IA Scientific Computing

2017/2018

Damon Wischik

Computer Laboratory, Cambridge

0. About this course

Goal of this course: There are some skills that you never really *get* until you try to apply them. Handling experimental data is such a skill. The aim of this course is to make sure you comfortable with getting hold of data (either from simulations or by importing it), manipulating it, and plotting it. With a minimum of fuss, and with decent speed, using the programming language Python.

0.0 What is scientific computing?

Scientific computing means using computers as a tool to gain understanding from data, either simulated or observed. The 'output' is improved understanding of a system, as opposed to software engineering where the output is a working system. This gives it a distinct flavour:

- Often we write small snippets of code, learn something, then discard the code and try something new, over and over again. We need tools that are concise and expressive, to make the gap between "what if..." and an answer as small as possible.
- It is mostly a personal effort to gain understanding, not a team effort. We're not bothered about "Will this codebase be robust when 20 people work on it and the requirements change each month?", we are bothered about "Is this result reproducible, and will people be able to re-run this same code and understand it a year from now?"
- Scientific computing is oriented around data. We need to always interact, and query, and visualize, and inspect. Compilers are out, interpreters are in.
- We always have to communicate whatever we've understood: there is an emphasis on visualisations, and on write-ups in the style of scientific reports. This course will use Jupyter Notebooks, "live documents" that mix code and text and plots.
- Programming languages for scientific computing tend to be glue-like, good at linking together assorted libraries and tools. This is what lets them be concise and expressive, and also fast at working with large amounts of data. It means you need to know your way around some big libraries, so you don't spend all your time endlessly rewriting boilerplate code. This course will use Python; the other two main languages for scientific computing are R and MATLAB.

Scientific computing will be useful in all the data-themed courses you study: machine learning, data science, signal processing, computer vision, etc. It will be useful for experimental subjects, such as aspects of HCI. It will be useful for subjects that involve simulations, such as computer systems modelling. In the short term, it is loosely linked to some other IA/IB courses:

- IA *Maths for Natural Sciences*. Science students have their own version of the *Scientific Computing* course, which uses MATLAB. The course you are reading is for 50%-CS and 75%-CS students, and it uses Python instead, and it has a greater emphasis on working with data.
- IA/IB *Machine Learning and Real World Data*. The MLRD course involves lots of scientific computing and plotting. It expects you to program your work in Java, but some of you might like to sketch out your ideas in Python first.
- IA *Foundations of Computer Science* and *Object Oriented Programming*. This course will introduce you to Python (if you don't know it already). Python is a general-purpose programming language, Appendix A1 discusses some of the similarities and differences with ML and Java.
- IA *Algorithms*. The algorithms you will study will be presented in pseudocode that closely resembles Python. If you want to become more fluent in Python, tinker with the code presented in that course.
- IA/IB *Databases*. Section 3 of this course is about handling tables of data, which closely resembles querying a database. Also, Appendix A3 gives examples of importing data by querying an SQL database, so you can use Python as a sandbox to familiarise yourself with SQL and to visualize query results.

0.1 Course structure

- **1. Programming in Python** …… Appendix: Python language choices
- **2. Numerical computation** … Assignment 2 …… Appendix: vectorized thinking
- **3. Working with data** … Assignment 3 …… Appendix: data import and cleanup

This course is for you to work through at your own pace. You may like to approach it as a problem-solving course, jumping straight into the assignments and skimming through the notes only as you need to. Or you may like to work through the notes systematically before going on to the assignments. It's up to you. There are three

appendices which aren't part of the course, but which you may find thought-provoking or useful as a reference in your later studies; and there are also further exercises scattered through each section of the notes.

Assessment. The course is marked out of 4. There are two assignments, each split into two parts, each part marked 0 or 1. (There's no assignment for Section 1, because your Python skills will be thoroughly tested by Sections 2 and 3.) The assignments are assessed in two ways:

- As you work through an assignment, you will submit answers to an **automatic grader**, which checks if the answers are correct. This is for you to use as a debugging aide: run the automatic grader as many times as you need until your code gives the right answers. You should aim to answer all the questions. The automatic grader will show you your progress.
- There is a final **ticking session**, when you present your work for the entire course to a (human) demonstrator. The demonstrator will choose some parts of the assignments, and ask you to explain your answers. The demonstrator will only ask you about the parts where you have submitted complete and correct answers to the autograder. The ticking session is for the examiners to guard against plagiarism. At the ticking session, you should have your notebooks ready and open in a browser, and your code should execute without errors. For scheduling of ticks, see Moodle.

Getting help. You must understand what your programs are doing, and in the ticking session you will be asked justify your answers to the assignments. However, it's fine to get help, and if you are having difficulty this is encouraged. The main help forum is on Moodle. In addition, there will be help sessions at the beginning of Lent term; see Moodle for timetabling information.

0.2 Running notebooks

This is a Jupyter notebook. Jupyter notebooks are "live documents" that intersperse text and running code. We will be using Jupyter hosted on the Microsoft Azure cloud computing environment. You can access it from any device — all you need is a web browser.

Click on Clone at the top of the page to get your own runnable version of the Library (consisting of all notebooks for this course). If you are not logged in, you will be prompted to log in: type in your Cambridge University CRSid e.g. `spqr1@cam.ac.uk`, click in the password field, and you will automatically be redirected to a Raven login page. Experiment freely with your clone of the library — you can always go back and read the master version.

> For academic staff at Cambridge: if your Raven password doesn't work, then you need to activate your Microsoft account first. For other users: you can sign up for a free Microsoft account on the login page.

Once you are logged in you can also create your own libraries and notebooks, using the `Libraries` link at the top left at `https://notebooks.azure.com/`. When you create a notebook you will be asked which language to use. This course uses Python 3.6, which is substantially different to Python 2.7, so be alert when you look for help online.

These notebooks are hosted on the cloud, and the first lesson of cloud computing is that **the cloud will fail when you need it most.** Take regular backups, by going to the `Libraries` page, clicking on the library, clicking on the line with the relevant notebook, then clicking on Download. You can also run the notebooks on your own machine: follow the Jupyter installation instructions, and clone the notebooks from their GitHub repository. More detailed instructions will be available on Moodle.

0.3 Using the automatic grader

Once you've answered some exercises and you're ready to check your answers, paste the following code at the top of your notebook and run it. Each section of the notes tells you what value to use for `section`. To illustrate how it works, we'll use `section=notes0`.

```
#!pip install ucamcl
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp', section='notes0')
# paste in whatever section is appropriate for the section of notes / assignment you're working on
```

> The first line installs a Python package `ucamcl`, with the functions used by the automatic grader. (On Azure, you are given a new machine every time you start a notebook, so you have to install the `ucamcl` package every time.) The next line makes it available in your current notebook. The third line will produce a login button, which you should click on. It will open a web page, showing you your progress so far.

Submitting answers to assessed exercises

A typical question will look something like this.

Question 1. Given an integer n and a value v, create a list containing n copies of v.

```
# Submit your answer:
q = GRADER.fetch_question('q1')
# Let x be a list consisting of q.n copies of q.v
GRADER.submit_answer(q, x)
```

When you call `GRADER.fetch_question`, the function returns an object `q` with the question parameters. Your fellow students might be given different parameters. You can call `print(q)` to see them, and to see what format the answer should take.

When you call `submit_answer`, the server checks if the answer is correct for the question parameters you were given. If not, it may tell you what answer it was expecting. If you get the question wrong, don't be tempted to just paste in the correct answer and resubmit — it won't do you any good in the ticking session, and anyway the system marks the question as "stale" and requires you to call `fetch_question` again to fetch new question parameters before you are allowed to resubmit.

Checking answers to non-assessed exercises

Some non-assessed exercises come with answers. Such exercises have labels: for example if you see Exercise (ex5) then the label is `ex5`. Check your answers by calling

It might print out the full answer. Or it might print out instructions for how you can submit your own answer for checking. In the latter case, print(q) will tell you what format it expects the answer to be in. Follow the instructions, and check your answer by running

1 Programming Python in a Jupyter notebook

Goal of this notebook: Familiarize yourself with how to use a notebook, how to use Python for simple calculation, and the basics of Python programming.

If you know Python already, skim through the section on comprehensions first and then jump ahead to Section 2.

To see how Python compares to other programming languages, particularly ML and Java, see Section A1.

1.0 Editing notebooks

This is a Jupyter notebook. Jupyter notebooks provide an interactive environment where you can mix text, equations, programming and visual outputs. The famous computer scientist Donald Knuth introduced this style of programming, mixing source code with explanations written in natural language, which he calls literate programming. It has become popular in data science and machine learning, wherever scientific explanation and coding go hand in hand. Scientific notebooks of course have a long and venerable history.

These notebooks are a mixture of text, input code, and output. * Edit a text cell by double-clicking on it * When you've finished editing a text cell, press `shift-enter` to make it display nicely, or choose `Cell | Run Cells` from the menu * Insert new cells using the `Insert` menu * Change a cell from text to input code and *vice versa* using the `Cell | Cell Type` menu

The text format is called Markdown.

Exercise. Try editing the cell above. How do you type in bullet lists, italics, and links? (If you can't edit cells, you need to clone the notebook, as described in §0.2 Running notebooks.)

1.1 Using Python for simple calculations

We will be programming in the language Python. At the top right of your notebook you should see Python 3 or Python 3.6 telling you which version you're running. If you're not running one or other of these, then use the menu `Kernel | Change kernel` to switch. Python 3 has slightly different syntax to Python 2, and when you look for help online please be aware that many older resources are for Python 2.

We can use Python like a calculator. Here are some simple expressions and their values. Try editing the expressions, then to evaluate the cell press `shift-enter` or choose `Cell | Run Cells` from the menu. (Remember, if you can't edit cells, you need to clone the notebook first.)

```
3 + 8
```

```
11
```

```
1.618 * 1e5
```

```
x = 3
y = 2.2
z = 1
x * y + z
```

```
(x,y,z) = (3, 2.2, 4)  # You can assign multiple values at once.
x * (y + z)            # And you can comment your code with the # symbol!
```

```
'hello ' + 'world ' * 2
```

Exercise. What happens when you type in an erroneous expression? Run the expression below to find out.

```
# This expression should produce an error message
'hello' + 5
```

In a notebook, we can pick which cells to execute when. We can define a variable lower down in the notebook, run that cell, then use it higher up in the notebook. Jupyter prints e.g. `In[42]` at the left to tell you about the order it executed cells. However, for the sanity of anyone reading your notebook (which includes you a week after

you wrote it), once you've got your code working you should rearrange it all into clean top-to-bottom order. The menu item `Kernel | Restart & Run All` should then produce all the right output.

1.2 Basic Python expressions

Maths and logic

All the usual mathematical operators work, though watch out for division which uses different syntax to Java. (The following code block uses `print` statements, to print multiple outputs from one cell.)

Exercise. Before running the code, guess what these statements produce. Then check your answers.

```
print('a:', 7 / 3)                      # floating point division
print('b:', 7 // 3)                     # integer division (rounds down)
print('c:', min(3,4), max(3,4))
print('d:', abs(-10), abs(3+4j))        # 3+4j is a complex number
print('e:', round(7.4), round(-7.4), round(3.4567, 2))
print('f:', 3**2)                       # power
print('g:', 5 << 1, 5 >> 2)            # bitwise shifting
print('h:', 7 & 1, 6 | 1)              # bitwise operations
print('i:', (3+4j).real, (3+4j).imag)
```

The usual logical operators work too, though the syntax is wordier than other languages.

```
(x,y) = (5,12)
print('a:', x < y or y < 10)
print('b:', x < y and not y < 15)
print('c:', 'lower' if x < y else 'higher')  # same as Java's (x < y) ? 'lower' : 'higher'
```

Some useful maths functions are found in the `math` module. To use them, you need to run `import math`. (It's common to put your import statements at the top of the notebook, as they only need to be run once per session, but they can actually appear anywhere.)

```
import math
print('a:', math.floor(-3.4), math.ceil(-3.4))
print('b:', math.pow(9, 0.5), math.sqrt(9))
print('c:', math.exp(2), math.log(math.e), math.log(101, 10))
print('d:', math.sin(math.pi*1.3), math.atan2(3,4))

import cmath   # for functions on complex numbers
print('e:', cmath.sqrt(-9))
print('f:', cmath.exp(math.pi * 1j) + 1)

import random  # for generating random numbers
print('g:', random.random(), random.random())
```

Strings and formatting

Python strings can be enclosed by either single quotes or double quotes. Strings (like everything else in Python) are objects, and they have methods for various string-processing tasks. See "String Methods" documentation for a full list.

```
print('a:', "shout".upper())
print('b:', "hitchhiker".replace('hi', 'ma'))
print('c:', 'i' in 'team')
```

```
x = '''
Also, a multi-line string can be
entered with triple-quotes.
'''
```

To control how values are printed, we can use the `str.format` method. Here are some examples.

```
print("My name is {} and I am {} years old".format('Zaphod', 27))                    # values
inserted by position
print("The value of π to 3 significant figures is {p:.3}, and to 5 is {p:.5}".format(p=math.pi))  # values
inserted by name
```

```
print(f"The value of e to 3 significant figures is {math.e:.3}, and to 5 is {math.e:.5}")        # string
interpolation (requires Python 3.6)
```

If you do any serious data processing in Python, you will likely find yourself needing regular expressions.

1.3 Collection types and control flow

Tuples, lists, and dictionaries

Python has three common types for storing collections of values: tuples, lists, and dictionaries. (Also two types of set which we won't cover in this course.)

In ML and Java we learned about lists *versus* arrays, and in Algorithms we will study the efficiency of various implementation choices. The Pythonic style is to just go ahead and code, and only worry about efficiency after we have working code. We'll simply use the built-in type for storing sequences, not even bothering whether it should be a list or an array, and we expect it to work reasonably well most of the time. If we have special needs we switch to a dedicated collection type, such as a deque or a heap or the specialized numerical types we'll learn about in §2. Numerical computation. As the famous computer scientist Donald Knuth said,

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of non-critical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.

Lists and tuples are both ways to store sequences of elements.

```
a = [1, 2, 'buckle my shoe']      # a list
b = (3, 4, 'knock at the door')   # a tuple
print('a:', a + list(b))
print('b:', tuple(a) + b)
print('c:', len(a), len(b))
print('d:', a[0], a[1], b[2])     # indexes start at 0
print('e:', a[-1], a[-2])         # negative indexes count from the end
print('f:', [a, 'then', b])
print('g:', 3 in a, 3 in b)
```

As you see, both lists and tuples can hold mixed types, including other lists or tuples. You can convert a list to a tuple and vice versa, and extract elements. The difference between them is that tuples are immutable whereas lists are mutable:

```
a[0] = 5
a.append('then')
a.extend(b)
print(a)

b[0] = 5  # error
print(b)
```

Exercise (ex1). What is the difference between the two commented lines? Do they give the same result?

Note: For labelled exercises, like this one, you can look up the answer or check your solution. The procedure is described in §0.3 Using the automatic grader, and for this notebook you should use section='notes1'.

Exercise (ex2). How do you type in a tuple of length 1? How about a tuple of length 0?

We can pick out subsequences using the *slice* notation, `x[start:end:sep]`.

```
x = list(range(10))     # creates a list with 10 elements, starting at 0
print('a:', x)
print('b:', x[1:3])     # start is inclusive and end is exclusive, so x[1:3] == [x[1],x[2]]
print('c:', x[:2])      # first two elements
print('d:', x[2:])      # everything after the first two
print('e:', x[-3:])     # last three elements
print('f:', x[:-3])     # everything prior to the last three
print('g:', x[::4])     # every fourth element
```

```
a: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
b: [1, 2]
c: [0, 1]
```

```
d: [2, 3, 4, 5, 6, 7, 8, 9]
e: [7, 8, 9]
f: [0, 1, 2, 3, 4, 5, 6]
g: [0, 4, 8]
```

We can sort a list.

```
names = ['adrian', 'chloe', 'guarav', 'shay', 'alexis', 'rebecca', 'zubin']

# The function sorted(list) returns a new list, and doesn't alter the original.
print(sorted(names))
print(names)

# The method list.sort() sorts the list in-place
names.sort()
print(names)
```

Another common operation is to concatenate a list of strings. Python's syntax for this is unusual:

```
', '.join(['alpher', 'bethe', 'gamov']) + 'wrote a famous paper on nuclear physics'
```

The other very useful data type is the dictionary.

```
room_alloc = {'Adrian': None, 'Laura': 32, 'John': 31}
room_alloc['Guarav'] = 19
print(room_alloc['Laura'])          # get an item
del room_alloc['John']              # remove an item
print(room_alloc)
print('Alexis' in room_alloc)       # does this dictionary contain the key 'Alexis'?
print(room_alloc.get('Alexis', 1))  # get room_alloc['Alexis'] if it exists, else default to 1
```

Control flow

Python supports the usual control flow statements: `for`, `while`, `continue`, `break`, `next`, `if`, `else`. Here's an example: suppose we're given a list of room allocations, and we want to find the occupants of each room.

```
room_alloc = {'adrian': 10, 'chloe': 5, 'guarav': 10, 'shay': 11,
              'alexis': 11, 'rebecca': 10, 'zubin': 5}
occupants = {}
for name, room in room_alloc.items():  # iterate over keys and values
    if room not in occupants:
        occupants[room] = []
    occupants[room].append(name)
for room, occupants_here in occupants.items():
    ns = ', '.join(occupants_here)
    print('Room {r} has {ns}'.format(r=room, ns=ns))
```

The above code iterates over the keys and values in a dictionary. We can also iterate over just the keys (`for name in room_alloc`), or just the values (`for room in room_alloc.values()`).

