MATH40006: An Introduction To Computation
Course notes, Volume 1

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

---

1 Getting Started

1.1 What this course is about

This course aims to teach you to program computers. The language we'll be using is Python, but this isn't really a course in the ins and outs of that particular language, although it may sometimes feel like that. What we hope you get from the course is an understanding of key **computational principles**, which you'll be able to use in future Maths work. You might, in that work, be using Python, or you might be using another language, such as R or C^{++}; what we want you to get from this course is a sense of how to get computers to do what we want them to do!

1.2 The software and how to launch it

There are three pieces of software we'll mostly be concerned with on this course: Python, Jupyter Notebook and Anaconda Navigator.

Python is a **programming language**. It's the real core of what we'll be doing.

Jupyter Notebook is a **coding environment** for Python. Actually, these days it's an environment for a whole lot of other things too, but Python was where it started (the "py" in the name is a clue), and Python is what we'll be using it for. It enables you to run Python code, write Python programs and create documents based around Python.

Anaconda Navigator is a **user interface**: it's our way in to Jupyter Notebooks (amongst other things) and thus ultimately to Python.

If you click on the Windows icon at the bottom left-hand corner of the screen, you should see something like Figure 1. The next thing to do is then to click where it says **Anaconda (64 bit)** and select, from the drop-down menu, **Anaconda Navigator**. Clicking on that should launch a window looking something like Figure 2.

This will offer you a bunch of options, each representing a different application, and many representing alternative ways of using Python. We will, later in the course, explore at least one other of them (namely **Spyder**), but for now and for the next few months we'll be working entirely in **Jupyter Notebook**, which you launch by clicking where it says **Launch**. Make sure you *don't* choose **JupyterLab** by mistake!

Then you should end up with a **landing page** looking something like Figure 3.

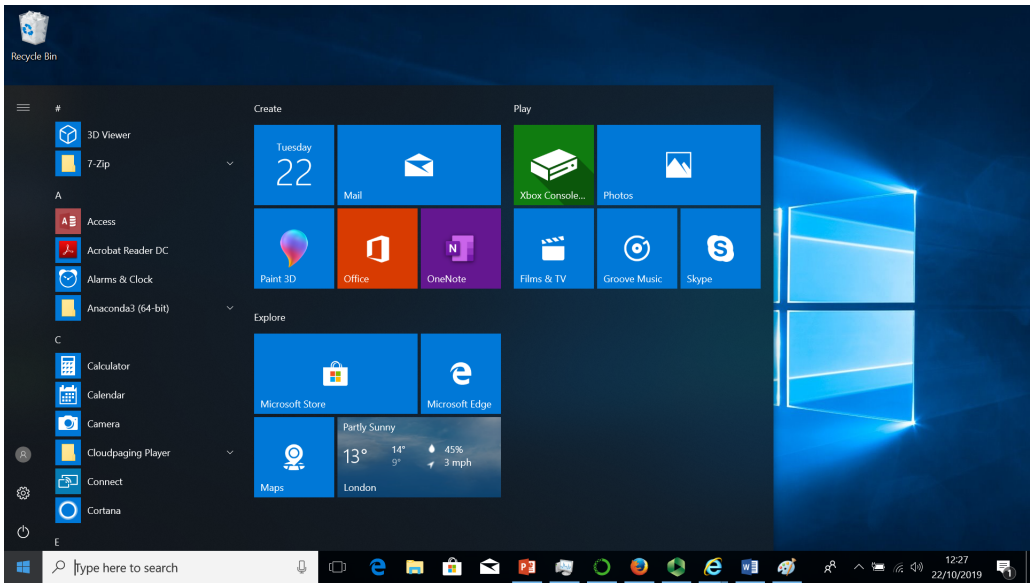


Figure 1: Screenshot: how to launch Anaconda Navigator

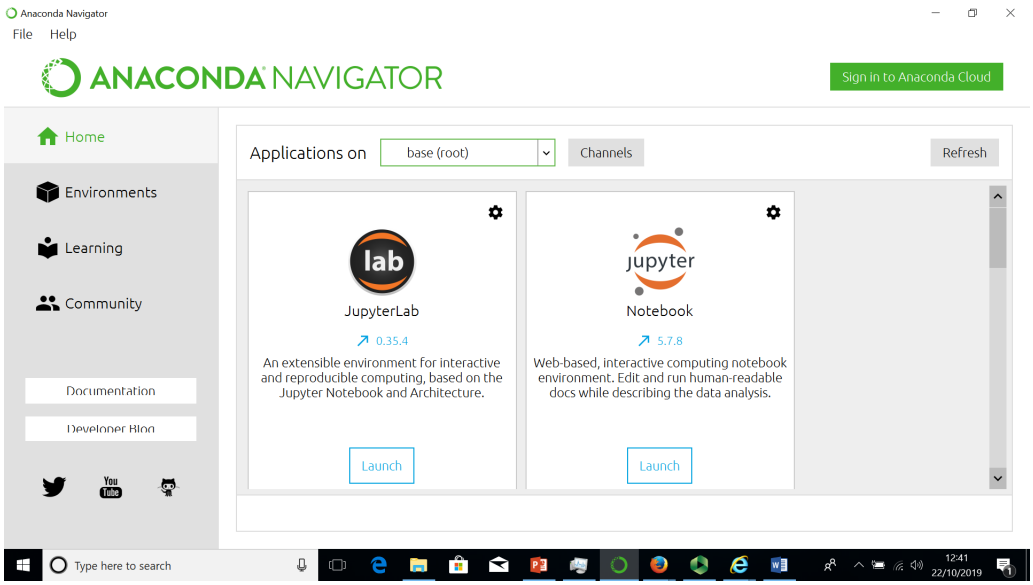


Figure 2: Screenshot: Anaconda Navigator

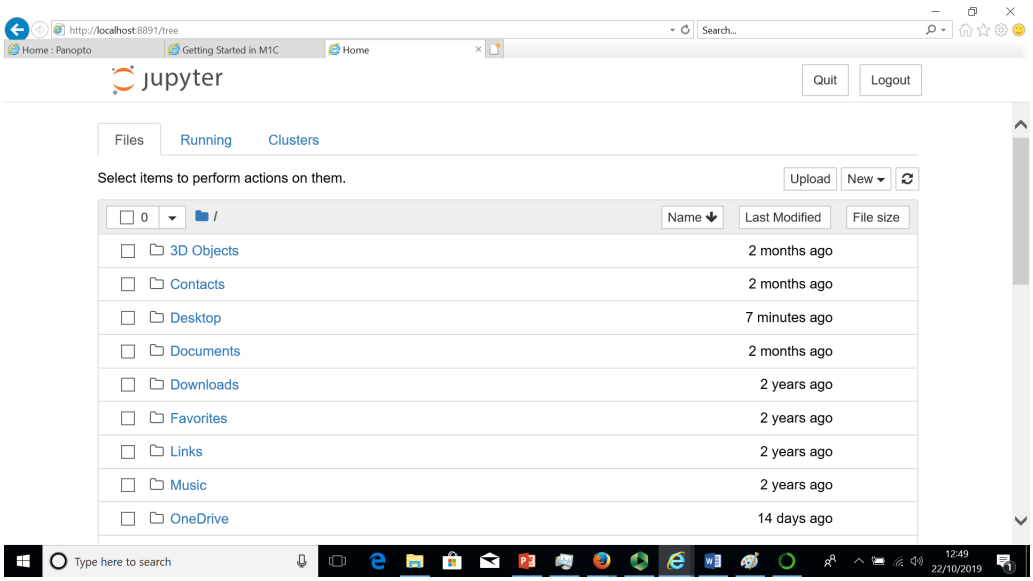


Figure 3: Screenshot: Jupyter landing page

One thing to notice straight away is that this is a *browser window*: Jupyter Notebook works entirely within web pages. The other thing to notice is the list of files and folders; make a note of where this is on the machine you're using.

The last stage is to click where it says **New**, near the top right-hand corner of the window, and select **Python 3**. This launches your first Jupyter notebook, which is also a browser window, and which should look something like Figure 4. Now you're ready to start coding!

1.3 Getting going in Python

The question "What is Python?" has rather a complicated answer, which in a sense we'll spend the whole course answering. But quite a good way to *start* thinking about it is that Python is a bit like a calculator.

Let's start by doing some simple arithmetic. In your new Jupyter notebook, in the box to the right of where it says `In [ ]`, type

```
2 + 6
```

Then (and this is important), **hold down the Shift key** and press the Return key. If you just press Return, all that'll happen is you'll get a new line. But if you "Shift-Return", you should get, with any luck, a piece of output, namely

```
8
```

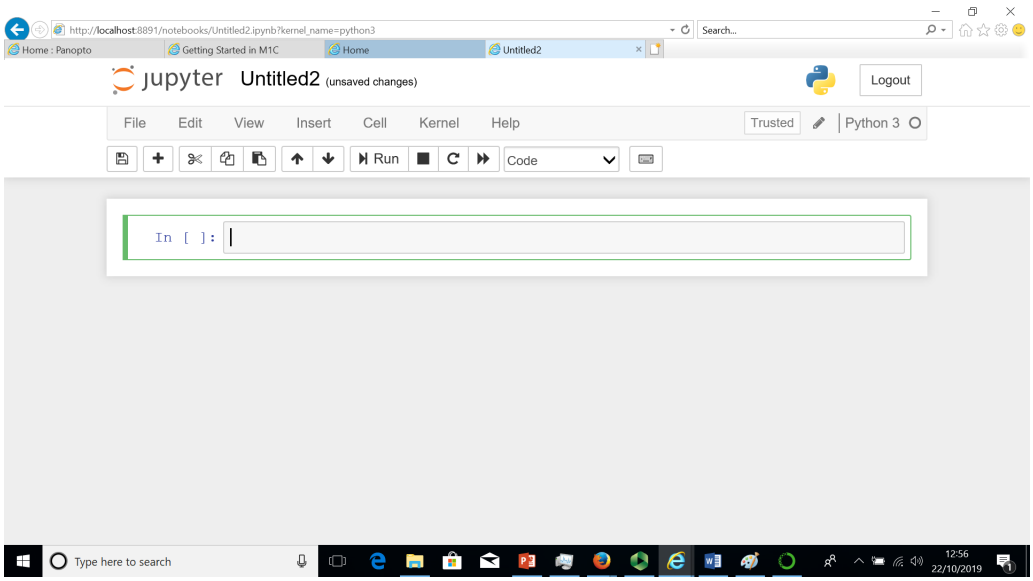


Figure 4: Screenshot: Jupyter notebook

You've just done your first piece of Python coding.

Jupyter notebooks are **editable**, rather like Word docs, spreadsheets, etc. If we now decide we didn't want to do an addition but a subtraction, we can go back and change the `2 + 6` into `2 - 6`, and then "Shift-Return" once more, which should give the output

```
-4
```

1.4 Arithmetic in Python

We've seen that the inputs `2 + 6` and `2 - 6` return the outputs 8 and `-4` respectively.

To multiply, we use the asterisk sign, `*` (as we do in most computing environments). If you type in

```
2 * 6
```

then Shift-Return, and don't get 12, let Dr Ramsden know **immediately**!

Division is slightly more complicated, and introduces us to our first real subtlety. If you type

```
6 / 2
```

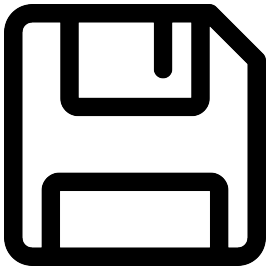


Figure 5: The save icon

you get `3.0`. Now, in Python, `3.0` is a different thing from `3`; more about that later in the course. If you want your output to be the integer `3`, you need to type

```
6 // 2
```

The `//` operator does **integer division**.

Question: what do you expect to see if you type `2 / 6`? What about `2 // 6`? What about `7 / 3` and `7 // 3`?

That just leaves us with **powers**. Now, in many computing languages, 2^6 would be rendered as `2 ^ 6`; but not in Python! If we do type that, we get `4`, which, whatever it *does* mean, is certainly not 2^6. For that we need to type

```
2 ** 6
```

which (sure enough) gives the output

64

1.5 Keeping your work

Jupyter notebooks autosave, but only every now and then. To save a fully up-to-date copy of your notebook, hit the save icon, as in Figure 5. But it'll save under the name **Untitled**, or something similar, which is not very helpful. To rename it, find the **File** menu (the one within the browser window) and choose **Rename**. Call your notebook something memorable like **Arithmetic**; it should then appear in your file listing on the landing page.

1.6 How to get it

Python is available in various **distributions**, each of which consists of software for developing, interpreting and executing code in the Python language. We'll be using Anaconda, which is available at

`https://www.anaconda.com/download/`

2 Introduction to Python

2.1 Modular arithmetic and the percent operator

Python's quite good at handling integers, and in particular, at **modular arithmetic**. For example, if we divide 344 by 3 we get a remainder of 2; that's because 342 is a multiple of 3, so 344 is 2 more than a multiple of 3.

In more hi-falutin' Maths terms, we say that the **residue** of 344, **reduced modulo** 3, is 2. In Python, that calculation looks like this:

```
344 % 3
```

2

The percent operator, %, means "reduced modulo".

We can do calculations like `(5 + 17) % 19`, `(5 * 17) % 19` and `(5 ** 17) % 19`; these mean, respectively, "add 5 and 17, then reduce modulo 19", "multiply 5 by 17, then reduce modulo 19" and "raise 5 to the power 17, then reduce modulo 19".

The last of these, though, can be done more efficiently using a built in Python function called `pow`:

```
pow(5, 17, 19)
```

4

If you do it the first way, Python first calculates 5^{17}, which is 762939453125, and only then reduces modulo 17. If you use `pow` like this, Python uses an efficient algorithm that involves reducing modulo 17 continually as it goes along. This doesn't matter much for small numbers, but (as you'll see) Python can handle genuinely pretty large integers, and for those, the `pow` method is much more efficient.

Question: How would you use the % operator to check whether 17 is a factor of 33677?

2.2 Mathematical functions: the `math` module

All we've done so far in Python is use it like a *very* basic calculator. We would hope it would at least offer us what a **scientific** calculator does, namely common mathematical functions like sine, log and e^x.

It does, but there's a catch. If you simply open a new Jupyter notebook and type, say,

```
sqrt(3)
```

it won't work: you get an error along the lines of

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-2f71726bed4c> in <module>()
----> 1 sqrt(3)

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

Yet `sqrt` is exactly the name of the function we want here. What's gone wrong?

The answer is that Python does offer us a square root function, but it's not available "out of the box". Instead, it sits in what's called a **module**, which is a specialised extension to Python's core (that core is kept really quite small, deliberately). What we need to type instead is

```
from math import *
sqrt(3)
```

(Here, the `*` character acts as a **wild card**; it means "from the `math` module, import *everything*.")

