLE
BANANA
ORANGE
```

Surprisingly often, we want to iterate over something that takes a moderately large amount of memory to store - for example, our map images in the green-graph example.

Our green-graph example involved making an array of all the maps between London and Birmingham. This kept them all in memory *at the same time*: first we downloaded all the maps, then we counted the green pixels in each of them.

This would NOT work if we used more points: eventually, we would run out of memory. We need to use a **generator** instead. This chapter will look at iterators and generators in more detail: how they work, when to use them, how to create our own.

7.3.1 Iterators

Consider the basic python `range` function:

```
In [2]: range(10)
```

```
Out[2]: range(0, 10)
```

```
In [3]: total = 0
        for x in range(int(1e6)):
            total += x

        total
```

```
Out[3]: 499999500000
```

In order to avoid allocating a million integers, `range` actually uses an **iterator**.
We don't actually need a million integers *at once*, just each integer *in turn* up to a million.
Because we can get an iterator from it, we say that a range is an **iterable**.
So we can `for`-loop over it:

```
In [4]: for i in range(3):
            print(i)
```

```
0
1
2
```

There are two important Python built-in functions for working with iterables. First is `iter`, which lets us create an iterator from any iterable object.

```
In [5]: a = iter(range(3))
```

Once we have an iterator object, we can pass it to the `next` function. This moves the iterator forward, and gives us its next element:

```
In [6]: next(a)
```

```
Out[6]: 0
```

```
In [7]: next(a)
```

```
Out[7]: 1
```

```
In [8]: next(a)
```

```
Out[8]: 2
```

When we are out of elements, a `StopIteration` exception is raised:

```
In [9]: next(a)
```

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

        StopIteration                             Traceback (most recent call last)

        <ipython-input-9-15841f3f11d4> in <module>
    ----> 1 next(a)


        StopIteration:
```

This tells Python that the iteration is over. For example, if we are in a `for i in range(3)` loop, this lets us know when we should exit the loop.
We can turn an iterable or iterator into a list with the `list` constructor function:

```
In [10]: list(range(5))
```

```
Out[10]: [0, 1, 2, 3, 4]
```

7.3.2 Defining Our Own Iterable

When we write `next(a)`, under the hood Python tries to call the `__next__()` method of `a`. Similarly, `iter(a)` calls `a.__iter__()`.

We can make our own iterators by defining *classes* that can be used with the `next()` and `iter()` functions: this is the **iterator protocol**.

For each of the *concepts* in Python, like sequence, container, iterable, the language defines a *protocol*, a set of methods a class must implement, in order to be treated as a member of that concept.

To define an iterator, the methods that must be supported are `__next__()` and `__iter__()`.

`__next__()` must update the iterator.

We'll see why we need to define `__iter__` in a moment.

Here is an example of defining a custom iterator class:

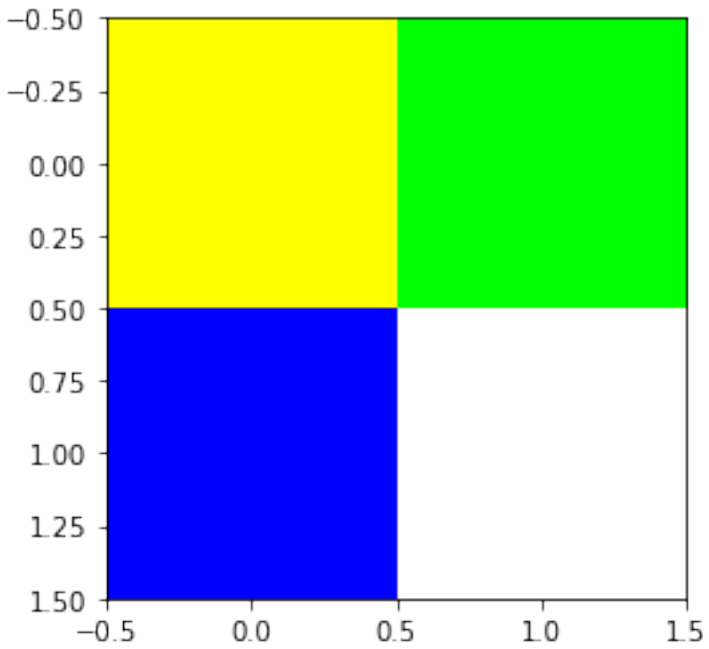
```
In [11]: class fib_iterator:
             """An iterator over part of the Fibonacci sequence."""

             def __init__(self, limit, seed1=1, seed2=1):
                 self.limit = limit
                 self.previous = seed1
                 self.current = seed2

             def __iter__(self):
                 return self

             def __next__(self):
                 (self.previous, self.current) = (self.current, self.previous + self.current)
                 self.limit -= 1
                 if self.limit < 0:
                     raise StopIteration()
                 return self.current

In [12]: x = fib_iterator(5)

In [13]: next(x)

Out[13]: 2

In [14]: next(x)

Out[14]: 3

In [15]: next(x)

Out[15]: 5

In [16]: next(x)

Out[16]: 8

In [17]: for x in fib_iterator(5):
             print(x)

2
3
5
8
13


In [18]: sum(fib_iterator(1000))

Out[18]: 297924218508143360336882819981631900915673130543819759032778173440536722190488904520034508163
```

7.3.3 A shortcut to iterables: the `__iter__` method

In fact, we don't always have to define both `__iter__` and `__next__`!

If, to be iterated over, a class just wants to behave as if it were some other iterable, you can just implement `__iter__` and return `iter(some_other_iterable)`, without implementing `next`. For example, an image class might want to implement some metadata, but behave just as if it were just a 1-d pixel array when being iterated:

```
In [19]: from numpy import array
         from matplotlib import pyplot as plt


         class MyImage(object):
             def __init__(self, pixels):
                 self.pixels = array(pixels, dtype='uint8')
                 self.channels = self.pixels.shape[2]

             def __iter__(self):
                 # return an iterator over just the pixel values
                 return iter(self.pixels.reshape(-1, self.channels))

             def show(self):
                 plt.imshow(self.pixels, interpolation="None")


         x = [[[255, 255, 0], [0, 255, 0]], [[0, 0, 255], [255, 255, 255]]]
         image = MyImage(x)

In [20]: %matplotlib inline
         image.show()
```



```
In [21]: image.channels

Out[21]: 3

In [22]: from webcolors import rgb_to_name
         for pixel in image:
             print(rgb_to_name(pixel))

yellow
lime
blue
white
```

See how we used `image` in a `for` loop, even though it doesn't satisfy the iterator protocol (we didn't define both `__iter__` and `__next__` for it)?

The key here is that we can use any *iterable* object (like `image`) in a `for` expression, not just iterators! Internally, Python will create an iterator from the iterable (by calling its `__iter__` method), but this means we don't need to define a `__next__` method explicitly.

The *iterator* protocol is to implement both `__iter__` and `__next__`, while the *iterable* protocol is to implement `__iter__` and return an iterator.

7.3.4 Generators

There's a fair amount of "boiler-plate" in the above class-based definition of an iterable.

Python provides another way to specify something which meets the iterator protocol: **generators**.

```
In [23]: def my_generator():
             yield 5
             yield 10


         x = my_generator()

In [24]: next(x)

Out[24]: 5

In [25]: next(x)

Out[25]: 10

In [26]: next(x)


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

         StopIteration                             Traceback (most recent call last)

         <ipython-input-26-92de4e9f6b1e> in <module>
     ----> 1 next(x)


         StopIteration:


In [27]: for a in my_generator():
             print(a)
```

```
5
10
```

```
In [28]: sum(my_generator())
```

```
Out[28]: 15
```

A function which has `yield` statements instead of a `return` statement returns **temporarily**: it automagically becomes something which implements `__next__`.

Each call of `next()` returns control to the function where it left off.

Control passes back-and-forth between the generator and the caller. Our Fibonacci example therefore becomes a function rather than a class.

```
In [29]: def yield_fibs(limit, seed1=1, seed2=1):
             current = seed1
             previous = seed2

             while limit > 0:
                 limit -= 1
                 current, previous = current + previous, current
                 yield current
```

We can now use the output of the function like a normal iterable:

```
In [30]: sum(yield_fibs(5))
```

```
Out[30]: 31
```

```
In [31]: for a in yield_fibs(10):
             if a % 2 == 0:
                 print(a)
```

```
2
8
34
144
```

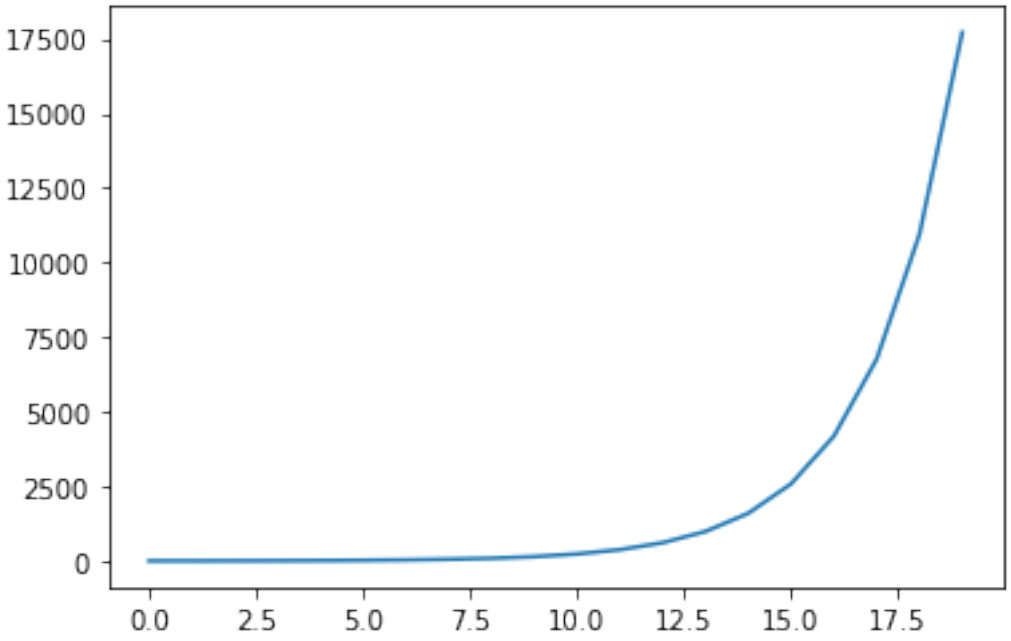
Sometimes we may need to gather all values from a generator into a list, such as before passing them to a function that expects a list:

```
In [32]: list(yield_fibs(10))
```

```
Out[32]: [2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

```
In [33]: plt.plot(list(yield_fibs(20)))
```

```
Out[33]: [<matplotlib.lines.Line2D at 0x7fa1155e4fd0>]
```



7.4 Related Concepts

Iterables and generators can be used to achieve complex behaviour, especially when combined with functional programming. In fact, Python itself contains some very useful language features that make use of these practices: context managers and decorators. We have already seen these in this class, but here we discuss them in more detail.

7.4.1 Context managers

We have seen before [notebook] that, instead of separately `open`ing and `close`ing a file, we can have the file be automatically closed using a context manager:

```
In [34]: %%writefile example.yaml
         modelname: brilliant
```

```
Writing example.yaml
```

```
In [35]: import yaml

         with open('example.yaml') as foo:
             print(yaml.safe_load(foo))
```

```
{'modelname': 'brilliant'}
```

In addition to more convenient syntax, this takes care of any clean-up that has to be done after the file is closed, even if any errors occur while we are working on the file.

How could we define our own one of these, if we too have clean-up code we always want to run after a calling function has done its work, or set-up code we want to do first?

We can define a class that meets an appropriate protocol:

```
In [36]: class verbose_context():
             def __init__(self, name):
                 self.name=name
             def __enter__(self):
                 print("Get ready, ", self.name)
             def __exit__(self, exc_type, exc_value, traceback):
                 print("OK, done")

         with verbose_context("Monty"):
             print("Doing it!")
```

```
Get ready,  Monty
Doing it!
OK, done
```

However, this is pretty verbose! Again, a generator with `yield` makes for an easier syntax:

```
In [37]: from contextlib import contextmanager

         @contextmanager
         def verbose_context(name):
             print("Get ready for action, ", name)
             yield name.upper()
             print("You did it")

         with verbose_context("Monty") as shouty:
             print(f"Doing it, {shouty}")
```

```
Get ready for action,  Monty
Doing it, MONTY
You did it
```

Again, we use `yield` to temporarily return from a function.

7.4.2 Decorators

When doing functional programming, we may often want to define mutator functions which take in one function and return a new function, such as our derivative example earlier.

```
In [38]: from math import sqrt


         def repeater(count):
             def wrap_function_in_repeat(func):

                 def _repeated(x):
                     counter = count
                     while counter > 0:
                         counter -= 1
                         x = func(x)
                     return x

                 return _repeated
             return wrap_function_in_repeat
```

```
        fiftytimes = repeater(50)

        fiftyroots = fiftytimes(sqrt)

        print(fiftyroots(100))
```

```
1.000000000000004
```

It turns out that, quite often, we want to apply one of these to a function as we're defining a class. For example, we may want to specify that after certain methods are called, data should always be stored:

Any function which accepts a function as its first argument and returns a function can be used as a **decorator** like this.

Much of Python's standard functionality is implemented as decorators: we've seen @contextmanager, @classmethod and @attribute. The @contextmanager metafunction, for example, takes in an iterator, and yields a class conforming to the context manager protocol.

```
In [39]: @repeater(3)
         def hello(name):
             return f"Hello, {name}"

In [40]: hello("Cleese")

Out[40]: 'Hello, Hello, Hello, Cleese'
```

7.5 Supplementary material

The remainder of this page contains an example of the flexibility of the features discussed above. Specifically, it shows how generators and context managers can be combined to create a testing framework like the one previously seen in the course.

7.5.1 Test generators

A few weeks ago we saw a test which loaded its test cases from a YAML file and asserted each input with each output. This was nice and concise, but had one flaw: we had just one test, covering all the fixtures, so we got just one . in the test output when we ran the tests, and if any test failed, the rest were not run. We can do a nicer job with a test **generator**:

```
In [41]: def assert_exemplar(**fixture):
             answer = fixture.pop('answer')
             assert_equal(greet(**fixture), answer)


         def test_greeter():
             with open(os.path.join(os.path.dirname(
                 __file__), 'fixtures', 'samples.yaml')
             ) as fixtures_file:
                 fixtures = yaml.safe_load(fixtures_file)

                 for fixture in fixtures:

                     yield assert_exemplar(**fixture)
```

Each time a function beginning with `test_` does a `yield` it results in another test.

7.5.2 Negative test contexts managers

We have seen this:

```
In [42]: from pytest import raises

         with raises(AttributeError):
             x = 2
             x.foo()
```

We can now see how `pytest` might have implemented this:

```
In [43]: from contextlib import contextmanager


         @contextmanager
         def reimplement_raises(exception):
             try:
                 yield
             except exception:
                 pass
             else:
                 raise Exception("Expected,", exception,
                                 " to be raised, nothing was.")

In [44]: with reimplement_raises(AttributeError):
             x = 2
             x.foo()
```

7.5.3 Negative test decorators

Some frameworks, like `nose`, also implement a very nice negative test decorator, which lets us marks tests that we know should produce an exception:

```
In [45]: import nose


         @nose.tools.raises(TypeError, ValueError)
         def test_raises_type_error():
             raise TypeError("This test passes")

In [46]: test_raises_type_error()

In [47]: @nose.tools.raises(Exception)
         def test_that_fails_by_passing():
             pass

In [48]: test_that_fails_by_passing()


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

         AssertionError                            Traceback (most recent call last)

         <ipython-input-48-627706dd82d1> in <module>
     ----> 1 test_that_fails_by_passing()
```

```
~/virtualenv/python3.7.5/lib/python3.7/site-packages/nose/tools/nontrivial.py in newfunc(*arg,
 65             else:
 66                 message = "%s() did not raise %s" % (name, valid)
---> 67                 raise AssertionError(message)
 68         newfunc = make_decorator(func)(newfunc)
 69         return newfunc


AssertionError: test_that_fails_by_passing() did not raise Exception
```

We could reimplement this ourselves now too, using the context manager we wrote above:

```
In [49]: def homemade_raises_decorator(exception):
             def wrap_function(func):  # Closure over exception
                 # Define a function which runs another function under our "raises" context:
                 def _output(*args):  # Closure over func and exception
                     with reimplement_raises(exception):
                         func(*args)
                 # Return it
                 return _output
             return wrap_function

In [50]: @homemade_raises_decorator(TypeError)
         def test_raises_type_error():
             raise TypeError("This test passes")

In [51]: test_raises_type_error()
```

7.6 Exceptions

When we learned about testing, we saw that Python complains when things go wrong by raising an "Exception" naming a type of error:

```
In [1]: 1/0


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

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-1-9e1622b385b6> in <module>
----> 1 1/0


ZeroDivisionError: division by zero
```

Exceptions are objects, forming a class hierarchy. We just raised an instance of the `ZeroDivisionError` class, making the program crash. If we want more information about where this class fits in the hierarchy, we can use Python's `inspect` module to get a chain of classes, from `ZeroDivisionError` up to `object`:

```
In [2]: import inspect
        inspect.getmro(ZeroDivisionError)
```

```
Out[2]: (ZeroDivisionError, ArithmeticError, Exception, BaseException, object)
```

So we can see that a zero division error is a particular kind of Arithmetic Error.

```
In [3]: x = 1

        for y in x:
            print(y)


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

        TypeError                                 Traceback (most recent call last)

        <ipython-input-3-127e9e41ff29> in <module>
          1 x = 1
          2
    ----> 3 for y in x:
          4     print(y)


        TypeError: 'int' object is not iterable
```

```
In [4]: inspect.getmro(TypeError)

Out[4]: (TypeError, Exception, BaseException, object)
```

7.6.1 Create your own Exception

When we were looking at testing, we saw that it is important for code to crash with a meaningful exception type when something is wrong. We raise an Exception with `raise`. Often, we can look for an appropriate exception from the standard set to raise.

However, we may want to define our own exceptions. Doing this is as simple as inheriting from Exception (or one of its subclasses):

```
In [5]: class MyCustomErrorType(ArithmeticError):
            pass


        raise(MyCustomErrorType("Problem"))


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

        MyCustomErrorType                         Traceback (most recent call last)

        <ipython-input-5-d49058201ff8> in <module>
          3
          4
    ----> 5 raise(MyCustomErrorType("Problem"))


        MyCustomErrorType: Problem
```

You can add custom data to your exception:

```
In [6]: class MyCustomErrorType(Exception):
            def __init__(self, category=None):
                self.category = category

            def __str__(self):
                return f"Error, category {self.category}"


        raise(MyCustomErrorType(404))


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

        MyCustomErrorType                         Traceback (most recent call last)

        <ipython-input-6-edbc5ba6fff8> in <module>
          7
          8
    ----> 9 raise(MyCustomErrorType(404))


        MyCustomErrorType: Error, category 404
```

The real power of exceptions comes, however, not in letting them crash the program, but in letting your program handle them. We say that an exception has been "thrown" and then "caught".