If we have a list, we sometimes want to iterate over items and their positions in the list. We use `enumerate` for this:

```
for i, name in enumerate(['adrian', 'chloe', 'guarav', 'shay', 'alexis', 'rebecca', 'zubin']):
    print('Person {} is in position {}'.format(name, i))
```

Comprehensions

Python has a distinctive piece of syntax called a *comprehension* for creating lists. It's a very common pattern to write code that transforms lists, e.g.

```
mylist = ...    # start with some list
newlist = []
for i in range(len(mylist)):
    x = mylist[i]
    if testfunc(x):
        newlist.append(mapfunc(x))
```

This is so common that Python has a special syntax for it:

```
newlist = [mapfunc(x) for x in mylist if testfunc(x)]
```

We can also use comprehension to create dictionaries and sets. Here are examples of list comprehension and dictionary comprehension:

```
xs = range(10)
print([x**2 for x in xs if x % 2 == 0])  # create a list, filtered to process only even values
print({x: x**2 for x in xs})             # create a dictionary
```

Exercise (ex3). If you go overboard with list comprehensions, your code becomes unreadable. What does the following code do?

Exercise (ex4). Write a single line of code to sort the names in this list

by length, breaking ties alphabetically, using list comprehension. Hint: make a list of (len(name), name) then sort it, where len(str) gives the length of a string. When Python sorts a list of tuples, it uses lexicographic ordering.

1.4 Functions

The code snippet below shows how we define a function in Python. There are several things to note: * The function is defined with a default argument, `c=0`. You can invoke it by either `roots(2,3,1)` or `roots(2,3)`. * Functions can be called with named arguments, `roots(b=3, a=2)`, in which case they can be provided in any order.

In scientific computing, we'll come across many functions that accept 10 or more arguments, all of them with sensible defaults, and typically we'll only specify a few of the arguments.

```
import math

def roots(a, b, c=0):
    """Return a list with the real roots of a + b*x + c*(x**2) == 0"""
    if b == 0 and c == 0:
        raise Exception("This polynomial is constant")
    if c == 0:
        return [-a/b]
    elif a == 0:
        return [0] + roots(b=c, a=b)
    else:
        discr = b**2 - 4*a*c
        if discr < 0:
            return []
        else:
            return [(-b+s*math.sqrt(discr))/2/a for s in [-1,1]]
```

Some more notes:

- This function either returns a value, or it throws an exception i.e. generates an error message and finishes. If your function finishes without an explicit `return` statement, it will return `None`. Unlike Java, it's possible for different branches of your function to return values of different types — at risk to your sanity.
- This function returns a single variable, namely a list. If you want to return several variables, return them in a tuple, and unpack the tuple using multiple assignment as shown above.
- It's conventional to document your function by providing a documentation string as the first line.

Exercise. Execute ?roots. What does it tell you? Look at some other functions, e.g. ?sorted.

We can pass functions as arguments to other functions, and we can use the keyword `lambda` to define so-called *anonymous* functions. Look back at Exercise 5 in ML, then try this code:

```
def twice(f, x):
    return f(f(x))
twice(lambda i: i+5, 3)
```

We often use anonymous functions as a way to fill in arguments:

```
def illustrate_func(f, xs):
    for x in xs:
        print(f'f({x}) = {f(x)}')

illustrate_func(lambda b: roots(1,b,2), range(5))
```

More exercises

The exercises in this notebook are to give you practice. They are optional, and do not contribute to your final grade. You can check your answers to labelled exercises by following the instructions in § 0.3 Using the automatic grader with `section='notes1'`.

Exercise (ex5). A simple queue can be simulated by the following equations. Let a_t be the queue size just before timestep t, let the service rate be C, and let a_t be the amount of work arriving in timestep t. Then

$$q_{t+1} = \max(q_t + a_t - C, 0).$$

This is called Lindley's Recursion. Write a function `sim(q0,C,a)` to compute the queue sizes at every timestep $t \geq 1$, where `q0` is the initial queue size, and `a` is the list $[a_0, a_1, \ldots]$. For example, `> sim(1, 3, [4, 1, 2, 8, 2, 3, 1])`

should produce the answer `> [2, 0, 0, 5, 4, 4, 2]`

Exercise (ex6). * Calculate the base 10 logarithm of 1200 (answer is 3.079) * Calculate the tangent of 60 degrees (answer is 1.7321) * Calculate the square root of -20 (answer is 0+4.4721i)

Exercise (ex7). Create the list of lists `[[1],[2],[3],...,[n]]` for `n=10`.

2 Numerical computation

Working with numbers is central to almost all scientific and engineering computing, from deep learning to image processing to climate simulation. We could use Python directly for numerical computation—but it's much faster to use Python just as 'glue', using it to write concise code and to quickly develop our thinking, and to rely on carefully optimized low-level libraries for the heavy lifting.

Goal of this notebook: Learn your way around the two main Python libraries for numerical work, NumPy and SciPy, and learn how to produce basic plots with matplotlib.

There are some exercises in this section of notes. They are for you to get practice on: they are optional, and do not contribute to your final grade. You can check your answers to labelled exercises as described in §0.3 using `section='notes2'`.

2.0 Preamble

At the top of almost every piece of scientific computing work, we'll import these standard modules.

```
# Import modules, and give them short aliases so we can write e.g. np.foo rather than numpy.foo
import math, random
import numpy as np
import matplotlib.pyplot as plt
import scipy
import scipy.optimize
# The next line is a piece of magic, to let plots appear in our Jupyter notebooks
%matplotlib inline
```

2.1 Vector calculations

In §1 we learnt about Python lists, which can store mixed data types e.g. integers mixed with strings and sublists and even functions. The flexibility comes at the price of performance. In scientific computing, it's better to use specialised classes for vectors, and to use functions that operate on entire vectors at once. This is called *vectorized thinking*, and it's a core skill for scientific computing. Once you get the hang of it, you will write code that is more concise and faster. Here are some simple examples.

```
x = np.array([1,2,5,3,2])  # create a numpy vector out of a Python list
y = np.ones(5)             # create a numpy vector [1,1,1,1,1]
x + y                      # iterates over all elements of x and y for you
```

```
array([ 2.,  3.,  6.,  4.,  3.])
```

All the elements of a vector have to be the same type. Use `x.dtype` to find this type, and `x.astype` to convert a vector from one type to another.

```
x = np.arange(10)
y = (x > 5)
z = np.ones_like(x)
(x.dtype, y.dtype, z.dtype)
```

```
(dtype('int64'), dtype('bool'), dtype('int64'))
```

To be good at writing vectorized code, we need to know what sort of calculations we can do on vectors. Here are some useful routines:

Maths:

- Normal mathematical expressions work on vectors, and you can mix vectors and scalars, e.g. `x + y ** 2 + 5 >= z`
- `np.sin`, `np.exp`, `np.floor`, ...

- `x @ y` gives the dot product, `np.linalg.norm(x)` is the norm
- `np.sum` and `np.prod`; `np.cumsum(x)` gives $[x_0, x_0 + x_1, x_0 + x_1 + x_2, \dots]$
- `np.min` and `np.max` for the overall min and max; `np.minimum(x,y)` for $[\min(x_0, y_0), \min(x_1, y_1), \dots]$
- and other maths and statistics functions

Create vectors:

- `np.array([1,2,3])` creates a numpy vector from a Python list
- `np.zeros(n)`, `np.ones(n)`, `numpy.full(n,fill_value)`
- `numpy.ones_like(a)` creates a vector of the same shape as `a`
- `np.arange` is like Python's `range`
- `np.linspace(start,stop,n)` creates n evenly-spaced points between `start` and `stop` inclusive, very useful for plotting
- `np.random.random(n)`, `np.random.choice(a,n)`, and other random number generators
- and other array creation routines

More elaborate example

Here's a more elaborate example: computing the correlation coefficient between two vectors x and y,

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}}$$

where x and y have the same length N, and

$$\bar{x} = \frac{1}{N}\sum_i x_i, \quad \bar{y} = \frac{1}{N}\sum_i y_i.$$

Here are two pieces of code, one written in Python-style, one written in scientific computing style, to compute ρ. The latter is roughly 15 times faster. (The magic command `%%time` at the top of a cell makes the notebook print out the execution time.)

```
# Set up some parameters.
# We'll use a random number seed so our code is reproducible. Python's hash function gives an integer,
# suitable for use as a seed.
N = 10000000
rand_seed = 1618033988
```

```
%%time
# Python-style code

random.seed(rand_seed)
# Create two lists of random numbers, xs and ys, where each y depends on the corresponding x
xs = [random.random() for i in range(N)]
ys = [xs[i] + random.random() for i in range(N)]
# Compute the various terms involved in the formula
xbar = sum(xs) / N  # sum(list) is built into Python
ybar = sum(ys) / N
sxy = sum([(x-xbar)*(y-ybar) for x,y in zip(xs,ys)])  # this is how to iterate over a pair of lists
sxx = sum([(x-xbar)**2 for x in xs])
syy = sum([(y-ybar)**2 for y in ys])
rho = sxy / math.sqrt(sxx) / math.sqrt(syy)
print(rho)
```

```
0.7070356005706819
CPU times: user 5.98 s, sys: 1.55 s, total: 7.53 s
Wall time: 7.53 s
```

```
%%time
# Vectorized code

np.random.seed(rand_seed)
# Create two random vectors x and y
x = np.random.random(N)
y = x + np.random.random(N)
```

```
# Compute the terms in the formula. Note: @ means "dot product"
xbar = np.sum(x) / N
ybar = np.sum(y) / N
rho = ((x-xbar) @ (y-ybar)) / math.sqrt(np.sum((x-xbar)**2)) / math.sqrt(np.sum((y-ybar)**2))
print(rho)
```

```
0.707201664199
CPU times: user 266 ms, sys: 219 ms, total: 484 ms
Wall time: 537 ms
```

(Though if we really did know our way round numpy, we'd have used `np.corrcoef` rather than write our own code for ρ!)

Exercise (ex1). Using Python-style code: let x and y be random numbers in the range $[-1,1]$, and let $d = \sqrt{x^2+y^2}$. Repeat this a million times, and find the mean and standard deviation of your values for d.

Exercise (ex2). Repeat Exercise ex1, but this time using vectorized code. Compare the speed of the two styles of code.

2.2 Vectorized thinking: `for` considered harmful

Vectorized thinking isn't just for mathematical calculations — there are all sorts of programming constructs that can be vectorized also. In general, whenever you find yourself writing a `for` loop or a Python list comprehension, stop and see if you can vectorize your code. You'll usually end up with something more flexible for scientific computing.

Programming:

- `len(x)` gives the length of a vector
- `np.any`, `np.all` and other logic functions
- `~x` is logical negation, the equivalent of Python's `not x`; `x & y` and `x | y` both work
- `np.count_nonzero(x)` counts the number of entries where `x` is `True` or non-zero
- `np.sort` sorts a vector; `np.argsort(x)` gives the vector `i` such that `x[i]` is sorted; also see other sort functions
- `np.argmax` and other search functions
- `np.where(cond,x,y)` is the vectorized version of `x if cond else y`
- `np.vectorize(f)` is a vectorized version of an arbitrary Python function f

Indexing:

- The usual slice notation works, e.g. `x[:10]` or `x[10:]` or `x[:-3]`
- We can index by a vector of booleans, e.g. `x[y>5] = 3`
- We can index by a vector of integers, e.g. `i=np.where(y>5); x[i]=3`
- `np.where(b)` gives a vector of indexes at which the boolean vector `b` is `True`
- `np.concatenate([v1,v2])` concatenates two or more vectors

More elaborate example

Here's an example, a vectorized version of Exercise ex4 from Section 1. Suppose we want to sort the following vector by length, breaking ties alphabetically.

```
names = np.array(['adrian', 'chloe', 'guarav', 'shay', 'alexis', 'rebecca', 'zubin'])
```

- First, get a vector with the length of each name. Numpy does have some routines for working with strings, but as the name suggests it is really oriented around numbers, and I prefer to use Python functions. I'll still wrap them up with `np.vectorize`, so that I can apply them to a whole vector without having to write a list comprehension.
- Second, work out how to put `lengths` in order, breaking ties alphabetically by `names`. This is called lexicographic sorting, and there is a function `np.lexsort` that like `np.argsort` returns a vector of integer indexes that will achieve this ordering.
- Last, pick out the `names` in the order specified by these indexes.

```
lengths = np.vectorize(len)(names)
indexes = np.lexsort([names, lengths])
names[indexes]
```

```
array(['shay', 'chloe', 'zubin', 'adrian', 'alexis', 'guarav', 'rebecca'],
      dtype='<U7')
```

At the end of this notebook you'll find some more challenging exercises in vectorized thinking.

2.3 Arrays

NumPy supports matrices and higher-dimensional arrays. (In fact, when we look up the help for any of the vectorized routines, we see that vectors are nothing more than one-dimensional arrays.) To enter a 2d array like

$$a = \begin{pmatrix} 2.2 & 3.7 & 9.1 \\ -4 & 3.1 & 1.3 \end{pmatrix}$$

we type in

```
a = np.array([[2.2, 3.7, 9.1], [-4, 3.1, 1.3]])
```

Use `a.shape` to find the dimensions of an array:

```
a.shape
```

To refer to a subarray, we can use an extended version of Python's slice notation.

```
a[:, :2]                  # all rows, first two columns
a[1, :]                   # second row (indexes start at 0), all columns
a[1]                      # another way to fetch the second row
a[:2, :2] = [[1,2],[3,4]] # assign to a submatrix
a
```

```
array([[ 1. ,  2. ,  9.1],
       [ 3. ,  4. ,  1.3]])
```

To refer to arbitrary sets of elements in the array, we can use boolean indexing e.g. `a[a>=2]`, or integer indexing as in the code snippet below. These are both called advanced indexing.

```
a = np.zeros((3,3), dtype=np.int)
a[[0,1,2], [1,0,2]] = [1,1,1]
a
```

For 1d vectors the only reshaping operations are slicing and concatenating, but for higher dimensional arrays there is a whole variety of reshaping functions such as stacking, tiling, transposing, etc. The most useful operations is adding a new dimension, for example to turn a one-dimensional vector into a column vector. The second most useful is stacking vectors to form an array.

```
x = np.array([1,2,3])  # one-dimensional, shape=(3,)
x[:, np.newaxis]       # two-dimensional, shape=(3,1)
```

```
array([[1],
       [2],
       [3]])
```

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

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

Exercise (ex2). What is the relationship between a.shape and len(a)?

Exercise (ex3). Look up the NumPy help for np.arange and np.reshape, and use these functions to produce the 3×5 matrix

$$b = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

Look up the help for np.sum, and compute the length-5 vector of column sums and the length-3 vector of row sums.

Exercise (ex4). Find two different ways to use NumPy to create the column vector array([[1],[2],...,[n]]).

Exercise (ex5). A permutation matrix is a square matrix of 0s and 1s, where each row contains exactly one 1, and each column likewise. (The code snippet above for advanced indexing creates a 3×3 permutation matrix.) Write code to generate a random $n \times n$ permutation matrix.

NumPy also has a powerful tool called broadcasting which generalizes "add a scalar to a vector", and which is used a lot in more advanced array-manipulating code. It's more advanced than we need for this course, but it's used a lot in machine learning and it's worth reading about.