We've now got a whole load of mathematical functions at our disposal; try typing `sqrt(5)`, `cos(pi/3)`, `sin(pi/3)`, `exp(1)`, `sinh(1)`, `log(exp(5))` and `atan(3)`.

Notes: Note that `log` means "logarithm to the base e, and that the inverse tangent function in the `math` module is called `atan`.

Questions: What happens if you type `csc(pi/6)` (usually, in computing, `csc` means "cosecant")? What about `sech(2)`? What can you do about this?

2.3 Variables and assignment

So, Python can be used as a basic calculator, and as a scientific calculator. We can also use it as a calculator with a **memory**: we can store values we want to use over and over again as named quantities called **variables**. Try this, for example:

```
a = 1
b = 6
c = 5
```

We can then perform as many calculations as we like with these quantities, for example:

```
(-b + sqrt(b**2-4*a*c))/(2*a)
```

```
-1.0
```

```
(-b - sqrt(b**2-4*a*c))/(2*a)
```

```
-5.0
```

Typing something like

```
sqrt(d)
```

simply generates an error, because `d` hasn't been given a value. (There's actually a way to get Python to understand purely symbolic quantities, but we'll leave that till later.)

Variable names can be as long as we like. In Maths, the culture is to give variables short names like x, α, ∞ or $\aleph_0$, presumably to save on ink, and allow two-dimensional expressions like

$$\sum_{r=1}^{\infty} \frac{1}{r^2}$$

to be written without filling the page. In Computing, the culture is to use long variable names like `client_address_line1`, so that code is what we call **self-documenting**: we can pass it to another programmer, or come back to it ourselves in three months' time, and have a sporting chance of knowing what the various variables mean and stand for.

As a Maths specialist who's learnt to program, I'll probably (a) encourage you to use long, verbose, self-documenting variable names, while (b) often failing to practice what I preach.

2.4 Three ways of importing a module

We've seen how to import all the functions in the math module using the "wild card" option, by typing

```
from math import *
```

This is fine, especially when we're using Python like a calculator, as we are now. But once you start programming, it's good practice to import only what you need. If all you want to do is the calculation

```
sin(pi/6) + 3 * cos(pi/6)
```

for example, then you can simply type

```
from math import sin, cos, pi
```

Then commands like the above will work. On the other hand, something like `exp(5)` won't work, because `exp` hasn't been imported.

This may sound like a silly way to do things, but in fact importing *everything* in a module carries a significant computational overhead, so being selective like this is good "lean and mean" practice, especially when writing programs.

There's a third way of doing it, which is very like the "wild card" import, but slightly different in its effects. To prepare for it, let's first restart the session (go to the **Kernel** menu and choose **Restart**). Then simply type

```
import math
```

Like the previous import command, this gives us all the functions and symbols in the `math` module. The difference is that

```
sin(pi/6) + 3 * cos(pi/6)
```

doesn't work, and instead we must type, in full

```
math.sin(math.pi/6) + 3 * math.cos(math.pi/6)
```

If this seems like an unacceptable faff, it's justified because of the following fact: *it's entirely possible for two or more modules to contain functions with the same name that do slightly different things*. This creates an obvious danger of a "namespace clash", and explicitly tying each function to the module it came from is one way of avoiding that.

We look at one important example next.

2.5 Complex numbers

Python supports complex numbers; slightly annoyingly for mathematicians, it uses the engineer's "j" notation. Try typing

```
z1 = (3  - 4j)
```

```
z2 = (1 + 2j)
```

and then try the calculations

```
z1 + z1
```

```
z1 - z2
```

```
z1 * z2
```

```
z1 / z2
```

If you like, check that Python's done the calculations right! (Notice that `z1 // z2` doesn't work.)

If you want to do calculations involving *mathematical functions* using complex numbers, things aren't quite so straightforward. The following, for example, generates an error:

```
from math import sqrt
sqrt(-4)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-aeffa2c96617> in <module>()
      1 from math import sqrt
----> 2 sqrt(-4)

ValueError: math domain error
```

Complex number functions are handled in Python using a module called `cmath`. If you type

```
from cmath import sqrt
sqrt(-4)
```

everything should work fine.

Now, notice that we've been working with two different functions with exactly the same name: the `sqrt` function from the `math` module, and the `sqrt` function from the `cmath` module. This could get pretty confusing, potentially: which one are we currently using?

To avoid this problem, we could restart Python (choose **Restart** from the **Kernel** menu), and then import the modules like this:

```
import math
import cmath
```

Then, the calculation

```
math.sqrt(4)
```

will use the real function, whereas

```
cmath.sqrt(-4)
```

will use the complex one; we've successfully avoided **namespace clash**.

2.6 Markdown

Jupyter notebooks can contain code and input; they can also contain text and headings. We get the latter using a mechanism called **Markdown**.

Open a new Jupyter notebook, and calculate `2 + 2`; with any luck, you'll get `4`. Now suppose you want to insert some text above this calculation; here's what to do.

1. Click where it says `2 + 2`; this selects the relevant **cell**.
2. Then go to the **Insert** menu, and select **Insert Cell Above**. This creates a new **cell**, just above the `2 + 2` one. Click in this cell.
3. Now find the dropdown menu that currently reads **Code**, and instead select **Markdown**.

You're now ready to type something like

```
This is a text cell, introducing a calculation.
```

To convert that to formatted text, simply "Shift-Return". To make it editable again, double-click. This time, edit it to read

````
This is a text cell, introducing the calculation
```python
2 + 2
```
````

It should then format, and look quite good. Now let's incorporate a second-level section heading: double-click again, and make it

````
## Section 1: Simple Calculations

This is a text cell, introducing the calculation
```python
````
````

````
2 + 2
```
````

(the resizing should happen automatically, and the text should turn blue). When you Shift-Return, there should now be a nice heading in the text cell.

Now convert the cell below `2 + 2` into a Markdown cell, and type

````
This is another text cell, introducing another calculation, $3^7$,
which we format as
```python
3 ** 7
```
````

Then Shift-Return; notice how the LaTeX expression `$3^7$` formats as 3^7.

Finally, use the **Insert** menu to put a cell at the top, convert it to Markdown, and type in

```
# An Introduction to Markdown
```

(again, the resizing should take care of itself). When you Shift-Enter, this will create a level-one heading, suitable for a title.

You can select more than one cell by selecting the one at the top of the group you want, and hitting Shift-J. You can then copy and paste groups of cells using the **Edit** menu, **as long as you stay in the same notebook**. If you want to copy *between* notebooks, you must double click, and select the cell's *contents*. Try it!

3 Data in Python

3.1 Data types

An important thing to understand in Computing is the idea of **data types**. Computers store data as binary code: as sequences of ones and zeros. There are then interfaces to convert that binary data into a human-readable form and vice versa, and also algorithms for operating with the data: for adding and multiplying two numbers, for example.

Exactly how the interfacing works, and exactly what the algorithms do, depends on the **type** of data we're talking about: computers don't set about adding two integers using the same algorithm as for adding two approximate decimals, for example.

In Python, this starts with three different types of **number**: `int`, `float` and `complex`.

- `int` is short for "**integer**"; an `int` can be as big as your computer can handle;
```

- `float` is short for "floating point number"; this is your computer's way of representing what humans read as approximate decimals; except, of course, the internal representation is *binary*;
- `complex` is self-explanatory; a `complex` has a real part and an imaginary part.

Let's start with `int`. Typing

```
int1 = 345
int2 = 456
```

and Shift-Returning, sets up two variables, called `int1` and `int2`, with integer values. Then typing

```
type(int1)
```

should return the output `int`. Tihis tells us how the computer is storing the number, and what algorithms are used when we do arithmetic with it; try

```
int1 + int2
```

801

```
int1- int2
```

-111

```
int1 * int2
```

157320

etc.

Remember that

```
int2 / int1
```

returns an approximate decimal (actually what we're now calling a `float`). To get the **integer quotient**, which is an `int`, you'd type

```
int2 // int1
```

```
1
```

The command `int2 % int1` returns the residue of `int2` modulo `int1`, which is `111`. Finally, `int1 ** int2` returns the value of 345^{456}, which is the rather hefty integer

```
17599784632566441467403397122135703843710061068378527893512480437253206697558693219307484884157138515663446492752441591450112416466266161051479230493334156255307739720847087765794478116736839562726727256906201576148714693763358177732769256696948249708192334143982318442194404761860395908512465553185810797821375665132179916621029172601642466431176963351584034018558113169085771276288402427763125673952147190047654184791610056076247393368471931585435363100733960092526832050233933523573009575494954041518958358386136586218387016935222874064395950274622170307870261518401789781188903912439331247690262779515803013069667135295225961268160887524492602879287569905536136684180417036582166533629197097032718038093790499904203806205001111401361122889580830826395899703059960307149230193867232087759903206273650133471173463551443535596719085348548203474959461345667261449718052111103677507240811022371892031517571941547930077396685510090556167807965941906225208937028904798543877656027483490475700657584782227039301120017582504922074128056015356548878729272628353474979649519183335779186554460678647826821811946047460394808390621301796841180475894361734390258789062 5
```

Notice that Python neither tries to round this number nor reports an error: an `int` can be as large as your computer will handle!

Now for `floats`. Let's set up

```
float1 = 345.0
float2 = 456.0
float3 = 23.456
```

and check that `type(float1)`, say, does indeed return the value `float`. We can then do arithmetic in much the same way as with `ints`, although behind the scenes the algorithms the computer is using will be different.

There are other, more obvious differences. If you try, for example

```
float3 ** float3
```

you should get the output

```
1.3825368655381606e+32
```

This is computer-ese for $1.3825368655381606 \times 10^{+32}$. As you see, Python is rounding, and representing the number in "standard form" (or "scientific notation", as you may know it); this is not something it ever does with `int`s. What's more, if you type

```
float2 ** float2
```

you'll get an **overflow error**: Python allows `int`s of arbitrary size, but for `float`s there's a size limit.

You can convert `int`s to `float`s and vice versa:

```
float(int1)
```

```
345.0
```

```
int(float1)
```

```
345
```

Typing `int(float3)` rounds down to 23.

Finally, `complex`. You can either define a complex number explicitly, like this:

```
comp1 = 2 - 1j
comp2 = 2.4 - 4j
```

or in terms of existing `int`s and `float`s, like this:

```
comp3 = complex(float1, float2)
print(comp3)
```

Something to notice is that

```
comp1.real
```

and

```
comp1.imag
```

return `2.0` and `-1.0` respectively; we might have expected `ints` here, but what we get is always `floats`.

3.2 Data structures

We can think of a a number as a single piece of data; we've met three **types** used in Python. I now want to look at collections of data. A collection of data that we hold together as a single thing is called a **data structure**. Python has many kinds of data structure, and you'll meet several more on this course, but for now let's focus on two **strings** and **lists**.

3.2.1 Strings

First strings. A string is an ordered collection (what in Python is called a **sequence**) of **characters**. Strings in Python can be represented using either single quotes...

```
string1 = 'Python makes me feel dumb'
```

... or double quotes:

```
string2 = "struck with admiration"
```

This flexibility allows us to embed quotes within a string:

```
string3 = "; I 'really mean' that!"
```

We join strings in Python using the + operator:

```
string1 + string2 + string3
```

It's possible to pull out individual characters from a string, using what's called **indexing**. If you type

```
string1[1]
```

what gets returned is the character of `string1` whose index is 1. One might think that that would be `'P'`, but actually, in Python the indexing starts from zero, so `string1[1]` is `'y'`. To get `'P'`, you'd need to type `string1[0]`.

We can also extract substrings; this is called **slicing**. If you type

```
string1[0:6]
```

this returns the substring `'Python'`.

But hang on a minute. This substring consists of six characters, meaning that the indexes represented are 0, 1, 2, 3, 4 and 5. This is also, perhaps, a bit unexpected: typing `string1[0:6]` returns the characters whose indexes are **greater than or equal to** 0, but **strictly less than** 6. This asymmetry in the indexing convention for slicing can take a bit of getting used to.

The input

```
string1[0:10:2]
```

returns `'Pto a`. This slice consists of the characters with indexes 0, 2, 4, 6 and 8: that is, those whose indexes are greater than or equal to 8, and strictly less than 10, **going up in steps of 2**.

To get the number of characters in a string, type

```
len(string1)
```

25

Something that turns out to be surprisingly useful is the ability to *split* a string at all occurrences of a certain character. To see what that does, type

```
splitstrings = string1.split('e')
print(splitstrings)
```

```
['Python mak', 's m', ' f', '', 'l dumb']
```

The output is a collection of substrings (actually a **list** of substrings; more about lists very soon). The boundaries of the substrings are where the character `'e'` occurs in the original string. If you don't specify a character in this way, the space character is used.

Notice that we *don't* type `split(string1, 'e')`, which is what we might expect. Something like `len` is called a Python **function**; something like `split`, which comes after a dot in this way, is called a `method`. More about functions later, and more about methods quite a lot later.

The method that undoes the effect of `split` is called, unsurprisingly, `join`, and it works like this:

```
'e'.join(splitstrings)
```

```
'Python makes me feel dumb'
```

The `replace` method works like this:

```
string1.replace('dumb', 'smart')
```

```
'Python makes me feel smart'
```

Typing `string1.lower()` converts all letters to lower case, and `string1.upper()` gives all caps.
Python has a feature called **string formatting**. Here's an illustration of how that works:

```
template = 'The radius of {} is {} metres.'
```

```
template.format('Jupiter', 69911000)
```

```
'The radius of Jupiter is 69911000 metres'
```

```
template.format('Earth', 6371000)'
```

```
'The radius of Earth is 6371000 metres'
```

3.2.2 Lists

A string is a sequence of characters. A **list** is a sequence of *anything*. Let's start by typing

```
list1 = [1, 2, 3, 4, 5]
```

This is a list of `int`s. If you type

```
list2 = ['one, 'two, 'three', 'four', 'five']
```

you'll have set up a list of strings. And if you type

```
list3 = list1 + list2
```

you get a list some of whose elements are `int`s, and some of which are strings: check it out by typing

```
list3
```

```
[1, 2, 3, 4, 5, 'one, 'two, 'three', 'four', 'five']
```

Literally anything can go in a list: we can even make a list of lists:

```
list4 = [list1, list2, list3]
list4
```

```
[[1, 2, 3, 4, 5'],
 ['one, 'two, 'three', 'four', 'five'],
 [1, 2, 3, 4, 5, 'one, 'two, 'three', 'four', 'five']]
```

We pick out elements by indexing in the same way as with strings:

```
list1[1]
```

2

(Note that the indexing conventions are the same as for strings; the element with index 1 is the second element!)