```
In [7]: import yaml

        try:
            config = yaml.safe_load(open("datasource.yaml"))
            user = config["userid"]
            password = config["password"]

        except FileNotFoundError:
            print("No password file found, using anonymous user.")
            user = "anonymous"
            password = None


        print(user)
```

```
No password file found, using anonymous user.
anonymous
```

Note that we specify only the error we expect to happen and want to handle. Sometimes you see code that catches everything:

```
In [8]: try:
            config = yaml.lod(open("datasource.yaml"))
            user = config["userid"]
            password = config["password"]
        except:
```

```
            user = "anonymous"
            password = None

        print(user)
```

```
anonymous
```

This can be dangerous and can make it hard to find errors! There was a mistyped function name there (‘`lod`’), but we did not notice the error, as the generic except caught it. Therefore, we should be specific and catch only the type of error we want.

7.6.2 Managing multiple exceptions

Let’s create two credential files to read

```
In [9]: with open('datasource2.yaml', 'w') as outfile:
            outfile.write('userid: eidle\n')
            outfile.write('password: secret\n')

        with open('datasource3.yaml', 'w') as outfile:
            outfile.write('user: eidle\n')
            outfile.write('password: secret\n')
```

And create a function that reads credentials files and returns the username and password to use.

```
In [10]: def read_credentials(source):
             try:
                 datasource = open(source)
                 config = yaml.safe_load(datasource)
                 user = config["userid"]
                 password = config["password"]
                 datasource.close()
             except FileNotFoundError:
                 print("Password file missing")
                 user = "anonymous"
                 password = None
             except KeyError:
                 print("Expected keys not found in file")
                 user = "anonymous"
                 password = None
             return user, password
```

```
In [11]: print(read_credentials('datasource2.yaml'))
```

```
('eidle', 'secret')
```

```
In [12]: print(read_credentials('datasource.yaml'))
```

```
Password file missing
('anonymous', None)
```

```
In [13]: print(read_credentials('datasource3.yaml'))
```

```
Expected keys not found in file
('anonymous', None)
```

This last code has a flaw: the file was successfully opened, the missing key was noticed, but not explicitly closed. It's normally OK, as Python will close the file as soon as it notices there are no longer any references to datasource in memory, after the function exits. But this is not good practice, you should keep a file handle for as short a time as possible.

```
In [14]: def read_credentials(source):
             try:
                 datasource = open(source)
                 config = yaml.safe_load(datasource)
                 user = config["userid"]
                 password = config["password"]
             except FileNotFoundError:
                 user = "anonymous"
                 password = None
             finally:
                 datasource.close()

             return user, password
```

The `finally` clause is executed whether or not an exception occurs.

The last optional clause of a `try` statement, an `else` clause is called only if an exception is NOT raised. It can be a better place than the `try` clause to put code other than that which you expect to raise the error, and which you do not want to be executed if the error is raised. It is executed in the same circumstances as code put in the end of the `try` block, the only difference is that errors raised during the `else` clause are not caught. Don't worry if this seems useless to you; most languages' implementations of try/except don't support such a clause.

```
In [15]: def read_credentials(source):
             try:
                 datasource = open(source)
             except FileNotFoundError:
                 user = "anonymous"
                 password = None
             else:
                 config = yaml.safe_load(datasource)
                 user = config["userid"]
                 password = config["password"]
             finally:
                 datasource.close()
             return user, password
```

Exceptions do not have to be caught close to the part of the program calling them. They can be caught anywhere "above" the calling point in the call stack: control can jump arbitrarily far in the program: up to the `except` clause of the "highest" containing try statement.

```
In [16]: def f4(x):
             if x == 0:
                 return
             if x == 1:
                 raise ArithmeticError()
             if x == 2:
```

```
            raise SyntaxError()
        if x == 3:
            raise TypeError()
```

```
In [17]: def f3(x):
        try:
            print("F3Before")
            f4(x)
            print("F3After")
        except ArithmeticError:
            print("F3Except ( )")
```

```
In [18]: def f2(x):
        try:
            print("F2Before")
            f3(x)
            print("F2After")
        except SyntaxError:
            print("F2Except ( )")
```

```
In [19]: def f1(x):
        try:
            print("F1Before")
            f2(x)
            print("F1After")
        except TypeError:
            print("F1Except ( )")
```

```
In [20]: f1(0)
```

```
F1Before
F2Before
F3Before
F3After
F2After
F1After
```

```
In [21]: f1(1)
```

```
F1Before
F2Before
F3Before
F3Except ( )
F2After
F1After
```

```
In [22]: f1(2)
```

```
F1Before
F2Before
F3Before
F2Except ( )
F1After
```

```
In [23]: f1(3)
```

```
F1Before
F2Before
F3Before
F1Except ( )
```

7.6.3 Design with Exceptions

Now we know how exceptions work, we need to think about the design implications… How best to use them.

Traditional software design theory will tell you that they should only be used to describe and recover from **exceptional** conditions: things going wrong. Normal program flow shouldn't use them.

Python's designers take a different view: use of exceptions in normal flow is considered OK. For example, all iterators raise a `StopIteration` exception to indicate the iteration is complete.

A commonly recommended Python design pattern is to use exceptions to determine whether an object implements a protocol (concept/interface), rather than testing on type.

For example, we might want a function which can be supplied *either* a data series *or* a path to a location on disk where data can be found. We can examine the type of the supplied content:

```
In [24]: import yaml


         def analysis(source):
             if type(source) == dict:
                 name = source['modelname']
             else:
                 content = open(source)
                 source = yaml.safe_load(content)
                 name = source['modelname']
             print(name)
```

```
In [25]: analysis({'modelname': 'Super'})
```

```
Super
```

```
In [26]: with open('example.yaml', 'w') as outfile:
             outfile.write('modelname: brilliant\n')
```

```
In [27]: analysis('example.yaml')
```

```
brilliant
```

However, we can also use the try-it-and-handle-exceptions approach to this.

```
In [28]: def analysis(source):
             try:
                 name = source['modelname']
             except TypeError:
                 content = open(source)
                 source = yaml.safe_load(content)
                 name = source['modelname']
             print(name)


         analysis('example.yaml')
```

```
brilliant
```

This approach is more extensible, and **behaves properly if we give it some other data-source which responds like a dictionary or string.**

```
In [29]: def analysis(source):
             try:
                 name = source['modelname']
             except TypeError:
                 # Source was not a dictionary-like object
                 # Maybe it is a file path
                 try:
                     content = open(source)
                     source = yaml.safe_load(content)
                     name = source['modelname']
                 except IOError:
                     # Maybe it was already raw YAML content
                     source = yaml.safe_load(source)
                     name = source['modelname']
             print(name)


         analysis("modelname: Amazing")
```

```
Amazing
```

Sometimes we want to catch an error, partially handle it, perhaps add some extra data to the exception, and then re-raise to be caught again further up the call stack.

The keyword "`raise`" with no argument in an `except:` clause will cause the caught error to be re-thrown. Doing this is the only circumstance where it is safe to do `except:` without catching a specific type of error.

```
In [30]: try:
             # Something
             pass
         except:
             # Do this code here if anything goes wrong
             raise
```

If you want to be more explicit about where the error came from, you can use the `raise from` syntax, which will create a chain of exceptions:

```
In [31]: def lower_function():
             raise ValueError("Error in lower function!")


         def higher_function():
             try:
                 lower_function()
             except ValueError as e:
                 raise RuntimeError("Error in higher function!") from e


         higher_function()
```

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

    ValueError                                Traceback (most recent call last)

    <ipython-input-31-88616a1de55f> in higher_function()
      6     try:
----> 7         lower_function()
      8     except ValueError as e:


    <ipython-input-31-88616a1de55f> in lower_function()
      1 def lower_function():
----> 2     raise ValueError("Error in lower function!")
      3


    ValueError: Error in lower function!


The above exception was the direct cause of the following exception:


    RuntimeError                              Traceback (most recent call last)

    <ipython-input-31-88616a1de55f> in <module>
     10
     11
---> 12 higher_function()


    <ipython-input-31-88616a1de55f> in higher_function()
      7         lower_function()
      8     except ValueError as e:
----> 9         raise RuntimeError("Error in higher function!") from e
     10
     11


    RuntimeError: Error in higher function!
```

It can be useful to catch and re-throw an error as you go up the chain, doing any clean-up needed for each layer of a program.

The error will finally be caught and not re-thrown only at a higher program layer that knows how to recover. This is known as the “throw low catch high” principle.

Chapter 8

Operator overloading

We've seen already during the course that some operators behave differently depending on the data type.

For example, `+` adds numbers but concatenates strings or lists:

```
In [1]: 4 + 2
```

```
Out[1]: 6
```

```
In [2]: '4' + '2'
```

```
Out[2]: '42'
```

`*` is used for multiplication, or repeated addition:

```
In [3]: 6 * 7
```

```
Out[3]: 42
```

```
In [4]: 'me' * 3
```

```
Out[4]: 'mememe'
```

`/` is division for numbers, and wouldn't have a real meaning on strings. However, it's used to separate files and directories on your file system. Therefore, this has been *overloaded* in the `pathlib` module:

```
In [5]: import os
        from pathlib import Path


        performance = Path('..') / 'ch08performance'
        os.listdir(performance)
```

```
Out[5]: ['040cython.ipynb',
         'deque_memory.png',
         'index.md',
         '020numpy.ipynb',
         '050scaling.ipynb',
         'list_memory.png',
         '015mandels.ipynb',
         'array_memory.png',
         '010intro.ipynb']
```

The above works because one of the elements is a `Path` object. Note, that the `/` works similarly to `os.path.join()`, so whether you are using Unix file systems or Windows, `pathlib` will know what path separator to use.

```
In [6]: performance = os.path.join('..', 'ch08performance')
```

8.1 Overloading operators for your own classes

Now that we have seen that in Python operators do different things, how can we use + or other operators on our own classes to achieve similar behaviour?

Let's go back to our Maze example, and simplify our room object so it's defined as:

```
In [7]: class Room:
            def __init__(self, name, area):
                self.name = name
                self.area = area
```

We can now create a room as:

```
In [8]: small = Room('small', 9)
        print(small)
```

```
<__main__.Room object at 0x7fe6281cdd50>
```

However, when we print it we don't get much infomation on the object. So, the first operator we are overloading is its string represenation defining `__str__`:

```
In [9]: class Room:
            def __init__(self, name, area):
                self.name = name
                self.area = area
            def __str__(self):
                return f"<Room: {self.name} {self.area}m²>"

In [10]: small = Room('small', 9)
         print(small)
```

```
<Room: small 9m²>
```

How can we add two rooms together? What does it mean? Let's define that the addition (+) of two rooms makes up one with the combined size. We produce this behaviour by defining the `__add__` method.

```
In [11]: class Room:
             def __init__(self, name, area):
                 self.name = name
                 self.area = area
             def __add__(self, other):
                 return Room(f"{self.name}_{other.name}", self.area + other.area)
             def __str__(self):
                 return f"<Room: {self.name} {self.area}m²>"

In [12]: small = Room('small', 9)
         big = Room('big', 21)
         print(small, big, small + big)
```

```
<Room: small 9m²> <Room: big 21m²> <Room: small_big 30m²>
```

Would the order of how the rooms are added affect the final room? As they are added now, the name is determined by the order, but do we want that? Or would we prefer to have:

```
small + big == big + small
```

That bring us to another operator, equal to: `==`. The method needed to produce such comparison is `__eq__`.

```
In [13]: class Room:
             def __init__(self, name, area):
                 self.name = name
                 self.area = area
             def __add__(self, other):
                 return Room(f"{self.name}_{other.name}", self.area + other.area)
             def __eq__(self, other):
                 return self.area == other.area and set(self.name.split('_')) == set(other.name.split('
```

So, in this way two rooms of the same area are "equal" if their names are composed by the same.

```
In [14]: small = Room('small', 9)
         big = Room('big', 21)
         large = Room('superbig', 30)
         print(small + big == big + small)
         print(small + big == large)
```

```
True
False
```

You can add the other comparisons to know which room is bigger or smaller with the following functions:

Operator	Function
<	__lt__(self, other)
<=	__le__(self, other)
>	__gt__(self, other)
>=	__ge__(self, other)

Let's add people to the rooms and check whether they are in one room or not.

```
In [15]: class Room:
             def __init__(self, name, area):
                 self.name = name
                 self.area = area
                 self.occupants = []
             def add_occupant(self, name):
                 self.occupants.append(name)

         circus = Room('Circus', 3)
         circus.add_occupant('Graham')
         circus.add_occupant('Eric')
         circus.add_occupant('Terry')
```

How do we know if John is in the room? We can check the `occupants` list:

```
In [16]: 'John' in circus.occupants
```

```
Out[16]: False
```

Or making it more readable adding a membership definition:

```
In [17]: class Room:
             def __init__(self, name, area):
                 self.name = name
                 self.area = area
                 self.occupants = []
             def add_occupant(self, name):
                 self.occupants.append(name)
             def __contains__(self, value):
                 return value in self.occupants

         circus = Room('Circus', 3)
         circus.add_occupant('Graham')
         circus.add_occupant('Eric')
         circus.add_occupant('Terry')

         'Terry' in circus
```

```
Out[17]: True
```

We can add lots more operators to classes. For example, `__getitem__` to let you index or access part of your object like a sequence or dictionary, *e.g.*, `newObject[1]` or `newObject["data"]`, or `__len__` to return a number of elements in your object. Probably the most exciting one is `__call__`, which overrides the `()` operator; this allows us to define classes that *behave like functions*! We call these **callables**.

```
In [18]: class Greeter(object):
             def __init__(self, greeting):
                 self.greeting = greeting

             def __call__(self, name):
                 print(self.greeting, name)

         greeter_instance = Greeter("Hello")

         greeter_instance("Eric")
```

```
Hello Eric
```

We've now come full circle in the blurring of the distinction between functions and objects! The full power of functional programming is really remarkable.

If you want to know more about the topics in this lecture, using a different language syntax, I recommend you watch the Abelson and Sussman "Structure and Interpretation of Computer Programs" lectures. These are the Computer Science equivalent of the Feynman Lectures!

Next notebook shows a detailed example of how to apply operator overloading to create your own symbolic algebra system.

Chapter 9

Operator overloading

Warning: Advanced Topic!

9.0.1 Setup for this notebook

We need to use a metaprogramming trick to make this teaching notebook work. I want to be able to put explanatory text in between parts of a class definition, so I'll define a decorator to help me build up a class definition gradually.

```
In [1]: def extend(class_to_extend):
            """ Metaprogramming to allow gradual implementation
            of class during notebook. Thanks to
            http://www.ianbicking.org/blog/2007/08/opening-python-classes.html """
            def decorator(extending_class):
                for name, value in extending_class.__dict__.items():
                    if name in ['__dict__', '__module__', '__weakref__', '__doc__']:
                        continue
                    setattr(class_to_extend, name, value)
                return class_to_extend
            return decorator
```

9.0.2 Operator overloading

Imagine we wanted to make a library to describe some kind of symbolic algebra system:

```
In [2]: class Term:
            def __init__(self, symbols=[], powers=[], coefficient=1):
                self.coefficient = coefficient
                self.data={symbol: exponent for symbol,exponent
                        in zip(symbols, powers)}
```

```
In [3]: class Expression:
            def __init__(self, terms):
                self.terms = terms
```

So that $5x^2y + 7x + 2$ might be constructed as:

```
In [4]: first = Term(['x', 'y'], [2, 1], 5)

        second = Term(['x'], [1], 7)

        third = Term([], [], 2)
```

```
result = Expression([first, second, third])
```

This is pretty cumbersome.
What we'd really like is to have `2x+y` give an appropriate expression.
First, we'll define things so that we can construct our terms and expressions in different ways.

```
In [5]: class Term:
            def __init__(self, *args):
                lead = args[0]
                if type(lead) == type(self):
                    # Copy constructor
                    self.data = dict(lead.data)
                    self.coefficient = lead.coefficient
                elif type(lead) == int:
                    self.from_constant(lead)
                elif type(lead) == str:
                    self.from_symbol(*args)
                elif type(lead) == dict:
                    self.from_dictionary(*args)
                else:
                    self.from_lists(*args)

            def from_constant(self, constant):
                self.coefficient = constant
                self.data = {}

            def from_symbol(self, symbol, coefficient=1, power=1):
                self.coefficient = coefficient
                self.data = {symbol: power}

            def from_dictionary(self, data, coefficient=1):
                self.data = data
                self.coefficient = coefficient

            def from_lists(self, symbols=[], powers=[], coefficient=1):
                self.coefficient = coefficient
                self.data={symbol: exponent for symbol,exponent
                           in zip(symbols, powers)}
```

```
In [6]: class Expression:
            def __init__(self, terms=[]):
                self.terms = list(terms)
```

We could define add() and multiply() operations on expressions and terms:

```
In [7]: @extend(Term)
        class Term:
            def add(self, *others):
                return Expression((self,) + others)
```

```
In [8]: @extend(Term)
        class Term:
            def multiply(self, *others):
```

```
            result_data = dict(self.data)
            result_coeff = self.coefficient
            # Convert arguments to Terms first if they are
            # constants or integers
            others = map(Term, others)

            for another in others:
                for symbol, exponent in another.data.items():
                    if symbol in result_data:
                        result_data[symbol] += another.data[symbol]
                    else:
                        result_data[symbol] = another.data[symbol]
                result_coeff *= another.coefficient

            return Term(result_data, result_coeff)
```

```
In [9]: @extend(Expression)
        class Expression:
            def add(self, *others):
                result = Expression(self.terms)

                for another in others:
                    if type(another) == Term:
                        result.terms.append(another)
                    else:
                        result.terms += another.terms

                return result
```

We can now construct the above expression as:

```
In [10]: x = Term('x')
         y = Term('y')

         first = Term(5).multiply(Term('x'), Term('x'), Term('y'))
         second = Term(7).multiply(Term('x'))
         third = Term(2)
         expr = first.add(second, third)
```

This is better, but we still can't write the expression in a 'natural' way. However, we can define what `*` and `+` do when applied to Terms!:

```
In [11]: @extend(Term)
         class Term:

             def __add__(self, other):
                 return self.add(other)

             def __mul__(self, other):
                 return self.multiply(other)
```

```
In [12]: @extend(Expression)
         class Expression:
             def multiply(self, another):
                 # Distributive law left as exercise
```

```
                pass

            def __add__(self, other):
                return self.add(other)
```

```
In [13]: x_plus_y = Term('x') + 'y'
         x_plus_y.terms[1]
```

```
Out[13]: 'y'
```

```
In [14]: five_x_ysq = Term('x') * 5 * 'y' * 'y'

         print(five_x_ysq.data, five_x_ysq.coefficient)
```

```
{'x': 1, 'y': 2} 5
```

This is called operator overloading. We can define what add and multiply mean when applied to our class.