In Easter term, you will study linear algebra in the *Maths for Natural Sciences* course. If you want to try out the maths, you'll find relevant functions in `np.linalg` and `np.dual`.

2.4 Numerical optimization and fitting

A common task in science and in machine learning is to find the minimum value of a function, which may have one or more variables. For example, we might have a collection of points that more or less follow a straight line, and we might want to use the equation $y = mx + c$. In this case, we'd like to tune the values of m and c so that the equation lies close to the data. We can achieve this by defining a function $L(m,c)$ that measures how far the points are from the straight line, and then choosing m and c to minimize $L(m,c)$.

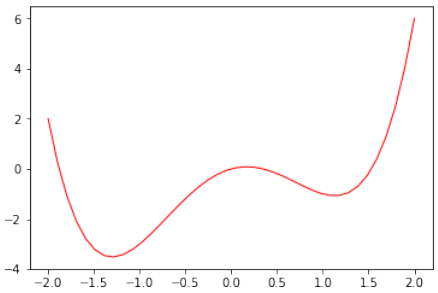
WARNING! The methods we discuss here sometimes work brilliantly, but sometimes are unstable. This is not the fault of Python or the libraries we are using. It's just the case that sometimes the equations in the algorithm and numerical issues in the data are not well balanced. This is something we need to bear in mind every time we use these methods, and we should check the output, for example by plotting graphs.

Let's start by minimizing a simple function of one variable. We could use calculus to find the minimum for a simple example like this, but let's do it with computer power instead.

```
def f(x, a, b, c):
    return a*x + b*(x**2) + c*(x**4)
```

We'll plot this function first, to get a rough idea of where the minimum should be. Visualisation is a crucial part of scientific computing, and we'll cover it in much more detail in §3 Working with Data, but for present purposes we'll just use some very simple plotting commands. The pyplot tutorial explains more options.

```
x = np.linspace(-2,2,40)          # 40 equally spaced points in the range [-2,2]
y = f(x, a=1, b=-3, c=1)          # f is already vectorized, because it only uses vectorized calculations
plt.plot(x, y, linestyle='-', linewidth=1, color='red')
plt.show()
```



The `scipy.optimize.fmin` function finds where the function achieves its minimum value, starting from an initial guess `x0`. The first argument is the function to optimize. In the snippet below we're giving it an anonymous function that is a version of `f` with the parameters `a`, `b` and `c` filled in.

```
scipy.optimize.fmin(lambda x: f(x,a=1,b=-3,c=1), x0=0.5)
```

```
Optimization terminated successfully.
         Current function value: -1.070230
         Iterations: 16
         Function evaluations: 32
```

```
array([ 1.13085938])
```

It found a local minimum, not the global minimum. This is often a problem with numerical optimization routines, and it's why it's helpful to look at the data first.

Exercise (ex6). What starting point do we need to give, so that it can find the global optimum? Run scipy.optimize.fmin for a range of values of x0 in the range $[-2,2]$. Tabulate x0 versus the minimizing x it finds. You can turn off the diagnostic output with the option disp=False.

Here is an example of a function of two variables. We'll try to fit the straight line $y = mx + c$ through a set of points. We'll define the *loss function*

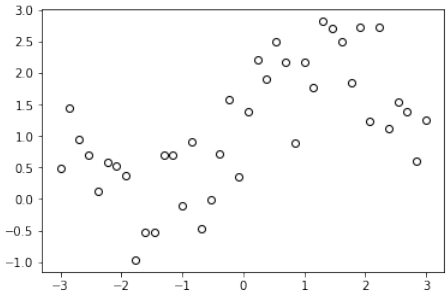
$$L(m,c) = \sum_i (mx_i + c - y_i)^2$$

and look for m and c to minimize it.

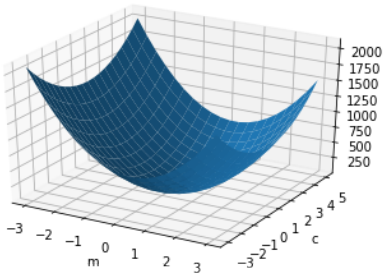
```
x = np.linspace(-3,3,40)
y = np.sin(x) + 2 * np.random.random(x.shape)

def loss(m,c):
    return np.sum((m*x+c - y)**2)
```

Before we do any numerical work, we should visualize what we're doing. Here are a plot of the points (x_i, y_i), and a surface plot of the loss function $L(m,c)$ as a function of m and c. The 3d plotting functions are somewhat mysterious, and you should look at relevant examples and copy them.

```
# Scatter plot of the data
plt.plot(x, y, marker='o', markerfacecolor='white', markeredgecolor='black', linestyle='None')
plt.show()
```




```
# Surface plot of loss(m,c)
# adapted from https://matplotlib.org/examples/mplot3d/surface3d_demo.html

# Create 2d arrays, one with each m value, one with each c value, one with loss
m,c = np.meshgrid(np.linspace(-3,3,20), np.linspace(-3,5,20))
l = np.zeros_like(m)
for i in range(l.shape[0]):
    for j in range(l.shape[1]):
        l[i,j] = loss(m[i,j], c[i,j])

from mpl_toolkits.mplot3d import axes3d   # import a library to allow 3d plots
axes = plt.figure().gca(projection='3d')  # say that we want axes for a 3d plot
axes.plot_surface(m, c, l)                # draw a surface plot
axes.set_xlabel('m')
axes.set_ylabel('c')
plt.show()
```



It doesn't look like $L(m,c)$ has any nasty surprises, so let's find the minimizing m and c.

```
# To optimize a function of several several variables, provide them as an array
# of the appropriate length.
optpars = scipy.optimize.fmin(lambda params: loss(params[0], params[1]), x0=[0,0])
optpars
```

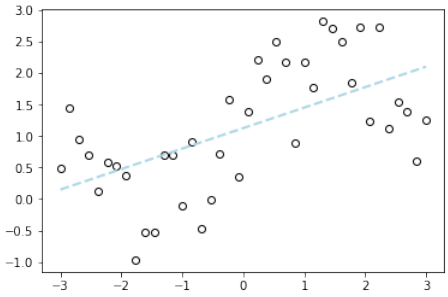
```
Optimization terminated successfully.
         Current function value: 25.961639
         Iterations: 58
```

```
         Function evaluations: 110
```

```
array([ 0.32481165,  1.12359353])
```

Now let's plot the data again, with the fitted straight line.

```
# Plot the points
plt.plot(x, y, marker='o', markerfacecolor='white', markeredgecolor='black', linestyle='None')
# Plot the fitted line
def fit(x):
    return optpars[0] * x + optpars[1]
plt.plot([-3,3], [fit(-3),fit(3)], linestyle='--', color='lightblue', linewidth=2)
# Show the graphic
plt.show()
```



2.5 Simulation

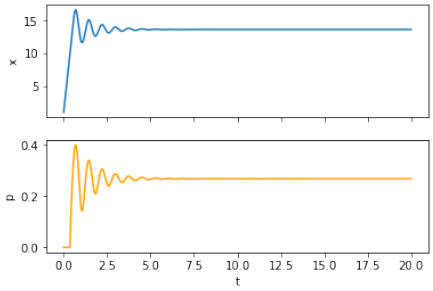
Simulation is a mainstay of scientific computing. A common style with numpy is to predefine an array to store the results, one row per timestep, and then iterate over timesteps gradually filling in the array. (This is the one case where `for` loops are appropriate.) Here's an example, a differential equation simulation. A model that has been proposed for TCP is

$$\frac{dx_t}{dt} = \frac{1}{\mathrm{RTT}^2} - p_{t-\mathrm{RTT}} x_{t-\mathrm{RTT}} \frac{x_t}{2}, \qquad p_t = \frac{\max(x_t - C, 0)}{x_t}$$

where x_t is the transmission rate of a sender at time t measured in packets per second, RTT is the round trip time, p_t is the packet drop probability, and C is the link capacity. We might simulate this as follows.

```
x0 = 1       # initial transmission rate, in pkt/sec
C = 10       # link capacity, in pkt/sec
T = 20       # simulated duration in seconds
RTT = 0.2    # round trip time in seconds
dt = 0.01    # timestep size
def P(x): return max(x-C,0) / x

# Initialization
num_iterations = np.ceil(T/dt).astype(int)
res = np.zeros((num_iterations, 3))  # a matrix to store t,x,p
res[0,1:] = [x0, P(x0)]
steps_back = int(RTT/dt)

# Loop
for i in range(1, num_iterations):
    (t,x,p),(xold,pold) = res[i-1], res[max(i-1-steps_back,0),1:]
    dx = 1/(RTT**2) - pold*xold*x / 2
    x = x + dx * dt
    p = P(x)
    res[i] = (t + dt, x, p)

# Plot the output (see Section 3 for more about plotting)
fig,(ax1,ax2) = plt.subplots(2, 1, sharex=True)
ax1.plot(res[:,0], res[:,1])
```

```
ax2.plot(res[:,0], res[:,2], color='orange')
ax1.set_ylabel('x')
ax2.set_ylabel('p')
ax2.set_xlabel('t')
plt.show()
```



- From a mathematical point of view this isn't ideal, because there are much more sophisticated numerical methods for solving differential equations.
- From a computer science point of view this isn't ideal, because the code tangles together the iteration logic with the logging logic. You may like to rewrite it using lazy lists.
- But from a scientific computing point of view, simulations like this are so easy to put together and learn from, that they are invaluable.

What we have written is called a discrete-time simulation, because time advances in fixed increments. In IA *Algorithms* you will study the *heap* data structure and event-driven simulation, in which time is pegged to changes in state.

More exercises

Exercise (ex7). In §1 you wrote a Pythonic simulator for a queue, based on Lindley's recursion

$$q_{t+1} = \max(q_t + a_t - C, 0).$$

It can be proved that this yields the same answer as

$$q_t = q_0 + x_t - \min(0, y_t),$$

where

$$x_t = \sum_{u=1}^{t} (a_u - C) \quad \text{and} \quad y_t = \min_{1 \le u \le t} (q_0 + x_u).$$

Compute $x = [x_1, x_2, \dots]$ using `np.cumsum`. Compute $y = [y_1, y_2, \dots]$ using `np.ufunc.accumulate`. Hence compute $q = [q_1, q_2, \dots]$. Check your answer against your Pythonic code.

Exercise. *Continuing from Assignment 2...* In the kinetic exchange model, the poorest and the richest might swap places after just one transaction, which isn't very likely. Consider a different model for exchange. As before, suppose that two individuals with wealth v and w respectively are paired, but now let their wealth be updated by

$$v_{\text{new}} = v + R \min(v, w), \quad w_{\text{new}} = w - R \min(v, w)$$

where R is now a random number in $[-1, 1]$, chosen independently for every pair at every timestep. The idea is that each party to the exchange puts up a certain amount of money, but no more than they can afford. Call this the *value transfer* model.

We can extend this model to include government intervention. Suppose the government charges a tax of say 20% on every exchange, collects all the tax revenue every timestep, and distributes it evenly to the entire population. This redistribution ought to have the effect of reducing inequality. Call this the *taxed value transfer* model. Here is a concrete example, for a population of size 6. $>$ 1. Initial wealth values are $[0, 2, 5, 3, 1, 2] > 1$. We pair adjacent individuals: $[0, 2; 5, 3; 1, 2] > 1$. Random exchange amounts pre-tax are $[0, 0; 2.6, -2.6; -0.4, 0.4] > 1$. Exchange amounts post-tax are $[0, 0; 2.08, -2.6; -0.4, 0.32] > 1$. Government revenue is $(2.6 - 2.08) + (0.4 - 0.32) = 0.6 >$ 1. Government redistributes $0.6/6 = 0.1$ to each person $>$ 1. Change in wealth is $[0.1, 0.1, 2.18, -2.5, -0.3, 0.42]$. $>$ 1. New wealth vector is $[0.1, 2.1, 7.18, 0.5, 0.7, 2.42]$.

Also, let's introduce another way to measure inequality. The Gini coefficient is unfamiliar to many people, and it's easier to communicate "The richest 1% of the population own x% of the wealth."

Implement the taxed value transfer model. Implement the "top x%" measure. Simulate the taxed value transfer model, and measure the Gini coefficient and the "richest 1%" statistic at each timestep. Plot these two measures, as a function of time, for tax rates 0%, 20% and 40%. See §3 for tips on plotting.

Exercise. *Continuing from the assignment...* Inflation means that wealth grows at an exponential rate, which is likely to exacerbate inequality. Compare a taxed value transfer model with tax rate 20% and no inflation, to a model with inflation of 3% per timestep. How much would tax have to increase, to compensate for this inflation?

Exercise. *Continuing from the assignment...* The economist Thomas Piketty argues that we have entered an age where the return on capital is greater than the growth due to income, and that this leads to higher inequality. We could incorporate income into the model by assigning each individual i a per-timestep income g_i, where the g_i are randomly chosen *a priori*. If there were no exchange, then wealth would grow linearly, and so the Gini coefficient would remain equal to `gini(g)`. Investigate what happens when we combine income, inflation, and random exchange. How well correlated are income and wealth? Do you agree with Piketty?

3. Working with data

Goal of this notebook:

Get used to the standard ways of thinking about data, *data frames* and *indexed arrays*

Learn how to use the Pandas library to handle data

See some more advanced plotting with matplotlib

There are some exercises in this section of notes. They are for you to get practice on: they are optional, and do not contribute to your final grade. You can check your answers to labelled exercises as described in §0.3 using `section='notes3'`.

3.0 Preamble

At the top of almost every piece of data-oriented scientific computing work, we'll import these standard modules.

```
# Import modules, and give them short aliases so we can write e.g. np.foo rather than numpy.foo
import numpy as np
import pandas
import matplotlib
import matplotlib.pyplot as plt
# The next line is a piece of magic, to let plots appear in our Jupyter notebooks
%matplotlib inline
```

The running example for this section is a dataset of stop-and-search records, made available by the UK home office. As it's a moderate-sized file (172MB) I like to download it to disk, so it's fast to reread it each time I restart the notebook. Here's how we can fetch a file from a url, using the Unix command-line tool `wget`. (The exclamation mark is called a *Jupyter magic*, and it means "Treat this line as though it were executed at the command prompt". In IB *Unix Tools* you'll learn more about the Unix command line.)

```
# Execute a unix command to download a file (if it's not already downloaded), and show download progress
import os.path
if os.path.exists('stop-and-search.csv'):
    print("file already downloaded")
else:
    !wget "https://teachingfiles.blob.core.windows.net/founds/stop-and-search.csv"
```

```
file already downloaded
```

3.1 What data looks like

We almost always work with data in the form of a spreadsheet-like table, referred to as a *dataframe*. Here's how to load a dataframe from a file and inspect it. (This dataframe will be used as a running example in the rest of §3.)

```
# Import a dataframe using the pandas library
stopsearch = pandas.read_csv('stop-and-search.csv')

# How many rows are there?
print("This dataset has", len(stopsearch), "rows")
# Display the first 3 rows. iloc[:3] means "select the first three rows"
stopsearch.iloc[:3]
```

```
This dataset has 808101 rows
```

```
            Type                        Date Part of a policing operation  \
0  Person search  2014-07-31T23:20:00+00:00                            NaN
1  Person search  2014-07-31T23:30:00+00:00                            NaN
2  Person search  2014-07-31T23:45:00+00:00                            NaN

   Policing operation   Latitude  Longitude Gender Age range  \
0                 NaN  50.938234  -1.388559   Male     25-34
1                 NaN  50.912978  -1.431990   Male   over 34
2                 NaN  51.005612  -1.497576   Male     10-17

                    Self-defined ethnicity Officer-defined ethnicity  \
0  Asian or Asian British - Pakistani (A2)                     Asian
1               White - White British (W1)                     White
2               White - White British (W1)                     White

                           Legislation  Object of search  \
0  Misuse of Drugs Act 1971 (section 23)  Controlled drugs
1  Misuse of Drugs Act 1971 (section 23)  Controlled drugs
2  Misuse of Drugs Act 1971 (section 23)  Controlled drugs

                           Outcome Outcome linked to object of search  \
0  Nothing found - no further action                              NaN
1         Suspect summonsed to court                              NaN
2  Nothing found - no further action                              NaN

  Removal of more than just outer clothing  year  month police_force
0                                      NaN  2014      8    hampshire
1                                      NaN  2014      8    hampshire
2                                      NaN  2014      8    hampshire
```

A dataframe is a collection of named columns. Each column has the same length, and all entries in a column have the same type, though different columns may have different types. If you have taken IA/IB *Databases*, you'll see that dataframes are similar to tables in a relational database. There are some differences:

- Scientific data is best thought of as logs of observations. Observed facts cannot be unobserved, so UPDATE and DELETE database operations are irrelevant, as are questions about database consistency.
- A dataframe has ordered rows, like an array, whereas a database table is unordered.
- We are often given messy badly structured data to work with; and we often create dataframes on the fly, work with them for a little while, then discard them. There is rarely a phase of entity-relationship modeling: instead we learn how to think about a dataset by working with it.