We get sublists by slicing in the same way as with strings too:

```
list3[0:7]
```

```
[1, 2, 3, 4, 5, 'one, 'two']
```

(these are the elements with indexes 0 to 6), or

```
list3[0:7:2]
```

```
[1, 3, 5, 'two']
```

(these are the elements with indexes 0 to 6, going up in steps of 2).

One neat thing we'll make a lot of use of is **appending**. If you type

```
list1.append('six')
```

you don't get any output; what's happened is that the value of `list1` has changed. You can see this by typing

```
list1
```

```
[1, 2, 3, 4, 5, 'six']
```

You can always, then, lengthen a list by one, by appending an extra element to it.

A special, very useful kind of list is one consisting of equally-spaced integers. We've already typed one of those in by hand, but here's a slicker way:

```
list5 = list(range(6))
```

If you now type

```
list5
```

you'll get the output

```
[0, 1, 2, 3, 4, 5]
```

4 Programming Essentials: Iteration and Branching

4.1 `for` loops

Let's start doing some programming.

One thing computers are brilliant at, and humans tend to find difficult and frustrating, is doing the same thing over and over again; what we call **iteration**. The simplest kind of iteration (in Python and many other languages) is probably the `for` loop. Here's an example.

Challenge 1: print "Hello World!" ten times.

```
for n in range(10):
    print('Hello World!')
```

```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

What happened was this. We set up a variable called `n`, which began by taking the value 0. Then the string "Hello World!" was printed. Then `n` took the value 1, and "Hello World!" was printed again. Then it took the value 2, and we got another "Hello World!". Then it became 3, then 4, and so on, up to its final value, which was 9. The string "Hello World!" was printed once for each value of `n` between 0 and 9 inclusive; that is, ten times.

Now, that's fine, if a bit dull. Let's be very slightly more ambitious, by writing a program that makes use of all these values of `n`.

Challenge 2: calculate n^2 for n from 0 to 9.

```
for n in range(10):
    print(n**2)
```

```
0
1
4
9
16
25
36
49
64
81
```

We might want to use string formatting here:

```
for n in range(10):
    print('The square of {} is {}'.format(n, n**2))
```

```
The square of 0 is 0
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
The square of 6 is 36
The square of 7 is 49
The square of 8 is 64
The square of 9 is 81
```

Now let's write something that might actually be useful. It turns out that if you evaluate the cosine of, say, 1.0, and then the cosine of that, and then the cosine of that, and so on, you get closer and closer to an angle in radians that is equal to its own cosine; that is, to a solution of the equation $x = \cos x$.

Challenge 3: find a value of x approximately equal to its own cosine.

First we better import a cosine function. Then we should set an initial value for x; we'll use 1.0. Then we're going to repeat the step `x = cos(x)` a set number of times (let's use 20). This step means "Let the new value of the variable `x` be equal to the cosine of the old value." Then, each time, let's print the current value of `x`.

```
from math import cos
x = 1.0
for n in range(20):
    x = cos(x)
    print('Iteration {}: {}'.format(n+1, x))
```

```
Iteration 1: 0.5403023058681398
Iteration 2: 0.8575532158463933
Iteration 3: 0.6542897904977792
Iteration 4: 0.7934803587425655
Iteration 5: 0.7013687736227566
Iteration 6: 0.7639596829006542
Iteration 7: 0.7221024250267077
Iteration 8: 0.7504177617637605
Iteration 9: 0.7314040424225098
Iteration 10: 0.7442373549005569
Iteration 11: 0.7356047404363473
Iteration 12: 0.7414250866101093
Iteration 13: 0.7375068905132428
Iteration 14: 0.7401473355678757
Iteration 15: 0.7383692041223232
Iteration 16: 0.739567202212256
Iteration 17: 0.7387603198742114
Iteration 18: 0.7393038923969057
Iteration 19: 0.7389377567153446
Iteration 20: 0.7391843997714936
```

(**Note**: there's a small bug in the version of this in the video tutorial; can you spot it? I'll correct this as soon as I can.)

Our fourth challenge now.

Challenge 4: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1}.$$

Here's a first go at this. We set up a variable called `total`, with initial value 0.0 (a `float`, notice). We then want to use the values of `n` between 0 and 100, meaning our `range` object

needs to be `range(101)`. We're going to calculate the value of the term for each of those values of `n`, then add it to `total`.

```
total = 0.0
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
    print(total)
```

However, if you run this, you get a massive printout of all 101 partial sums for values of `n` between 0 and 100. That seems excessive. How can we tweak this code so that we only see the final value of `total`?

In programming terms, what we're trying to do is move the command `print(total)` **outside the loop**, so that it only executes once, when 101 iterations have taken place. Different computer languages have different ways of marking where a block of code begins and ends; Python uses **indentation**. To move `print(total)` outside the loop, we simply remove its indentation, so that it begins at the start of the line.

```
total = 0.0
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
print(total)
```

```
3.1514934010709914
```

(Notice this is not far from π. That's no accident: this summation does converge to π, though really rather slowly.)

Now for our final challenge.

Challenge 5: iterate the two-dimensional Hénon map

$$\begin{aligned} x_{n+1} &= 1 - 1.4\,{x_n}^2 + y_n, \\ y_{n+1} &= 0.3\,x_n \end{aligned}$$

20 times, starting with $x_0 = y_0 = 0.5$.

For this, we're going to make use of a neat trick, which not every programming language allows, which lets us assign values to more than one variable at the same time. To initialise the values of `x` and `y`, for example, we simply need to type

```
x, y = 0.5, 0.5
```

Here's the full code.

```
x, y = 0.5, 0.5
for n in range(20):
    x, y = 1 - 1.4*x**2 + y, 0.3*x
    print('({}, {})'.format(x, y))
```

```
(1.15, 0.15)
(-0.7014999999999995, 0.345)
(0.656056850000001, -0.21044999999999983)
(0.18697517339530675, 0.1968170550000003)
(1.1478734533473132, 0.05609255201859203)
(-0.7885662988406887, 0.34436203600419396)
(0.47379050526997063, -0.2365698896522066)
(0.4491616903102298, 0.1421371515809912)
(0.8596924379217113, 0.13474850709306893)
(0.1000489841453833, 0.2579077313765134)
(1.243894012456581, 0.030014695243614987)
(-1.1361665446718507, 0.3731682037369743)
(-0.4340559803872273, -0.3408499634015552)
(0.39538360484456087, -0.13021679411616818)
(0.650923732912, 0.11861508145336826)
(0.5254326929580385, 0.1952771198736)
(0.8087657991128091, 0.15762980788741154)
(0.24188684294699858, 0.24262973973384272)
(1.1607167970266303, 0.07256605288409958)
(-0.813602823175564, 0.34821503910798907)
```

(There's not much apparent order in these numbers, and that's no accident: there's strong evidence that this map is what we call **chaotic**.)

4.2 Using append

The `for` loops in the last section are all very well, but all they do is print values. Often, we'd prefer it if instead our programs built a **data structure** containing all the output; that way, it's available for us to do calculations with. Let's start with a slightly silly example.

Challenge 1: create a list containing 10 copies of the string "Hello World!".

This is exactly like the challenge in the last section, except that we don't want a printout of ten "Hello World!" strings, but a list containing them. Our tactic will be to set up a variable that begins its life as an empty list, and then to use the `append` method to lengthen this list by one on each turn of the loop.

```
hw_list = []
for n in range(10):
    hw_list.append('Hello World!')
    print(hw_list)
```

```
['Hello World!']
['Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!']
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

That shows quite nicely what happens: at each turn of the loop, the list gets another copy of the string appended to it. However, in practice, we're unlikely to want to see all those intermediate values of the list, with one, two, three elements etc; let's move the `print` command outside the loop, so we only see the final version, with ten elements.

```
hw_list = []
for n in range(10):
    hw_list.append('Hello World!')
print(hw_list)
```

```
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

Challenge 2: create a list of all the squares of the integers between 0 and 9.

```
sq_list = []
for n in range(10):
    sq_list.append(n**2)
print(sq_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note: this probably isn't the best way to do this task; later in the course we'll look at **comprehensions**, which give us a slicker method.

One nice thing about having all these numbers in a list like that is that we can do calculations with them. For example, we could plot them. More about plotting, in systematic detail, later in the course, but for now, here's how we could create an appropriate plot. We need to tell Jupyter that we want graphics to appear in the notebook rather than a separate window; we then want to import the `pyplot` submodule of the `matplotlib` module; finally, we want to create a **point plot**, using the integers from 0 to 9 on the horizontal axis, and these squares we've just calculated on the vertical one.

Here's the code:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(10), sq_list, '.')
```

The image is shown in Figure 6.

Notice that if we'd simply printed those squares, we wouldn't have been able to do that; using `append` like this gives us our values in a form we can use for calculations. All programming languages allow programs to create data structures in broadly this way, though the precise details of how it works vary rather a lot from language to language.

Challenge 3: iterate $x_{n+1} = \cos x_n$ 20 times, starting with $x = 1.0$, this time creating a list of all the iterates

One slight difference here is that we'll set up our list of values not as an empty list but as a list containing only the initial value, 1.0. This is a *design decision*; we could leave out this value if we liked. However, I think it makes more sense to include it.

```
from math import cos
```

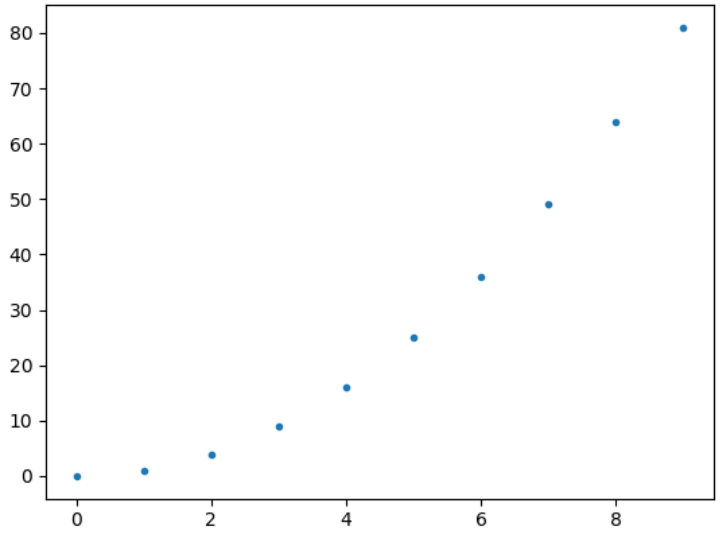


Figure 6: Plot of n^2 against n for $n = 0, 1, 2, \ldots, 9$

```
x = 1.0
x_list = [x]
for n in range(20):
    x = cos(x)
    x_list.append(x)
print(x_list)
```

```
[1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792,
0.7934803587425655, 0.7013687736227566, 0.7639596829006542,
0.7221024250267077, 0.7504177617637605, 0.7314040424225098,
0.7442373549005569, 0.7356047404363473, 0.7414250866101093,
0.7375068905132428, 0.7401473355678757, 0.7383692041223232,
0.739567202212256, 0.7387603198742114, 0.7393038923969057,
0.7389377567153446, 0.7391843997714936]
```

What about the plotting code? Let's create a line plot this time (we simply leave out the `'.'`). Also, we needn't include the first two lines, in which we tell Juypter to put the image in the notebook and import the relevant submodule, because we've already done that this session; that's unless you've either broken session, moved to a new notebook or restarted the Kernel since creating the last plot (if you have, simply include those two lines).

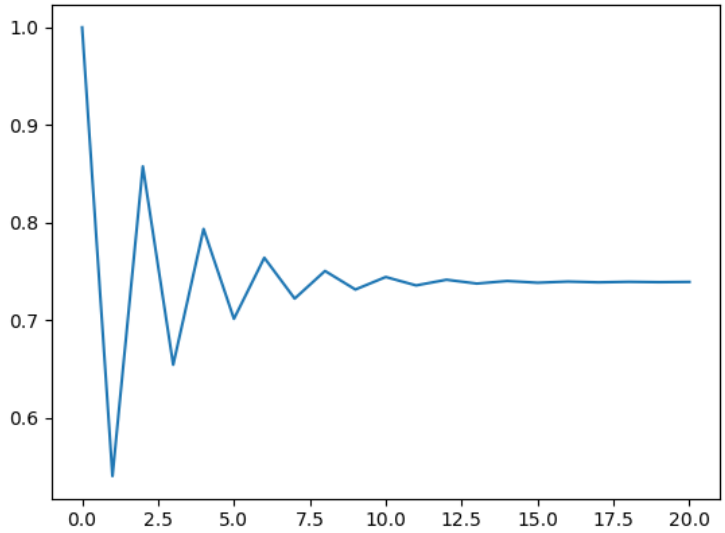


Figure 7: Plot of iterates of $x_{n+1} = \cos x_n$

```
plt.plot(range(21), x_list)
```

Notice that it's `range(21)` and not `range(20)`; this is a consequence of our decision to include the initial value, which appears as "iterate zero". The image is shown in Figure 7.

Challenge 4: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1},$$

this time creating a list of all the partial sums

```
total = 0.0
partial_sums = []
for n in range(101):
    total = total + (4*(-1)**n)/(2*n + 1)
```

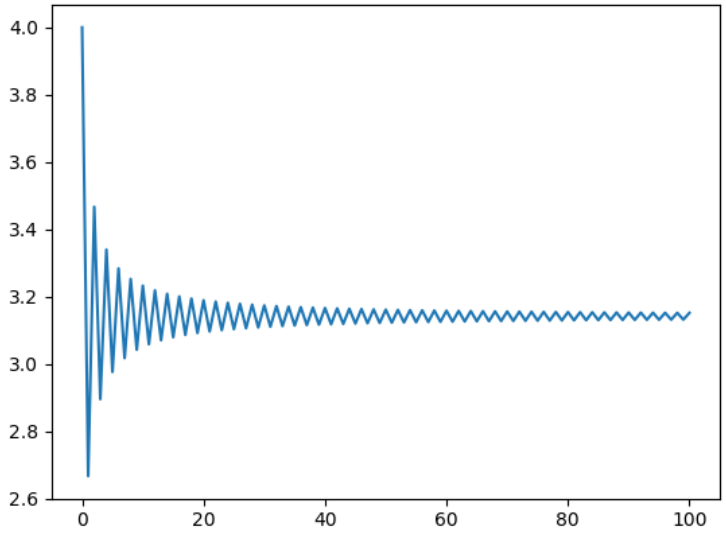


Figure 8: Plot of partial sums in series that converges slowly to π

```
    partial_sums.append(total)
print(partial_sums)
```

This time, I won't include the printout of the list, because it's massive. Here's the plotting code:

```
plt.plot(range(101), partial_sums)
```

Again, it's easy to put `range(100)` instead of `range(101)`; this "off by one" type of coding error is one of the easiest to make in computing! The plot appears in Figure 8; you can see clearly how slow the convergence is.

Challenge 5: iterate the two-dimensional Hénon map

$$\begin{aligned} x_{n+1} &= 1 - 1.4\,{x_n}^2 + y_n, \\ y_{n+1} &= 0.3\,x_n \end{aligned}$$

10000 times, this time creating lists of coordinate pairs.

The idea this time is to create two separate lists, `x_list` and `y_list`, containing, respectively, the iterates $x_0, x_1, x_2, \ldots$ and $y_0, y_1, y_2, \ldots$. We'll initialise these two lists as "list containing x_0" and "list containing y_0" respectively.