Note that this only works so far if we multiply on the right-hand-side! However, we can define a multiplication that works backwards, which is used as a fallback if the left multiply raises an error:

```
In [15]: @extend(Expression)
         class Expression:
             def __radd__(self, other):
                 return self.__add__(other)
```

```
In [16]: @extend(Term)
         class Term:
             def __rmul__(self, other):
                 return self.__mul__(other)

             def __radd__(self, other):
                 return self.__add__(other)
```

```
In [17]: 5 * Term('x')
```

```
Out[17]: <__main__.Term at 0x7f85e8452e50>
```

It's not easy at the moment to see if these things are working!

```
In [18]: fivex = 5 * Term('x')
         fivex.data, fivex.coefficient
```

```
Out[18]: ({'x': 1}, 5)
```

We can add another operator method `__str__`, which defines what happens if we try to print our class:

```
In [19]: @extend(Term)
         class Term:
             def __str__(self):
                 def symbol_string(symbol, power):
                     if power == 1:
                         return symbol
                     else:
                         return f"{symbol}^{power}"
```

```
                symbol_strings=[symbol_string(symbol, power)
                        for symbol, power in self.data.items()]

                prod = '*'.join(symbol_strings)

                if not prod:
                    return str(self.coefficient)
                if self.coefficient == 1:
                    return prod
                else:
                    return f"{self.coefficient}*{prod}"
```

```
In [20]: @extend(Expression)
         class Expression:
             def __str__(self):
                 return '+'.join(map(str, self.terms))
```

```
In [21]: first = Term(5) * 'x' * 'x' * 'y'
         second = Term(7) * 'x'
         third = Term(2)
         expr = first + second + third
```

```
In [22]: print(expr)
```

```
5*x^2*y+7*x+2
```

9.1 Metaprogramming

Warning: Advanced topic!

9.1.1 Metaprogramming globals

Consider a bunch of variables, each of which need initialising and incrementing:

```
In [1]: bananas = 0
        apples = 0
        oranges = 0
        bananas += 1
        apples += 1
        oranges += 1
```

The right hand side of these assignments doesn't respect the DRY principle. We could of course define a variable for our initial value:

```
In [2]: initial_fruit_count = 0
        bananas = initial_fruit_count
        apples = initial_fruit_count
        oranges = initial_fruit_count
```

However, this is still not as DRY as it could be: what if we wanted to replace the assignment with, say, a class constructor and a buy operation:

```
In [3]: class Basket:
            def __init__(self):
                self.count = 0
```

```
            def buy(self):
                self.count += 1

        bananas = Basket()
        apples = Basket()
        oranges = Basket()
        bananas.buy()
        apples.buy()
        oranges.buy()
```

We had to make the change in three places. Whenever you see a situation where a refactoring or change of design might require you to change the code in multiple places, you have an opportunity to make the code DRYer.

In this case, metaprogramming for incrementing these variables would involve just a loop over all the variables we want to initialise:

```
In [4]: baskets = [bananas, apples, oranges]
        for basket in baskets:
            basket.buy()
```

However, this trick **doesn't** work for initialising a new variable:

```
In [5]: from pytest import raises
        with raises(NameError):
            baskets = [bananas, apples, oranges, kiwis]
```

So can we declare a new variable programmatically? Given a list of the **names** of fruit baskets we want, initialise a variable with that name?

```
In [6]: basket_names = ['bananas', 'apples', 'oranges', 'kiwis']

        globals()['apples']

Out[6]: <__main__.Basket at 0x7fd8016a6750>
```

Wow, we can! Every module or class in Python, is, under the hood, a special dictionary, storing the values in its **namespace**. So we can create new variables by assigning to this dictionary. globals() gives a reference to the attribute dictionary for the current module

```
In [7]: for name in basket_names:
            globals()[name] = Basket()


        kiwis.count

Out[7]: 0
```

This is **metaprogramming**.

I would NOT recommend using it for an example as trivial as the one above. A better, more Pythonic choice here would be to use a data structure to manage your set of fruit baskets:

```
In [8]: baskets = {}
        for name in basket_names:
            baskets[name] = Basket()

        baskets['kiwis'].count
```

```
Out[8]: 0
```

Or even, using a dictionary comprehension:

```
In [9]: baskets = {name: Basket() for name in baskets}
        baskets['kiwis'].count

Out[9]: 0
```

Which is the nicest way to do this, I think. Code which feels like metaprogramming is needed to make it less repetitive can often instead be DRYed up using a refactored data structure, in a way which is cleaner and more easy to understand. Nevertheless, metaprogramming is worth knowing.

9.1.2 Metaprogramming class attributes

We can metaprogram the attributes of a **module** using the globals() function.

We will also want to be able to metaprogram a class, by accessing its attribute dictionary.

This will allow us, for example, to programmatically add members to a class.

```
In [10]: class Boring:
             pass
```

If we are adding our own attributes, we can just do so directly:

```
In [11]: x = Boring()

         x.name = "Michael"

In [12]: x.name

Out[12]: 'Michael'
```

And these turn up, as expected, in an attribute dictionary for the class:

```
In [13]: x.__dict__

Out[13]: {'name': 'Michael'}
```

We can use `getattr` to access this special dictionary:

```
In [14]: getattr(x, 'name')

Out[14]: 'Michael'
```

If we want to add an attribute given it's name as a string, we can use setattr:

```
In [15]: setattr(x, 'age', 75)

         x.age

Out[15]: 75
```

And we could do this in a loop to programmatically add many attributes.

The real power of accessing the attribute dictionary comes when we realise that there is *very little difference* between member data and member functions.

Now that we know, from our functional programming, that **a function is just a variable that can be *called* with `()`**, we can set an attribute to a function, and it becomes a member function!

```
In [16]: setattr(Boring, 'describe', lambda self: f"{self.name} is {self.age}")
```

```
In [17]: x.describe()

Out[17]: 'Michael is 75'

In [18]: x.describe

Out[18]: <bound method <lambda> of <__main__.Boring object at 0x7fd8009d8550>>

In [19]: Boring.describe

Out[19]: <function __main__.<lambda>(self)>
```

Note that we set this method as an attribute of the class, not the instance, so it is available to other instances of `Boring`:

```
In [20]: y = Boring()
         y.name = 'Terry'
         y.age  = 78

In [21]: y.describe()

Out[21]: 'Terry is 78'
```

We can define a standalone function, and then **bind** it to the class. Its first argument automagically becomes `self`.

```
In [22]: def broken_birth_year(b_instance):
             import datetime
             current = datetime.datetime.now().year
             return current - b_instance.age

In [23]: Boring.birth_year = broken_birth_year

In [24]: x.birth_year()

Out[24]: 1945

In [25]: x.birth_year

Out[25]: <bound method broken_birth_year of <__main__.Boring object at 0x7fd8009d8550>>

In [26]: x.birth_year.__name__

Out[26]: 'broken_birth_year'
```

9.1.3 Metaprogramming function locals

We can access the attribute dictionary for the local namespace inside a function with `locals()` but this *cannot be written to.*

Lack of safe programmatic creation of function-local variables is a flaw in Python.

```
In [27]: class Person:
             def __init__(self, name, age, job, children_count):
                 for name, value in locals().items():
                     if name == 'self':
                         continue
                     print(f"Setting self.{name} to {value}")
                     setattr(self, name, value)
```

```
In [28]: terry = Person("Terry", 78, "Screenwriter", 0)

Setting self.name to Terry
Setting self.age to 78
Setting self.job to Screenwriter
Setting self.children_count to 0


In [29]: terry.name

Out[29]: 'Terry'
```

9.1.4 Metaprogramming warning!

Use this stuff **sparingly**!

The above example worked, but it produced Python code which is not particularly understandable. Remember, your objective when programming is to produce code which is **descriptive of what it does**.

The above code is **definitely** less readable, less maintainable and more error prone than:

```
In [30]: class Person:
             def __init__(self, name, age, job, children_count):
                 self.name = name
                 self.age = age
                 self.job = job
                 self.children_count = children_count
```

Sometimes, metaprogramming will be **really** helpful in making non-repetitive code, and you should have it in your toolbox, which is why I'm teaching you it. But doing it all the time overcomplicated matters. We've talked a lot about the DRY principle, but there is another equally important principle:

> **KISS**: *Keep it simple, Stupid!*

Whenever you write code and you think, "Gosh, I'm really clever",you're probably *doing it wrong*. Code should be about clarity, not showing off.

Chapter 10

Performance programming

We've spent most of this course looking at how to make code readable and reliable. For research work, it is often also important that code is efficient: that it does what it needs to do *quickly*.

It is very hard to work out beforehand whether code will be efficient or not: it is essential to *Profile* code, to measure its performance, to determine what aspects of it are slow.

When we looked at Functional programming, we claimed that code which is conceptualised in terms of actions on whole data-sets rather than individual elements is more efficient. Let's measure the performance of some different ways of implementing some code and see how they perform.

10.1 Two Mandelbrots

You're probably familiar with a famous fractal called the Mandelbrot Set.

For a complex number c, c is in the Mandelbrot set if the series $z_{i+1} = z_i^2 + c$ (With $z_0 = c$) stays close to 0. Traditionally, we plot a color showing how many steps are needed for $|z_i| > 2$, whereupon we are sure the series will diverge.

Here's a trivial python implementation:

```
In [1]: def mandel1(position, limit=50):

            value = position

            while abs(value) < 2:
                limit -= 1
                value = value**2 + position
                if limit < 0:
                    return 0

            return limit
```

```
In [2]: xmin = -1.5
        ymin = -1.0
        xmax = 0.5
        ymax = 1.0
        resolution = 300
        xstep = (xmax - xmin) / resolution
        ystep = (ymax - ymin) / resolution
        xs = [(xmin + (xmax - xmin) * i / resolution) for i in range(resolution)]
        ys = [(ymin + (ymax - ymin) * i / resolution) for i in range(resolution)]
```

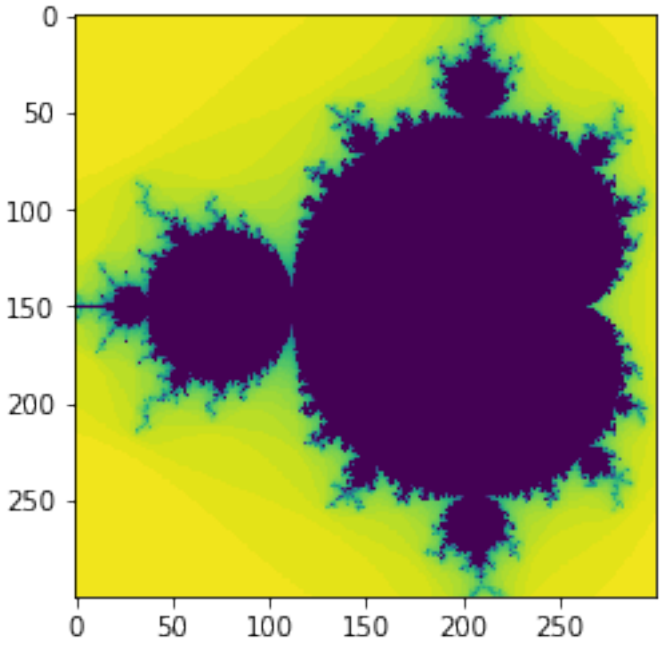
```
In [3]: %%timeit
        data = [[mandel1(complex(x, y)) for x in xs] for y in ys]
```

```
523 ms ± 3.97 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [4]: data1 = [[mandel1(complex(x, y)) for x in xs] for y in ys]
```

```
In [5]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.imshow(data1, interpolation='none')
```

```
Out[5]: <matplotlib.image.AxesImage at 0x7f774da58150>
```



We will learn this lesson how to make a version of this code which works Ten Times faster:

```
In [6]: import numpy as np
        def mandel_numpy(position,limit=50):
            value = position
            diverged_at_count = np.zeros(position.shape)
            while limit > 0:
                limit -= 1
                value = value**2+position
                diverging = value * np.conj(value) > 4
                first_diverged_this_time = np.logical_and(diverging, diverged_at_count == 0)
                diverged_at_count[first_diverged_this_time] = limit
                value[diverging] = 2

            return diverged_at_count
```

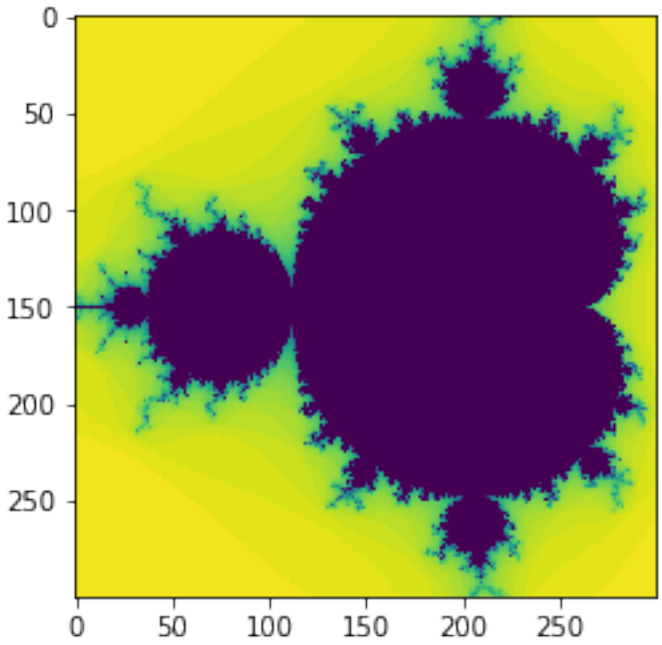
```
In [7]: ymatrix, xmatrix = np.mgrid[ymin:ymax:ystep, xmin:xmax:xstep]
```

```
In [8]: values = xmatrix + 1j * ymatrix
```

```
In [9]: data_numpy = mandel_numpy(values)
```

```
In [10]: %matplotlib inline
         import matplotlib.pyplot as plt
         plt.imshow(data_numpy, interpolation='none')
```

```
Out[10]: <matplotlib.image.AxesImage at 0x7f77715c0850>
```




```
In [11]: %%timeit
         data_numpy = mandel_numpy(values)
```

```
44.1 ms ± 621 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Note we get the same answer:

```
In [12]: sum(sum(abs(data_numpy - data1)))
```

```
Out[12]: 0.0
```

```
In [1]: xmin = -1.5
        ymin = -1.0
        xmax = 0.5
        ymax = 1.0
        resolution = 300
        xstep = (xmax - xmin) / resolution
        ystep = (ymax - ymin) / resolution
        xs = [(xmin + (xmax - xmin) * i / resolution) for i in range(resolution)]
        ys = [(ymin + (ymax - ymin) * i / resolution) for i in range(resolution)]
```

```
In [2]: def mandel1(position, limit=50):
            value = position
            while abs(value) < 2:
```

```
            limit -= 1
            value = value**2 + position
            if limit < 0:
                return 0
        return limit
```

```
In [3]: data1 = [[mandel1(complex(x, y)) for x in xs] for y in ys]
```

10.2 Many Mandelbrots

Let's compare our naive python implementation which used a list comprehension, taking 662ms, with the following:

```
In [4]: %%timeit
        data2 = []
        for y in ys:
            row = []
            for x in xs:
                row.append(mandel1(complex(x, y)))
            data2.append(row)
```

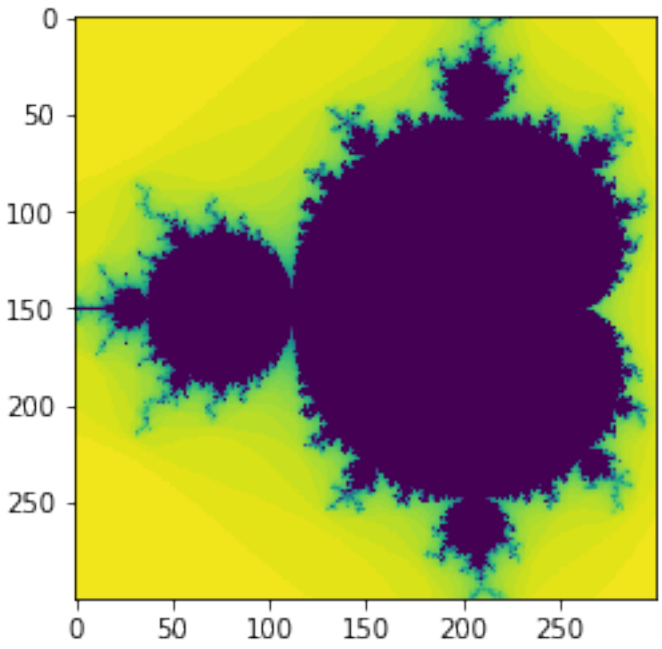
```
509 ms ± 8.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [5]: data2 = []
        for y in ys:
            row = []
            for x in xs:
                row.append(mandel1(complex(x, y)))
            data2.append(row)
```

Interestingly, not much difference. I would have expected this to be slower, due to the normally high cost of **appending** to data.