In Python, there are several choices about how to represent dataframes. A simple choice is as a dictionary of lists:

```
mydata = {'police_force': ['hampshire', 'hampshire', 'hampshire', ...],
          'Age range': ['25-34', 'over 34', '10-17', ...],
          'year': [2014, 2014, 2014, ...],
          ...}
```

We won't use this representation. Instead we'll use the Pandas library, designed specifically for working with data. It has several benefits:

- Data import and export has lots of fiddly corner cases. Even printing a dataframe takes a surprisingly large amount of code to do well.
- For fast numerical computation and concise code, `numpy` is best as we saw in §2. Pandas stores dataframe columns as `numpy` vectors.
- Some simple operations, like selecting a subset of rows, takes a lot of boilerplate code if implemented in pure Python. Much better to use a Pandas dataframe, which lets us write e.g. `stopsearch.iloc[:3]` to select some entries from each column, without our even having to think about iterating over columns.

It has the disadvantage of being yet another library to learn. It also has some idiosyncratic notation for indexing, which in my experience can lead to rather cryptic error messages when plotting, and which will be discussed in §3.2. It has poor support for missing values in data, which it inherits from `numpy`. Despite these problems, it's the best choice at this stage in Python's evolution.

3.2 Importing, exporting, and creating dataframes

It's very easy to import data from a simple comma-separated value (CSV) file. A CSV file looks like this:

```
"Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species"
5.1,3.5,1.4,0.2,"setosa"
```

```
4.9,3,1.4,0.2,"setosa"
4.7,3.2,1.3,0.2,"setosa"
4.6,3.1,1.5,0.2,"setosa"
5,3.6,1.4,0.2,"setosa"
```

i.e. a header line, then one line per row of the data frame, with values separated by commas. We've already seen how to import a CSV, using `pandas.read_csv`. If your file is nearly a CSV but has some quirks such as comments or a missing header row, experiment with the 55 options in `pandas.read_csv`. We can use the same function to read CSV files from remote urls (though if you're using Azure Notebooks, be aware that Azure only permits you to connect to Azure web servers.)

```
import urllib.request  # standard Python library for url requests
iris = pandas.read_csv(urllib.request.urlopen('https://teachingfiles.blob.core.windows.net/scicomp/iris.csv'))
```

In my experience, around 70% of the time you spend working with data will be fighting to import it and clean it up. See §A3 for a collection of recipes for web scraping, reading from a database, and parsing log files.

To write a CSV file,

```
iris.to_csv('iris.csv', index=False)
```

If you're running this notebook with Azure Notebooks, you would then use the `Data | Download` menu to download the file from Azure Notebooks to your local machine.

To create a dataframe from scratch, pass in a dictionary of columns. Python dictionaries are unordered, so you can optionally specify the column order you want with the `columns` argument.

```
iris = pandas.DataFrame({'species': ['setosa', 'virginica', 'virginica', 'setosa', 'versicolor'],
                         'Petal.length': [1.0, 5.0, 5.8, 1.7, 4.2],
                         'Petal.width': [0.2, 1.9, 1.6, 0.5, 1.2]},
                        columns = ['species', 'Petal.length', 'Petal.width'])
```

Or you can create a dataframe from a list of tuples. Now the `columns` argument is needed to say what the column names are.

```
iris = pandas.DataFrame([('setosa', 1.0, 0.2), ('virginica', 5.0, 1.9), ('virginica', 5.8, 1.6),
                         ('setosa', 1.7, 0.5), ('versicolor', 4.2, 1.2)],
                        columns = ['species', 'Petal.length', 'Petal.width'])
```

A dataframe behaves like a dictionary of vectors, and you can add and remove columns using the same syntax you'd use for dictionaries. When you add new columns, Pandas converts them to `numpy` vectors for you, so you can use the usual `numpy` operations. (The columns aren't actually plain `numpy` vectors, as §3.3 explains, and the difference will bite you whenever you try to subset a column.)

```
iris.keys()                                              # what column names are present?
iris['Sepal.length'] = [4.6, 6.3, 7.2, 5.1, 5.7]         # add a column
if 'Petal.width' in iris: del iris['Petal.width']        # delete a column (if present)
iris['P/S'] = iris['Petal.length'] / iris['Sepal.length']  # vectorized whole-column operation
iris
```

```
       species  Petal.length  Sepal.length       P/S
0       setosa           1.0           4.6  0.217391
1    virginica           5.0           6.3  0.793651
2    virginica           5.8           7.2  0.805556
3       setosa           1.7           5.1  0.333333
4   versicolor           4.2           5.7  0.736842
```

To modify an entry in a dataframe, I prefer to "think vectorized" and replace the entire column, e.g.

```
iris['Petal.length'] = np.where(np.arange(5)==0, 100, iris['Petal.length'])
```

If we modify only some of the rows of a dataframe, then (depending how exactly we do it) Pandas will tell us off, warning us that the operation may be inefficient.

```
iris['Petal.length'][0] = 200
```

```
/home/djw/.local/lib/python3.6/site-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-
versus-copy
  """Entry point for launching an IPython kernel.
```

3.3 Selecting from a dataframe

Dataframes have a triple identity: part array, part database table, part dictionary. Because of this, there are several ways to select subsets of rows and columns.

Selecting columns is easy. We've already seen how to select one column. To select several columns,

```
# Select three columns, and print out the first 5 rows of the resulting dataframe
stopsearch[['Date', 'Object of search', 'Legislation']].iloc[:5]
```

```
                       Date          Object of search  \
0  2014-07-31T23:20:00+00:00          Controlled drugs
1  2014-07-31T23:30:00+00:00          Controlled drugs
2  2014-07-31T23:45:00+00:00          Controlled drugs
3  2014-08-01T00:40:00+00:00              Stolen goods
4  2014-08-01T02:05:00+00:00  Article for use in theft

                                    Legislation
0              Misuse of Drugs Act 1971 (section 23)
1              Misuse of Drugs Act 1971 (section 23)
2              Misuse of Drugs Act 1971 (section 23)
3  Police and Criminal Evidence Act 1984 (section 1)
4  Police and Criminal Evidence Act 1984 (section 1)
```

To select subsets of rows by row number use `.iloc[row_indexes]`. This can take any slice or list of integers.

```
stopsearch.iloc[:3]       # the first 3 rows; stopsearch[:3] is an abbreviation for this
stopsearch.iloc[[0,3,5]]  # select several rows, by providing a list of row numbers
stopsearch.iloc[[5]]      # select a dataframe object consisting of just one row
stopsearch.iloc[:3][['Date', 'Object of search']]  #first a row selector, then a column selector
```

To select subsets of rows by a boolean vector use `.loc[rows_indicator]` where `rows_indicator` is a boolean vector as long as the dataframe. You can optionally also specify a list of columns to select.

```
wantcols = ['Date', 'Object of search', 'Outcome']
wantrows = stopsearch['police_force'] == 'cambridgeshire'  # a boolean vector
stopsearch.loc[wantrows, wantcols][:3]
```

There is a third way to select rows from a dataframe, which in my experience is the source of endless confusion: selecting by row index.

When you see a Pandas dataframe printed out, there is a column at the left printed in bold. These aren't row numbers, they are *row indexes*, which behave like the keys in a dictionary. In all the examples we've seen so far the indexes happen to be numbers, but they could be any other Python object. Pandas remembers row indexes, even when you pull out a single column, and it always tries to match indexes. This is usually not what we want. I recommend that you **always use .values when you are working with subsets of rows.** This gives you the actual `numpy` vector behind the column, not the confusing Pandas vector-plus-index object.

We won't be using row indexes in this course, but it's worth knowing they exist so you can understand the cryptic errors and error messages you will undoubtedly come across.

```
df = pandas.DataFrame({'x': [3,3,4,8,2,7,6]})

# This looks like it's adding [3,3,4,8] and [8,2,7,6] -- but it's not!
df['x'][:4] + df['x'][-4:]

# This is the way to do it
df['x'][:4].values + df['x'][-4:].values
```

3.4 Tabulations and indexed arrays

The pattern behind much data processing is split-apply-combine-join: split your data into pieces, apply a transformation to each piece, combine the pieces, and join results from different datasets together. We could code this explicitly with a `for` loop, but it would involve lots of boilerplate code — and I hope you have been persuaded by §2 that `for` loops are considered harmful. Instead, let's see how to do it with Pandas.

The following line of code performs a cross-tabulation: it splits the data into a separate dataframe for each combination of officer-defined ethnicity and gender, applies the `len` function to each sub-dataframe to get the number of rows it contains, and combines the results into a single indexed object.

```
x = stopsearch.groupby(['Officer-defined ethnicity', 'Gender']).apply(len)
x
```

```
Officer-defined ethnicity  Gender
Asian                      Female      2230
                           Male       72458
                           Other         85
Black                      Female      7497
                           Male      151988
                           Other       189
Mixed                      Female        22
                           Male         327
Other                      Female       974
                           Male       16221
                           Other        147
White                      Female     47947
                           Male      414428
                           Other        986
dtype: int64
```

- If you have a function you want to apply to only one column, use `dataframe.groupby(...)[colname].apply(fun)`.
- For this course, we will only `apply` functions that return simple Python values. It's more complicated to apply functions that return dataframes or Pandas columns or indexed arrays.

The groupby/apply command has produced an *indexed array*. An indexed array is a cross between a normal numpy array and a dataframe. We access elements and sub-arrays by dimension, like a numpy array — but we use `.loc[]` and the indexes aren't integer positions, they're values from the underlying column. Also, the array might be 'incomplete', as in the example above which has no entry for `['Mixed','Other']`.

```
x.loc['Asian']              # select a sub-array indexed just by Gender
x.loc[:, 'Other']           # select a sub-array indexed by Ethnicity (all levels)
x.loc[['Mixed','White']]    # select a sub-array indexed by Ethnicity (two levels) and Gender (all levels)
# The index labels can be accessed with  x.index.levels[0].values and x.index.levels[1].values
```

To pretty-print an indexed array, use `unstack()`. It will by default fill in any missing values with `NaN` (not a number), and you can override this with `fill_value`.

```
x[['Mixed','White']].unstack(fill_value=0)
```

```
Gender                     Female    Male  Other
Officer-defined ethnicity
Mixed                          22     327      0
White                       47947  414428    986
```

When you first start working with data, I recommend you do all your calculations on dataframes rather than indexed arrays. If you want to do calculations on an indexed array, first turn it into a dataframe. There are two ways to do this, depending on whether you want it *long form* or *wide form*.

```
# Convert an indexed array into a long-form dataframe
x[['Mixed','White']].reset_index()
```

```
  Officer-defined ethnicity  Gender       0
0                     Mixed  Female      22
1                     Mixed    Male     327
2                     White  Female   47947
3                     White    Male  414428
4                     White   Other     986
```

```
# Convert an indexed array into a wide-form dataframe. See note below about rename_axis
x[['Mixed','White']].unstack(fill_value=0).reset_index().rename_axis(None, axis=1)
```

```
  Officer-defined ethnicity  Female    Male  Other
0                     Mixed      22     327      0
1                     White   47947  414428    986
```

As you get deeper into working with data, you'll discover that the skill in working with data is knowing which representation works best for your task, dataframe or indexed array. Also,

- Pandas blurs the boundary between dataframes and indexed arrays
- Both rows and columns can have hierarchical indexes, called multi-indexes
- For clever tricks with higher-dimensional indexed arrays, read the documentation for `unstack` and `rename_axis`
- When you read the documentation or look for help, please note that what I'm calling an indexed array, Pandas calls a `Series`.

Exercise (ex1). Here is a function for finding the most frequent value in a vector:

Generate a table showing the most frequent age range for each combination of ethnicity and gender, in the stop-and-search dataset.

Exercise (ex2). Given the dataframe

pretty-print an indexed array that has rows for A, columns for B, and shows values of X. [Hint. Make sure your apply function returns a simple Python value, not a Pandas column or dataframe.]

3.5 Joining dataframes

Suppose we've got a table that lists, for each ethnic group and possible outcome, the number of such cases:

```
# The 'Outcome' column has lots of possible values -- let's simplify it to just two
stopsearch['outcome'] = np.where(stopsearch['Outcome'] == 'Nothing found - no further action', 'nothing',
'find')
x = stopsearch.groupby(['Officer-defined ethnicity', 'outcome']).apply(len).reset_index(name='n')
x
```

	Officer-defined ethnicity	outcome	n
0	Asian	find	23864
1	Asian	nothing	55628
2	Black	find	52161
3	Black	nothing	111695
4	Mixed	find	212
5	Mixed	nothing	138
6	Other	find	6117
7	Other	nothing	12363
8	White	find	157927
9	White	nothing	325545

We might want to know `outcome==find` is equally likely for each ethnic group (to test for ethnic bias in police behaviour). We could achieve this with a `for` loop, looking at each ethnic group in turn — but `for` loops are considered harmful, and a more idiomatic way to do it is by combining tables, as follows. First, create a summary table that lists, for each ethnic group, the total number of cases:

```
y = x.groupby('Officer-defined ethnicity')['n'].apply(sum).reset_index(name='ntot')
y
```

	Officer-defined ethnicity	ntot
0	Asian	79492
1	Black	163856
2	Mixed	350
3	Other	18480
4	White	483472

Second, divide the values in the first table by the corresponding values in the second table. The key word here is *corresponding*. This is the equivalent of `JOIN` in SQL, which you learn about in IA/IB *Databases*.

```
z = x.merge(y, on='Officer-defined ethnicity')  # you can join on a column name, or a list of column names

# Compute the ratio n/ntot, and display percent_find for each ethnicity
z['percent_find'] = z['n'] / z['ntot'] * 100
z.loc[z['outcome']=='find', ['Officer-defined ethnicity','percent_find']]
# (If we select rows first and then add a column, Pandas will tell us off for trying to modify part of a
dataframe.)
```

	Officer-defined ethnicity	percent_find
0	Asian	30.020631
2	Black	31.833439
4	Mixed	60.571429
6	Other	33.100649
8	White	32.665180

Pandas also lets us join indexed arrays on their common indices, and that would be a more natural way to write this calculation; but that counts as more advanced Pandas usage than we will cover here.

Exercise (ex3). Use merge to produce a dataframe with one row per ethnic group, one column for the number of cases of outcome==find, and one column for the number of cases with outcome==nothing. Use this table to compute the percent_find column.

3.6 Extended example

Here is an extended example, a deeper look at the stop and search dataset. The goal of this section is to see how everything we've learnt comes together. Every piece of data work is different, so don't worry about the details of this example. Instead you should skim through and just look at the pictures, and come back to use this section as a reference if you find yourself wondering how to produce a similar picture. The general themes illustrated in this example are:

- What are common steps we might take when approaching a new dataset?
- How do all the various pieces of Python and numpy come together when tidying a dataset?
- See a more involved use of `merge`
- Get a flavour of advanced plotting with matplotlib

Matplotlib is a huge and not very coherent library. You'll see some of the most important parts here, but you'll need to make frequent use of Google, Stack Overflow, the matplotlib gallery, maybe even the documentation.