```
x, y = 0.5, 0.5
x_list, y_list = [x], [y]
for n in range(10000):
    x, y = 1 - 1.4*x**2 + y, 0.3*x
    x_list.append(x)
    y_list.append(y)
```

I won't bother printing these enormous lists, each of which has 10001 elements. Instead, let's do a point plot of `x_list` (horizontal axis) against the corresponding values of `y_list` (vertical axis), using the optional **keyword argument** `markersize` to make the points as small as we can.

```
plt.plot(x_list, y_list, '.', markersize=0.1)
```

The image appears as Figure 9. This shape, which is called a **strange attractor**, is extremely intricate, and has some fascinating mathematical properties.

4.3 `while` loops

The `for` loop is a wonderful thing: it allows us to get a computer to do the same thing over and over again, which they're great at and we hate. But they only work if we know in advance exactly how many times we want the thing, whatever it is, to be done. And sometimes we don't; sometimes we want to say to the computer "Do this thing over and over again until the job is done, but I don't know how many times that will be yet." Here's an example.

Challenge 1: the iteration

$$x_{n+1} = \frac{x_n + 2}{{x_n}^2 + 1}$$

converges fairly slowly to $\sqrt[3]{2}$. Run this iteration 20 times, starting with $x = 1.0$, printing the successive values of x.
Then run it, instead, not for a fixed number of times but until x^3 is within 0.00005 of 2.

First the fixed number of times; a `for` loop is ideal here.

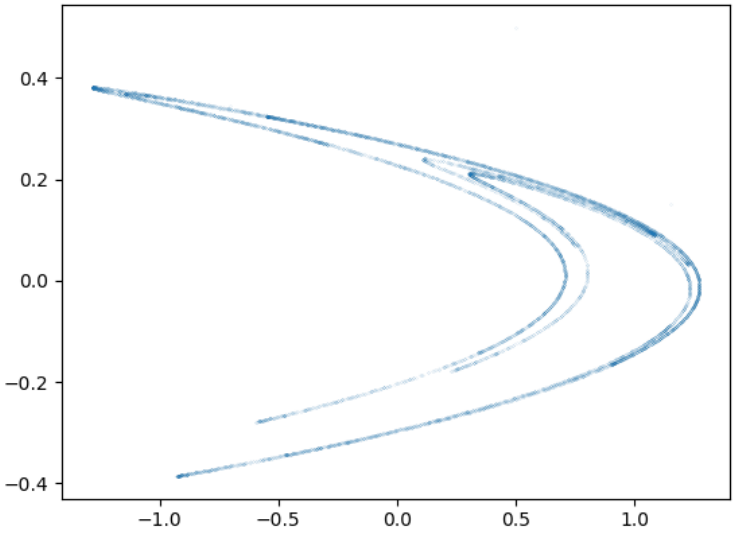


Figure 9: Strange attractor of the Hénon map

```
x = 1.0
for n in range(20):
    x = (x + 2.0)/(x**2 + 1.0)
    print(x)
```

```
1.5
1.0769230769230769
1.4246575342465755
1.1303809228863424
1.3743236803695795
1.1680849791244732
1.339897989633841
1.1948149323951562
1.3160478127293076
1.2137883779591594
1.2994022047163227
1.2272525074812513
1.2877338385533594
1.2367999252342665
1.2795324040899612
```

```
1.2435645601831735
1.2737579666268228
1.2483541288095155
1.269687822783779
1.2517433678387995
```

For the other part of the task, where we want to run it until x^3 gets close to 2, we replace the `for` loop with what's called a `while` loop:

```
x = 1.0
while abs(x**3 - 2) > 0.00005:
    x = (x + 2.0)/(x**2 + 1.0)
    print(x)
```

```
1.5
1.0769230769230769
1.4246575342465755
1.1303809228863424

...

1.259930405269743
```

(There are many values here, and I've cut most of them!)

What this means is "Do this thing for as long as the absolute value of $(x^3 - 2)$ is greater than 0.00005".

Note: A while loop always includes a **condition**. This condition starts off **true**, otherwise the loop won't run at all. And it ends up **false**, otherwise the loop will run forever.

OK, strictly, neither of those things is quite right: sometimes, we want a loop that fails to run in certain circumstances, and sometimes, surprisingly, we want one that runs forever, or at any rate until we force a break.

But those are the exceptions: almost always we want a while loop that executes at least once, but only a finite number of times, which means the condition must start off true, and **change its value** during the execution of the loop in such a way that it ends up false. This is a good example: we start with a value of x whose cube is a long way from 2, making the condition `abs(x**3 - 2) > 0.00005` true, and we have an iteration we know converges to $\sqrt[3]{2}$, meaning that eventually the condition will be false.

Note: If the condition in a while loop goes from being true to being false, it must depend on the value of at least one **variable**, and that value must **change** during the execution of the loop.

This seems almost obvious, right? But you'd be surprised how many attempted `while` loops I see don't obey these rules! So do check, when you write a `while` loop: does its condition depend on the value of a variable, and does that value change when the loop is executed? Your `while` loop *may* not work even if both those things are true, but it definitely won't work if they're not. (Except for those special cases I talked about; but let's worry about those later.)

Challenge 2: run the iteration

$$x_{n+1} = \cos x_n$$

starting with $x = 1.0$, until successive iterates lie within 0.00005 of each other.

This one is a little different. It's one thing to check whether we're near the cube root of 2 by comparing x^3 with 2, but how do we check we're near a solution of $x = \cos x$? The only way, really, is to carry on iterating until successive iterates don't change very much; we call this a **heuristic convergence criterion**.

Here's a first crack at the code:

```
from math import cos
x = 1.0
while abs(x - cos(x)) > 0.00005:
    x = cos(x)
    print(x)
```

```
0.5403023058681398
0.8575532158463933
0.6542897904977792
0.7934803587425655
0.7013687736227566
0.7639596829006542
0.7221024250267077
0.7504177617637605
0.7314040424225098
0.7442373549005569
0.7356047404363473
0.7414250866101093
0.7375068905132428
0.7401473355678757
```

```
0.7383692041223232
0.739567202212256
0.7387603198742114
0.7393038923969057
0.7389377567153446
0.7391843997714936
0.7390182624274122
0.7391301765296711
0.7390547907469175
0.7391055719265361
```

This is fine, except for one thing: it's not very efficient. That's because the cosine function is evaluated twice at each turn of the loop: once when the condition

```
abs(x - cos(x)) > 0.0005
```

is checked, and once when the update step,

```
x = cos(x)
```

is executed inside the loop.

This is a very minor problem for a small, fast program like this, but it's good to get into good habits early on. How do we solve it? One way is to have not one value of `x` but two on the go; let's call them `oldx` and `newx`, where at any one time, `newx = cos(oldx)`. The condition will then be

```
abs(oldx - newx) > 0.00005
```

which doesn't involve the evaluation of a cosine. Here's the complete program:

```
from math import cos
oldx = 1.0
newx = cos(oldx)
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = cos(oldx)
    print(oldx)
```

```
0.5403023058681398
0.8575532158463933
0.6542897904977792
0.7934803587425655
0.7013687736227566
0.7639596829006542
0.7221024250267077
0.7504177617637605
0.7314040424225098
0.7442373549005569
```

```
0.7356047404363473
0.7414250866101093
0.7375068905132428
0.7401473355678757
0.7383692041223232
0.739567202212256
0.7387603198742114
0.7393038923969057
0.7389377567153446
0.7391843997714936
0.7390182624274122
0.7391301765296711
0.7390547907469175
0.7391055719265361
```

Actually, we can improve it slightly by printing the value of `newx` instead of that of `oldx`; that way, we squeeze out one more iterate, and get what's probably a more accurate result.

> **Note**: The **heuristic convergence criterion** is impossible to justify in general, and indeed, in general it's false. The fact that iterates aren't changing very much tells us nothing rigorous about whether or not we're close to a solution. But it's extremely widely used anyway, because exceptions are rare.

Here's an example of an exception. If you run the following code

```
oldx = 0.9
newx = -oldx**4 + 4*oldx**3 -5*oldx**2+3*oldx+0.00001
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = -oldx**4 + 4*oldx**3 -5*oldx**2+3*oldx+0.00001
    print(newx)
```

you get quite a long printout, ending in

```
0.993773880037735
```

And yet there's no solution of $x = -x^4 + 4\,x^3 - 5\,x^2 + 3\,x + 0.00001$ anywhere near 0.99377; the real solutions are about $x = -0.00005$ and $x = 2.00005$. So be a bit wary of the heuristic convergence criterion!

Challenge 3: run the `while` loop iteration from Challenge 1 again, this time putting the iterates into a list.

```
x = 1.0
x_list = [x]
while abs(x**3 - 2) > 0.00005:
    x = (x + 2.0)/(x**2 + 1.0)
    x_list.append(x)
print(x_list)
```

I won't bother showing the values!

Challenge 4: run the `while` loop iteration from Challenge 2 again, this time putting the iterates into a list.

```
from math import cos
oldx = 1.0
newx = cos(oldx)
x_list = [oldx, newx]
while abs(oldx - newx) > 0.00005:
    oldx = newx
    newx = cos(oldx)
    x_list.append(newx)
print(x_list)
```

Again, let's not bother with the values.

4.4 `if` and branching

One of the main ideas of programming is **iteration**: getting the computer to do the same thing over and over again. The other big idea to get to grips with is **branching**: getting the computer to make a choice about what to do dependent on some condition or other.

Challenge 1: write a program that checks whether 8 is a factor of 24, and if it is, prints an appropriate message. Repeat for 9 instead of 8.

Clearly, 8 *is* a factor of 24: if we type

```
24 % 8
```

we get 0, and if we type

```
24 % 8 == 0
```

we get `True`. (Notice, to check whether two quantities are equal in Python, we use the double equals sign, ==; the single equals sign is reserved for **assigning a value to a variable**.)

So: what we want is for Python to run a check to find out whether 8 is a factor of 24, and if it is, print something (and if it isn't, do nothing). We might do that like this.

```
if 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
```

```
Hooray! 8 is a factor of 24
```

If you replace the 8s with 9s...

```
if 24 % 9 == 0:
    print("Hooray! 8 is a factor of 24")
```

... you get no output at all.

Note: don't get `if` mixed up with `while`!

In some ways, `if` and `while` are a bit similar. Both involve the computer checking the value of a condition, and if that value is `True`, executing some code. But with `if`, the code is executed just once; with `while`, the code is executed repeatedly until the condition becomes `False`. If the program was

```
while 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
```

it would run forever, printing the string again and again until we force-broke it. So this can be quite a costly mistake to make!

In this first example, the logic was "If the condition is `True`, execute this code; it it isn't, do nothing at all." We can also build programs for which the logic is "If this condition is true, execute this piece of code, and if it isn't, execute this other piece of code."

Challenge 2: write a program that checks whether 8 is a factor of 24, prints an appropriate message depending on whether it is or not. Repeat for 9 instead of 8.

First for 8:

```
if 24 % 8 == 0:
    print("Hooray! 8 is a factor of 24")
else:
    print("Oh no! 8 is not a factor of 24")
```

```
Hooray! 8 is a factor of 24
```

Now for 9:

```
if 24 % 9 == 0:
    print("Hooray! 9 is a factor of 24")
else:
    print("Oh no! 9 is not a factor of 24")
```

```
Oh no! 9 is not a factor of 24
```

Now let's embed a branch in a loop.

Challenge 3: write a program that iterates through the integers between 1 and 24, printing the values of any that are factors of 24.

We need to be a bit careful with our `range` object here. Remember that we want those integers that are greater than or equal to 1, and strictly less than 25; the right way to get those is `range(1, 25)`. Here's a program for printing all of them:

```
for n in range(1, 25):
    print(n)
```

```
1
2
3
4
5
```

```
6

...

24
```

That's not what we want, though; we want to print only those numbers that are factors of 24. For that we need an `if`, inside the `for`:

```
for n in range(1, 25):
    if 24 % n == 0:
        print(n)
```

```
1
2
3
4
6
8
12
24
```

Notice the double-indenting for the line of code inside the `if`, inside the `for`.

Challenge 4: write a program that iterates through the integers between 1 and 24, appending the values of any that are factors of 24 to a list.

```
fac_list = []
for n in range(1, 25):
    if 24 % n == 0:
        fac_list.append(n)
print(fac_list)
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

Challenge 5: write a program that iterates through the integers between 1 and 24, appending the values of any that are factors of 24 to one list, and those of any that *aren't* factors to another.

```
fac_list = []
nonfac_list = []
for n in range(1, 25):
    if 24 % n == 0:
        fac_list.append(n)
    else:
        nonfac_list.append(n)
print(fac_list)
print(nonfac_list)
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
[5, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
```

Finally, let's write a program for calculating the maximum of a list of numbers.

Challenge 6: write a program to calculate the maximum of the numbers

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
   34, 6, 29, 74, 86, 23]
```

One thing to realise is that we can run a `for` loop not only over a `range`, but also over a list. So we're allowed to type

```
for x in x_list
```

and the variable `x` will take on, successively, the values 43, 62, 53 and so on.

So here's our tactic. Set up a variable, representing the "maximum so far"; give it the initial value 0, which is less than all the positive integers in this list. Then, loop through `x_list`, checking whether each value is greater than the "maximum so far". If it is, update the value of the "maximum so far", making it this new number. If it isn't, do nothing.

Here's the code:

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
       34, 6, 29, 74, 86, 23]
max_x = 0
for x in x_list:
    if x > max_x:
        max_x = x
print(max_x)
```

```
97
```

4.5 Iterable objects

We've now written quite a few `for` loops. Most of them have been over `range` objects, but the last one we wrote was over a **list**.

The structure of a `for` loop is always

```
for x in <something>:
    <do a thing>
```

What we're concerned with here is what the allowable `<something>`s are!