```
In [6]: from matplotlib import pyplot as plt
        %matplotlib inline
        plt.imshow(data2, interpolation='none')
```

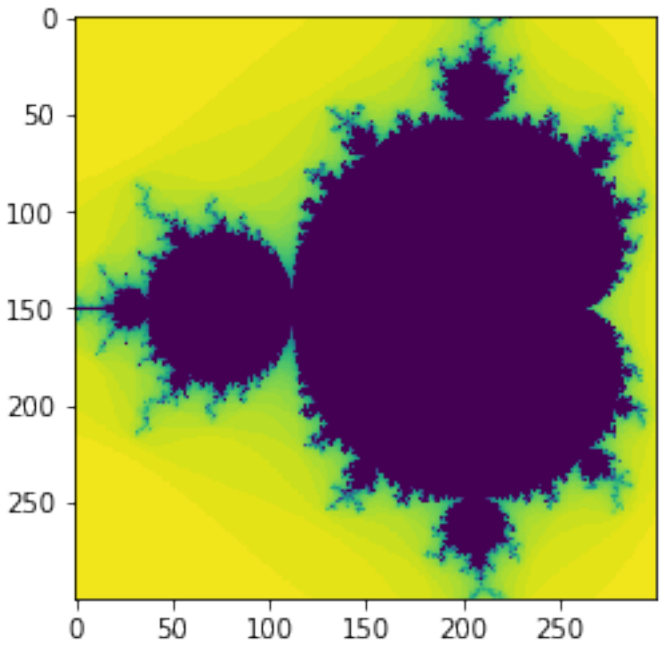
```
Out[6]: <matplotlib.image.AxesImage at 0x7fdc4e1514d0>
```



We ought to be checking if these results are the same by comparing the values in a test, rather than re-plotting. This is cumbersome in pure Python, but easy with NumPy, so we'll do this later.

Let's try a pre-allocated data structure:

```
In [7]: data3 = [[0 for i in range(resolution)] for j in range(resolution)]

In [8]: %%timeit
        for j, y in enumerate(ys):
            for i, x in enumerate(xs):
                data3[j][i] = mandel1(complex(x, y))

511 ms ± 5.51 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)


In [9]: for j, y in enumerate(ys):
            for i, x in enumerate(xs):
                data3[j][i] = mandel1(complex(x, y))

In [10]: plt.imshow(data3, interpolation='none')

Out[10]: <matplotlib.image.AxesImage at 0x7fdc742a6390>
```



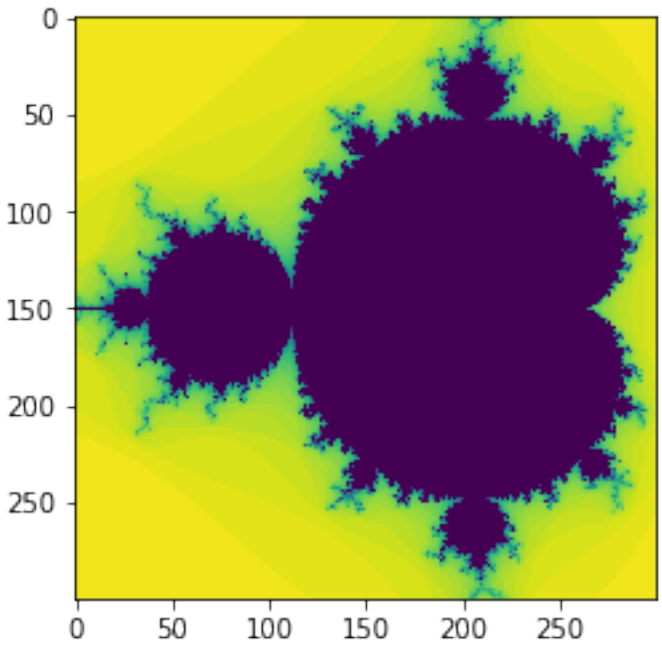
Nope, no gain there.

Let's try using functional programming approaches:

```
In [11]: %%timeit
         data4 = []
         for y in ys:
             bind_mandel = lambda x: mandel1(complex(x, y))
             data4.append(list(map(bind_mandel, xs)))
```

```
521 ms ± 8.39 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [12]: data4 = []
         for y in ys:
             bind_mandel = lambda x: mandel1(complex(x, y))
             data4.append(list(map(bind_mandel, xs)))

In [13]: plt.imshow(data4, interpolation='none')

Out[13]: <matplotlib.image.AxesImage at 0x7fdc74213c90>
```



That was a tiny bit slower.

So, what do we learn from this? Our mental image of what code should be faster or slower is often wrong, or doesn't make much difference. The only way to really improve code performance is empirically, through measurements.

10.3 NumPy for Performance

10.3.1 NumPy constructors

We saw previously that NumPy's core type is the `ndarray`, or N-Dimensional Array:

```
In [1]: import numpy as np
        np.zeros([3, 4, 2, 5])[2, :, :, 1]

Out[1]: array([[0., 0.],
               [0., 0.],
               [0., 0.],
               [0., 0.]])
```

The real magic of numpy arrays is that most python operations are applied, quickly, on an elementwise basis:

```
In [2]: x = np.arange(0, 256, 4).reshape(8, 8)

In [3]: y = np.zeros((8, 8))

In [4]: %%timeit
        for i in range(8):
            for j in range(8):
                y[i][j] = x[i][j] + 10
```

```
51.6 µs ± 926 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
In [5]: x + 10
```

```
Out[5]: array([[ 10,  14,  18,  22,  26,  30,  34,  38],
               [ 42,  46,  50,  54,  58,  62,  66,  70],
               [ 74,  78,  82,  86,  90,  94,  98, 102],
               [106, 110, 114, 118, 122, 126, 130, 134],
               [138, 142, 146, 150, 154, 158, 162, 166],
               [170, 174, 178, 182, 186, 190, 194, 198],
               [202, 206, 210, 214, 218, 222, 226, 230],
               [234, 238, 242, 246, 250, 254, 258, 262]])
```

Numpy's mathematical functions also happen this way, and are said to be "vectorized" functions.

```
In [6]: np.sqrt(x)
```

```
Out[6]: array([[ 0.        ,  2.        ,  2.82842712,  3.46410162,  4.        ,
                  4.47213595,  4.89897949,  5.29150262],
               [ 5.65685425,  6.        ,  6.32455532,  6.63324958,  6.92820323,
                  7.21110255,  7.48331477,  7.74596669],
               [ 8.        ,  8.24621125,  8.48528137,  8.71779789,  8.94427191,
                  9.16515139,  9.38083152,  9.59166305],
               [ 9.79795897, 10.        , 10.19803903, 10.39230485, 10.58300524,
                 10.77032961, 10.95445115, 11.13552873],
               [11.3137085 , 11.48912529, 11.66190379, 11.83215957, 12.        ,
                 12.16552506, 12.32882801, 12.489996  ],
               [12.64911064, 12.80624847, 12.9614814 , 13.11487705, 13.26649916,
                 13.41640786, 13.56465997, 13.7113092 ],
               [13.85640646, 14.        , 14.14213562, 14.28285686, 14.4222051 ,
                 14.56021978, 14.69693846, 14.83239697],
               [14.96662955, 15.09966887, 15.23154621, 15.3622915 , 15.49193338,
                 15.62049935, 15.74801575, 15.87450787]])
```

Numpy contains many useful functions for creating matrices. In our earlier lectures we've seen `linspace` and `arange` for evenly spaced numbers.

```
In [7]: np.linspace(0, 10, 21)
```

```
Out[7]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
                5.5,  6. ,  6.5,  7. ,  7.5,  8. ,  8.5,  9. ,  9.5, 10. ])
```

```
In [8]: np.arange(0, 10, 0.5)
```

```
Out[8]: array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. ,
               6.5, 7. , 7.5, 8. , 8.5, 9. , 9.5])
```

Here's one for creating matrices like coordinates in a grid:

```
In [9]: xmin = -1.5
        ymin = -1.0
        xmax = 0.5
        ymax = 1.0
        resolution = 300
        xstep = (xmax - xmin) / resolution
        ystep = (ymax - ymin) / resolution

        ymatrix, xmatrix = np.mgrid[ymin:ymax:ystep, xmin:xmax:xstep]
```

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

```
[[-1.         -1.         -1.         … -1.         -1.
  -1.        ]
 [-0.99333333 -0.99333333 -0.99333333 … -0.99333333 -0.99333333
  -0.99333333]
 [-0.98666667 -0.98666667 -0.98666667 … -0.98666667 -0.98666667
  -0.98666667]
 …
 [ 0.98        0.98        0.98       …  0.98        0.98
   0.98      ]
 [ 0.98666667  0.98666667  0.98666667 …  0.98666667  0.98666667
   0.98666667]
 [ 0.99333333  0.99333333  0.99333333 …  0.99333333  0.99333333
   0.99333333]]
```

We can add these together to make a grid containing the complex numbers we want to test for membership in the Mandelbrot set.

```
In [11]: values = xmatrix + 1j * ymatrix
```

```
In [12]: print(values)
```

```
[[-1.5       -1.j         -1.49333333-1.j         -1.48666667-1.j
  …  0.48      -1.j           0.48666667-1.j
   0.49333333-1.j        ]
 [-1.5       -0.99333333j -1.49333333-0.99333333j -1.48666667-0.99333333j
  …  0.48      -0.99333333j  0.48666667-0.99333333j
   0.49333333-0.99333333j]
 [-1.5       -0.98666667j -1.49333333-0.98666667j -1.48666667-0.98666667j
  …  0.48      -0.98666667j  0.48666667-0.98666667j
   0.49333333-0.98666667j]
 …
 [-1.5       +0.98j       -1.49333333+0.98j       -1.48666667+0.98j
  …  0.48      +0.98j         0.48666667+0.98j
   0.49333333+0.98j      ]
 [-1.5       +0.98666667j -1.49333333+0.98666667j -1.48666667+0.98666667j
  …  0.48      +0.98666667j  0.48666667+0.98666667j
   0.49333333+0.98666667j]
 [-1.5       +0.99333333j -1.49333333+0.99333333j -1.48666667+0.99333333j
  …  0.48      +0.99333333j  0.48666667+0.99333333j
   0.49333333+0.99333333j]]
```

10.3.2 Arraywise Algorithms

We can use this to apply the mandelbrot algorithm to whole *ARRAYS*

```
In [13]: z0 = values
         z1 = z0 * z0 + values
         z2 = z1 * z1 + values
         z3 = z2 * z2 + values
```

```
In [14]: print(z3)
```

```
[[24.06640625+20.75j       23.16610231+20.97899073j
  22.27540349+21.18465854j … 11.20523832 -1.88650846j
  11.5734533  -1.6076251j  11.94394738 -1.31225596j]
 [23.82102149+19.85687829j 22.94415031+20.09504528j
  22.07634812+20.31020645j … 10.93323949 -1.5275283j
  11.28531994 -1.24641067j 11.63928527 -0.94911594j]
 [23.56689029+18.98729242j 22.71312709+19.23410533j
  21.86791017+19.4582314j  … 10.65905064 -1.18433756j
  10.99529965 -0.90137318j 11.33305161 -0.60254144j]
 …
 [23.30453709-18.14090998j 22.47355537-18.39585192j
  21.65061048-18.62842771j … 10.38305264 +0.85663867j
  10.70377437 +0.57220289j 11.02562928 +0.27221042j]
 [23.56689029-18.98729242j 22.71312709-19.23410533j
  21.86791017-19.4582314j  … 10.65905064 +1.18433756j
  10.99529965 +0.90137318j 11.33305161 +0.60254144j]
 [23.82102149-19.85687829j 22.94415031-20.09504528j
  22.07634812-20.31020645j … 10.93323949 +1.5275283j
  11.28531994 +1.24641067j 11.63928527 +0.94911594j]]
```

So can we just apply our `mandel1` function to the whole matrix?

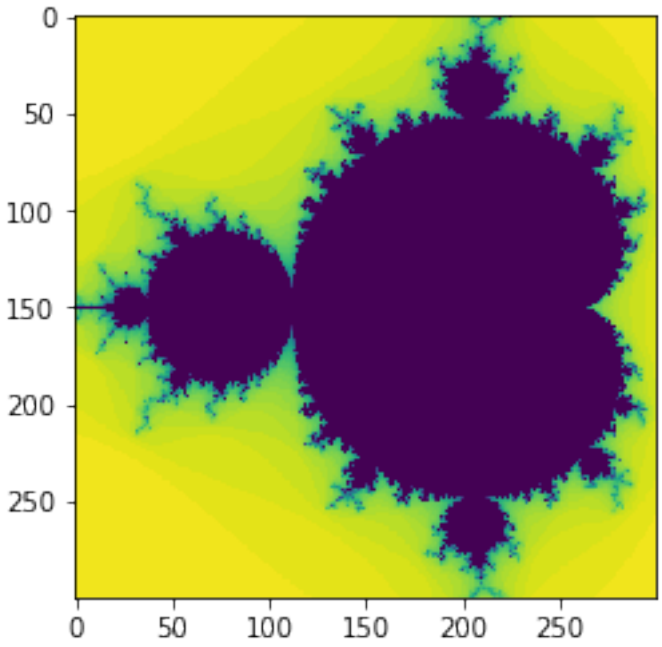
```
In [15]: def mandel1(position,limit=50):
             value = position
             while abs(value) < 2:
                 limit -= 1
                 value = value**2 + position
                 if limit < 0:
                     return 0
             return limit
```

```
In [16]: mandel1(values)
```

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

ValueError                                Traceback (most recent call last)

<ipython-input-16-484a82ca909a> in <module>
----> 1 mandel1(values)


<ipython-input-15-9fde680e0c52> in mandel1(position, limit)
      1 def mandel1(position,limit=50):
      2     value = position
----> 3     while abs(value) < 2:
      4         limit -= 1
      5         value = value**2 + position


ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or
```

No. The *logic* of our current routine would require stopping for some elements and not for others.
We can ask numpy to **vectorise** our method for us:

```
In [17]: mandel2 = np.vectorize(mandel1)

In [18]: data5 = mandel2(values)

In [19]: from matplotlib import pyplot as plt
         %matplotlib inline
         plt.imshow(data5, interpolation='none')

Out[19]: <matplotlib.image.AxesImage at 0x7f47fd61df10>
```



Is that any faster?

```
In [20]: %%timeit
         data5 = mandel2(values)

476 ms ± 2.92 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

This is not significantly faster. When we use *vectorize* it's just hiding an plain old python for loop under the hood. We want to make the loop over matrix elements take place in the "**C Layer**".

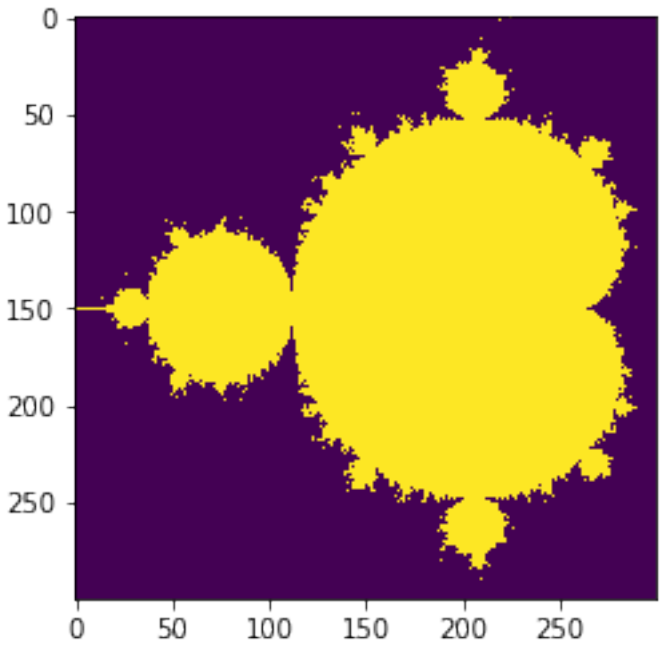
What if we just apply the Mandelbrot algorithm without checking for divergence until the end:

```
In [21]: def mandel_numpy_explode(position, limit=50):
             value = position
             while limit > 0:
                 limit -= 1
                 value = value**2 + position
                 diverging = abs(value) > 2


             return abs(value) < 2
```

```
In [22]: data6 = mandel_numpy_explode(values)

/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/ipykernel_launcher.py:5: RuntimeWarning
  """
/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/ipykernel_launcher.py:5: RuntimeWarning
  """
/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/ipykernel_launcher.py:6: RuntimeWarning

/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/ipykernel_launcher.py:6: RuntimeWarning

/home/travis/virtualenv/python3.7.5/lib/python3.7/site-packages/ipykernel_launcher.py:9: RuntimeWarning
  if __name__ == '__main__':
```

OK, we need to prevent it from running off to ∞

```
In [23]: def mandel_numpy(position, limit=50):
             value = position
             while limit > 0:
                 limit -= 1
                 value = value**2 + position
                 diverging = abs(value) > 2
                 # Avoid overflow
                 value[diverging] = 2

             return abs(value) < 2

In [24]: data6 = mandel_numpy(values)

In [25]: %%timeit

         data6 = mandel_numpy(values)

62 ms ± 382 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)


In [26]: from matplotlib import pyplot as plt
         %matplotlib inline
         plt.imshow(data6, interpolation='none')

Out[26]: <matplotlib.image.AxesImage at 0x7f4821be6e10>
```



Wow, that was TEN TIMES faster.
There's quite a few NumPy tricks there, let's remind ourselves of how they work:

```
In [27]: diverging = abs(z3) > 2
         z3[diverging] = 2
```

When we apply a logical condition to a NumPy array, we get a logical array.

```
In [28]: x = np.arange(10)
         y = np.ones([10]) * 5
         z = x > y

In [29]: x

Out[29]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [30]: y

Out[30]: array([5., 5., 5., 5., 5., 5., 5., 5., 5., 5.])

In [31]: print(z)

[False False False False False False  True  True  True  True]
```

Logical arrays can be used to index into arrays:

```
In [32]: x[x>3]

Out[32]: array([4, 5, 6, 7, 8, 9])

In [33]: x[np.logical_not(z)]
```

```
Out[33]: array([0, 1, 2, 3, 4, 5])
```

And you can use such an index as the target of an assignment:

```
In [34]: x[z] = 5
         x

Out[34]: array([0, 1, 2, 3, 4, 5, 5, 5, 5, 5])
```

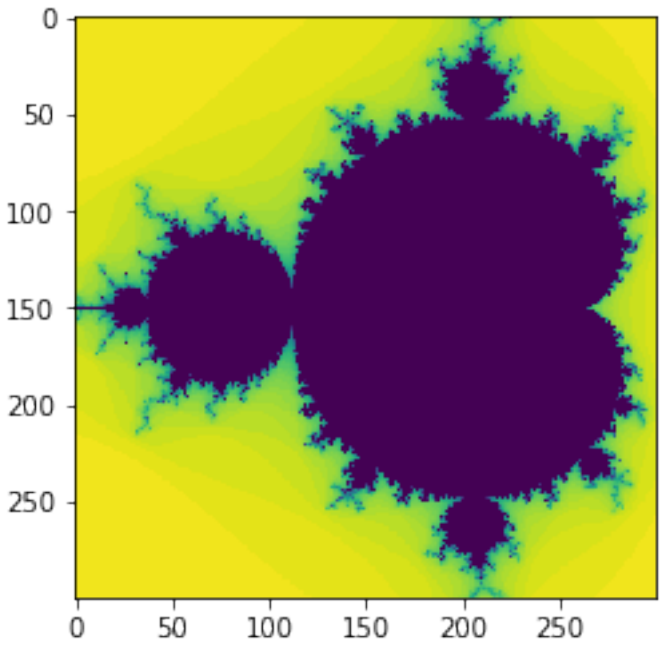
Note that we didn't compare two arrays to get our logical array, but an array to a scalar integer – this was broadcasting again.

10.3.3 More Mandelbrot

Of course, we didn't calculate the number-of-iterations-to-diverge, just whether the point was in the set.

Let's correct our code to do that:

```
In [35]: def mandel4(position,limit=50):
             value = position
             diverged_at_count = np.zeros(position.shape)
             while limit > 0:
                 limit -= 1
                 value = value**2 + position
                 diverging = abs(value) > 2
                 first_diverged_this_time = np.logical_and(diverging,
                                                           diverged_at_count == 0)
                 diverged_at_count[first_diverged_this_time] = limit
                 value[diverging] = 2

             return diverged_at_count

In [36]: data7 = mandel4(values)

In [37]: plt.imshow(data7, interpolation='none')

Out[37]: <matplotlib.image.AxesImage at 0x7f47f9df5c50>
```



```
In [38]: %%timeit

         data7 = mandel4(values)
```

```
65.4 ms ± 753 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Note that here, all the looping over mandelbrot steps was in Python, but everything below the loop-over-positions happened in C. The code was amazingly quick compared to pure Python.