The first step is analysing a dataframe is always to look at a few of the rows, as we did right at the top of this notebook. I like to look at a random sample of rows, to get a picture of variety in the data.

```
stopsearch.iloc[np.random.choice(len(stopsearch), size=3)]
```

```
                Type                       Date Part of a policing operation  \
364933  Person search  2016-01-22T00:10:00+00:00                         False
741425  Person search  2017-04-29T14:50:00+00:00                         False
222012  Person search  2015-09-01T12:00:00+00:00                           NaN

        Policing operation   Latitude  Longitude Gender Age range  \
364933                 NaN  51.252699  -0.015391   Male     18-24
741425                 NaN  51.473878  -0.088951   Male     18-24
222012                 NaN        NaN        NaN   Male     25-34

                   Self-defined ethnicity Officer-defined ethnicity  \
364933          White - White British (W1)                      White
741425  White - Any other White background                        NaN
222012          White - White British (W1)                      White

                                Legislation  Object of search  \
364933                                  NaN      Stolen goods
741425  Misuse of Drugs Act 1971 (section 23)  Controlled drugs
222012  Misuse of Drugs Act 1971 (section 23)               NaN

                                      Outcome  \
364933       Nothing found - no further action
741425  Offender given drugs possession warning
222012       Nothing found - no further action

       Outcome linked to object of search  \
364933                                NaN
741425                                NaN
222012                                NaN

       Removal of more than just outer clothing  year  month  police_force  \
364933                                      NaN  2016      1        surrey
741425                                      NaN  2017      4  metropolitan
222012                                      NaN  2015      9  metropolitan

        outcome
364933  nothing
741425     find
222012  nothing
```

(a) Investigate categorical columns

Let's look at one column in detail, `Age range`. Entries in this column take one of a small number of possible values; it is what is known as a *categorical* variable. In other languages we'd store it as an enumeration type, but Python and numpy have poor support for enumerations, and it's typical to either store it as a string or (if the dataset is huge) to code it as an integer.

We could use any numpy routines we like to summarize the column, but Pandas column objects have a helpful method `describe()`, which is polymorphic and displays appropriate information for most types of column. (If we want to call `describe` on a numpy vector, we have to convert it to a Pandas column object first: `pandas.Series(x).describe()`.)

```
stopsearch['Age range'].describe()
```

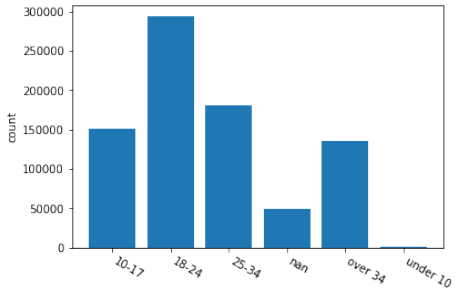
```
count     759820
unique         5
top        18-24
freq      292998
Name: Age range, dtype: object
```

It tells us, somewhat unhelpfully, that this column is of type `object`. Numpy insists that each item in a vector or array have exactly the same length, whereas strings are variable length, so Pandas has stored it as a vector of *pointers* to string objects, so it displays as `dtype=object`. Numpy can also store vectors of fixed-length strings, but this is rarely a good idea. In my experience, numpy has poor support for strings (as one might guess from its name), and I prefer to use general-purpose Python functions for string handling.

The `describe()` method has gotcha behaviour when there are missing values. When Pandas reads a CSV, if there are missing values then Pandas stores them as NaN (Not a Number — a floating point value, despite its name!). The count printed out by `describe()` doesn't include NaNs.

The standard way to depict a categorical variable is with a histogram. In matplotlib, we have to do the counting ourself and then draw bars with `matplotlib.pyplot.bar`. Missing values are a gotcha: we can either convert NaN into the string `'nan'` and show it in the plot, or we can discard NaN values using `pandas.isnull`.

```
x = stopsearch['Age range']

# Count the number of occurrences of each value in the column
vs,ns = np.unique(x[~pandas.isnull(x)], return_counts=True)  # discard NaN values,
vs,ns = np.unique(x.astype(str), return_counts=True)         # or convert NaN to 'nan'

xs = np.arange(len(vs))                                      # pick x coordinates
plt.bar(xs, ns, align='center')                              # you can also set width=...
plt.xticks(xs, vs, rotation=-30, ha='left')                  # specify and label the x ticks
plt.ylabel('count')
plt.show()
```



Here's another display, a table showing the most frequent items and their counts. We're using `np.argsort(-ns)` to rearrange the rows in the order of decreasing `ns`, and keeping the first three rows.

```
pandas.DataFrame({'Age range':vs, 'n':ns}, columns=['Age range','n']).iloc[np.argsort(-ns)[:3]]
```

```
  Age range       n
1     18-24  292998
2     25-34  180725
0     10-17  150286
```

(b) Investigate numerical columns

Numerical columns also have a `describe()` method, which prints out some standard numerical summaries. The only numerical columns in this dataset are latitude and longitude, which it's daft to summarize numerically, but here goes.

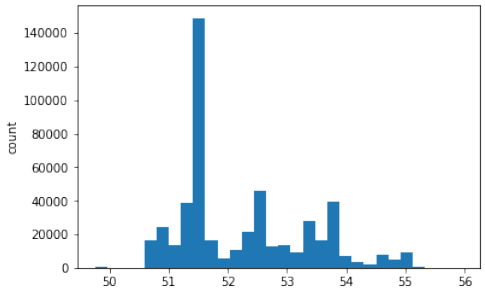
```
stopsearch['Latitude'].describe()
```

```
count    495557.000000
mean         52.253553
std           1.099470
min          49.766800
25%          51.469594
```

```
50%          51.653199
75%          53.161358
max          55.945400
Name: Latitude, dtype: float64
```

The obvious way to plot a numerical column is with a histogram. The matplotlib `hist` command doesn't work if there are missing values, so they must be removed first.

```
x = stopsearch['Latitude']
plt.hist(x[~pandas.isnull(x)], bins=30)
plt.ylabel('count')
plt.show()
```



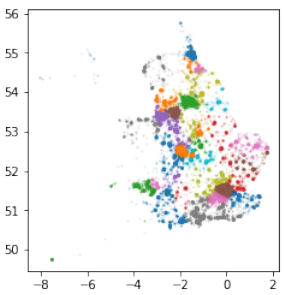
It's sometimes helpful to split a numerical column into categories. Here is a cheap-and-cheerful hack at geocoding. We could achieve the same effect ourselves by fiddling with `np.digitize`, but `pandas.cut` does everything for us. If we want to cut into categories of roughly equal size, use `pandas.qcut`, or fiddle around with `np.nanpercentile`,

```
region = pandas.cut(stopsearch['Latitude'], bins=[-np.inf, 51.98, 55.27, np.inf],
labels=['south','mid/north','scotland'])
ns,vs = np.unique(region.astype(str), return_counts=True)
pandas.DataFrame({'region':vs, 'n':ns})
```

```
            n  region
0  mid/north  231792
1         nan  312544
2   scotland     304
3      south  263461
```

Here's a more intelligent plot of latitude and longitude, a scatter plot, coloured by police force. We iterate through the police forces and draw a scatter plot for each. The scatterplot automatically picks a new colour each time. (You can also use a mapping library like gmplot, which puts nice interactive maps in your Jupyter notebook, using Google map tiles.)

```
# Plot a sample of points, because there's no point plotting loads of overlaps
df = stopsearch.iloc[np.random.choice(len(stopsearch), size=100000)]

for pf in np.unique(df['police_force']):
    i = df['police_force'] == pf
    plt.scatter(df['Longitude'][i], df['Latitude'][i], s=1, alpha=.1) # control size and alpha

# Set the aspect ratio, based on the UK's average latitude
plt.gca().set_aspect(1/np.cos(54/360*2*np.pi))
plt.show()
```



(c) Investigate date/time columns

Dates and times are always a nuisance because of timezones, as xkcd observes — and also because of unequal months and leap years and so on, which make it hard work to get axes right in plots. But they're also very common, so here are a few recipes.

Let's look at some typical values in the `Date` column.

```
np.random.choice(stopsearch['Date'], 5)
```

```
array(['2016-08-26T21:13:00+00:00', '2015-11-15T17:45:00+00:00',
       '2014-12-02T13:10:00+00:00', '2015-11-20T20:00:00+00:00',
       '2015-04-10T13:45:00+00:00'], dtype=object)
```

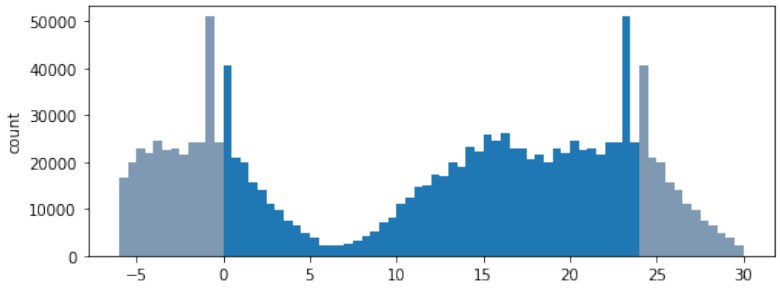
To find the time of day of each entry (measured in seconds since midnight), we can just trim the string, split it at colons, and add up the pieces. By using `np.vectorize` we can apply this to the entire vector.

```
def as_timeofday(s):
    return sum(int(x)*y for x,y in zip(s[11:19].split(':'), [3600,60,1]))
timeofday = np.vectorize(as_timeofday)(stopsearch['Date'])
```

Here's a plot, a histogram of the time of day of stop-and-search events. There are two fancy things going on here:

- The plot size is set using `plt.rc_context`, which lets you control size and many other options.
- Colours can be specified by name, by string for a grey-level, or by an (r,g,b) tuple.
- Since stop-and-search events happen throughout the night, it's good to show context around midnight. Here we're showing three versions of the histogram in the same plot, and controlling the colour to draw the viewer's attention to a 24-hour window.

```
ONEHOUR = 3600
with plt.rc_context({'figure.figsize': (8,3)}):
    plt.hist(timeofday/ONEHOUR, bins=np.linspace(0,24,48+1))
    plt.hist(timeofday[timeofday>18*ONEHOUR]/ONEHOUR-24, bins=np.linspace(-6,0,12+1), color=(.5,.6,.7))
    plt.hist(timeofday[timeofday<6*ONEHOUR]/ONEHOUR+24, bins=np.linspace(24,30,12+1), color=(.5,.6,.7))
plt.ylabel('count')
plt.show()
```



For the fancy plots that follow, we'll want to work with dates. In my experience, there is only one way to maintain your sanity when working with dates and times:

- Store dates and datetimes as Unix timestamps, i.e. as seconds since 1970-01-01 00:00:00.
- If timezones are irrelevant in your dataset, then just pretend everything is all UTC.
- If timezones are relevant in your dataset, then store UTC timestamps, and store an additional column for timezone offset. Do not be tempted to store local time.

Every language and every library and every database has its own quirks at dealing with datetimes, and it's not worth wasting your time on them, except for learning how to convert to a timestamp.

```
import datetime, pytz

def as_timestamp(s):
    t = datetime.datetime.strptime(s[:10], '%Y-%m-%d')  # create a naive datetime (with no timezone specified)
    u = t.replace(tzinfo=pytz.UTC)  # set timezone=UTC, otherwise timestamp() is unpredictable
    return int(u.timestamp())

# Create a column with the date, stored as a Unix timestamp
stopsearch['t'] = np.vectorize(as_timestamp)(stopsearch['Date'])
```

(d) Clever joins

Let's analyse in detail the number of stop-and-search events per day. The naive method, `stopsearch.groupby('Date').apply(len)`, doesn't make sense: it only aggregates the data there is, so it won't produce a row saying "On date *d*, there were zero events". We need to somehow augment this table to include all possible dates.

Further challenge: this dataset happens to be uneven, with different dates of coverage for different police forces, as this very terse snippet shows. (You'll find yourself writing very powerful one-liners for quick looks at the data. Don't leave them in your finished code, because no one else will be able to understand them. This one is left in as an exercise!)

```
stopsearch.groupby('police_force')['t'].apply(lambda t:
(max(t)-min(t))/(3600*24)).astype(int).reset_index(name='days')[:5]
```

	police_force	days
0	avon-and-somerset	912
1	bedfordshire	851
2	btp	852
3	cambridgeshire	731
4	cheshire	853

To simplify the problem, let's restrict attention to a single police force. We'll assume that the data covers all incidents from the earliest date to the latest date present. This assumption should be verified by checking the dataset's original documentation — and remember, "absence of evidence is not evidence of absence".

```
# Count the number of stop-and-search per day, grouped by outcome, for Cambridgeshire police force
df = stopsearch.loc[stopsearch['police_force'] == 'cambridgeshire', ['t', 'Outcome']]
df['outcome'] = np.where(df['Outcome'] == 'Nothing found - no further action', 'nothing', 'find')
df = df.groupby(['t', 'outcome']).apply(len).reset_index(name='n')

# Build a new dataframe consisting of all possible combinations of 't' and 'find'
# In database language, this is called an 'outer join', and Pandas forces us to do it on a dummy key.
alldays = pandas.DataFrame({'t': range(np.min(df['t']), np.max(df['t']), 3600*24)})
allfind = pandas.DataFrame({'outcome': ['nothing', 'find']})
alldays['k'] = 0
allfind['k'] = 0
allrows = alldays.merge(allfind, on='k', how='outer')[['t','outcome']]

# Merge the dataframe of counts, and the dataframe of all possible rows.
# The 'n' column will be filled with NaN for dates/outcomes not present in df, so turn them to 0.
df = df.merge(allrows, on=['t','outcome'], how='outer')
df['n'] = np.where(pandas.isnull(df['n']), 0, df['n'])

# Reshape the dataframe, using merge, so it has one row per date, and columns for each outcome.
# (This could also be done by converting a suitable indexed array to a wide-form dataframe.)
x = df.loc[df['outcome'] == 'nothing', ['t','n']]
y = df.loc[df['outcome'] == 'find', ['t','n']]
df = x.merge(y, on='t', suffixes=('_nothing','_find'))
```

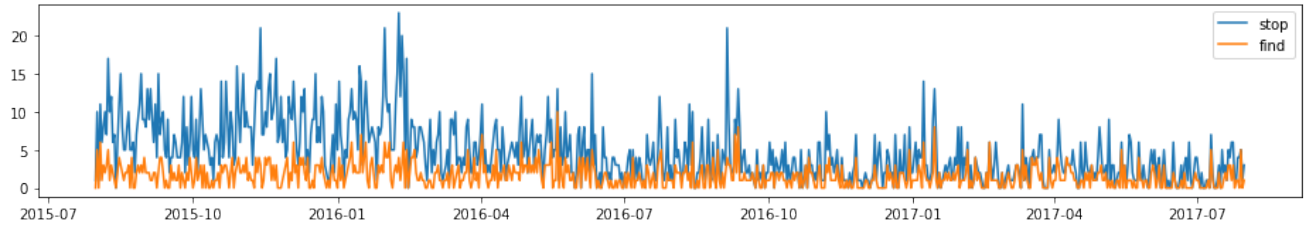
(c) Fancy plots

If you want to save any of these plots, you can right-click on them in the browser and choose "save as", to save a bitmap. To save as PDF, use `plt.savefig(filename, transparent=True, bbox_inches='tight', pad_inches=0)` and then (on Azure Notebooks) choose the `Data | Download` menu option.

How many stops are there each day by the Cambridgeshire police force, and how any of them result in something being found? The plot below is a *grouped plot*, showing multiple pieces of data. For each piece, we call the usual plot command, and we also provide `label`. This picks a different colour for each piece, and it remembers the association between colours and labels, so that `plt.legend()` can draw the right thing.

```
# Sort the data by date, because otherwise the plotted lines could go back and forth across the plot
df = df.iloc[np.argsort(df['t'])]

# Make real date objects, so that matplotlib knows how to format the axis nicely
dates = np.vectorize(datetime.datetime.utcfromtimestamp)(df['t'])

# A grouped plot
```

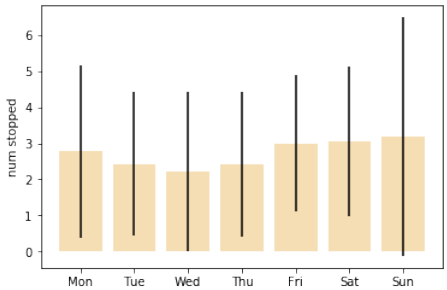
```
with plt.rc_context({'figure.figsize': (16,2.5)}):
    plt.plot(dates, df['n_find']+df['n_nothing'], label='stop')
    plt.plot(dates, df['n_find'], label='find')
plt.legend()
plt.show()
```



Let's next look at police activity by day of the week, as measured by average number of stop-and-search events, and let's also show the variability as measured by the standard deviation. This involves computing two summaries of the data, combining them, and plotting the result.