In fact, we can write a `for` loop over a `range` object, or a list, or a string; and that's just the start of it. Lists first:

Challenge 1: for the list

```
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
   34, 6, 29, 74, 86, 23]
```

write a `for` loop that prints out all the square roots.

```
from math import sqrt
x_list = [43, 62, 53, 75, 53, 3, 97, 44, 69, 47, 84, 63, 90, 80,
       34, 6, 29, 74, 86, 23]
for x in x_list:
    print(sqrt(x))
```

```
6.557438524302
7.874007874011811
7.280109889280518
8.660254037844387
7.280109889280518

...

4.795831523312719
```

Now strings. For this next example, you need to know that the function called `ord` associates a unique numerical **character code** with each alphanumeric character; for example, `ord('n')` returns 110.

Challenge 2: for the list

```
spy_string = 'My name is Bond, James Bond.'
```

write a `for` loop that prints out each character code.

```
spy_string = 'My name is Bond, James Bond.'
for character in spy_string:
    print(ord(character))
```

```
77
121
32
110

...

46
```

(Notice that I've used the variable name `character`. Now, I needn't have; I could have called this variable `x`, or `ian`, or `spoon`. But this variable stands for a character, so calling it `character` helps with the readability of our code; helps to make it **self-documenting**.)

Here's our final challenge in this section.

Challenge 3: for each of $x = 33, 35, 37, 39, 41, 43, 45, 47$, print the integer value of

$$\frac{3\,x+1}{2}.$$

Do this in five different ways.

One way is by cheekily copying and pasting from the notes (this is rather frowned upon, but let's go crazy).

```
x_list = [33, 35, 37, 39, 41, 43, 45, 47]
for x in x_list:
    print((3*x+1)//2)
```

```
50
53
56
```

```
59
62
65
68
71
```

However, these numbers are equally spaced, meaning we could create `x_list` like this:

```
x_list = list(range(33, 48, 2))
```

So here's a second implementation:

```
x_list = list(range(33, 48, 2))
for x in x_list:
    print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

(You could argue this isn't really different from the first implementation, but let's let that slide.)

But lists aren't the only things we're allowed to loop over; there are also `range` objects. Now, a `range` object isn't the same thing as a list. You can tell that if you type

```
x_range = range(33, 48, 2)
print(x_range)
```

```
range(33, 48, 2)
```

The difference is that a list's contents are all explicit and visible; it's what we call a **transparent** sequence. A `range` object, by contrast, keeps its contents hidden; indeed, it doesn't even create them till they're needed. These **opaque** sequences take up much less memory, which is why it's a good idea to use them when we can.

Here's our third implementation, using a `range`:

```
x_range = range(33, 48, 2)
for x in x_range:
    print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

The idea of opaque sequences is quite powerful. For example, because such sequences only create their contents when they're needed, it's even possible to have transparent sequences that are, at least potentially, **infinite**. One such construct is the `count` object, which needs to be important from a module called `itertools`:

```
from itertools import count
x_count = count(33, 2)
```

Now, this object contains the integers 33, 35, 37, 39, etc, **going on to infinity**. The reason it can do that is because it's opaque, meaning we don't need infinite memory; each number is created only when it's needed, so this infinity is merely a *potential* one.

But it still has to be treated with care; it can still cause an infinite loop if mishandled. Here's a program that uses `count`, forcing a break when we've got to our final number:

```
from itertools import count
x_count = count(33, 2)
for x in x_count:
    if x > 47:
        break
    else:
        print((3*x+1)//2)
```

```
50
53
56
```

```
59
62
65
68
71
```

That's four implementations so far. For the last one, we'll import another function from `itertools`, called `islice`. This takes a **slice** out of a `count` object, making it finite again; here, we want to slice out the first eight elements of `count(33, 2)`.

```
from itertools import count, islice
x_slice = islice(count(33, 2), 8)
for x in x_slice:
     print((3*x+1)//2)
```

```
50
53
56
59
62
65
68
71
```

You may wonder what the point is of creating an infinite sequence and then slicing it so it's finite; more on that later in the course.

So, to summarise, thinsg it's possible to write `for` loops over include:

- lists;
- strings;
- `range` objects;
- `count` objects;
- `islice` objects.

There are several others. These are all examples of **sequences**, and all have the property of being **iterable**. Lists and strings are **transparent** sequences; the others are **opaque**. (You may, if you read up on Python, also come across the term **iterator**; more about that later.)

As an illustration, let's set up four iterable objects:

```
r = range(10)
l = list(r)
c = count()
i = islice(c, 10)
```

Then we can check the values of these things:

```
print(r)
print(l)
print(c)
print(i)
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
count(0)
<itertools.islice object at 0x00000000052305E8>
```

Finally, we can check that they all belong to the Python **collection** known as `Iterable`, as follows:

```
from collections import Iterable
print(isinstance(r, Iterable)
print(isinstance(l, Iterable)
print(isinstance(c, Iterable)
print(isinstance(i, Iterable)
```

```
True
True
True
True
```

This means, as we've seen, that `for` loops can be written across them all; the same, as you'll see soon, is true of the powerful Python-specific programming construct called the **comprehension**.

MATH40006: An Introduction To Computation
Course notes, Volume 2

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

5 Further programming

5.1 Tuples and immutability

Most computing languages support some kind of data structure in the form of a one-dimensional sequence of pieces of data, though the details of how this works vary enormously from language to language. One of the basic data structures in Python is, as you've seen, the **list**.

Python actually supports another data structure that is in most respects extremely similar to the list, with a few crucial differences (actually, really, with just one crucial difference). This is called the **tuple**.

The most obvious difference between a list and a tuple is in the way they're represented to users: a list uses square brackets and a tuple uses round ones.

Challenge 1: create and print a list, `list1` and a tuple, `tuple1`, each containing the data `1, 'two', 3.0, (4+0j) and 'V'`.

```
list1 = [1, 'two', 3.0, (4+0j), 'V']
tuple1 = (1, 'two', 3.0, (4+0j), 'V')
print(list1)
print(tuple1)
```

```
[1, 'two', 3.0, (4+0j), 'V']
(1, 'two', 3.0, (4+0j), 'V')
```

That's not a very big difference, though, and in many respects they're extremely similar.

Challenge 2: create and print a list, `list2` and a tuple, `tuple2`, each containing the integers $31, 34, 37, 40, \ldots, 103$.

```
list2 = list(range(31,104,3))
tuple2 = tuple(range(31,104,3))
print(list2)
print(tuple2)
```

```
[31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67,
        70, 73, 76, 79, 82, 85, 88, 91, 94, 97, 100, 103]
(31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61, 64, 67,
        70, 73, 76, 79, 82, 85, 88, 91, 94, 97, 100, 103)
```

Challenge 3: extract and print element number 4 (starting with 0) of `list2` and `tuple2`.

```
print(list2[4])
print(tuple2[4])
```

```
43
43
```

Challenge 4: print a list consisting of elements $0, 2, 4, 6$ and 8 of `list2`, and a tuple consisting of elements $0, 2, 4, 6$ and 8 of `tuple2`.

```
print(list2[0:9:2])
print(tuple2[0:9:2])
```

```
[31, 37, 43, 49, 55]
(31, 37, 43, 49, 55)
```

You can even run a `for` loop over a tuple, and it behaves the same way as when you run one over a list.

Challenge 5: go through `list2`, and print out each square root. Then do the same with `tuple2`.

```
from math import sqrt
print('First the list')
print('--------------')
for n in list2:
    print(sqrt(n))
print('')
print('Now the tuple')
print('-------------')
for n in tuple2:
    print(sqrt(n))
```

```
First the list
--------------
5.5677643628300215
5.830951894845301
6.082762530298219

...

10.14889156509222

Now the tuple
-------------
5.5677643628300215
5.830951894845301
6.082762530298219

...

10.14889156509222
```

So what's the difference between them? Well, here's an illustration.

Challenge 6: append the value 6 to `list1`, then try to do that with `tuple1`

Now, if we go first with the list, it works fine:

```
list1.append(6)
print(list1)
```

```
[1, 'two', 3.0, (4+0j), 'V', 6]
```

But if we try it with the tuple, it all goes wrong:

```
tuple1.append(6)
print(tuple1)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-29-34955e41cdef> in <module>()
----> 1 tuple1.append(6)
      2 print(tuple1)

AttributeError: 'tuple' object has no attribute 'append'
```

So that's one difference: tuples are like lists except you can't append to them. If we wanted to solve this problem, we'd need to do something like this:

```
tuple1 = tuple1 + (6,)
print(tuple1)
```

```
(1, 'two', 3.0, (4+0j), 'V', 6)
```

(Note in passing that if we want a tuple with one element, like the tuple containing just 6, we need to type `(6,)` and not `(6)`; the latter, annoyingly, is the same as 6.) Another solution would be to convert our tuple into a list and back again:

```
tuple1 = (1, 'two', 3.0, (4+0j), 'V')
temp_list = list(tuple1)
temp_list.append(6)
tuple1 = tuple(temp_list)
print(tuple1)
```

```
(1, 'two', 3.0, (4+0j), 'V', 6)
```

The other main difference (actually, it's really another aspect of the same difference) is shown by this example:

Challenge 7: with these new six-element definitions of `list1` and `tuple1`, change the final element, 6, to a float.

Once again, with the list, it works fine:

```
list1[5] = 6.0
print(list1)
```

```
[1, 'two', 3.0, (4+0j), 'V', 6.0]
```

But again if we try it with the tuple, no such luck:

```
tuple1[5] = 6.0
print(tuple1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-33-c3c524102390> in <module>()
----> 1 tuple1[5] = 6.0
      2 print(tuple1)

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

So tuples are the same as lists, except (a) we can't append to them and (b) we can't give their elements new values. We say that lists are **mutable** and tuples are **immutable**.

But isn't this a bit like saying tuple are the same as lists, except worse? What's the point of having an immutable data type? Unfortunately, the answer to that one is going to have to wait till later in the course; there is an advantage, in certain contexts, in having data whose value you know is stable. I mention tuples this early purely because they crop up all the time (and indeed you'll already have met them if you've looked at the `zip` function.)

5.2 `True` and `False`

You remember we talked about three **data types** in Python: ints, floats and complexes? Actually, **strings** are technically counted as a data type too (even though I introduced them as a data *structure*), so that makes four you've met.

Now for a fifth: what we call type **Boolean**. Actually, you've met these before; but let's look at them a but more systematically.

The Boolean data type consists of two values, `True` and `False`. These values serve as the outputs for certain kinds of input in Python, including those based on the operators <, >, <=, >= and ==:

```
print(2 < 4)
print(4 < 2)
print(5 < 5)
print(5 <= 5)
print(5 == 5)
```

```
True
False
False
True
True
```

Just as we can combine numerical data using operators such as +, −, *, /, // and %, so we can combine Boolean data. The two most important operators are `and` and `or`. Typing

```
<expr1> and <expr2>
```

returns `True` if both `<expr1>` and `<expr2>` are `True`, and `False` otherwise.

```
print(2 < 4 and 4 % 2 == 0)
print(2 > 4 and 4 % 2 == 0)
print(2 < 4 and 4 % 2 == 1)
print(2 > 4 and 4 % 2 == 1)
```

```
True
False
False
False
```

Typing

```
<expr1> or <expr2>
```

returns `True` if one or other of `<expr1>` or `<expr2>`, or both, are `True`, and `False` otherwise.

```
print(2 < 4 or 4 % 2 == 0)
print(2 > 4 or 4 % 2 == 0)
print(2 < 4 or 4 % 2 == 1)
print(2 > 4 or 4 % 2 == 1)
```

```
True
True
True
False
```

There's also `not`, which changes `True` into `False` and vice versa.

```
print(not 2 < 4)
print(not 2 > 4)
```

```
False
True
```

Something that can be `True` or `False` is called a **Boolean expression**. Boolean expressions are what we use in `while` loops and when setting up `if` ... `else` blocks.

Challenge 1: create a list consisting of all the integers between 2 and 100 inclusive, then create another list from which the even numbers other than 2 have been removed.

```
ints1 = list(range(2,101))
ints2 = []
for n in ints1:
    if n == 2 or not n % 2 == 0:
        ints2.append(n)
print(ints2)
```

```
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37,
39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71,
73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]
```

Challenge 2: create a third list from which the multiples of 3 other than 3 itself have been removed. Carry on until you get the primes.

```
ints3 = []
for n in ints2:
    if n == 3 or not n % 3 == 0:
        ints3.append(n)
print(ints3)
ints4 = []
for n in ints3:
    if n == 5 or not n % 5 == 0:
        ints4.append(n)
print(ints4)
```

```
ints5 = []
for n in ints4:
    if n == 7 or not n % 7 == 0:
        ints5.append(n)
print(ints5)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49, 53,
55, 59, 61, 65, 67, 71, 73, 77, 79, 83, 85, 89, 91, 95, 97]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61,
67, 71, 73, 77, 79, 83, 89, 91, 97]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
71, 73, 79, 83, 89, 97]
```

That's enough now; the next thing we'd be trying to eliminate is multiples of 11 other than 11 itself, and we've already got rid of those (22 is a multiple of 2, 33 is a multiple of 3, 44 is a multiple of 2, etc).

Note: this is not the best way to do either of these two things; the best way uses **comprehensions** (see the next subsection).

5.3 Comprehensions

You've now met the two key components of the style of computer programming known as **procedural**, namely iteration and branching. There are some programming languages that aren't procedural, and that rely instead on other constructs, but many, many languages are procedural, and have either `for` and `while` loops, and things like `if`, or something very similar. So none of that stuff is really unique to Python.

There are certain tasks, though, that, while they can certainly be done with loops, are more easily done *in Python* using a construct that most other languages don't have, called a **comprehension**. Comprehensions are appropriate when you start with an **iterable sequence**, and want to do the same thing to each of its elements, ending up with a list.

Challenge 1: create a list containing 10 copies of the string "Hello World!".

We've already done this by appending a copy of the string to an empty list ten times using a `for` loop. Here's another way of doing it, which I think you'll agree is a bit neater.

```
hw_list = ['Hello World!' for n in range(10)]
```

```
print(hw_list)
```

```
['Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!', 'Hello World!', 'Hello World!',
'Hello World!', 'Hello World!']
```

Challenge 2: create a list of all the squares of the integers between 0 and 9.

```
sq_list = [n**2 for n in range(10)]
print(sq_list)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Now let's do our summation that converges slowly to π.

Challenge 3: calculate

$$\sum_{n=0}^{100} \frac{4 \times (-1)^n}{2\,n + 1}.$$

```
terms = [(4*(-1)**n)/(2*n + 1) for n in range(101)]
print(sum(terms))
```

```
3.1514934010709914
```

This may actually not be the best way to do this; the `for` loop version we've already done has the advantage that it doesn't have to create a list of 101 terms, but only a running total.

You can set up a comprehension over any iterable sequence: that is, over anything you can do a `for` loop over.