Can we do better by avoiding a square root?

```
In [39]: def mandel5(position, limit=50):
             value = position
             diverged_at_count = np.zeros(position.shape)
             while limit > 0:
                 limit -= 1
                 value = value**2 + position
                 diverging = value * np.conj(value) > 4
                 first_diverged_this_time = np.logical_and(diverging, diverged_at_count == 0)
                 diverged_at_count[first_diverged_this_time] = limit
                 value[diverging] = 2

             return diverged_at_count
```

```
In [40]: %%timeit

         data8 = mandel5(values)
```

```
44.3 ms ± 303 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Probably not worth the time I spent thinking about it!

10.3.4 NumPy Testing

Now, let's look at calculating those residuals, the differences between the different datasets.

```
In [41]: data8 = mandel5(values)
         data5 = mandel2(values)
```

```
In [42]: np.sum((data8 - data5)**2)
```

```
Out[42]: 0.0
```

For our non-numpy datasets, numpy knows to turn them into arrays:

```
In [43]: xmin = -1.5
         ymin = -1.0
         xmax = 0.5
         ymax = 1.0
         resolution = 300
         xstep = (xmax-xmin)/resolution
         ystep = (ymax-ymin)/resolution
         xs = [(xmin + (xmax - xmin) * i / resolution) for i in range(resolution)]
         ys = [(ymin + (ymax - ymin) * i / resolution) for i in range(resolution)]
         data1 = [[mandel1(complex(x, y)) for x in xs] for y in ys]
         sum(sum((data1 - data7)**2))
```

```
Out[43]: 0.0
```

But this doesn't work for pure non-numpy arrays

```
In [44]: data2 = []
         for y in ys:
             row = []
             for x in xs:
                 row.append(mandel1(complex(x, y)))
             data2.append(row)

In [45]: data2 - data1


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

         TypeError                                 Traceback (most recent call last)

         <ipython-input-45-b9ae9db328ea> in <module>
     ----> 1 data2 - data1


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

So we have to convert to NumPy arrays explicitly:

```
In [46]: sum(sum((np.array(data2) - np.array(data1))**2))

Out[46]: 0
```

NumPy provides some convenient assertions to help us write unit tests with NumPy arrays:

```
In [47]: x = [1e-5, 1e-3, 1e-1]
         y = np.arccos(np.cos(x))
         y

Out[47]: array([1.00000004e-05, 1.00000000e-03, 1.00000000e-01])

In [48]: np.testing.assert_allclose(x, y, rtol=1e-6, atol=1e-20)

In [49]: np.testing.assert_allclose(data7, data1)
```

10.3.5 Arraywise operations are fast

Note that we might worry that we carry on calculating the mandelbrot values for points that have already diverged.

```
In [50]: def mandel6(position, limit=50):
             value = np.zeros(position.shape) + position
             calculating = np.ones(position.shape, dtype='bool')
             diverged_at_count = np.zeros(position.shape)
             while limit > 0:
                 limit -= 1
                 value[calculating] = value[calculating]**2 + position[calculating]
                 diverging_now = np.zeros(position.shape, dtype='bool')
                 diverging_now[calculating] = value[calculating] * \
```

```
                                        np.conj(value[calculating])>4
                calculating = np.logical_and(calculating,
                                        np.logical_not(diverging_now))
                diverged_at_count[diverging_now] = limit

            return diverged_at_count
```

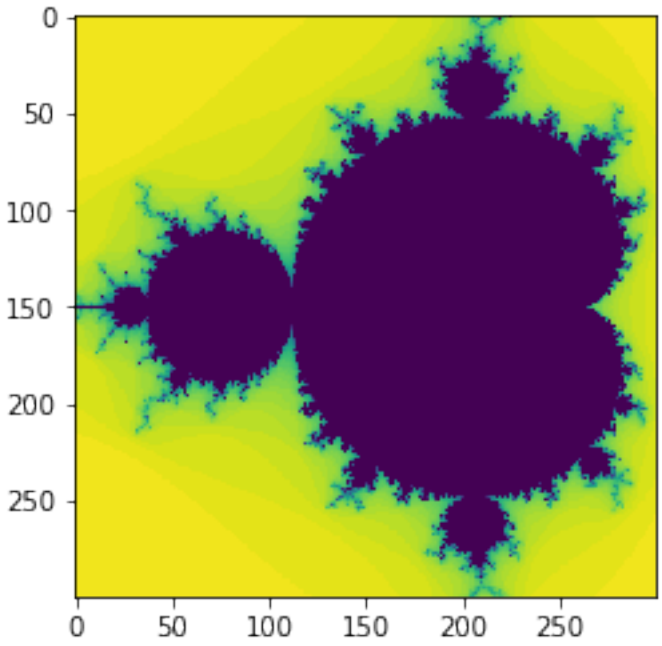
```
In [51]: data8 = mandel6(values)
```

```
In [52]: %%timeit

         data8 = mandel6(values)
```

```
58.8 ms ± 530 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [53]: plt.imshow(data8, interpolation='none')
```

```
Out[53]: <matplotlib.image.AxesImage at 0x7f47f9dbc390>
```



This was **not faster** even though it was **doing less work**

This often happens: on modern computers, **branches** (if statements, function calls) and **memory access** is usually the rate-determining step, not maths.

Complicating your logic to avoid calculations sometimes therefore slows you down. The only way to know is to **measure**

10.3.6 Indexing with arrays

We've been using Boolean arrays a lot to get access to some elements of an array. We can also do this with integers:

```
In [54]: x = np.arange(64)
         y = x.reshape([8,8])
         y

Out[54]: 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]])

In [55]: y[[2, 5]]

Out[55]: array([[16, 17, 18, 19, 20, 21, 22, 23],
                [40, 41, 42, 43, 44, 45, 46, 47]])

In [56]: y[[0, 2, 5], [1, 2, 7]]

Out[56]: array([ 1, 18, 47])
```

We can use a : to indicate we want all the values from a particular axis:

```
In [57]: y[0:4:2, [0, 2]]

Out[57]: array([[ 0,  2],
                [16, 18]])
```

We can mix array selectors, boolean selectors, :s and ordinary array seqeuencers:

```
In [58]: z = x.reshape([4, 4, 4])
         z

Out[58]: 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]]])

In [59]: z[:, [1, 3], 0:3]
```

```
Out[59]: array([[[ 4,  5,  6],
                 [12, 13, 14]],

                [[20, 21, 22],
                 [28, 29, 30]],

                [[36, 37, 38],
                 [44, 45, 46]],

                [[52, 53, 54],
                 [60, 61, 62]]])
```

We can manipulate shapes by adding new indices in selectors with np.newaxis:

```
In [60]: z[:, np.newaxis, [1, 3], 0].shape

Out[60]: (4, 1, 2)
```

When we use basic indexing with integers and : expressions, we get a **view** on the matrix so a copy is avoided:

```
In [61]: a = z[:, :, 2]
         a[0, 0] = -500
         z

Out[61]: array([[[   0,    1, -500,    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]]])
```

We can also use … to specify ": for as many as possible intervening axes":

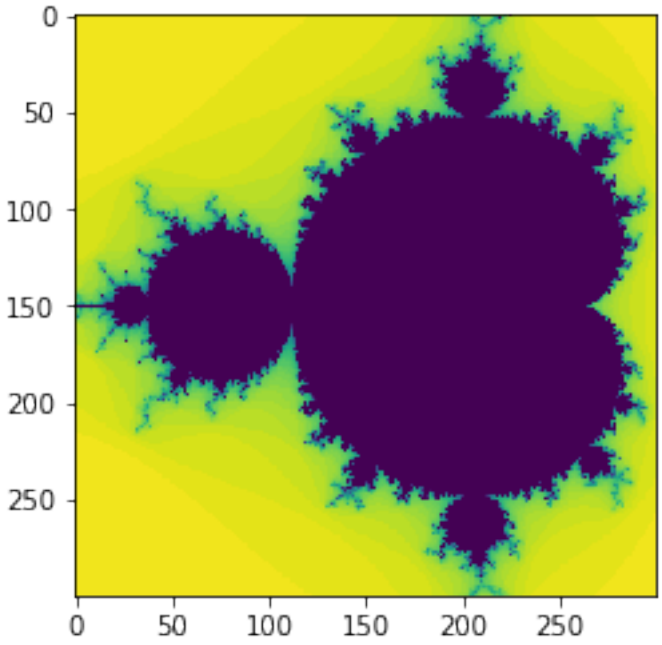
```
In [62]: z[1]

Out[62]: array([[16, 17, 18, 19],
                [20, 21, 22, 23],
                [24, 25, 26, 27],
                [28, 29, 30, 31]])

In [63]: z[...,2]
```

```
Out[63]: array([[-500,    6,   10,   14],
                [  18,   22,   26,   30],
                [  34,   38,   42,   46],
                [  50,   54,   58,   62]])
```

However, boolean mask indexing and array filter indexing always causes a copy.

Let's try again at avoiding doing unnecessary work by using new arrays containing the reduced data instead of a mask:

```
In [64]: def mandel7(position, limit=50):
             positions = np.zeros(position.shape) + position
             value = np.zeros(position.shape) + position
             indices = np.mgrid[0:values.shape[0], 0:values.shape[1]]
             diverged_at_count = np.zeros(position.shape)
             while limit > 0:
                 limit -= 1
                 value = value**2 + positions
                 diverging_now = value * np.conj(value) > 4
                 diverging_now_indices = indices[:, diverging_now]
                 carry_on = np.logical_not(diverging_now)

                 value = value[carry_on]
                 indices = indices[:, carry_on]
                 positions = positions[carry_on]
                 diverged_at_count[diverging_now_indices[0,:],
                                   diverging_now_indices[1,:]] = limit

             return diverged_at_count

In [65]: data9 = mandel7(values)

In [66]: plt.imshow(data9, interpolation='none')

Out[66]: <matplotlib.image.AxesImage at 0x7f47f9cf3710>
```



```
In [67]: %%timeit

         data9 = mandel7(values)
```

```
65.7 ms ± 441 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Still slower. Probably due to lots of copies – the point here is that you need to *experiment* to see which optimisations will work. Performance programming needs to be empirical.

10.4 Profiling

We've seen how to compare different functions by the time they take to run. However, we haven't obtained much information about where the code is spending more time. For that we need to use a profiler. IPython offers a profiler through the `%prun` magic. Let's use it to see how it works:

```
In [68]: %prun mandel7(values)
```

`%prun` shows a line per each function call ordered by the total time spent on each of these. However, sometimes a line-by-line output may be more helpful. For that we can use the `line_profiler` package (you need to install it using `pip`). Once installed you can activate it in any notebook by running:

```
In [69]: %load_ext line_profiler
```

And the `%lprun` magic should be now available:

```
In [70]: %lprun -f mandel7 mandel7(values)
```

Here, it is clearer to see which operations are keeping the code busy.

10.5 Cython

Cython can be viewed as an extension of Python where variables and functions are annotated with extra information, in particular types. The resulting Cython source code will be compiled into optimized C or C++ code, and thereby yielding substantial speed-up of slow Python code. In other words, Cython provides a way of writing Python with comparable performance to that of C/C++.

10.5.1 Start Coding in Cython

Cython code must, unlike Python, be compiled. This happens in the following stages:

- The cython code in `.pyx` file will be translated to a `C` file.
- The `C` file will be compiled by a C compiler into a shared library, which will be directly loaded into Python.

In a Jupyter notebook, everything is a lot easier. One needs only to load the Cython extension (`%load_ext Cython`) at the beginning and put `%%cython` mark in front of cells of Cython code. Cells with Cython mark will be treated as a `.pyx` code and consequently, compiled into C.

For details, please see Building Cython Code.

Pure python Mandelbrot set:

```
In [1]: xmin = -1.5
        ymin = -1.0
        xmax = 0.5
        ymax = 1.0
        resolution = 300
        xstep = (xmax - xmin) / resolution
        ystep = (ymax - ymin) / resolution
        xs = [(xmin + (xmax - xmin) * i / resolution) for i in range(resolution)]
        ys = [(ymin + (ymax - ymin) * i / resolution) for i in range(resolution)]
```

```
In [2]: def mandel(position, limit=50):
            value = position
            while abs(value) < 2:
                limit -= 1
                value = value**2 + position
                if limit < 0:
                    return 0
            return limit
```

Compiled by Cython:

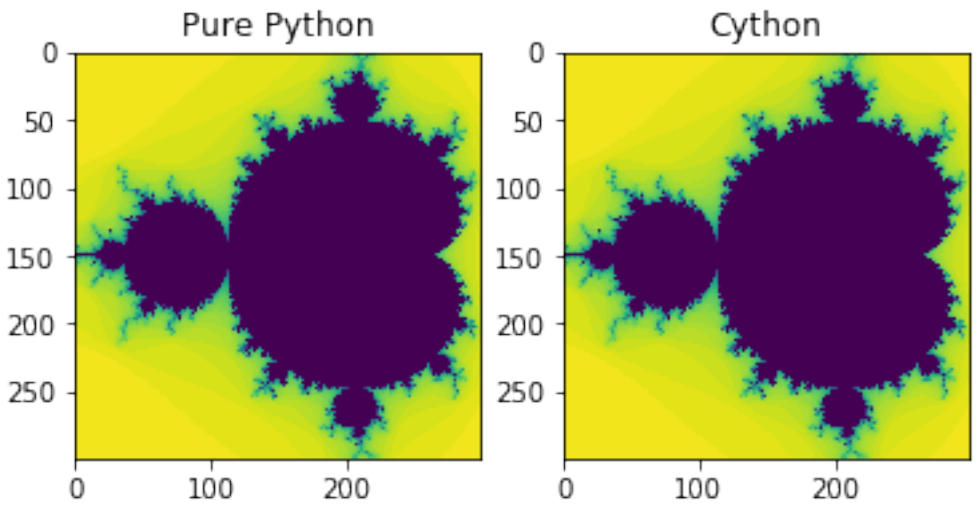
```
In [3]: %load_ext Cython
```

```
In [4]: %%cython

        def mandel_cython(position, limit=50):
            value = position
            while abs(value) < 2:
                limit -= 1
                value = value**2 + position
                if limit < 0:
                    return 0
            return limit
```

Let's verify the result

```
In [5]: from matplotlib import pyplot as plt
        %matplotlib inline
        f, axarr = plt.subplots(1, 2)
        axarr[0].imshow([[mandel(complex(x, y)) for x in xs] for y in ys], interpolation='none')
        axarr[0].set_title('Pure Python')
        axarr[1].imshow([[mandel_cython(complex(x, y)) for x in xs] for y in ys], interpolation='none')
        axarr[1].set_title('Cython')
```

```
Out[5]: Text(0.5, 1.0, 'Cython')
```




```
In [6]: %timeit [[mandel(complex(x,y)) for x in xs] for y in ys] # pure python
        %timeit [[mandel_cython(complex(x,y)) for x in xs] for y in ys] # cython
```

```
537 ms ± 5.64 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
303 ms ± 6.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

We have improved the performance of a factor of 1.5 by just using the Cython compiler, **without changing the code**!

10.5.2 Cython with C Types

But we can do better by telling Cython what C data type we would use in the code. Note we're not actually writing C, we're writing Python with C types.

typed variable

```
In [7]: %%cython
        def var_typed_mandel_cython(position, limit=50):
            cdef double complex value # typed variable
            value = position
            while abs(value) < 2:
                limit -= 1
                value = value**2 + position
                if limit < 0:
                    return 0
            return limit
```

typed function + typed variable

```
In [8]: %%cython
        cpdef call_typed_mandel_cython(double complex position,
                                       int limit=50): # typed function
            cdef double complex value # typed variable
            value = position
            while abs(value)<2:
```

```
            limit -= 1
            value = value**2 + position
            if limit < 0:
                return 0
        return limit
```

performance of one number:

```
In [9]: # pure python
        %timeit a = mandel(complex(0, 0))
```

```
10.5 µs ± 59.4 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [10]: # primitive cython
         %timeit a = mandel_cython(complex(0, 0))
```

```
5.95 µs ± 67.6 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [11]: # cython with C type variable
         %timeit a = var_typed_mandel_cython(complex(0, 0))
```

```
2.97 µs ± 5.62 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
In [12]: # cython with typed variable + function
         %timeit a = call_typed_mandel_cython(complex(0, 0))
```

```
656 ns ± 3.89 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

10.5.3 Cython with numpy ndarray

You can use NumPy from Cython exactly the same as in regular Python, but by doing so you are losing potentially high speedups because Cython has support for fast access to NumPy arrays.

```
In [13]: import numpy as np
         ymatrix, xmatrix = np.mgrid[ymin:ymax:ystep, xmin:xmax:xstep]
         values = xmatrix + 1j * ymatrix
```

```
In [14]: %%cython
         import numpy as np
         cimport numpy as np

         cpdef numpy_cython_1(np.ndarray[double complex, ndim=2] position,
                              int limit=50):
             cdef np.ndarray[long,ndim=2] diverged_at
             cdef double complex value
             cdef int xlim
             cdef int ylim
             cdef double complex pos
             cdef int steps
             cdef int x, y

             xlim = position.shape[1]
             ylim = position.shape[0]
```

```
    diverged_at = np.zeros([ylim, xlim], dtype=int)
    for x in xrange(xlim):
        for y in xrange(ylim):
            steps = limit
            value = position[y,x]
            pos = position[y,x]
            while abs(value) < 2 and steps >= 0:
                steps -= 1
                value = value**2 + pos
            diverged_at[y,x] = steps

    return diverged_at
```

Note the double import of numpy: the standard numpy module and a Cython-enabled version of numpy that ensures fast indexing of and other operations on arrays. Both import statements are necessary in code that uses numpy arrays. The new thing in the code above is declaration of arrays by np.ndarray.