In this code I'm doing my own arithmetic on timestamps to get day of week. I find it's easier to do a little bit of arithmetic than to wade through library documentation about datetime utility functions.

```
# Compute the weekday for each row.
# Timestamp is measured in seconds since Thu 1970-01-01 00:00, so this computation gives me 0=Mon, 1=Tue, ...
df['weekday'] = (df['t']//(24*3600) - 4) % 7
weekday_names = ['Mon','Tue','Wed','Thu','Fri','Sat','Sun']

# The previous plot suggests there's a change in policing after 2016-09,
# so let's restrict attention to that range
df['stops'] = df['n_find'] + df['n_nothing']
t0 = datetime.datetime.strptime('2016-10-01', '%Y-%m-%d').replace(tzinfo=pytz.utc).timestamp()
df2 = df.loc[df['t'] >= t0]

# Compute the two statistics we want to plot: mean and standard deviation of number of stops
x = df2.groupby('weekday')['stops'].apply(np.mean).reset_index(name='mean_stops')
y = df2.groupby('weekday')['stops'].apply(np.std).reset_index(name='sd_stops')
stats = x.merge(y, on='weekday')

# A bar plot with error bars
plt.bar(stats['weekday'], stats['mean_stops'], align='center', facecolor='wheat')
plt.errorbar(stats['weekday'], stats['mean_stops'], yerr=stats['sd_stops'], linestyle='None', ecolor='black')
plt.xticks(range(0,7), weekday_names)
plt.ylabel('num stopped')
plt.show()
```

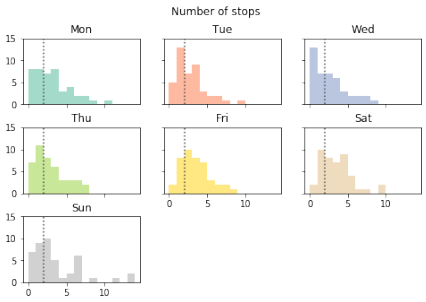


Our final plot is called a *facet plot* or a *panel plot* or a *small multiples* plot. According to the plotting guru Edward Tufte,

> At the heart of quantitative reasoning is a single question: Compared to what? Small multiple designs, multivariate and data bountiful, answer directly by visually enforcing comparisons of changes, of the differences among objects, of the scope of alternatives. For a wide range of problems in data presentation, small multiples are the best design solution.

Let's plot a histogram for each day of the week, showing the distribution of the number of stops made.

```
with plt.rc_context({'figure.figsize': (8,5), 'figure.subplot.hspace': 0.35}):

    # Get a colour map with 7 pretty colours: see https://matplotlib.org/users/colormaps.html
    cols = plt.get_cmap('Set2', 7)

    fig = plt.figure()
    for i, weekday in enumerate(range(7)):
        # Create a new facet
        ax = fig.add_subplot(3, 3, i+1)

        # Plot the histogram. Force the x-axis bins and y-axis range to be identical in each facet
        ax.hist(df2.loc[df2['weekday']==weekday, 'stops'], alpha=.6, facecolor=cols(i), bins=range(15))
        ax.set_ylim(0,15)

        # Some annotations.
        # Suppress the tick labels, except on the outside facets.
        ax.axvline(x=np.median(df2['stops']), linestyle='dotted', color='.3')
        ax.set_title(weekday_names[weekday])
        if i < 4: ax.set_xticklabels([])
        if (i % 3) != 0: ax.set_yticklabels([])

fig.suptitle('Number of stops')
plt.show()
```



There are two ways to generate a facet plot.

- When we want facets for one variable, as is the case here for the variable `weekday`, use `ax = fig.add_subplot(nrows, ncols, i)` to add facets one by one. This creates a new facet in position `i`, from a `nrows * ncols` grid arrangement.
- When we want facets for two variables, use `fig,ax_array = plt.subplots(nrows,ncols)`, as in these demos, to create a full grid of facets in one go.

Previously we have used commands like `plt.hist()` to draw histograms, but this code uses `ax.hist()` where `ax` is an `Axes` object. All the matplotlib `plt.*` plotting routines are actually `Axes` methods, and the `plt.*` names are just aliases. For a plot with a single facet, you can use either style.

A1. Python language choices

Goal of this notebook: See how Python fits in with the ideas you learnt in IA *Concepts in programming languages* and *Object oriented programming*.

Python development documents are called *Python Enhancement Proposals* (PEPs). Significant changes in the language, or in the standard libraries, are discussed in mailing lists and written up for review as a PEP. PEPs typically suggest several ways a feature might be implemented, and give the reason for choosing one of them. If consensus is not reached to accept the PEP, then the reasons for its rejection are also documented. They are fascinating reading if you are interested in programming language design.

The PEPs are a lot of reading, and so this document is a quick summary of certain features of Python and how they relate to ML and Java.

A1.1 Dynamic typing

Consider the functions

```
def double_items(xs):
    return [x*2 for x in xs]
def goodfunc():
    return double_items([1,2,[3,4]]) + double_items("hello world")
def badfunc():
    return double_items(10)
```

Python uses *dynamic typing*, which means that values are tagged with their types during execution and checked only then. then we won't be told of any errors until `badfunc()` is invoked, even though it's clear before executing it that `badfunc` will fail.

Python programmers are encouraged to use *duck typing*, which means that you should test values for what they can do rather than what they're tagged as. "If it walks like a duck, and it quacks like a duck, then it's a duck". In this example, `double_items(l)` iterates through `xs` and applies `*2` to every element, so it should apply to any `xs` that supports iteration and whose elements all support `*2`. These mean different things: iterating over a list returns its elements while iterating over a string returns its characters; doubling a number is an arithmetical operation, doubling a string or list repeats it. Python does allow you to test the type of a value with e.g. `if isinstance(x, list): ...`, but programmers are encouraged not to do this.

Python's philosophy is that library designers are providing a service, and programmers are adults. If a library function uses comparison and addition, and if the end-user programmer invents a new class that supports comparison and addition, then why on earth shouldn't the programmer be allowed to use the library function? (I've found this useful for simulators: I replaced "real-valued timestamp" with "rich timestamp class that supports auditing, listing which events depended on which other events", and I didn't have to change a single line of the simulator body.) Some statically typed languages like Haskell and Scala support this via *dynamic type classes*, but their syntax is rather heavy.

To make duck typing useful, Python has a long list of special method names so that you can create custom classes supporting the same operations as numbers, or as lists, or as dictionaries. For example, if you define a new class with the method `__iter__` then your new class can be iterated over just like a list.

Example: trees

Suppose we want to define a tree whose leaves are integers and whose branches can have an arbitrary number of children. Actually, in Python, there's nothing to define: we can just start using it, using a list to denote a branch node.

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

To flatten a list like this we can use duck typing: given a node `n`, try to iterate over its children, and if this fails then the node must be a leaf so just return `[n]`.

```
def flatten(n):
    try:
        return [y for child in n for y in flatten(child)]
    except TypeError as e:
        return [n]
```

```
flatten(x)  # returns [1, 2, 4, 3, 9, 5, 6, 7, 8]
```

This would work perfectly well for trees containing arbitrary types — unless the end-user programmer puts in leaves which are themselves iterable, in which case the duck typing test doesn't work — unless that is the user's intent all along, to be able to attach new custom sub-branches...

A solution is to define a custom class for branch nodes, and use `isinstance` to test each element to see if it's a branch node. This is not very different to the ML solution, which is to declare nodes to be of type "either leaf or branch" — except that Python would still allow leaves of arbitrary mixed type.

A1.2 None, Maybe, Enumeration types

It's often handy for functions to be able to return either a value, or a marker that there is no value. For example, `head(list)` should return a value unless the list is empty in which case there's nothing to return. A common pattern in statically typed languages is to have a datatype that explicitly supports this, for example we'd define `head` to return an enumeration datatype with a constructor function, `None | Some['a]`. This forces everyone who uses `head` to check whether or not the answer is `None`.

In Python, the return type of a function isn't constrained. It's a common convention to return `None` if you have nothing to return, and a value otherwise, and to trust that the person who called you will do the appropriate checks.

Enumeration types are also used for general type restriction, e.g. to limit what can be placed in a list. When we actually do want to achieve this, Python isn't much help. It does have an add-on library for enumeration types, but it's a lot of work for little benefit.

One situation where enumeration types are very useful is when working with categorical values in data Section 3. When working with data, the levels of the enumeration are decided at runtime (by the contents of the data we load in), so a compile-time type annotation is no use; but Python's dynamic typing is also no help in preventing errors.

A1.3 Functions as values

In Python, as in ML, functions are values. A common pattern is to define functions that return parameterized functions, e.g.

```
def percentile(p):
    def f(xs):
        return xs[int(p*len(xs))]
    return f
```

I can define `median = percentile(.5)` and `lower_quartile = percentile(.25)`, and then apply `median` and `lower_quartile` to lists. The thing returned by `percentile` is a function `f` which also remembers the value `p`; this 'function plus variables it needs' is called a *closure*. In Python, the outer function is called a `decorator`.

In Python web servers, it's common to define decorators for purposes like user authentication, cross-site scripting checks, etc.

A1.4 Lists, lazy lists and generators

Python has lists and lazy lists, which it calls *generators*. The simplest way to define a generator is like a list comprehension, but using `($\cdot$)` rather than `[$\cdot$]`. Consider the following code:

```
def f(x):
    if x == 5:
        raise Exception("Bad value!")
    else:
        return x + 1

xs = range(10)
ys = (f(x) for x in xs)   # doesn't raise exception
zs = [f(x) for x in xs]   # raises exception
```

This code raises an exception on the last line. The preceding line doesn't, because `ys` is a lazy list, not evaluated until you need it. We could for example pull out the first three values of `ys` by

```
[next(ys) for _ in range(3)]
```

and no exception would be raised because we haven't hit the bad value. (Here, `_` is a variable name, like any other. It's a convention to use `_` for loop variables that you don't use for anything.)

Python also has a simple syntax for defining infinite lazy lists, using the `yield`. For example,

```
def fib():
    a,b = 1,1
    while True:
       yield a
       a,b = b, a+b

f = fib()
[next(f) for _ in range(20)]  # produces the first 20 Fibonnaci numbers
```

This is like a closure, but richer: the `f` object remembers not only the variables `a` and `b` in its scope, but also the point where it should resume when next `next(f)` is called.

A1.5 Object-oriented programming

Python is an object-oriented programming language. Every value is an object. You can see the class of an object by calling `x.__class__`. For example,

```
x = 10
x.__class__  # reports int
dir(x)       # gives a list of x's methods and attributes
```

It supports inheritance and multiple inheritance, and static methods, and class variables, and so on. It doesn't support interfaces, because they don't make sense in a duck typing language.

Here's a quick look at a Python object, and at how it might be used for the `flatten` function earlier.

```
class Branch(object):
    def __init__(self, children):
        self.children = children

def flatten(n):
    if isinstance(n, Branch):
        return [y for child in n.children for y in flatten(child)]
    else:
        return [n]

x = Branch([10,Branch([3,2]),"hello"])
flatten(x)

y = Branch([])
y.my_label = "added an attribute"
```

Every method takes as its first argument a variable referring to the current object, `this` in Java.

The last two lines are surprising. You can "monkey patch" an object, after it has been created, to change its attributes or give it new attributes. Like so many language features in Python, this is sometimes tremendously handy, and sometimes the source of infuriating bugs.

Python doesn't support private and protected access modifiers, except by convention. The convention is that attributes and functions whose name beings with an underscore are considered private, and may be changed in future versions of the library.

A2. Vectorized thinking

Goal of this notebook: Understand some more of the systems context behind vectorized thinking.

Vectorized thinking is great for conciseness. Surely no one would prefer

```
assert len(xs) == len(ys)
[xs[i] + ys[i] for i in range(len(xs))]
```

when they can just write

```
xs + ys
```

But vectorized coding is perhaps more important from the point of view of performance. Suppose `xs` and `ys` are both very large vectors. With list comprehension, it's almost impossible for the compiler to figure out that the operation can be parallelized, and each core of a multicore machine could be working on a chunk of the data.

Here's another example. Suppose we have a vector `xs` where each entry is the contents of a web page, and the entire vector is 100 TB and it's sharded across machines, and we want to compute

```
m = -inf
for x in xs:
    m = max(m, f(x))
```

where `f` is some scoring function that is slow to evaluate. Again, it's hard for a compiler to see how to achieve parallelism when the function is written out as an explicit iteration. (Maybe not hard in this case, but hard in the general case!) But it's easy to see how the computation *could* be distributed: tell each machine in the cluster to work on the fragment of data it can reach and compute the maximum over its fragment, then assemble the maximums from each machine. If we write the computation abstractly, e.g.

```
reduce(pairwise_max, -inf, map(f, xs))
```

then the compiler can figure out how to distribute it.

Vectorized thinking means avoiding `for` loops and instead writing our computations in a way that shows our intention more clearly, to give the compiler a chance to figure out what can be distributed and parallelized.

A3. Data import and cleanup

In my experience, around 75% of the time you spend working with data will be fighting to import it and clean it up. For the most part this is just general-purpose programming, but there are a few library routines that will save you from reinventing the wheel.

Treat this section as a collection of recipes and pointers to useful library routines. If you find yourself needing them, you should read the recipe, try it out, then look online for more information about the library functions it suggests.

A3.0 Preamble

```
# Import modules, and give them short aliases so we can write e.g. np.foo rather than numpy.foo
import math, random
import numpy as np
import matplotlib.pyplot as plt
import pandas
# The next line is a piece of magic, to let plots appear in our Jupyter notebooks
%matplotlib inline
```

A3.1 Reading from a csv file

When your data is a very simple comma-separated value (CSV) file then it's very easy to import. A CSV file looks like this: a header line, then one line per row of the data frame, values separated by commas.

```
"Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species"
5.1,3.5,1.4,0.2,"setosa"
4.9,3,1.4,0.2,"setosa"
4.7,3.2,1.3,0.2,"setosa"
4.6,3.1,1.5,0.2,"setosa"
5,3.6,1.4,0.2,"setosa"
```

Here is code to import a simple CSV file.

```
with open('res/iris.csv') as f:
    df = pandas.read_csv(f)
```

The `with` syntax is useful for files and similar objects which need to be closed/released after we've finished with them, to avoid memory leaks. It's equivalent to calling `f=open('data/iris.csv')`, then running the other commands, then calling `f.close()`.

If your file is nearly a CSV but has some quirks such as comments or a missing header row, experiment with the options in `pandas.read_csv` or `pandas.read_table`. For extreme quirks you may need to use the raw Python `csv.reader`.

A3.2 Reading from a URL or other file-like object

You can use the same `pandas.read_csv` function to read any file-like thing, such as a files retrieved over the web.

```
import urllib.request  # standard library for web requests
my_url = "https://teachingfiles.blob.core.windows.net/scicomp/iris.csv"
with urllib.request.urlopen(my_url) as f:
    df = pandas.read_csv(f)
```

You can also read from a string as if it were a file using `io.StringIO`. This is sometimes handy for debugging.

```
import io
f = io.StringIO('''
x,y
10,3
2,5
''')
df = pandas.read_csv(f)
```

A3.3 Parsing a text log file

A typical line from a web server log might look like this

```
207.46.13.169 - - [27/Aug/2017:06:52:11 +0000] "GET /marcus/essay/st&h2.html HTTP/1.1" 200 3881 "-" "Mozilla/5
```

where (according to the Apache web server documentation) the pieces are

- **207.46.13.169** The IP address that made the request
- **-** The identity of the client; - means not available
- **-** The userid of the logged-in user who made the request; - means not available
- **[27/Aug/2017:06:52:11 +0000]** The time the request was received
- **"GET /marcus/essay/st&h2.html HTTP/1.1"** The type of request, what was requested, and the connection type
- **200** The http status code (200 means OK)
- **3881** The size of the object returned to the client, in bytes
- **-** The referrer URL; - means not available
- **"Mozilla/5.0 (...)"** The browser type. The substring `bingbot` here tells us that the request comes from Microsoft Bing's web crawler.