Challenge 4: create a list containing all the character codes in the string

```
spy_string = 'My name is Bond, James Bond.'
```

```
spy_string_ords = [ord(character) for character in spy_string]
print(spy_string_ords)
```

```
[77, 121, 32, 110, 97, 109, 101, 32, 105, 115, 32, 66, 111, 110,
100, 44, 32, 74, 97, 109, 101, 115, 32, 66, 111, 110, 100, 46]
```

Note, though, that the output from a comprehension is always a list. (Actually, that's not strictly true, as you'll see later in the course; but let's for the moment act as if it is: certainly, the output from a comprehension can't be a string, or a tuple, or a `range` object, or anything like that.

5.3.1 Filtering conditions

It's possible to attach a **condition** to a comprehension, which means it only operates on, and the list at the end is only constructed from, some of the elements.

Challenge 5: create a list containing all the character codes in the string

```
spy_string = 'My name is Bond, James Bond.'
```

but only those in lower case.

```
spy_string_ords = [ord(character) for character in spy_string
                   if character==character.lower()]
print(spy_string_ords)
```

```
[121, 32, 110, 97, 109, 101, 32, 105, 115, 32, 111, 110, 100, 44, 32, 97,
109, 101, 115, 32, 111, 110, 100, 46]
```

Notice how this works; for each character, it checks whether it's equal to its own lower-case version, and if it is, sends it through to the `ord` function.

Often, all we want to do is filter.

Challenge 6: create a list containing all the lower case characters in the string

```
spy_string = 'My name is Bond, James Bond.'
```

Then make a new string out of them.

```
lower_cases = [character for character in spy_string
               if character==character.lower()]
print(lower_cases)
lc_string = ''.join(lower_cases)
print(lc_string)
```

```
['y', ' ', 'n', 'a', 'm', 'e', ' ', 'i', 's', ' ', 'o', 'n', 'd',
',', ' ', 'a', 'm', 'e', 's', ' ', 'o', 'n', 'd', '.']
y name is ond, ames ond.
```

Challenge 7: create a list consisting of all the integers between 2 and 100 inclusive, then remove all the even numbers other than 2.

```
ints = list(range(2,101))
ints = [n for n in ints if n==2 or not n % 2 == 0]
print(ints)
```

```
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37,
39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71,
73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99]
```

5.4 Functions

Python comes with a number of **functions**, such as `pow` and `ord`; modules like `math` and `cmath` contain many more. A function takes Python data as its input, and **returns** Python data as its output, as in

```
ord('M')
```

77

or

```
pow(2, 7, 7)
```

2

The next step in Python programming is *writing your own functions*.

Challenge 1: write and test a function called `absolute_difference`, which takes as its arguments two numbers `x` and `y`, and returns the value of `|x - y|`.

Notice that this challenge is different from any you've done before, in that you're not writing a one-off program that will run only once. The aim here is to *teach Python a new function*. That means that the process has at least two stages: write the function, and then use it.

The syntax for functions in Python is this:

```
def <function_name>:
    <code>
    <code>
    <code>
    return <value>
```

In this case, that looks like this:

```
def absolute_difference(x, y):
    return abs(x - y)
```

Python has now "learned" a new function, called `absolute_difference`. Let's test it:

```
absolute_difference(5, 7)
```

```
2
```

```
absolute_difference(5.3, 2.9)
```

```
2.4
```

```
absolute_difference((8 + 1j), (4 - 2j))
```

```
5.0
```

Unlike all the other programs you've written on this course, this one is available for multiple re-use. Python will forget it when you close the session, but you've kept the code, so you

can always "Shift-Return" on it again (and there are other ways to keep it, which we'll come to later in the course).

The bulk of your Python programming from now on will consist of writing and using Python functions.

Challenge 2: write and test a function called `square_list`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns a list of the squares between 0 and `(n-1)` inclusive.

```
def square_list(n):
    return [r**2 for r in range(n)]
```

Testing:

```
square_list(12)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

```
square_list(1)
```

```
[0]
```

```
square_list(0)
```

```
[]
```

It's always important to **test** your functions. But how much testing is the right amount of testing?

That's a large question. For complex software, the testing process needs to be exhaustive, painstaking and mainly automated, and that includes sophisticated Python functions; the cleverer and more intricate the code, the more there is to go wrong. This is right at the simple end, though, so all I've done is manually test one typical case, `n=12`, and two **edge cases**, `n=1` and `n=0`. The function has performed adequately on all three tests.

Challenge 3: write and test a function called `pi_sum`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns the value of

$$\sum_{r=0}^{n} \frac{4 \times (-1)^r}{2r+1}.$$

```
def pi_sum(n):
    total = 0.0
    for r in range(n+1):
        total += (4*(-1)**r)/(2*r+1)
    return total
```

A couple of things to notice here. One is the tricksy use of

`total += (4*(-1)**r)/(2*r+1)`

to mean

`total = total + (4*(-1)**r)/(2*r+1)`

This is called **augmented assignment**, and we'll make fairly free use of it from now on. (There are some subtleties associated with it, which we'll look at.) Not all languages offer augmented assignment, though many do.

The second, which is more general, concerns how we've made sure that the output from the function is our desired `total`, using the line

`return total`

Nearly every function you'll write will have a line, at or near the end, beginning with the word `return`, telling Python what it is you'd like the function to output. In this case, it was simply the final value of the variable called `total`.

All languages that allow you to write your own functions like this use a mechanism along these lines allowing you to specify the function's output, though how it works in detail can vary a lot from language to language.

Now for some testing:

```
pi_sum(100)
```

```
3.1514934010709914
```

```
pi_sum(10000)
```

```
3.1416926435905346
```

```
pi_sum(0)
```

```
4.0
```

```
pi_sum(1)
```

```
2.666666666666667
```

Here, I've tested the case `n=100`, for which I already know what the answer should be; a larger value of `n`, to check the convergence to π, and a couple of small values of `n`, which serve as edge cases and also allow me to check by hand-calculation.

The question of what tests to run is largely a matter of judgement. In assessed tasks on this course, you will be expected to run tests on any functions you write. We'll mark different decisions about testing fairly generously, but we will expect you to make sensible ones.

Challenge 4: write and test a function called `cos_nest`, which takes as its arguments a number `x0`, assumed to be a float or an int, and a second number `n`, assumed to be a non-negative integer, and returns the iterates x_0, x_1, x_2, ..., x_n of the iteration

$$x_{r+1} = \cos x_r,$$

starting with `x=x0`.

```
def cos_nest(x0, n):
    from math import cos
    x = x0
    x_list = [x]
    for r in range(n):
        x = cos(x)
        x_list.append(x)
    return x_list
```

Here, notice, what we want to be returned is the final value of `x_list`. Notice too that we've imported the `cos` function inside our function definition; that means that we don't have to remember to do that every time we use the function. It's good practice, within function code, to import anything you know you're going to need (but only that).

Testing:

```
cos_nest(1.0, 20)
```

```
[1.0,
 0.5403023058681398,
 0.8575532158463933,
 0.6542897904977792,
 0.7934803587425655,
 0.7013687736227566,
 0.7639596829006542,
 0.7221024250267077,
 0.7504177617637605,
 0.7314040424225098,
 0.7442373549005569,
 0.7356047404363473,
 0.7414250866101093,
 0.7375068905132428,
 0.7401473355678757,
 0.7383692041223232,
 0.739567202212256,
 0.7387603198742114,
 0.7393038923969057,
 0.7389377567153446,
 0.7391843997714936]
```

```
cos_nest(0.0, 0)
```

```
[0.0]
```

Certainly, the function *seems* to be working. But we need to tread slightly carefully. One very frequent error that people often make when they're new to functions is replacing the `return` line with a call to `print`, as in the following piece of code:

```
def cos_nest_with_bug(x0, n):
    from math import cos
    x = x0
    x_list = [x]
    for r in range(n):
        x = cos(x)
```

```
        x_list.append(x)
    print(x_list)
```

The behaviours of our two functions look superficially similar:

```
cos_nest(1.0, 5)
```

```
[1.0,
 0.5403023058681398,
 0.8575532158463933,
 0.6542897904977792,
 0.7934803587425655,
 0.7013687736227566]
```

```
cos_nest_with_bug(1.0, 5)
```

```
[1.0, 0.5403023058681398, 0.8575532158463933,
0.6542897904977792, 0.7934803587425655,
0.7013687736227566]
```

The problems arise if we try to perform calculations using the "output" from the second function. With the first function, for example, plotting the iterates works fine:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(21), cos_nest(1.0, 20))
```

produces an image identical to Figure **??** above. However, if we try

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(range(21), cos_nest_with_bug(1.0, 20))
```

we get a printout of our list of iterates, and then a very scary-looking error message:

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

```
ValueError                                Traceback (most recent call last)
<ipython-input-7-2bc00b1d27d2> in <module>()
      1 get_ipython().run_line_magic('matplotlib', 'qt')
      2 import matplotlib.pyplot as plt
----> 3 plt.plot(range(21), cos_nest_with_bug(1.0, 20))

C:\Anaconda3\lib\site-packages\matplotlib\pyplot.py in plot(*args, **kwargs)
   3315                      mplDeprecation)
   3316     try:
-> 3317         ret = ax.plot(*args, **kwargs)
   3318     finally:
   3319         ax._hold = washold

C:\Anaconda3\lib\site-packages\matplotlib\__init__.py in inner(ax, *args, **kwargs)
   1895                     warnings.warn(msg % (label_namer, func.__name__),
   1896                                   RuntimeWarning, stacklevel=2)
-> 1897             return func(ax, *args, **kwargs)
   1898         pre_doc = inner.__doc__
   1899         if pre_doc is None:

C:\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in plot(self, *args, **kwargs)
   1404         kwargs = cbook.normalize_kwargs(kwargs, _alias_map)
   1405
-> 1406         for line in self._get_lines(*args, **kwargs):
   1407             self.add_line(line)
   1408             lines.append(line)

C:\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _grab_next_args(self, *args, *
    405                 return
    406             if len(remaining) <= 3:
--> 407                 for seg in self._plot_args(remaining, kwargs):
    408                     yield seg
    409                 return

C:\Anaconda3\lib\site-packages\matplotlib\axes\_base.py in _plot_args(self, tup, kwargs)
    366         # downstream.
    367         if any(v is None for v in tup):
--> 368             raise ValueError("x and y must not be None")
    369
    370         kw = {}

ValueError: x and y must not be None
```

Wow. What's happened here and why?

The main lesson is that **printing is not the same as returning**. The values that a function *returns* are available for calculation; the values that it *prints* can only be read by humans like us.

Printing is what we call a **side-effect** of using the function. It's actually very useful that functions can have side-effects, and it's even useful that printing can be one of them (as you may find out when you start to **debug** your functions), but what we typically want is for a value to be *returned*. Functions should have inputs and outputs!

(Actually, like many of my sweeping generalisations, this isn't strictly true; some functions don't have inputs, and some functions don't have outputs. But those are the exceptions; for now, assume that every function you write ought to have a line that begins with the word `return`.)

Challenge 5: write and test a function called `cos_fixedpoint`, which takes as its arguments a number `x0`, assumed to be a float or an int, and a second number `tolerance`, assumed to be a positive float, and returns the iterates of the iteration

$$x_{r+1} = \cos x_r,$$

starting with `x=x0`, until successive iterates are within `tolerance` of one another.

```
def cos_fixedpoint(x0, tolerance):
    from math import cos
    oldx = x0
    newx = cos(x0)
    x_list = [oldx, newx]
    while abs(oldx-newx) > tolerance:
        oldx = newx
        newx = cos(oldx)
        x_list.append(newx)
    return x_list
```

Testing:

```
cos_fixedpoint(1.0, 0.00005)
```

```
[1.0,
 0.5403023058681398,
 0.8575532158463933,
 0.6542897904977792,
 0.7934803587425655,
```

```
 0.7013687736227566,
 0.7639596829006542,
 0.7221024250267077,
 0.7504177617637605,
 0.7314040424225098,
 0.7442373549005569,
 0.7356047404363473,
 0.7414250866101093,
 0.7375068905132428,
 0.7401473355678757,
 0.7383692041223232,
 0.739567202212256,
 0.7387603198742114,
 0.7393038923969057,
 0.7389377567153446,
 0.7391843997714936,
 0.7390182624274122,
 0.7391301765296711,
 0.7390547907469175,
 0.7391055719265361,
 0.739071365298945]
```

```
cos_fixedpoint(1.0, 0.005)
```

```
[1.0,
 0.5403023058681398,
 0.8575532158463933,
 0.6542897904977792,
 0.7934803587425655,
 0.7013687736227566,
 0.7639596829006542,
 0.7221024250267077,
 0.7504177617637605,
 0.7314040424225098,
 0.7442373549005569,
 0.7356047404363473,
 0.7414250866101093,
 0.7375068905132428]
```

```
cos_fixedpoint(0.0, 5.0)
```

```
[0.0, 1.0]
```

This function is behaving as we would wish.

Challenge 6: write and test a function called `henon_iterates`, which takes as its arguments `a` and `b`, assumed to be floats or ints, `x0` and `y0`, assumed to be floats, and `n`, assumed to be a non-negative int, and returns the iterates up to x_n, y_n of the iteration

$$\begin{aligned} x_{r+1} &= 1 - a\,{x_r}^2 + y_r, \\ y_{r+1} &= b\,x_r \end{aligned}$$

starting with `x=x0`, `y=y0`.
Your function should return a **tuple** containing a list of x-values and a list of y-values.

```
def henon_iterates(a, b, x0, y0, n):
    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)
    return (x_list, y_list)
```

Our test strategy here, I think, should be to try, first, one with a small value of `n`, which we can check by hand if we like and, secondly, one with the values we used when we generated Figure **??** above. First, then:

```
henon_iterates(2.0, 0.5, 1.0, 1.0, 5)
```

```
([1.0, 0.0, 1.5, -3.5, -22.75, -1035.875],
 [1.0, 0.5, 0.0, 0.75, -1.75, -11.375])
```

And then, for the plot:

```
xy_values = henon_iterates(1.4, 0.3, 0.5, 0.5, 10000)
plt.plot(xy_values[0], xy_values[1], '.', markersize=0.1)
```

This does indeed produce a copy of Figure **??**.

Lastly, a nice simple one that I hope you'll find useful:

Challenge 7: write and test a function called `multiple_filter`, which takes as its arguments `ints`, assumed to be a list of ints and `n`, assumed to be an int greater than 1, and returns a list containing all the elements of `ints`, with multiples of `n` other than `n` itself removed.

```
def multiple_filter(ints, n):
    return [r for r in ints if r == n or not r % n == 0]
```

Test this one yourselves!

5.5 Commenting and documenting

Here's a nice quote from a Python website:

> When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important. If you're like me, you've probably opened up old codebases and wondered to yourself, "What in the world was I thinking?" If you're having a problem reading your own code, imagine what your users or other developers are experiencing when they're trying to use or contribute to your code.

We've put this off too long! Now that our code is becoming more complex, it's going to be useful to us to add **comments** and **docstrings** to it.

5.5.1 Docstrings

A docstring is a description of what the function does. That description can be a one-liner, or something more complicated; this is a design decision. Docstrings begin and end with three double quotes, `"""`.

Challenge 1: write a version of `henon_iterates` with a one-line docstring.