```
In [15]: %timeit data_cy = [[mandel(complex(x,y)) for x in xs] for y in ys] # pure python
```

```
513 ms ± 1.04 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [16]: %timeit data_cy = [[call_typed_mandel_cython(complex(x,y)) for x in xs] for y in ys] # typed cy
```

```
46.6 ms ± 135 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [17]: %timeit numpy_cython_1(values) # ndarray
```

```
27.8 ms ± 226 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

A trick of using `np.vectorize`

```
In [18]: numpy_cython_2 = np.vectorize(call_typed_mandel_cython)
```

```
In [19]: %timeit numpy_cython_2(values) #  vectorize
```

```
33.2 ms ± 61.9 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

10.5.4 Calling C functions from Cython

Example: compare `sin()` from Python and C library

```
In [20]: %%cython
         import math
         cpdef py_sin():
             cdef int x
             cdef double y
             for x in range(1e7):
                 y = math.sin(x)
```

```
In [21]: %%cython
         from libc.math cimport sin as csin # import from C library
         cpdef c_sin():
             cdef int x
             cdef double y
             for x in range(1e7):
                 y = csin(x)
```

```
In [22]: %timeit [math.sin(i) for i in range(int(1e7))] # python

1.75 s ± 15.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)


In [23]: %timeit py_sin()                                  # cython call python library

917 ms ± 18.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)


In [24]: %timeit c_sin()                                   # cython call C library

4.53 ms ± 2.24 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

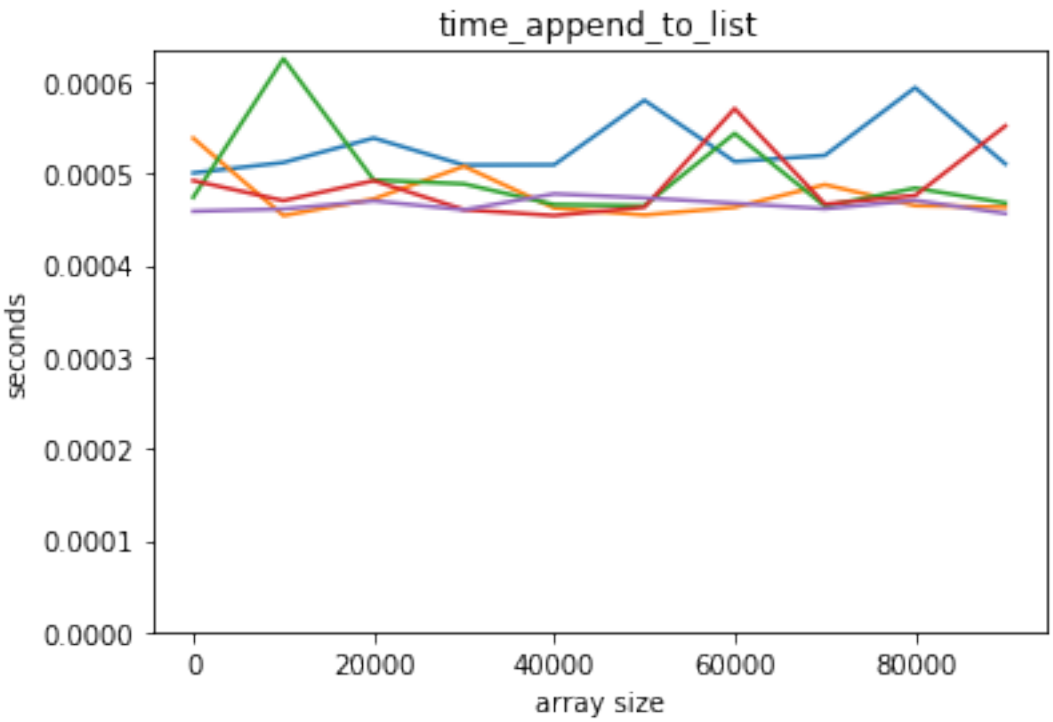
10.6 Scaling for containers and algorithms

We've seen that NumPy arrays are really useful. Why wouldn't we always want to use them for data which is all the same type?

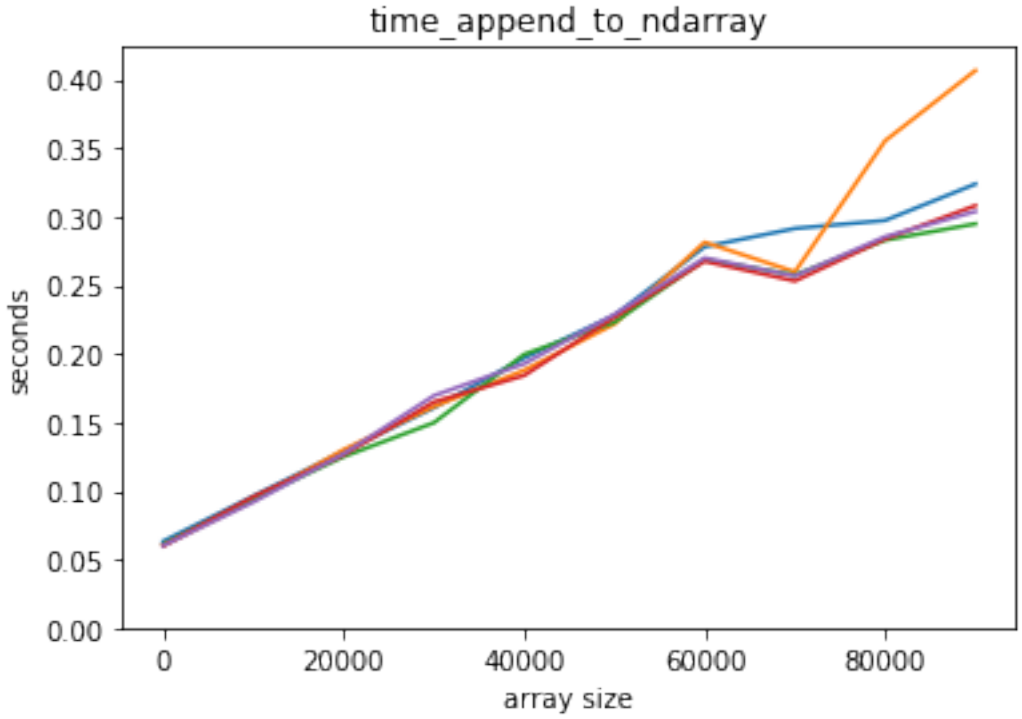
```
In [1]: import numpy as np
        from timeit import repeat
        from matplotlib import pyplot as plt
        %matplotlib inline
```

Let's look at appending data into a NumPy array, compared to a plain Python list:

```
In [2]: def time_append_to_ndarray(count):
            # the function repeat does the same that the `%timeit` magic
            # but as a function; so we can plot it.
            return repeat('np.append(before, [0])',
                          f'import numpy as np; before=np.ndarray({count})',
                          number=10000)

In [3]: def time_append_to_list(count):
            return repeat('before.append(0)',
                          f'before = [0] * {count}',
                          number=10000)

In [4]: counts = np.arange(1, 100000, 10000)

        def plot_time(function, counts, title=None):
            plt.plot(counts, list(map(function, counts)))
            plt.ylim(bottom=0)
            plt.ylabel('seconds')
            plt.xlabel('array size')
            plt.title(title or function.__name__)

In [5]: plot_time(time_append_to_list, counts)
```



In [6]: `plot_time(time_append_to_ndarray, counts)`



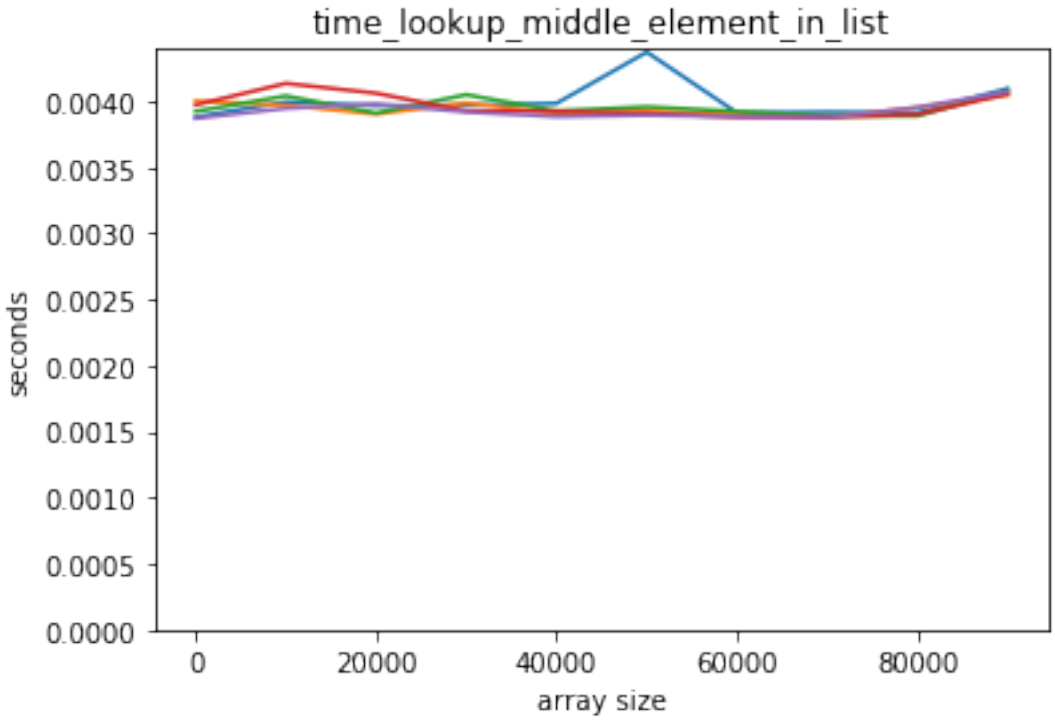
Adding an element to a Python list is way faster! Also, it seems that adding an element to a Python list is independent of the length of the list, but it's not so for a NumPy array.

How do they perform when accessing an element in the middle?

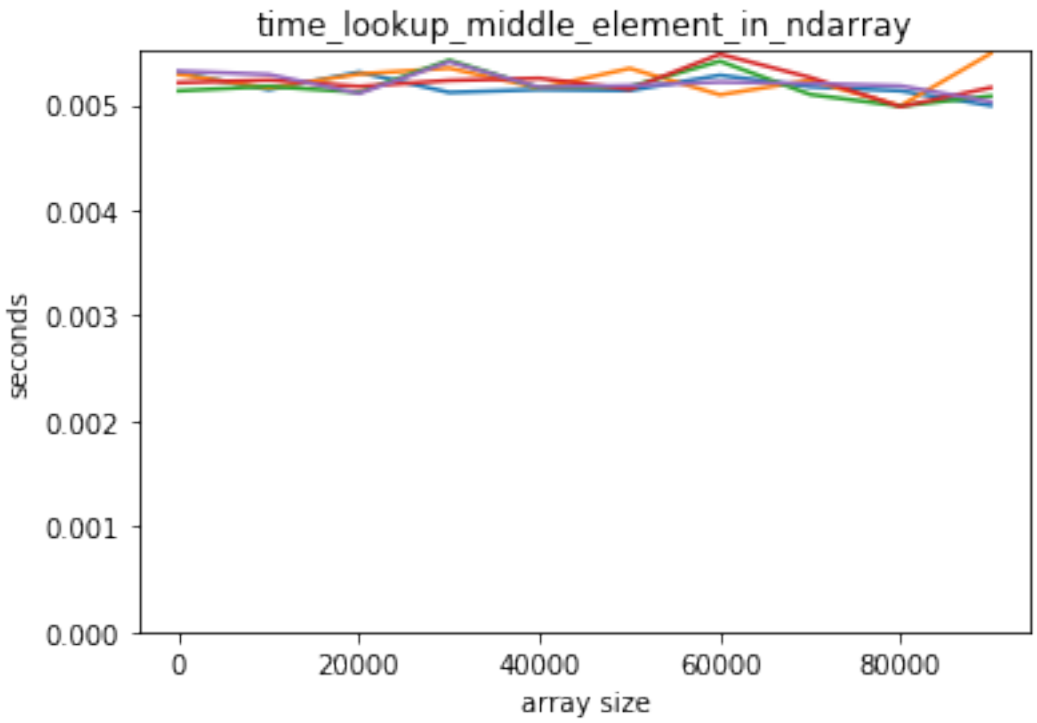
```
In [7]: def time_lookup_middle_element_in_list(count):
            before = [0] * count
            def totime():
                x = before[count // 2]
            return repeat(totime, number=10000)
```

```
In [8]: def time_lookup_middle_element_in_ndarray(count):
            before = np.ndarray(count)
            def totime():
                x = before[count // 2]
            return repeat(totime, number=10000)
```

```
In [9]: plot_time(time_lookup_middle_element_in_list, counts)
```




```
In [10]: plot_time(time_lookup_middle_element_in_ndarray, counts)
```

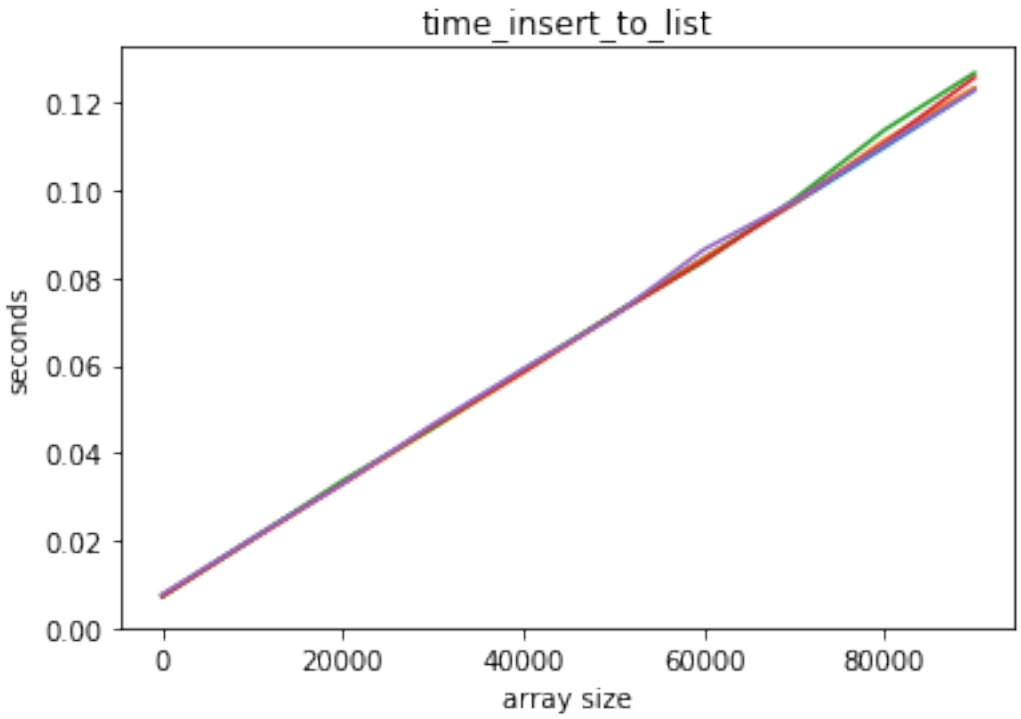


Both scale well for accessing the middle element.

What about inserting at the beginning?

If we want to insert an element at the beginning of a Python list we can do:

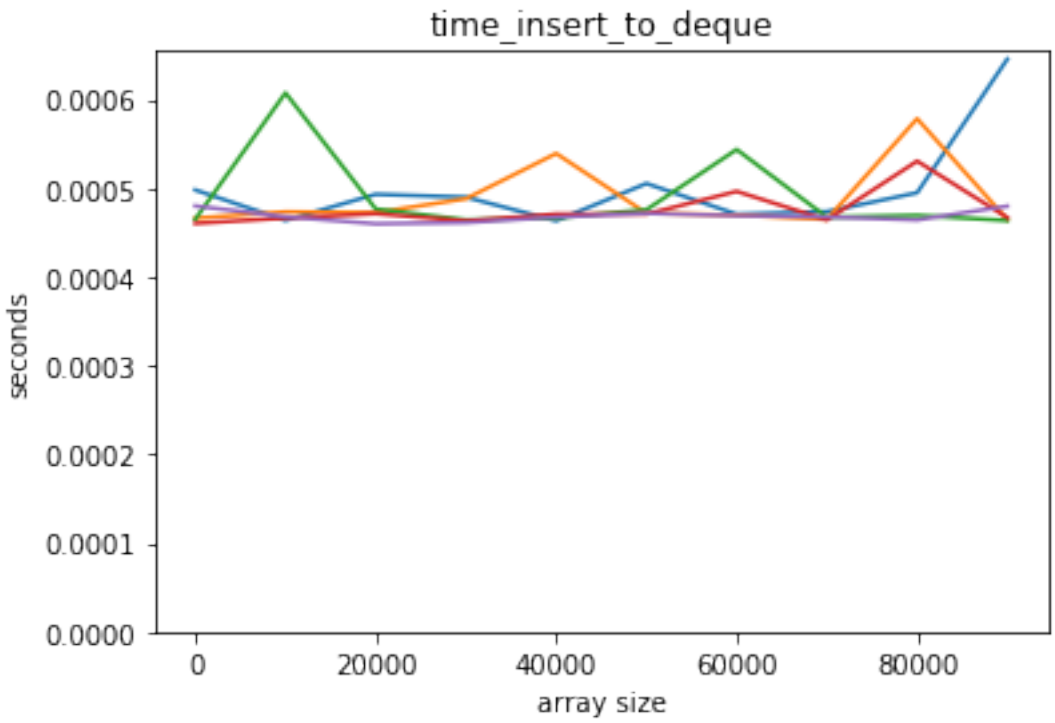
```
In [11]: x = list(range(5))
         x

Out[11]: [0, 1, 2, 3, 4]

In [12]: x[0:0] = [-1]
         x

Out[12]: [-1, 0, 1, 2, 3, 4]

In [13]: def time_insert_to_list(count):
             return repeat('before[0:0] = [0]',
                           f'before = [0] * {count}',number=10000)

In [14]: plot_time(time_insert_to_list, counts)
```



`list` performs **badly** for insertions at the beginning!