To extract these pieces from a single line of the log file, the best tool is regular expressions, a mini-language for string matching that is common across many programming languages. The syntax is terse and takes a lot of practice. I like to start with a small string pattern and incrementally build it up, testing as I go.

```
import re    # standard Python module for regular expressions
s = """
207.46.13.169 - - [27/Aug/2017:06:52:11 +0000] "GET /marcus/essay/st&h2.html HTTP/1.1"
200 3881 "-" "Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X)
AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53
(compatible; bingbot/2.0; +http://www.bing.com/bingbot.htm)"
"""


# First attempt: match the first two items in the log line.
# If my pattern is right, re.match returns an object.
# If my pattern is wrong, re.match returns None.
pattern_test = r'\s*(\S+)\s*(\S+)'
re.match(pattern_test, s)
```

```
<_sre.SRE_Match object; span=(0, 16), match='\n207.46.13.169 -'>
```

```
# This is the full pattern I built up to. Python lets us add verbose comments
# to the pattern, which is handy for remembering what your code does when you look
# at it the next morning.
pattern = r'''(?x)  #   flag saying that this pattern has comments
\s*                 #   any whitespace at the start of the string
(?P<ip>\S+)         # one or more non-space characters: the IP address
\s+                 #   one or more spaces
(?P<client>\S+)     # the client identity
\s+
(?P<user>\S+)       # the userid
\s+
\[(?P<t>[^\]]*)\]   # [, then any number of not-] characters, then ]: the timestamp
\s+
"(?P<req>[^"]*)"    # ", then any number of not-" characters, then ": the request string
\s+
(?P<status>\d+)     # one or more numerical digits: the http status code
\s+
(?P<size>\d+)       # one or more numerical digits: the size
\s+
"(?P<ref>[^"]*)"    # the referrer URL
```

```
\s+
"(?P<ua>[^"]*)"     # the user agent i.e. browser type
'''
m = re.match(pattern, s)
m.groupdict()       # returns a dictionary of all the named sub-patterns
```

```
{'client': '-',
 'ip': '207.46.13.169',
 'ref': '-',
 'req': 'GET /marcus/essay/st&h2.html HTTP/1.1',
 'size': '3881',
 'status': '200',
 't': '27/Aug/2017:06:52:11 +0000',
 'ua': 'Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Mac OS X)\nAppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safa
 'user': '-'}
```

Exercise. Why not use s.split(' ')?

How do we extract the fields from a full log file? The standard Python code is

```
with open(myfile) as f:
    for line in f:
        m = re.match(pattern, line)
        # store the fields from m.groups() or m.groupdict() somewhere appropriate
```

Alternatively, `numpy` has a handy shortcut for reading in an entire file and splitting it via a regular expression:

```
# Split the file into an array, one row per line, one column per field in the pattern
df = np.fromregex('res/webaccess_short.log', pattern, dtype=np.unicode_)
# Make a dictionary out of the columns, according to the named fields in the pattern
df = pandas.DataFrame({v: df[:,i-1] for v,i in re.compile(pattern).groupindex.items()})

df[:4]
```

	client	ip	ref	\
0	-	207.46.13.169	-	
1	-	31.218.127.114	https://www.google.ae/	
2	-	31.218.127.114	http://www.wischik.com/damon/recipe/oventemper…	
3	-	31.218.127.114	http://www.wischik.com/damon/recipe/oventemper…	

	req	size	status	\
0	GET /marcus/essay/st&h2.html HTTP/1.1	3881	200	
1	GET /damon/recipe/oventemperatures.html HTTP/1.1	879	200	
2	GET /damon/recipe/recipe.css HTTP/1.1	1118	200	
3	GET /favicon.ico HTTP/1.1	503	404	

	t	\
0	27/Aug/2017:06:52:11 +0000	
1	27/Aug/2017:06:52:43 +0000	
2	27/Aug/2017:06:52:43 +0000	
3	27/Aug/2017:06:52:44 +0000	

	ua	user
0	Mozilla/5.0 (iPhone; CPU iPhone OS 7_0 like Ma…	-
1	Mozilla/5.0 (Linux; Android 4.3; 5050X Build/J…	-
2	Mozilla/5.0 (Linux; Android 4.3; 5050X Build/J…	-
3	Mozilla/5.0 (Linux; Android 4.3; 5050X Build/J…	-

A3.4 Reading json from a web data service

More and more forward-thinking companies and government services make data available by simple web requests. Here is an example, importing river levels from the UK's real-time flood monitoring API.

The first step is to import the Python module for making web requests. When I'm developing data code I like to build it up in small steps, which means lots of repeated requests, so I also like to use another Python module which caches responses. This means I don't hammer the service unnecessarily.

```
import requests, requests_cache
requests_cache.install_cache('floodsystem', backend='memory')
```

The flood monitoring API documentation tells us the URL for fetching a list of all stations, `{root}/id/stations`. Let's try it.

```
# Make the request. This should print out <Response [200]>, meaning successfully retrieved
stations_resp = requests.get('http://environment.data.gov.uk/flood-monitoring/id/stations')
stations_resp
```

```
<Response [200]>
```

Now let's look at the body of the response. It's likely to be very long, so we'll only print out the first 300 characters.

```
stations_resp.text[:300]
```

```
'{ \n  "@context" : "http://environment.data.gov.uk/flood-monitoring/meta/context.jsonld" ,\n  "meta" : { \n    "publisher" : "Enviro
```

It looks like JSON, "JavaScript Object Notation", a common format for web data services. It's easy to turn it it into Python dictionaries and lists, and to explore what it contains. (Alternatively, just read the web service documentation, if you trust it!)

```
x = stations_resp.json()
print(x.__class__)                      # The top-level response is a dictionary
print(x.keys())                         # What are its keys? 'items' sounds promising.
print(x['items'].__class__)             # 'items' is a list,
print(len(x['items']))                  # with 4363 elements.

stations = stations_resp.json()['items']  # This is what we're really after
stations[0]                             # Look at a single sensor
```

```
<class 'dict'>
dict_keys(['@context', 'meta', 'items'])
<class 'list'>
4346
```

```
{'@id': 'http://environment.data.gov.uk/flood-monitoring/id/stations/1029TH',
 'RLOIid': '7041',
 'catchmentName': 'Cotswolds',
 'dateOpened': '1994-01-01',
 'easting': 417990,
 'label': 'Bourton Dickler',
 'lat': 51.874767,
 'long': -1.740083,
 'measures': [{'@id': 'http://environment.data.gov.uk/flood-monitoring/id/measures/1029TH-level-downstage-i-15_min-mASD',
   'parameter': 'level',
   'parameterName': 'Water Level',
   'period': 900,
   'qualifier': 'Downstream Stage',
   'unitName': 'mASD'},
  {'@id': 'http://environment.data.gov.uk/flood-monitoring/id/measures/1029TH-level-stage-i-15_min-mASD',
   'parameter': 'level',
   'parameterName': 'Water Level',
   'period': 900,
   'qualifier': 'Stage',
   'unitName': 'mASD'}],
 'northing': 219610,
 'notation': '1029TH',
 'riverName': 'Dikler',
 'stageScale': 'http://environment.data.gov.uk/flood-monitoring/id/stations/1029TH/stageScale',
 'stationReference': '1029TH',
 'status': 'http://environment.data.gov.uk/flood-monitoring/def/core/statusActive',
 'town': 'Little Rissington',
 'wiskiID': '1029TH'}
```

Data from web services is often patchy and inconsistent, so your code for processing it should be full of error handling. For example,

```
rivers = [s['riverName'] for s in stations]
```

will fail because `riverName` isn't present for all stations. You might use one of these instead:

```
rivers = [s.get('riverName', None) for s in stations]
rivers = [s['riverName'] for s in stations if 'riverName' in s]
```

Here's how we might turn the data into a data frame:

```
df = pandas.DataFrame({
    'label': np.array([s['label'] for s in stations]),
    'riverName': np.array([s.get('riverName',None) for s in stations]),
    'town': np.array([s.get('town',None) for s in stations])
})
```

```
# Each station has zero or more measures. Pick out the first 'water level' measure at each.
# The field measure['@id'] is the url to use to get water level readings.
measures = [[m for m in s.get('measures',{}) if m['parameter']=='level'] for s in stations]
df['url'] = np.array([ms[0]['@id'] if len(ms)>0 else None for ms in measures])
```

A3.5 Scraping a website with xpath

There are fascinating stories to be discovered from public data, and sometimes you have to work to scrape it from web pages. Here's an acount by a BBC data journalist. We'll work with a very simple example: extracting results of the Oxford / Cambridge boat race from the Wikipedia table.

I recommend using XPath queries from the `lxml` module. XPath is a powerful mini-language for extracting data from hierarchical documents, with wide support across many programming languages — think of it as regular expressions but for html rather than plain text. If you want to scrape websites then it's worth finding a tutorial and learning XPath. For this course, we'll just see how to use XPath in Python.

The first step is to install `lxml`, which is not included with Python.

```
!pip install lxml
```

Now we'll fetch the web page and parse the contents. Most web pages are badly-formatted html (sections not properly closed, etc.), and `lxml.html.fromstring` makes a reasonable attempt to make sense of it.

```
import requests
import lxml.html
boatrace_url = 'https://en.wikipedia.org/wiki/List_of_The_Boat_Race_results'
resp = requests.get(boatrace_url)
doc = lxml.html.fromstring(resp.content)
```

This gives us `doc`, the root `<html>` element, which we can inspect.

```
print(doc.tag)               # the type of element
print(len(doc))              # the number of children
print([n.tag for n in doc])  # tags of its children, <head> and <body>
print(doc.attrib)            # get the attributes, e.g. <html class="client-nojs" lang="en" dir="ltr">
print(doc.text, doc.tail)    # any text directly under in this element
```

```
html
2
['head', 'body']
{'class': 'client-nojs', 'lang': 'en', 'dir': 'ltr'}

 None
```

We want to pull out a particular element from the document, namely the table with boat race results, so we need to work out how to refer to it in XPath. The Chrome webbrowser has a handy tool to help with this. Go to the page you're interested in, and click on View | Developer | Developer Tools. Click on the element-selector button at the top left: Go back to the web page, and click on a piece close to what you want to select. I clicked on the top left cell of the table: Go back to the developer tools window, and navigate to the exact element you want. Here, we want the table. Right-click and choose Copy | Copy XPath. It gave me the XPath location `"//*[@id="mw-content-text"]/div/table[2]"`. Now we can extract the data.

```
# Pick out the table element.
# (XPath queries return lists, but I only want one item, hence the [0].)
table = doc.xpath('//*[@id="mw-content-text"]/div/table[2]')[0]

# Get a list of all rows i.e. <tr> elements inside the table.
# Print one, to check things look OK.
rows = table.xpath('.//tr')
print(lxml.etree.tostring(rows[1], encoding='unicode'))

# Extract the timestamp and winner columns.
# The timestamp is in the second child, in a <span> element with class "sortkey".
# The winner is in the third child.
df = {'t': [row[1].xpath('.//span[contains(@class, "sortkey")]')[0].text for row in rows[1:]],
      'winner': [row[2].text for row in rows[1:]]}
df = pandas.DataFrame(df)
df[:5]
```

```
<tr>
<td align="center"><a href="/wiki/The_Boat_Race_1829" title="The Boat Race 1829">1</a></td>
<td align="center"><span class="sortkey" style="display:none;speak:none">000000001829-06-10-0000</span><span
style="white-space:nowrap">10 June 1829</span> <img alt="dagger"
src="//upload.wikimedia.org/wikipedia/commons/3/37/Dagger-14-plain.png" width="9" height="14" data-file-
width="9" data-file-height="14" /><br />
<span style="color:red;font-size:85%">1830 1835 no race</span></td>
<td style="background:#004685; color:#FFFFFF" align="center">Oxford</td>
<td align="center">14:03</td>
<td align="center"><span style="display:none">999</span>Easily</td>
<td align="center">1</td>
<td align="center">0</td>
</tr>
```

```
                         t     winner
0  000000001829-06-10-0000     Oxford
1  000000001836-06-17-0000  Cambridge
2  000000001839-04-03-0000  Cambridge
3  000000001840-04-15-0000  Cambridge
4  000000001841-04-14-0000  Cambridge
```

You should consider the ethics of your web scraping. Here are some thoughts: from Sophie Chou at the MIT Media Lab, and the data journalist Näel Shiab.

A3.6 Querying an SQL database

Once your data is in an SQL database, access is easy. Here's an example with Postgresql, a dialect of SQL. I don't like to include secret credentials in an Azure notebook. Instead, I store the credentials in a file, and I make sure the file isn't cloned in a public repository. The file looks something like

```
{"user": "***", "password": "***", "host": "***", "dbname": "***"}
```

so that Python can load it in as a simple dictionary. The syntax `connect(**creds)` passes each item in the dictionary as a named argument, i.e. `connect(user="***", ...)`.

```
import psycopg2    # module for connecting to a Postgresql database
import json        # standard module for reading json files

creds = json.load(open('res/secret_creds.json'))
conn = psycopg2.connect(**creds)
```

You can run arbitrary SQL queries. Pass in parameters with the `%(name)s` quoting mechanism, to keep yourself safe from SQL injection attacks.

```
cmd = '''
SELECT *
FROM flood_stations AS s JOIN flood_measures AS m ON (m.station_uri = s.uri)
WHERE river = %(river)s OR town = %(town)s
LIMIT 3
'''
pandas.read_sql(cmd, conn, params={'river': 'River Cam', 'town': 'Cambridge'})
```

```
   index                                          uri  \
0    345  http://environment.data.gov.uk/flood-monitorin…
1    800  http://environment.data.gov.uk/flood-monitorin…
2   1272  http://environment.data.gov.uk/flood-monitorin…

                label      id                                catchment  \
0     Great Chesterford  E21778  Cam and Ely Ouse (Including South Level)
1      Weston Bampfylde   52113   Parrett, Brue and West Somerset Streams
2  Cambridge Baits Bite  E60101  Cam and Ely Ouse (Including South Level)

       river              town        lat       lng  index  measure_id  \
0  River Cam  Great Chesterford  52.061730  0.194279    397         398
1  River Cam   Weston Bampfylde  51.023159 -2.565568    918         919
2  River Cam            Milton  52.236542  0.176925   1454        1455

                                          uri  \
0  http://environment.data.gov.uk/flood-monitorin…
1  http://environment.data.gov.uk/flood-monitorin…
2  http://environment.data.gov.uk/flood-monitorin…
```

```
                                     station_uri qualifier    parameter  \
0  http://environment.data.gov.uk/flood-monitorin…     Stage  Water Level
1  http://environment.data.gov.uk/flood-monitorin…     Stage  Water Level
2  http://environment.data.gov.uk/flood-monitorin…     Stage  Water Level

   period  unit      valuetype    low   high
0   900.0     m  instantaneous  0.109  0.333
1   900.0     m  instantaneous  0.026  0.600
2   900.0  mASD  instantaneous  0.218  0.294
```

Assignment 2. Econophysics simulator

Economic inequality is one of the defining social issues of our age. Yet we have a poor grasp of the scale of inequality, as described in Scientific American and nicely shown in this video:

How does inequality arise? Is it an inevitable outcome of liberal economics, and if so how can it be mitigated by economic policy? These questions have been studied by economists and more recently by physicists. In this assignment you will investigate a simple "econophysics" model of inequality. The notes for §2 suggest ways to take the investigation further.

About this assignment: This assignment tests your vectorized thinking. You will be asked to run simulations on a population of hundreds of thousands of individuals, over many timesteps. Your code *must* use NumPy vectorized operations rather than iterating over the population. You may use Python iteration over timesteps. You can organize your code however you like. Please create a new notebook for your answers to this assignment.

Part A: kinetic exchange model

This section is worth 1 mark. Check your answers as described in §0.3 using `section='assignment2a'`.

Here is a simple model. There are N individuals in the population, each with an initial wealth of £1. Every timestep, we randomly group them into $N/2$ pairs. (Assume N is even.) For every pair, we simulate an economic exchange, as follows. Let the two paired individuals have wealth v and w, and update their wealth according to

$$v_{\text{new}} = R(v+w), \quad w_{\text{new}} = (1-R)(v+w)$$

where R is a random number in $[0,1]$, chosen independently for every pair and at every timestep. This model is loosely inspired by the physics of gases, in which two gas molecules exchange a random amount of energy whenever they collide.

We can measure inequality with the Gini coefficient,

$$G = 2\frac{\sum_{i=1}^{N} i\, w_{(i)}}{N\sum_i w_{(i)}} - \left(1+\frac{1}{N}\right)$$

where $w_{(1)}$ is the smallest value, $w_{(2)}$ the second smallest etc. If everyone has the same wealth then $G=0$; if one person has all the wealth then $G = 1-1/N$.

Question 1. The model needs us to randomly group the population into $N/2$ pairs. We can do this by randomly permuting the vector $[0,\ldots,N-1]$, letting the vector `m1` consist of the first $N/2$ integers and `m2` consist of the rest, and interpreting it as "`m1[i]` is paired with `m2[i]`".