```
def henon_iterates(a, b, x0, y0, n):
    """Returns x and y iterates of the Henon map."""
    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
```

```
        y_list.append(y)
    return (x_list, y_list)
```

The docstring is then available to users who want to find out about the function, as follows:

```
help(henon_iterates)
```

```
Help on function henon_iterates in module __main__:

henon_iterates(a, b, x0, y0, n)
    Returns x and y iterates of the Henon map.
```

There are "Pythonic" conventions about how a multi-line docstring should be set up, namely

```
    """
    Summary line.

    Extended description of function.

    Parameters:
    arg1 (type, if appropriate): Description of first argument
    arg2 (type, if appropriate): Description of second argument
    ...

    Returns:
    type: Description of return value

    """
```

Challenge 2: write a version of `henon_iterates` with a multi-line docstring.

```
def henon_iterates(a, b, x0, y0, n):
    """
    Returns x and y iterates of the Henon map.

    For real  parameters a and b, iterates the Henon map,
    (x, y) -> (1 - a*x**2 + y, b*x),
```

```
    n times, starting with (x, y) = (x0, y0), returning a tuple
    containing a list of x-values and a list of y-values.

    Parameters:
    a (float): A real system parameter
    b (float): A real system parameter
    x0 (float): Initial value of state variable x
    y0 (float): Initial value of state variable y
    n (int, non-negative): Number of times to iterate

    Returns:
    tuple of list of float: (x_list, y_list) where
        x_list = [x0, x1, ...],
        y_list = [y0, y1, ...]

    """

    x, y = x0, y0
    x_list, y_list = [x], [y]
    for r in range(n):
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)
    return (x_list, y_list)
```

A one-line docstring is fine for simple functions, especially if it's only you who's ever going to use them. A multi-line docstring becomes appropriate if the function is more complex, especially if you're designing for clients, or end-users other than yourself. This is largely a matter of judgement.

5.5.2 Comments

Docstrings are mostly about documenting your code for users (which of course includes yourself). They also help out **developers** (which also includes yourself), but developers often need a little more. It's useful to put into your code **comments**, which are pieces of text that Python ignores but that are readable by human beings. In Python, comment lines begin with a hashtag, #.

Challenge 3: write a version of `henon_iterates` with a one-line docstring *and comments*.

```
def henon_iterates(a, b, x0, y0, n):
    """Returns x and y iterates of the Henon map."""

    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]

    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)

    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

I hope you can see how this hugely improves the readability of your code! Let's use comments, and at least single-line docstrings, from now on.

5.5.3 Commenting out

The main use of comments is in order to improve the readability of your code, both to other developers and to yourself. But there are other reasons why it might be useful to be able to make lines in a program invisible to Python without having to delete them.

For example, suppose you've written a "cosine iteration" function (yes, yet another) which delivers not a set of iterates but a final value, and another one that you think may converge more rapidly. It can be useful to insert a `print` statement into your functions, *on a temporary basis*, so that you can view the iterates and test this hunch, as in

```
def cosine_fixedpoint1(x0, tolerance):
    """Finds solution of x = cos x by iteration."""

    # import cosine function
    from math import cos

    # initial values of oldx and newx
    oldx = x0
    newx = cos(oldx)
```

```
    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate cosine, updating oldx and newx
        oldx = newx
        newx = cos(oldx)
        print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint1(1.0, 0.0005)
```

```
0.8575532158463933
0.6542897904977792
0.7934803587425655
0.7013687736227566
0.7639596829006542
0.7221024250267077
0.7504177617637605
0.7314040424225098
0.7442373549005569
0.7356047404363473
0.7414250866101093
0.7375068905132428
0.7401473355678757
0.7383692041223232
0.739567202212256
0.7387603198742114
0.7393038923969057
0.7389377567153446

0.7389377567153446
```

As contrasted with

```
def cosine_fixedpoint2(x0, tolerance):
    """Finds solution of x = cos x by iteration."""
```

```
    # import cosine and sine functions
    from math import cos, sin

    # initial values of oldx and newx
    oldx = x0
    newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))

    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate function, updating oldx and newx
        oldx = newx
        newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))
        print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint2(1.0, 0.0005)
```

```
0.7391128909113617
0.7390851333852839

0.7390851333852839
```

So, yes, our second one is way more efficient! The thing is, now we know that, we probably don't need the `print` statements any more. But rather than deleting them (which would mean that if we did turn out to want them again, we would have to rewrite them), we could simply comment them out, as here:

```
def cosine_fixedpoint2(x0, tolerance):
    """Finds solution of x = cos x by iteration."""

    # import cosine and sine functions
    from math import cos, sin

    # initial value of oldx and newx
    oldx = x0
```

```
    newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))

    # loop until successive iterates are within tolerance...
    while abs(oldx - newx) > tolerance:
        # iterate function, updting oldx and newx
        oldx = newx
        newx = (cos(oldx) + oldx * sin(oldx))/(1 + sin(oldx))
        # print(newx)

    # return final value of newx
    return newx
```

Then

```
cosine_fixedpoint2(1.0, 0.0005)
```

```
0.7390851333852839
```

This is also quite a good **debugging strategy**: if your function is going wrong and you're not sure why, consider printing the values of key variables so you can "look under the hood" (or bonnet!) and see what's really going on. Then, as you stop seeming to need those `print` statements, act cautiously by commenting them out instead of deleting them; delete them finally only when you're sure your code is working as designed.

6 More about functions

6.1 Function arguments

6.1.1 Specifying arguments by keyword

Recall our `henon_iterates` function (this is the version with the one-line docstring).

```
def henon_iterates(a, b, x0, y0, n):
    """Returns x and y iterates of the Henon map."""

    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]
```

```
    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)

    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

In the past, we've called it with statements like

```
henon_iterates(1.4, 0.3, 0.5, 0.5, 5)
```

```
([0.5,
  1.15,
  -0.7014999999999995,
  0.656056850000001,
  0.18697517339530675,
  1.1478734533473132],
 [0.5,
  0.15,
  0.345,
  -0.21044999999999983,
  0.1968170550000003,
  0.05609255201859203])
```

This specifies the function's arguments **by position**. The disadvantage of doing things this way is that the user has to remember which argument goes where and stands for what. If they get mixed up and type

```
henon_iterates(0.5, 0.5, 1.4, 0.3, 5)
```

it'll all go horribly wrong.

Luckily, Python allows the user to specify arguments **by keyword** instead: the user could have typed:

```
henon_iterates(x0=0.5, y0=0.5, a=1.4, b=0.3, n=5)
```

or

```
henon_iterates(a=1.4, b=0.3, x0=0.5, y0=0.5, n=5)
```

or even something crazy like

```
henon_iterates(b=0.3, n=5, y0=0.5, a=1.4, x0=0.5)
```

and it would have worked just fine.

This kind of flexibility is very, very unusual, and is a distinctive thing that Python offers.

6.1.2 Default arguments

It's possible to give certain arguments **default values**, so that if the user leaves them out, Python will assume they meant those values. You've seen an example of this in the built-in **split** method, where if no substring is specified, Python assumes you mean the space character, `' '`.

If you want to equip your own function with default arguments, there's only one rule: any ordinary non-default arguments have to be listed first.

Challenge 1: write and test a function called `henon_iterates2` that uses by default the values $a = 1.4$, $b = 0.3$, but assigns no default values to any of the other arguments.

```
def henon_iterates2(x0, y0, n, a=1.4, b=0.3):
    """Returns x and y iterates of the Henon map."""

    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]

    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)

    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

Then

```
henon_iterates2(0.5, 0.5, 5, 1.4, 0.3)
```

and

```
henon_iterates2(0.5, 0.5, 5)
```

generate exactly the same output, but

```
henon_iterates2(0.5, 0.5, 5, 1.2, 0.1)
```

will use different values for a and b.

Note that default arguments can be referred to by position or keyword, just like ordinary arguments: the input

```
henon_iterates2(x0=0.5, b=0.1, y0=0.5, n=5)
```

works just fine, for example (and will use the default value for a but not for b).

6.1.3 Keyword-only arguments

Python always *allows* arguments to be referred to by keyword, but it's actually possible to *require* this.

Challenge 2: write and test a function called `henon_iterates3` that uses the keyword-only arguments a and b, with default values 1.4 and 0.3 respectively.

```
def henon_iterates3(x0, y0, n, *,  a=1.4, b=0.3):
    """Returns x and y iterates of the Henon map."""

    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]

    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
```

```
        y_list.append(y)
    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

Then this like

```
henon_iterates3(0.5, 0.5, 5)
```

and

```
henon_iterates3(0.5, 0.5, 5, b=0.1)
```

and

```
henon_iterates3(0.5, 0.5, 5, b=0.1, a=1.5)
```

and

```
henon_iterates3(b=0.1, a = 1.5, x0=0.5, y0=0.5, n=5)
```

all work fine, but Python won't allow a purely positional call like

```
henon_iterates3(0.5, 0.5, 5, 1.5, 0.1)
```

Our a and b are now strictly **keyword-only arguments**.

Note: In this example we've give our keyword-only arguments default values, but there's nothing to say we have to do that; this is a question of design. The following would be perfectly acceptable:

```
def henon_iterates3(x0, y0, n, *,  a, b):
    """Returns x and y iterates of the Henon map."""
```

```
    # initial values of the state variables x and y
    x, y = x0, y0
    # initial values of the iterate lists
    x_list, y_list = [x], [y]

    # loop n times...
    for r in range(n):
        # iterate Henon map and append new values to lists
        x, y = 1 - a*x**2 + y, b*x
        x_list.append(x)
        y_list.append(y)

    # return tuple containing lists of values of x and y
    return (x_list, y_list)
```

The user would then have to specify the values of a and b, by name, when using the function: the followuing would work:

```
henon_iterates3(0.5, 0.5, 5, a=1.4, b=0.3)
```

but the following wouldn't:

```
henon_iterates3(0.5, 0.5, 5)
```

So when you're writing a function, you have two decisions to make concerning each of the function's arguments: do I want this argument to have a default value, and do I want it to be keyword-only? Those decisons are separate from, and independent from, each other.

That said, it's *usual* for keyword-only arguments to be given default values.

6.2 "Infinite" loops

When introducing `while` loops, I gave it as a rule that the condition at the top of the loop should start off **true**, and end up **false**. But in fact, this isn't an absolutely iron rule. The reason we can relax it is that every time a function returns a value it stops; that means that we don't need to be sure that our condition will become false, as long as we're sure that we'll eventually return a value.

Challenge 1: write and test two versions of a Hailstone sequence function, one using a terminating `while` loop and the other using an infinite `while` loop that terminates through a `return`.

First the conventional implementation:

```
def hailstone_sequence1(a):
    """Returns iterates of the Hailstone sequence."""

    # initialize list of values
    alist = [a]

    # loop until value 1 is reached
    while a > 1:

        # Hailstone step
        if a % 2 == 0:
            a //= 2
        else:
            a = 3*a+1

        # append to list of values
        alist.append(a)

    # when loop is complete, return list of values
    return alist
```

Now the wacky infinite-loop one:

```
def hailstone_sequence2(a):
    """Returns iterates of the Hailstone sequence."""

    # initialize list of values
    alist = [a]

    # loop 'forever'
    while True:

        # Hailstone step
        if a % 2 == 0:
            a //= 2
        else:
            a = 3*a+1

        # append to list
        alist.append(a)
```

```
        # if value 1 has been reached, return list of values and stop
        if a==1:
            return alist
```

Then, for example,

```
hailstone_sequence1(35)
```

and

```
hailstone_sequence2(35)
```

both return

```
[35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Note the nifty way of setting up an "infinite" loop using

`while True:`

Note, too, that the loop isn't really infinite; the program always terminates because all (known) Hailstone sequences hit 1 eventually, forcing the `return`. So this is only a *potentially* infinite loop, not an *actually* infinite loop. But it should go without saying that

`while True:`

should be handled with caution, in case your potentuially infinite loop really does loop forever!

The existence of the `itertools` module even allows us to set up infinite `for` loops.

Challenge 2: write and test three versions of a function for calculating the **integer square root** of a non-negative integer a: that is, the largest integer n such that $n^2 \leq a$.

First a conventional implementation with an ordinary `while` loop:

```
def int_sqrt1(a):
    """Returns integer square root of a."""

    # initialize n
    n = 0
```

```
    # increment until n-squared > a
    while n**2 <= a:
        n+=1

    # return previous value of n
    return n-1
```

Now one using `while True`:

```
def int_sqrt2(a):
    """Returns integer square root of a."""

    # initialize n
    n = 0

    # increment 'forever'
    while n**2 <= a:
        n+=1

        # if n-squared > a, return previous value of n
        if n**2 > a:
            return n-1
```

And finally a rather neat version that uses a (potentially) infinite `for` loop:

```
def int_sqrt3(a):
    """Returns integer square root of a."""

    from itertools import count

    # loop over n = 0, 1, 2, ... 'forever'
    for n in count():

        # if n-squared > a, return previous value of n
        if n**2 > a:
            return n-1
```

All three functions will behave in the same way: give them the input 10, and they'll return 3, and give them the input 20, and they'll return 4. The last one is probably my favourite.

Note: There's a far more efficient way of implementing the integer square root, which we may look at later in the course.

6.3 Recursion

So far, when we've wanted Python to do the same thing over and over again, we've written a loop, using either `while` or `for`: this is called **iteration**. There exists an entirely different approach, called **recursion**, which works by getting a function to call itself: that is, by having the function's code include the function itself.

On the face of it, this seems paradoxical; almost crazy: if a function is defined in terms of itself, is that not a circular definition? But actually, it's not really a circle; it's more like a helix! Let me show you what I mean.

Challenge 1a: write and test an old-skool iterative function called `square_sum1` that takes as its argument an integer `n`, and returns the sum of the squares from 0 to `n`.

```
def square_sum1(n):
    """Returns the sum of the first n squares."""

    # initial value of total
    total = 0

    # loop n times...
    for r in range(1,n+1):
        # increment total by r**2
        total += r**2

    # return final total
    return total
```

Then testing:

```
square_sum1(10)
```

```
385
```

```
square_sum1(1)
```

1

```
square_sum1(0)
```

0

Challenge 1b: write and test a **recursive** function called `square_sum2` that dpes the same thing.

```
def square_sum2(n):
    """Returns the sum of the first n squares."""

    # base case
    if n==0:
        return 0

    # recursion step
    else:
        return n**2 + square_sum2(n-1)
```

Then testing:

```
square_sum2(10)
```

385

```
square_sum2(1)
```

1

```
square_sum2(0)
```

0

So, how does this work? The answer is that when we type

`square_sum2(10)`

this becomes

`10**2 + square_sum2(9)`

and

`10**2 + 9**2 + square_sum2(8)`

and so on, until we eventually get

`10**2 + 9**2 + 8**2 + ... + 1**2 + square_sum2(0)`

and `square_sum2(0)` is defined as zero! There's a downward spiral until Python hits a value it knows.