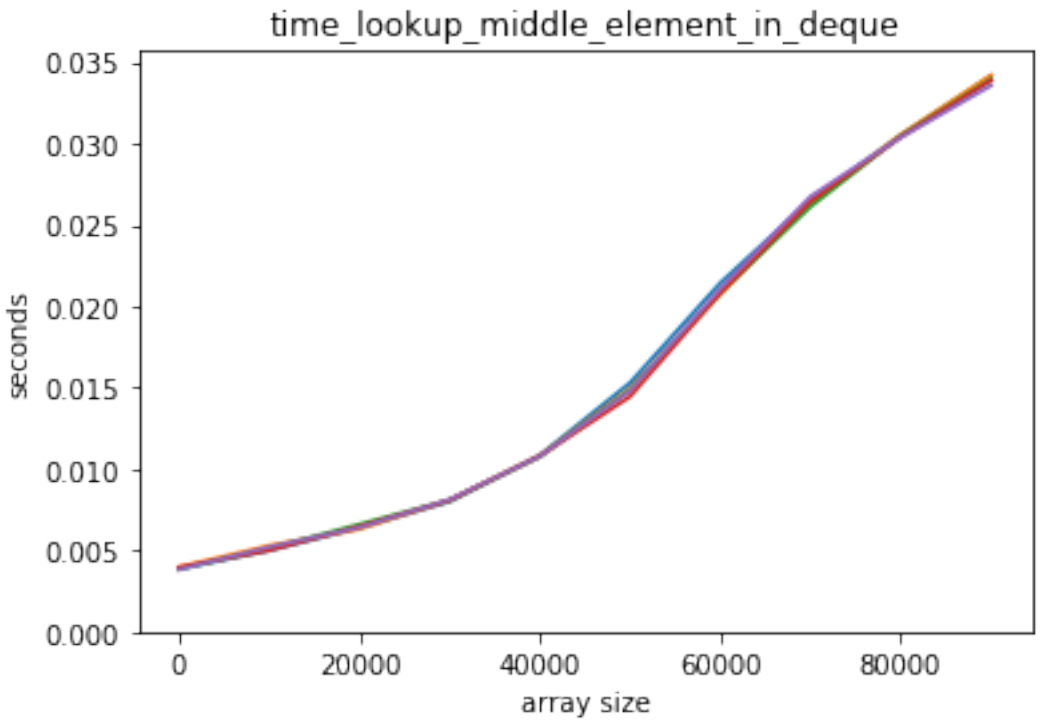
There are containers in Python that work well for insertion at the start:

```
In [15]: from collections import deque

In [16]: def time_insert_to_deque(count):
             return repeat('before.appendleft(0)',
                           f'from collections import deque; before = deque([0] * {count})',
                           number=10000)

In [17]: plot_time(time_insert_to_deque, counts)
```



But looking up in the middle scales badly:

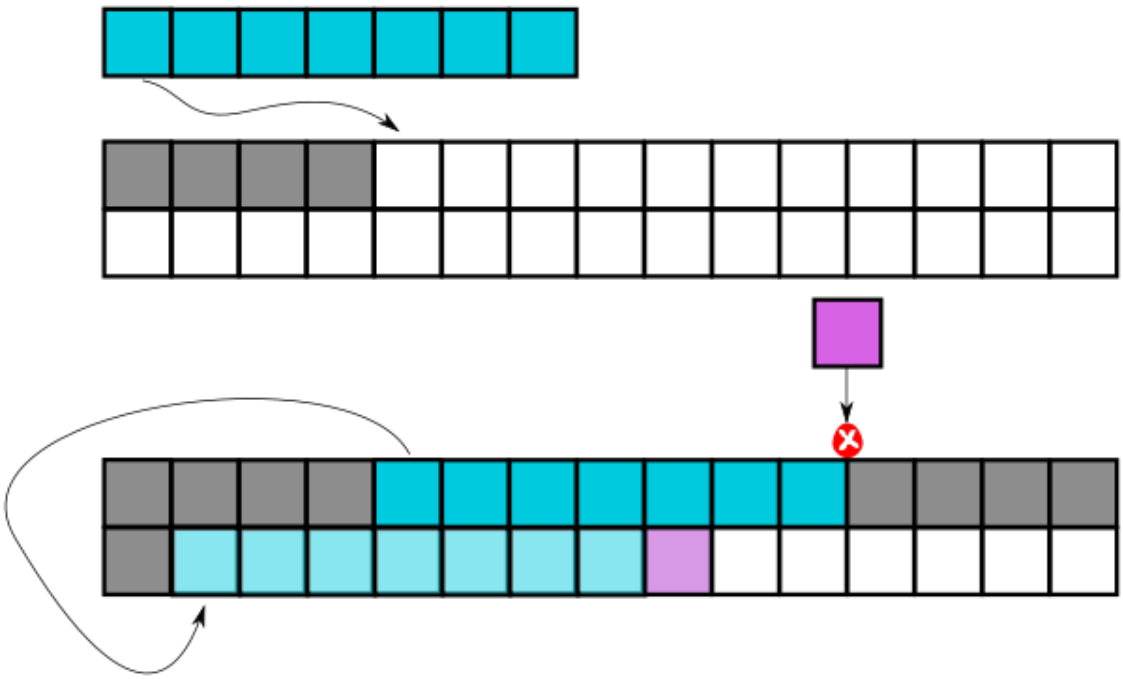
```
In [18]: def time_lookup_middle_element_in_deque(count):
             before = deque([0] * count)
             def totime():
                 x = before[count // 2]
             return repeat(totime, number=10000)

In [19]: plot_time(time_lookup_middle_element_in_deque, counts)
```



What is going on here?

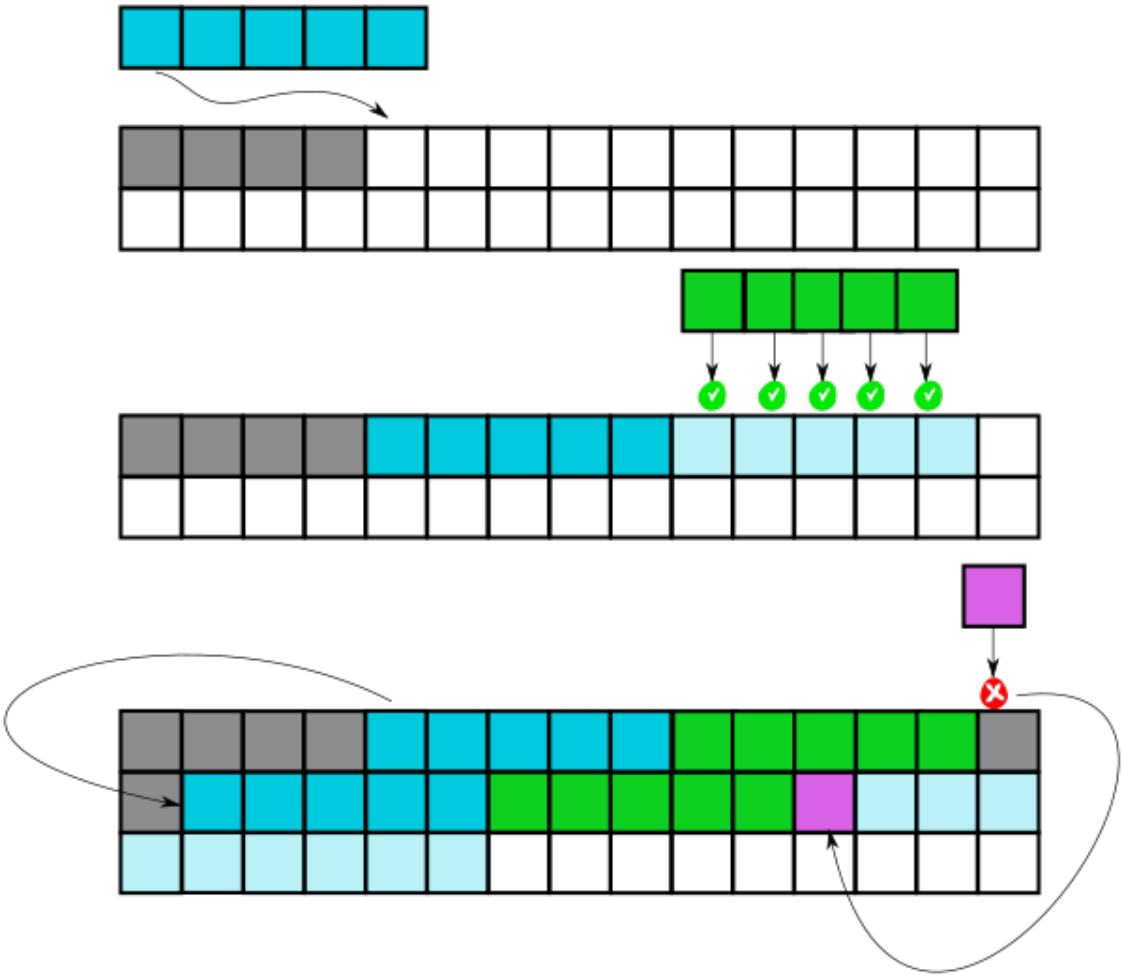
Arrays are stored as contiguous memory. Anything which changes the length of the array requires the whole array to be copied elsewhere in memory.

This copy takes time proportional to the array size.



The Python `list` type is **also** an array, but it is allocated with **extra memory**. Only when that memory

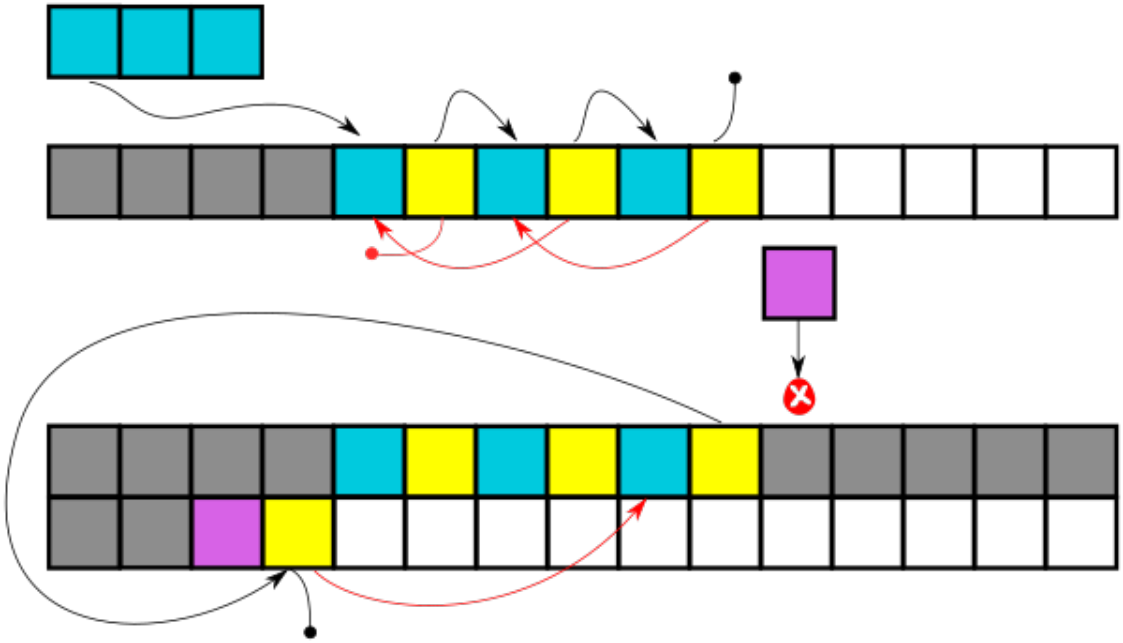
is exhausted is a copy needed.



If the extra memory is typically the size of the current array, a copy is needed every 1/N appends, and costs N to make, so **on average** copies are cheap. We call this **amortized constant time**.

This makes it fast to look up values in the middle. However, it may also use more space than is needed.

The deque type works differently: each element contains a pointer to the next. Inserting elements is therefore very cheap, but looking up the Nth element requires traversing N such pointers.



10.6.1 Dictionary performance

For another example, let's consider the performance of a dictionary versus a couple of other ways in which we could implement an associative array.

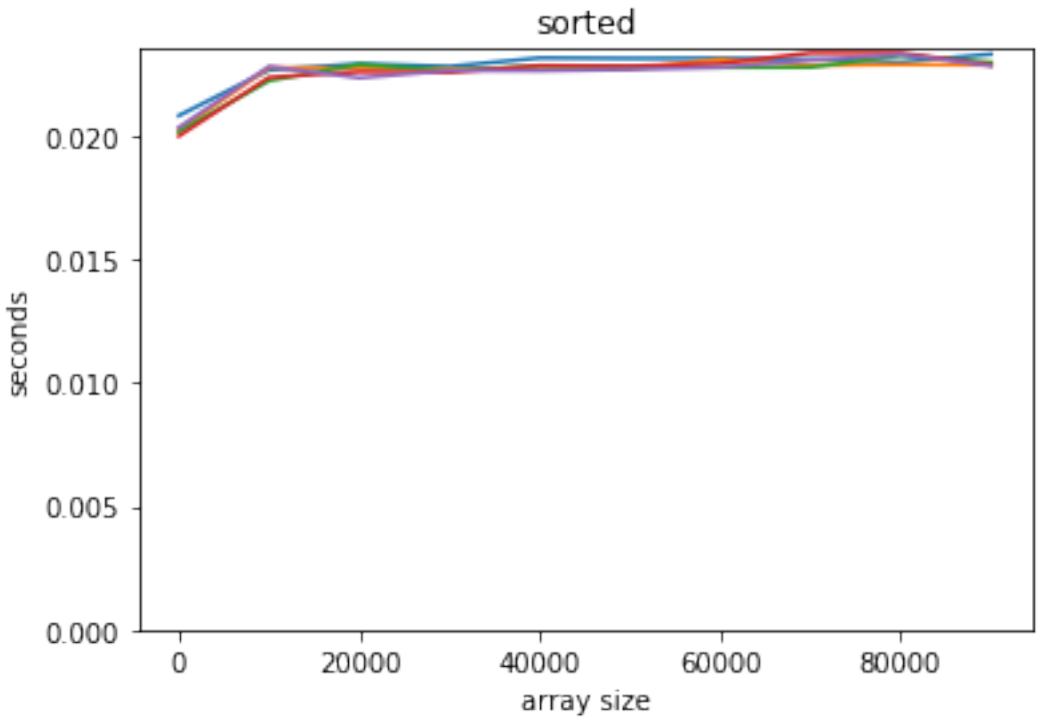
```
In [20]: class evildict:
             def __init__(self, data):
                 self.data = data

             def __getitem__(self, akey):
                 for key, value in self.data:
                     if key == akey:
                         return value
                 raise KeyError()
```

If we have an evil dictionary of N elements, how long would it take - on average - to find an element?

```
In [21]: eric = [["Name", "Eric Idle"], ["Job", "Comedian"], ["Home", "London"]]

In [22]: eric_evil = evildict(eric)

In [23]: eric_evil["Job"]

Out[23]: 'Comedian'

In [24]: eric_dict = dict(eric)

In [25]: eric_evil["Job"]

Out[25]: 'Comedian'

In [26]: x = ["Hello", "License", "Fish", "Eric", "Pet", "Halibut"]

In [27]: sorted(x, key=lambda el: el.lower())

Out[27]: ['Eric', 'Fish', 'Halibut', 'Hello', 'License', 'Pet']
```

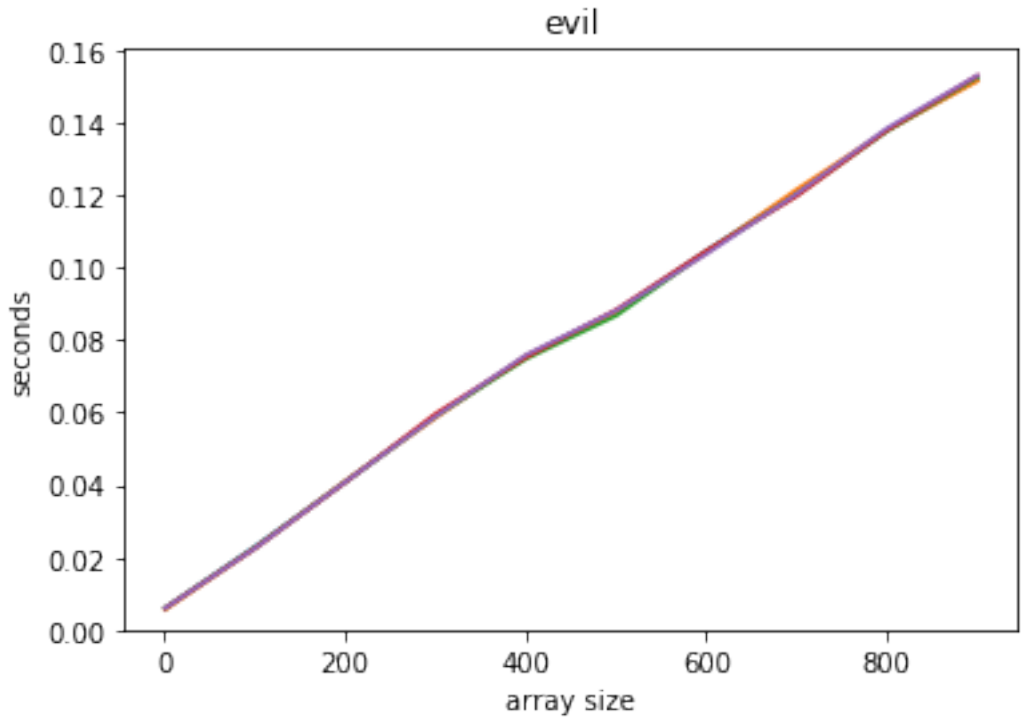
What if we created a dictionary where we bisect the search?

```
In [28]: class sorteddict:
             def __init__(self, data):
                 self.data = sorted(data, key = lambda x:x[0])
                 self.keys = list(map(lambda x:x[0], self.data))

             def __getitem__(self,akey):
                 from bisect import bisect_left
                 loc = bisect_left(self.keys, akey)

                 if loc != len(self.data):
                     return self.data[loc][1]

                 raise KeyError()

In [29]: eric_sorted = sorteddict(eric)

In [30]: eric_sorted["Job"]

Out[30]: 'Comedian'

In [31]: def time_dict_generic(ttype, count, number=10000):
             from random import randrange
             keys = list(range(count))
             values = [0] * count
             data = ttype(list(zip(keys, values)))
             def totime():
                 x = data[keys[count // 2]]
             return repeat(totime, number=10000)

In [32]: time_dict = lambda count: time_dict_generic(dict, count)
         time_sorted = lambda count: time_dict_generic(sorteddict, count)
         time_evil = lambda count: time_dict_generic(evildict, count)

In [33]: plot_time(time_sorted, counts, title='sorted')
```