Write a function `pairs(N)` that returns a tuple `(m1,m2)` where `m1` and `m2` are both vectors of length $N/2$ as described above. For example, if you run `pairs(6)`, you might get the output

```
(array[(3, 0, 1]), array([2, 4, 5]))
```

```
# Submitting your answer:
q = GRADER.fetch_question('q1')
m1,m2 = pairs(q.n)
ans = {'n': np.unique(np.concatenate([m1,m2])), 's': np.std(np.abs(m1-m2))}
GRADER.submit_answer(q, ans)
```

Question 2. Write a function `kinetic_exchange(v,w)` which takes two wealth vectors `v` and `w`, each of length $N/2$, and returns a tuple `(vnew, wnew)` with two new vectors, according to the kinetic exchange model.

```
# Submitting your answer:
q = GRADER.fetch_question('q2')
v,w = np.linspace(1,5,q.n), np.linspace(1,2,q.n)**q.p
vnew,wnew = kinetic_exchange(v,w)
ans = {'m1': np.mean(vnew), 's2': np.std(wnew)}
GRADER.submit_answer(q, ans)
```

Question 3. Write a function `gini(w)` which takes a vector `w` and returns the Gini coefficient.

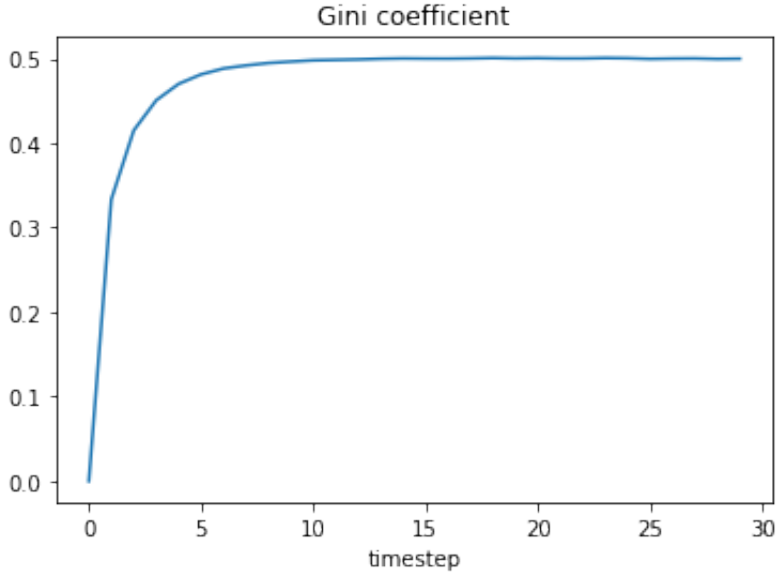
```
# Submitting your answer:
q = GRADER.fetch_question('q3')
```

```
w = np.linspace(0,1,q.n)**q.p
g = gini(w)
GRADER.submit_answer(q, {'g': g})
```

Question 4. Write a function `sim(N, T)` which runs the kinetic exchange model on a population of N individuals for T timesteps. It should return a pair `(w, gs)` where `w` is the wealth vector after T timesteps, and `gs` is a length T vector where `gs[i]` is the Gini coefficient at timestep i.

```
# Submitting your answer:
q = GRADER.fetch_question('q4')
w,gs = sim(q.n, q.t)
ans = {'gm': np.mean(gs[(q.t/2):]), 'gs': np.std(gs[(q.t/2):]), 'ws': np.std(w)}
GRADER.submit_answer(q, ans)
```

Question 5. Simulate a population of 500,000 over 30 iterations. Plot the Gini coefficient as a function of timestep. (You don't have to submit your plot, but it may be assessed in the ticking session.) Your plot should look something like this:



Part B: economic mobility

This section is worth 1 mark. Check your answers as described in §0.3 using `section='assignment2b'`.

Some degree of inequality might be acceptable if economic mobility were high, i.e. if everyone had similar chances of reaching either end of the wealth distribution. Economic mobility is often measured by splitting the population into five equal brackets, and measuring the chance of moving between brackets. From the Wikipedia article on economic mobility:

> in terms of relative mobility a report stated: "contrary to American beliefs about equality of opportunity, a child's economic position is heavily influenced by that of his or her parents." 42% of children born to parents in the bottom fifth of the income distribution ("quintile") remain in the bottom, while 39% born to parents in the top fifth remain at the top.

Let's measure economic mobility by recording the wealth distribution at one timepoint, and again some number of timesteps later, splitting the two distributions into quintiles, and counting what fraction of the population moved by more than one quintile from beginning to end. (In each timestep a median individual might find their wealth increasing or decreasing by around 50%, so one timestep corresponds roughly to several years of human life.) For example, if we have a population of 5000 and we draw up a matrix A where A_{ij} is the number of people who start in quintile i and end up in quintile j, we might get

$$A = \begin{pmatrix} 344 & 313 & 243 & 100 & 0 \\ 266 & 261 & 302 & 167 & 4 \\ 212 & 260 & 225 & 272 & 31 \\ 147 & 143 & 183 & 331 & 196 \\ 31 & 23 & 47 & 130 & 769 \end{pmatrix}$$

(A quick check: the row sums and column sums are all 1000.) The number who moved by more than one quintile is 1148, which is 23% of the population.

Question 6. In a perfectly mobile economy, where everyone has equal chance of reaching any quintile, what fraction of people are expected to move by more than one quintile?

```
q = GRADER.fetch_question('q6')
GRADER.submit_answer(q, your_answer)
```

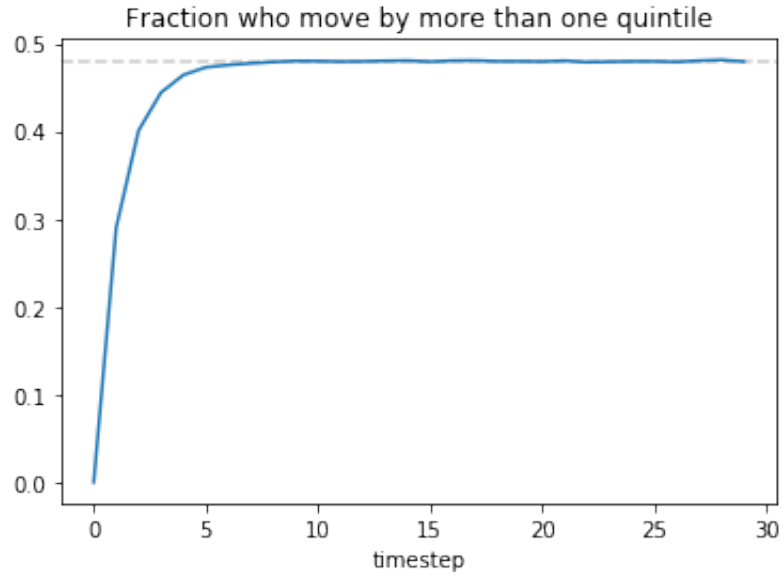
Question 7. Write a function `mobility(v,w)` that returns the proportion of people who moved by more than one quintile, where `v[i]` and `w[i]` measure respectively the wealth of individual i at the beginning and end of a time period. Hint: look up `np.percentile` and `np.digitize`.

```
# Submitting your answer:
q = GRADER.fetch_question('q7')
v,w = np.arange(q.n)**q.a, np.arange(q.n)**q.a * np.random.random(q.n)
GRADER.submit_answer(q, mobility(v,w))
```

Question 8. Simulate the kinetic exchange model long enough for it to stabilize, and measure the wealth vector `w0`. Run it t timesteps further, and find the wealth vector `v`, and compute `mobility(w0,v)`. It's up to you to decide how to judge stabilization; you don't have to explain your method but you do have to submit a correct answer.

```
# Submitting your answer:
q = GRADER.fetch_question('q8')
# For a population size q.n, measure mobility over q.t timesteps
GRADER.submit_answer(q, your_answer)
```

Question 9. Compute the social mobility for a population of 500,000 as in Question 8, over a sequence of timesteps. Plot a graph of economic mobility as a function of time. (You don't have to submit your plot, but it may be assessed in the ticking session.) Your plot should look something like this:



Assignment 3. Analysis of flood data

Jesus Green lock

This assignment asks you to analyse data provided by the UK Environment Agency concerning flooding. The agency offers an API for near real-time data covering: * flood warnings and flood alerts * flood areas which to which warnings or alerts apply * measurements of water levels and flows * information on the monitoring stations providing those measurements

In this assignment we will be working with historical data of water level measurements, at several monitoring stations in Cambridge and on the Cam. The dataset is available as a CSV file at `https://teachingfiles.blob.core.windows.net/scicomp/flood.csv`. If you go home over Christmas and are worried about flooding, and want to extend these analyses to your home area, see A3. Data import and cleanup for details of how to fetch data from a web API.

Image by N. Chadwick.

Goal of the assignment. This assignment tests your skill at manipulating dataframes and indexed arrays, and your flair at plotting data. You should use Pandas and numpy operations for data manipulation, rather than for loops, wherever possible. You can organize your code however you like. Please create a new notebook for your answers to this assignment.

Part A

This section is worth 1 mark. Check your answers as described in §0.3 using `section='assignment3a'`.

Question 1. Import the CSV file and print out a few lines, choosing the lines at random using `np.random.choice`. The file mistakenly includes the village Cam, near Bristol. Remove these rows, and store what's as the variable `flood`. How many rows are left?

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q1'), num_rows)
```

Question 2. Complete this table of the number of entries in this dataset for each town and river.

Bin Brook	
River Cam	

```
# Submit your answer, as an unstacked indexed array:
GRADER.submit_answer(GRADER.fetch_question('q2'), your_answer.as_matrix())
```

Question 3. Each measurement station has a unique `measure_id` and `label`. Complete this table of the number of measurement stations for each town and river. Use only the Pandas operations for split-apply-combine, don't use any numpy operations or Python `for` loops or list comprehensions.

Bin Brook	
River Cam	

```
# Submit your answer. Let your_answer be an unstacked indexed array.
GRADER.submit_answer(GRADER.fetch_question('q3'), your_answer.as_matrix())
```

Question 4. Each measurement station has low and high reference levels, in columns `low` and `high`. In this dataset, the reference levels are stored for every measurement, but we can verify that every `measure_id` has a unique pair `(low,high)` with

```
assert all(flood.groupby(['measure_id','low','high']).apply(len).groupby('measure_id').apply(len) == 1), "Ref
```

Add a column `norm_value`, by rescaling `value` linearly so that `value=low` correponds to `norm_value=0` and `value=high` corresponds to `norm_value=1`. Use `np.nanpercentile` to find the *tercile points*, the two values that split the entire `norm_value` column into three roughly equal parts.

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q4'), [tercile1, tercile2])
```

Question 5. Complete the following dataframe, listing the number of observations in each tercile and the total number of observations per station.

label
Bin Brook
Bin Brook
Bin Brook
Cambridge

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q5'), your_dataframe)
```

Question 6. Complete this dataframe listing the fraction of observations in each tercile per station:

label
Bin Brook
Cambridge

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q6'), your_dataframe)
```

Question 7. Fill in the rest of this indexed array, giving the `low` and `high` values for each measurement station:

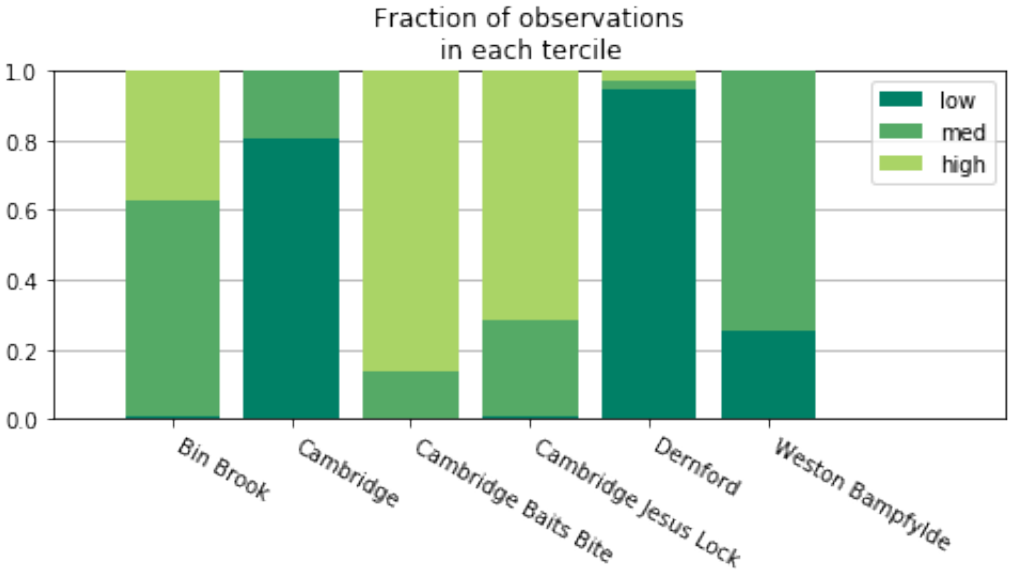
label
Bin Brook
Cambridge

```
# Submit your answer. Let your_answer be an indexed array.
GRADER.submit_answer(GRADER.fetch_question('q6'), your_answer.reset_index())
```

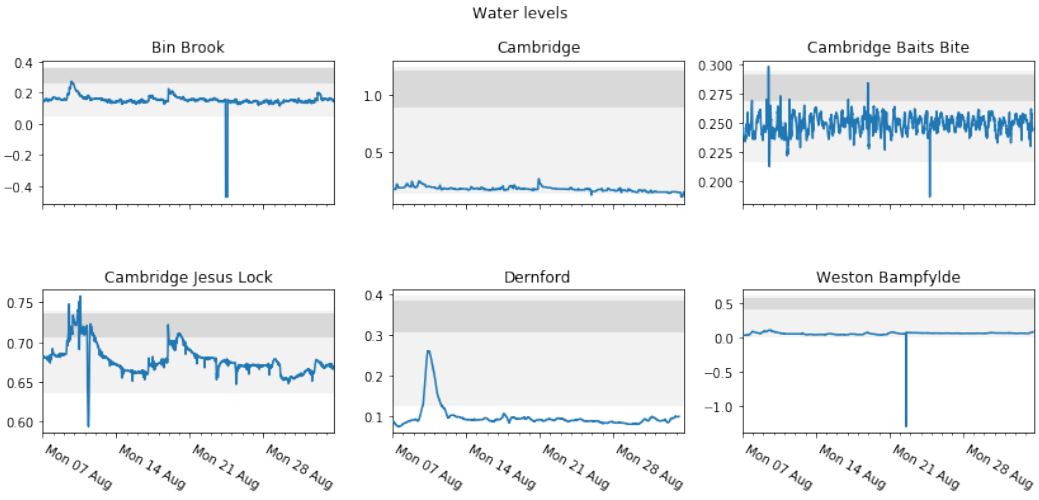
Part B

This section is worth 1 mark. There is no automated testing of your answers here, but your code may be assessed in the ticking session.

You should pay attention to axis ranges, axis labelling, colour schemes, etc. in reproducing the plot, though you shouldn't aim to be pixel-perfect. You will need to spend time searching how to control matplotlib plots.



Question 8. Reproduce this plot:



Question 9. Reproduce this plot:

The light shaded area shows the range from `low` to `high` for each station. The dark shaded area shows the inter-tercile range, `low+tercile1*(high-low)` to `low+tercile2*(high-low)` where `tercile1` and `tercile2` are your answers to Question 4. They can be plotted with `ax.axhspan`. Here are some code snippets for working with datetimes that may be helpful.

```
# Create a column with datetime objects
import datetime, pytz
def as_datetime(s): return datetime.datetime.strptime(s, '%Y-%m-%dT%H:%M:%SZ').replace(tzinfo=pytz.UTC)
flood['datetime'] = np.vectorize(as_datetime)(flood['t'])

# Date-axis control, taken from http://matplotlib.org/examples/api/date_demo.html
# Given a matplotlib axis, print out date labels nicely
ax.xaxis.set_major_locator(matplotlib.dates.WeekdayLocator(byweekday=matplotlib.dates.MO, tz=pytz.UTC))
ax.xaxis.set_minor_locator(matplotlib.dates.DayLocator(tz=pytz.UTC))
ax.xaxis.set_major_formatter(matplotlib.dates.DateFormatter('%a %d %b'))
# then, at the end,
fig.autofmt_xdate(bottom=0.2, rotation=-30, ha='left')
```