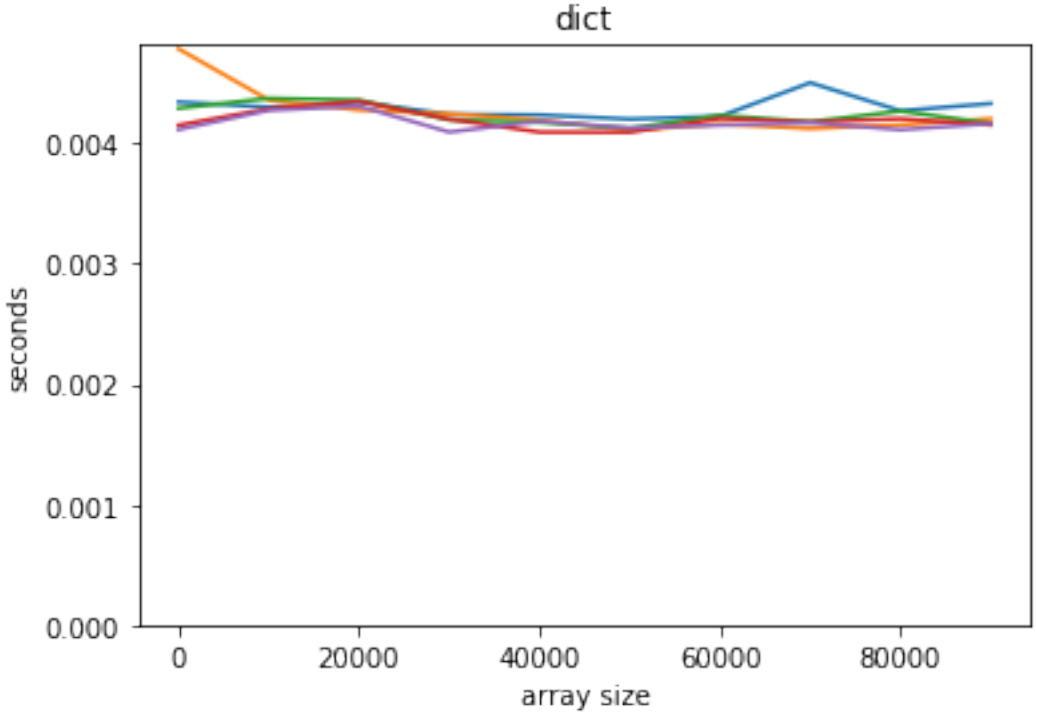
We can't really see what's going on here for the sorted example as there's too much noise, but theoretically we should get **logarithmic** asymptotic performance. We write this down as $O(\ln N)$. This doesn't mean there isn't also a constant term, or a term proportional to something that grows slower (such as $\ln(\ln N)$): we always write down just the term that is dominant for large N. We saw before that `list` is $O(1)$ for appends, $O(N)$ for inserts. Numpy's `array` is $O(N)$ for appends.

```
In [34]: counts = np.arange(1, 1000, 100)
         plot_time(time_evil, counts, title='evil')
```



The simple check-each-in-turn solution is $O(N)$ - linear time.

```
In [35]: counts = np.arange(1, 100000, 10000)
         plot_time(time_dict, counts, title='dict')
```



Python's built-in dictionary is, amazingly, O(1): the time is **independent** of the size of the dictionary.

This uses a miracle of programming called the *Hash Table*: you can learn more about these issues at this video from Harvard University. This material is pretty advanced, but, I think, really interesting!

Optional exercise: determine what the asymptotic peformance for the Boids model in terms of the number of Boids. Make graphs to support this. Bonus: how would the performance scale with the number of dimensions?

Chapter 11

An Adventure In Packaging: An exercise in research software engineering.

In this exercise, you will convert the already provided solution to the programming challenge defined in this Jupyter notebook, into a proper Python package.

The code to actually solve the problem is already given, but as roughly sketched out code in a notebook.

Your job will be to convert the code into a formally structured package, with unit tests, a command line interface, and demonstrating your ability to use `git` version control.

The exercise will be semi-automatically marked, so it is *very* important that you adhere in your solution to the correct file and folder structure, as defined in the rubric below. An otherwise valid solution which doesn't work with our marking tool will **not** be given credit.

First, we set out the problem we are solving, and it's informal solution. Next, we specify in detail the target for your tidy solution. Finally, to assist you in creating a good solution, we state the marks scheme we will use.

Chapter 12

Treasure Hunting for Beginners: an AI testbed

We are going to look at a simple game, a modified version of one with a long history. Games of this kind have been used as test-beds for development of artificial intelligence.

A *dungeon* is a network of connected *rooms*. One or more rooms contain *treasure*. Your character, the *adventurer*, moves between rooms, looking for the treasure. A *troll* is also in the dungeon. The troll moves between rooms at random. If the troll catches the adventurer, you lose. If you find treasure before being eaten, you win. (In this simple version, we do not consider the need to leave the dungeon.)

The starting rooms for the adventurer and troll are given in the definition of the dungeon.

The way the adventurer moves is called a *strategy*. Different strategies are more or less likely to succeed.

We will consider only one strategy this time - the adventurer will also move at random.

We want to calculate the probability that this strategy will be successful for a given dungeon.

We will use a "monte carlo" approach - simply executing the random strategy many times, and counting the proportion of times the adventurer wins.

Our data structure for a dungeon will be somewhat familiar from the Maze example:

```
In [1]: dungeon1 = {
            'treasure' : [1], # Room 1 contains treasure
            'adventurer': 0, # The adventurer starts in room 0
            'troll': 2, # The troll starts in room 2
            'network': [[1], #Room zero connects to room 1
                        [0,2], #Room one connects to rooms 0 and 2
                        [1] ] #Room 2 connects to room 1
        }
```

So this example shows a 3-room linear corridor: with the adventurer at one end, the troll at the other, and the treasure in the middle.

With the adventurer following a random walk strategy, we can define a function to update a dungeon:

```
In [2]: import random

        def random_move(network, current_loc):
            targets=network[current_loc]
            return random.choice(targets)
```

```
In [3]: def update_dungeon(dungeon):
            dungeon['adventurer']=random_move(dungeon['network'], dungeon['adventurer'])
            dungeon['troll']=random_move(dungeon['network'], dungeon['troll'])
```

```
In [4]: update_dungeon(dungeon1)

        dungeon1

Out[4]: {'treasure': [1], 'adventurer': 1, 'troll': 1, 'network': [[1], [0, 2], [1]]}
```

We can also define a function to test if the adventurer has won, died, or if the game continues:

```
In [5]: def outcome(dungeon):
            if dungeon['adventurer']==dungeon['troll']:
                return -1
            if dungeon['adventurer'] in dungeon['treasure']:
                return 1
            return 0

In [6]: outcome(dungeon1)

Out[6]: -1
```

So we can loop, to determine the outcome of an adventurer in a dungeon:

```
In [7]: import copy

        def run_to_result(dungeon):
            dungeon=copy.deepcopy(dungeon)
            max_steps=1000
            for _ in range(max_steps):
                result= outcome(dungeon)
                if result != 0:
                    return result
                update_dungeon(dungeon)
            # don't run forever, return 0 (e.g. if there is no treasure and the troll can't reach the a
            return result

In [8]: dungeon2 = {
            'treasure' : [1], # Room 1 contains treasure
            'adventurer': 0, # The adventurer starts in room 0
            'troll': 2, # The troll starts in room 2
            'network': [[1], #Room zero connects to room 1
                        [0,2], #Room one connects to rooms 0 and 2
                        [1,3], #Room 2 connects to room 1 and 3
                        [2]] # Room 3 connects to room 2

        }

In [9]: run_to_result(dungeon2)

Out[9]: -1
```

Note that we might get a different result sometimes, depending on how the adventurer moves, so we need to run multiple times to get our probability:

```
In [10]: def success_chance(dungeon):
             trials=10000
             successes=0
             for _ in range(trials):
                 outcome = run_to_result(dungeon)
```

```
            if outcome == 1:
                successes+=1
        success_fraction = successes/trials
        return success_fraction
```

```
In [11]: success_chance(dungeon2)
```

```
Out[11]: 0.5002
```

Make sure you understand why this number should be a half, given a large value for `trials`.

```
In [12]: dungeon3 = {
            'treasure' : [2], # Room 2 contains treasure
            'adventurer': 0, # The adventurer starts in room 0
            'troll': 4, # The troll starts in room 4
            'network': [[1], #Room zero connects to room 1
                        [0,2], #Room one connects to rooms 0 and 2
                        [1,3], #Room 2 connects to room 1 and 3
                        [2, 4], # Room 3 connects to room 2 and 4
                        [3]] # Room 4 connects to room 3

        }
```

```
In [13]: success_chance(dungeon3)
```

```
Out[13]: 0.4044
```

[Not for credit] Do you understand why this number should be 0.4? Hint: The first move is always the same. In the next state, a quarter of the time, you win. 3/8 of the time, you end up back where you were before. The rest of the time, you lose (eventually). You can sum the series: $\frac{1}{4}(1 + \frac{3}{8} + (\frac{3}{8})^2 + ...) = \frac{2}{5}$.

Chapter 13

Packaging the Treasure: your exercise

You must submit your exercise solution to **Moodle** as a single uploaded **Zip** format archive. (You must use only the *zip* tool, **not** any other archiver, such as `.tgz` or `.rar`. If we cannot unzip the archiver with `zip`, you will receive zero marks.)

The folder structure inside your zip archive must have a single top-level folder, whose **folder name is your student number**, so that on running `unzip` this folder appears. This top level folder must contain all the parts of your solution. You will lose marks if, on unzip, your archive creates other files or folders at the same level as this folder, as we will be unzipping all the assignments in the same place on our computers when we mark them!

Inside your top level folder, you should create a setup.py file to make the code installable. You should also create some other files, per the lectures, that should be present in all research software packages. (Hint, there are three of these.)

Your tidied-up version of the solution code should be in a sub-folder called `adventure` which will be the python package itself. It will contain an **init**.py file, and the code itself must be in a file called dungeon.py. This should define a class `Dungeon`: instead of a data structure and associated functions, you must refactor this into a class and methods.

Thus, if you run python in your top-level folder, you should be able to `from adventure.dungeon import Dungeon`. If you cannot do this, you will receive zero marks.

You must create a command-line entry point, called hunt. This should use the entry_points facility in setup.py, to point toward a module designed for use as the entry point, in adventure/command.py. This should use the `Argparse` library. When invoked with `hunt mydungeon.yml --samples 500` the command must print on standard output the probability of finding the treasure in the specified dungeon, using the random walk strategy, after the specified number of test runs.

The dungeon.yml file should be a yml file containing a structure representing the dungeon state. Use the same structure as the sample code above, even though you'll be building a `Dungeon` object from this structure rather than using it directly.

You must create unit tests which cover a number of examples. These should be defined in `adventure/tests/test_dungeon.py`. Don't forget to add an **init.py** file to that folder too, so that at the top of the test file you can " `from ..dungeon import Dungeon`." If your unit tests use a fixture file to DRY up tests, this must be called `adventure/tests/fixtures.yml`. For example, this could contain a yaml array of many dungeon structures.

You should `git init` inside your student-number folder, as soon as you create it, and `git commit` your work regularly as the exercise progresses.

Due to our automated marking tool, **only** work that has a valid git repository, and follows the folder and file structure described above, will receive credit.

Due to the need to avoid plagiarism, do *not* use a public github repository for your work - instead, use git on your local disk (with `git commit` but not `git push`), and *ensure the secret `.git` folder is part of your zipped archive.

Chapter 14

Marks Scheme

Note that because of our automated marking tool, a solution which does not match the standard solution structure defined above, with file and folder names exactly as stated, may not receive marks, even if the solution is otherwise good. "Follow on marks" are **not** guaranteed in this case.

- Code in dungeon.py, implementing the random walk strategy (5 marks)
- Which works (1 mark)
- Cleanly laid out and formatted - PEP8 (1 mark)
- Defining the class `Dungeon` with a valid object oriented structure (1 mark)
- Breaking down the solution sensibly into subunits (1 mark)
- Structured so that it could be used as a base for other strategies (1 mark)
- Command line entry point (4 marks)
- Accepting a dungeon definition text file as input (1 mark)
- With an optional parameter to control sample size (1 mark)
- Which prints the result to standard out (1 mark)
- Which correctly uses the `Argparse` library (1 mark)
- Which is itself cleanly laid out and formatted (1 mark)
- setup.py file (5 marks)
- Which could be used to `pip install` the project (1 mark)
- With appropriate metadata, including version number and author (1 mark)
- Which packages code (but not tests), correctly. (1 mark)
- Which specifies library dependencies (1 mark)
- Which points to the entry point function (1 mark)
- Three other metadata files: (3 marks)
- Hint: Who did it, how to reference it, who can copy it.
- Unit tests: (5 marks)
- Which test some obvious cases (1 mark)
- Which correctly handle approximate results within an appropriate tolerance (1 mark)
- Which test how the code fails when invoked incorrectly (1 mark)
- Which use a fixture file or other approach to avoid overly repetitive test code (1 mark)
- Which are themselves cleanly laid out code (1 mark)
- Version control: (2 marks)
- Sensible commit sizes (1 mark)
- Appropriate commit comments (1 mark)

Total: 25 marks

```
In [ ]:
```

Chapter 15

Refactoring Trees: An exercise in Research Software Engineering

In this exercise, you will convert badly written code, provided here, into better-written code.

You will do this not through simply writing better code, but by taking a refactoring approach, as discussed in the lectures.

As such, your use of `git` version control, to make a commit after each step of the refactoring, with a commit message which indicates the refactoring you took, will be critical to success.

You will also be asked to look at the performance of your code, and to make changes which improve the speed of the code.

The script as supplied has its parameters hand-coded within the code. You will be expected, in your refactoring, to make these available as command line parameters to be supplied when the code is invoked.

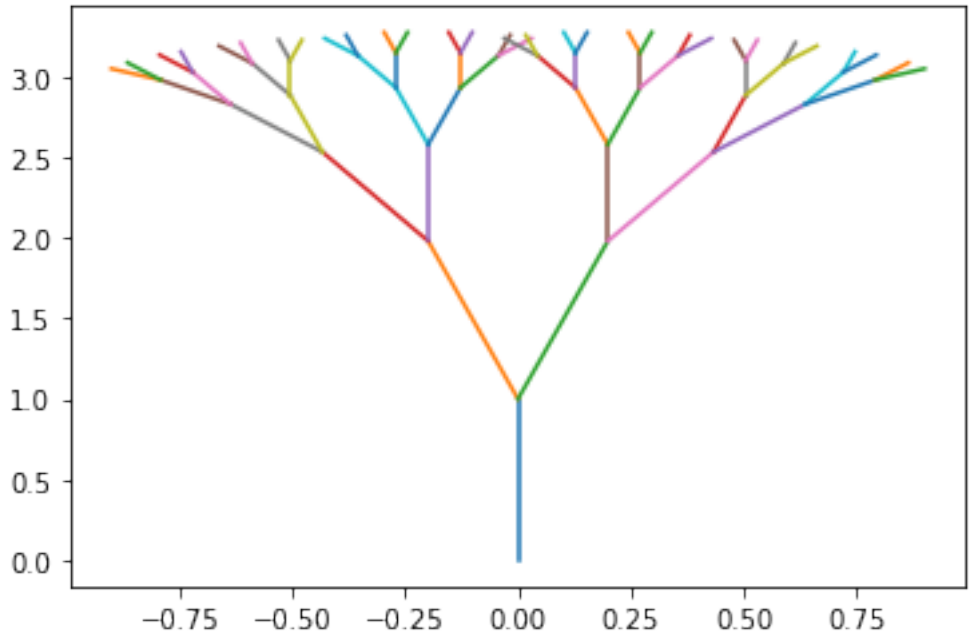
Chapter 16

Some terrible code

Here's our terrible code:

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

```
In [2]: from math import sin, cos
        from matplotlib import pyplot as plt
        s=1
        d=[[0,1,0]]
        plt.plot([0,0],[0,1])
        for i in range(5):
            n=[]
            for j in range(len(d)):
                n.append([d[j][0]+s*sin(d[j][2]-0.2), d[j][1]+s*cos(d[j][2]-0.2), d[j][2]-0.2])
                n.append([d[j][0]+s*sin(d[j][2]+0.2), d[j][1]+s*cos(d[j][2]+0.2), d[j][2]+0.2])
                plt.plot([d[j][0], n[-2][0]],[d[j][1], n[-2][1]])
                plt.plot([d[j][0], n[-1][0]],[d[j][1], n[-1][1]])
            d=n
            s*=0.6
        plt.savefig('tree.png')
```



Chapter 17

Rubric and marks scheme

17.1 Part one: Refactoring (15 marks)

- Copy the code above into a file tree.py, invoke it with python tree.py, and verify it creates an image tree.png which looks like that above.
- Initialise your git repository with the raw state of the code. [1 mark]
- Identify a number of simple refactorings which can be used to improve the code, *reducing repetition* and *improving readability*. Implement these one by one, with a git commit each time.
  - 1 mark for each refactoring, 1 mark for each git commit, at least five such: ten marks total.
- Do NOT introduce NumPy or other performance improvements yet (see below.)
- Identify which variables in the code would, more sensibly, be able to be input parameters, and use Argparse to manage these.
- 4 marks: 1 for each of four arguments identified.

17.2 Part two: performance programming (10 marks)

- For the code as refactored, prepare a figure which plots the time to produce the tree, versus number of iteration steps completed. Your code to produce this figure should run as a script, which you should call perf_plot.py, invoking a function imported from tree.py. The script should produce a figure called perf_plot.png. Comment on your findings in a text file, called comments.md. You should turn off the actual plotting, and run only the mathematical calculation, for your performance measurements. (Add an appropriate flag.)
- 5 marks: [1] Time to run code identified [1] Figure created [1] Figure correctly formatted [1] Figure auto-generated from script [1] Performance law identified.
- The code above makes use of `append()` which is not appropriate for NumPy. Create a new solution (in a file called tree_np.py) which makes use of NumPy. Compare the performance (again, excluding the plotting from your measurements), and discuss in comments.md
  - 5 marks: [1] NumPy solution uses array-operations to subtract the change angle from all angles in a single minus sign, [1] to take the sine of all angles using np.sin [1] to move on all the positions with a single vector displacement addition [1] Numpy solution uses `hstack` or similar to create new arrays with twice the length, by composing the left-turned array with the right-turned array [1] Performance comparison recorded

As with assignment one, to facilitate semi-automated marking, submit your code to moodle as a single Zip file (not .tgz, nor any other zip format), which unzips to produce files in a folder named with your **student number**.