This is the basic structure of a recursive function, then:

```
def <recursive_function>(<args>):

    # base case
    if <base_condition>(<args>):
        return <base_value>

    # recursion step
    else:
        <code>
        <code>
        ...
        return <recursive_function>(<changed_args>)
```

Just as with a `while` loop you have have a condition that ends up false, with a recursion you have to have a condition that ends up true: that is, you need to be sure that the base case is eventually reached.

Challenge 2a: write and test an iterative function called `my_sqrt1` that takes as

its arguments an number `a`, a starting value `x0` and a `tolerance`, and returns an approximation to the square root of `a`, obtained by iterating

$$x_{n+1} = \frac{x_n + a}{x_n + 1}$$

until $|x^2 - a|$ is within `tolerance`.

```
def my_sqrt1(a, x0, tolerance):
    """Calculates an approximation to sqrt(a)."""

    # initialize x
    x = x0

    # while loop
    while abs(x**2 - a) > tolerance:
        # update x
        x = (x + a)/(x + 1.0)

    # return final value of x
    return x
```

Then testing:

```
my_sqrt1(5, 5, 0.000005)
```

```
2.236067059356593
```

Challenge 2b: write and test a **recursive** function called `my_sqrt2` that does the same thing.

```
def my_sqrt2(a, x0, tolerance):
    """Calculates an approximation to sqrt(a)."""

    # base case
    if abs(x0**2 - a) <= tolerance:
        return x0
```

```
    # recursion step
    else:
        # update x
        x = (x0 + a)/(x0 + 1.0)

        # return function evaluated on new value of x
        return my_sqrt2(a, x, tolerance)
```

Then testing:

```
my_sqrt2(5, 5, 0.000005)
```

```
2.236067059356593
```

Sometimes, a recursive implementation needs two functions: an inner one, implementing the actual recursion, and an outer "wrapper". A good example is when you want to return a list; somehow, the recursion needs access to the "list so far".

Challenge 3a: write a recursive function called `cos_nestlist_inner` that takes as its arguments a list of iterates `xlist` and a non-negative integer `n`, and returns an iterate list for $x_{r+1} = \cos x_r$, lengthened by a further `n` iterates.

```
def cos_nestlist_inner(xlist, n):
    from math import cos

    # base case
    if n==0:
        return xlist

    # recursion step
    else:
        # last value of x
        x = xlist[-1]

        # next value of x
        x = cos(x)

        # append to list
```

```
        xlist.append(x)

        # return using n-1
        return cos_nestlist_inner(xlist, n-1)
```

Then testing:

```
cos_nestlist_inner([0.0, 1.0], 3)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

```
cos_nestlist_inner([0.0], 4)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

Challenge 3b: hence, write a function called `cos_nestlist2` that takes as its arguments an iniitial value `x0` and a non-negative integer `n`, and returns a list of the iterates

$$x_0, x_1, x_2, \ldots, x_n.$$

```
def cos_nestlist2(x0, n):
    return cos_nestlist_inner([x0], n)
```

Then testing:

```
cos_nest2(0.0, 4)
```

```
[0.0, 1.0, 0.5403023058681398, 0.8575532158463933, 0.6542897904977792]
```

6.4 Output versus side-effects

As you've seen, Python functions and methods can do two things: return output, or have some kind of *side-effect*, such as changing the order of a list, or placing a graphic in the notebook. How do you make your own functions have side-effects?

There are at least two ways: have your function call side-effect functions or methods, or work with **global variables**. The former works especially well if the variable you want to

change is a *list* (or any other kind of *mutable* data), and if it's one of the *arguments* of your function. The latter works for anything, mutable or immutable, and is appropriate if your variable isn't an argument, but is instead assigned a value *inside* your function.

6.4.1 Changing the value of a list argument

Suppose you want to write your own function for reversing the order of a list. So far, your experience would lead you to write a function that takes a list as its input and returns a list as its output:

Challenge 1: write a function called `my_reverse1` which takes a list as its input and returns that list, reversed, as its output.

```
def my_reverse1(lis):
    """Outputs elements of lis in reverse order"""

    # initialize output list
    new_list = []

    # iterate backwards through input list, appending to output list
    for n in range(1,len(lis)+1):
        new_list.append(lis[-n])

    # return final output list
    return new_list
```

Now, if we type

```
my_list = [1, 2, 4, 1, 0]
my_reverse1(my_list)
```

the output is

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

Now, this is fine as far as it goes, but a real Python function, or method, would probably not do that; instead, it would produce no output, but would simply reorder the items in the input list.

Challenge 2: write a function called `my_reverse2` which takes a list as its input and reverses that list, returning no output.

Here's a first go. Warning: this isn't going to work.

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""

    # initialize sorted list
    new_list = []

    # iterate backwards through input list, appending to sorted list
    for n in range(1,len(lis)+1):
        new_list.append(lis[-n])

    # set input list equal to sorted list
    lis = new_list
```

This looks good, but it fails: look.

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

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

Why hasn't this worked? The point of failure is the line `lis = new_list`. We want this to reset the value of the **global** variable represented by `lis`, but a line beginning "`lis =`" never does that; instead, it assigns a value to a separate **local** variable with the same name. So our solution, whatever it is, can't involve a line beginning "`lis =`".

One approach is to empty `lis` as we go, so that by the end of the loop, it's simply the list `[]`; we can then use **augmented assignment**, which does work globally in the way we want. There's a list method called `pop` that does exactly what we need: `lis.pop(4)` outputs element number 4 from `lis`, and removes it at the same time.

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""
```

```
    # initialize sorted list
    new_list = []

    N = len(lis)

    # backwards through input list, removing elements
    # and appending to sorted list
    for n in range(1,len(lis)+1):
        new_list.append(lis.pop(N-n))

    # lis is now empty; use augmented assignment to make it
    # the same as the sorted list
    lis += new_list
```

This now works:

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

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

We can do even better, though; how about this for a neat implementation?

```
def my_reverse2(lis):
    """Reverses the order of elements in lis"""

    N = len(lis)

    # iterate backwards through input list, removing elements
    # and appending to input list
    for n in range(1,len(lis)+1):
        lis.append(lis.pop(N-n))
```

Then

```
my_list = [1, 2, 4, 1, 0]
my_reverse2(my_list)
print(my_list)
```

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

6.4.2 Global variables

By default, the variables we use within functions are **local** to those functions: they have no effect on any variables you may be using outside the function that may happen to have the same name. Nearly always, that's exactly what we want, but on occasion we may wish our function to have the **side-effect** of changing the value of a global variable. When that's the case, we need to declare it `global`.

To see how this works, compare the following two examples. Here's a function that returns a list of "Hello World!" strings, and at the same time has the utterly random effect of assigning the value 3 to the variable `x`.

```
def hw_list(n):
    x = 3
    return ["Hello World!" for r in range(n)]
```

What would you expect to see if you typed

```
x = 7
print(hw_list(5))
print(x)
```

You might think the answer would be

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
3
```

because calling the function should, as a side-effect, give `x` the value 3. But no: what you see is

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
7
```

That's because the variable called `x` inside the function isn't the same as the variable `x` outside the function: it's purely **local**, meaning that we can give it the value 3 all night; the one outside will still have the value 7.

If we want to change that, we need to explicitly declare `x` **global**:

```
def hw_list(n):
    global x
    x = 3
    return ["Hello World!" for r in range(n)]
```

Then

```
x = 7
print(hw_list(5))
print(x)
```

does indeed give us

```
["Hello World!", "Hello World!", "Hello World!",
"Hello World!", "Hello World!"]
3
```

But that's a slightly silly example. How about doing something useful with this stuff?

Challenge 3: write and test a function called `cos_and_count` which takes a list as its input a number (float or int) and returns its cosine, at the same time increasing the value of the global variable `evaluation_count` by 1.

Use your function to compare the efficiency of two implementations of a `cos_fixedpoint` function, one of which uses the condition

```
abs(x - cos(x)) > tolerance
```

and the other of which uses

```
abs(oldx - newx) > tolerance
```

First the `cos_and_count` function:

```
def cos_and_count(x):
```

```
    global evaluation_count
    from math import cos
    evaluation_count += 1
    return cos(x)
```

Then

```
evaluation_count = 0
print(cos_and_count(0.5))
print(cos_and_count(0.5))
print(evaluation_count)
```

```
0.8775825618903728
0.8775825618903728
2
```

Now the tests:

```
def cos_fixedpoint_inefficient(x0, tolerance):
    x = x0
    while abs(x - cos_and_count(x)) > tolerance:
        x = cos_and_count(x)
    return x

def cos_fixedpoint_efficient(x0, tolerance):
    oldx = x0
    newx = cos_and_count(oldx)
    while abs(oldx - newx) > tolerance:
        oldx = newx
        newx = cos_and_count(oldx)
    return newx
```

Then

```
evaluation_count = 0
print(cos_fixedpoint_inefficient(1.0, 0.00001))
print(evaluation_count)
```

```
0.7390893414033927
57
```

```
evaluation_count = 0
print(cos_fixedpoint_efficient(1.0, 0.00001))
print(evaluation_count)
```

```
0.7390822985224023
29
```

The efficient version is indeed more efficient!

MATH40006: An Introduction To Computation
Course notes, Volume 3

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

---

7 More about modules

7.1 The `numpy` module

You've already met the NumPy module, but it's time we took a deeper dive into it. NumPy is, of course, short for Numerical Python, and is the key module for scientific computing in Python. So useful is it that we'll often want to import it in its entirety, and there's a convention that says that we do that like this:

```
import numpy as np
```

We can then put the shortened prefix `np`, instead of `numpy`, in front of any functions or constants we use.

7.1.1 Arrays

At the heart of NumPy is the specialised data structure called an **array**. The simplest way to make one is from a list or tuple, using the constructor function `array`:

```
arr1 = np.array([1, 4, 7])
arr2 = np.array((5, -6, 2])
```

Arrays look superficially like lists or tuples, but they behave very differently, as you'll see. Two major differences are

- a list or tuple can contain a variety of different types of data, whereas an array must consist of data of just one type;
- arrays behave radically differently from lists or tuples when added, or when multiplied by a scalar.

To see the former property, try typing

```
mixed_list = [1, 2.0, 3, 4.0]
mixed_array = np.array(mixed_list)
print(mixed_list)
print(mixed_array)
```

```
[1, 2.0, 3, 4.0]
[1. 2. 3. 4.]
```

Notice that the elements of `mixed_array` aren't mixed at all; they're all floats! (Notice, too, in passing, that the `print` function displays the elements of an array without separating commas.)

To see the second property, contrast

```
[1, 4, 7] + [5, -6, 2]
```

```
[1, 4, 7, 5, -6, 2]
```

and

```
np.array([1, 4, 7]) + np.array([5, -6, 2])
```

```
array([6, -2, 9])
```

Contrast, too,

```
3 * [1, 4, 7]
```

```
[1, 4, 7, 1, 4, 7]
```

and

```
3 * np.array([1, 4, 7])
```

```
array([3, 12, 21])
```

NumPy arrays don't concatenate like lists and tuples; instead, addition, or scalar multiplication, works as if the arrays were **vectors**. (This is no accident, as we'll see.)

If you base your array on a **list of lists**, it will end up **two-dimensional**, like a matrix (indeed, *very* like a matrix, as we'll explore later):

```
array2d = np.array([[1, 3, 5],[2, 4, 6], [1,-1,1]])
print(array2d)
```

```
[[ 1  3  5]
 [ 2  4  6]
 [ 1 -1  1]]
```

These behave in a similar way to 1D arrays when added, or multiplied by a scalar:

```
print(array2d + array2d)
```

```
[[ 2  6 10]
 [ 4  8 12]
 [ 2 -2  2]]
```

```
print(3.1 * array2d)
```

```
[[ 3.1  9.3 15.5]
 [ 6.2 12.4 18.6]
 [ 3.1 -3.1  3.1]]
```

It's even possible to create 3-dimensional, or 4-dimensional, or 5-dimensional arrays, or whatever, via lists of lists of lists of . . . :

```
array3d = np.array([[[1, 3, 5],[2, 4, 6], [1,-1,1]],
        [[6, 0, -1],[0,1, -6], [1, 2, 1]]])
print(array3d)
```

```
[[[ 1  3  5]
  [ 2  4  6]
  [ 1 -1  1]]

 [[ 6  0 -1]
  [ 0  1 -6]
  [ 1  2  1]]]
```

7.1.2 The `linspace`, `arange`, `zeros` and `ones` functions

There are other ways of creating NumPy arrays. It's often useful to have an array all of whose elements are equally spaced, such as values between 0 and $2\,\pi$ in steps of $\pi/6$. To create a *list* containing those values, we'd have to use a loop or, better, a comprehension:

```
from math import pi
x_list = [i * pi/6 for i in range(13)]
print(x_list)
```

```
[0.0, 0.5235987755982988, 1.0471975511965976, 1.5707963267948966,
2.0943951023931953, 2.6179938779914944, 3.141592653589793,
3.665191429188092, 4.1887902047863905, 4.71238898038469,
5.235987755982989, 5.759586531581287, 6.283185307179586]
```

But with NumPy arrays, it's far easier. We can either use the `linspace` function, which allows us to specify the first value, the last value and the number of values ...

```
x_arr = np.linspace(0, 2*np.pi, 13)
print(x_arr)
```

```
[0.         0.52359878 1.04719755 1.57079633 2.0943951  2.61799388
 3.14159265 3.66519143 4.1887902  4.71238898 5.23598776 5.75958653
 6.28318531]
```

... or the `arange` function, which allows us to set up a range of values—a little like the `range` function, except that the step size is allowed to be a float.

```
x_arr2 = np.arange(0, 2*np.pi+0.01, np.pi/6)
print(x_arr2)
```

```
[0.         0.52359878 1.04719755 1.57079633 2.0943951  2.61799388
 3.14159265 3.66519143 4.1887902  4.71238898 5.23598776 5.75958653
 6.28318531]
```

Notice that `arange` obeys the standard Python system: it delivers those numbers, in steps of $\pi/6$, that are greater than or equal to zero but **strictly less than** $2\,\pi + 0.01$. For this reason,

```
x_arr2 = np.arange(0, 2*np.pi, np.pi/6)
print(x_arr2)
```

wouldn't have worked.

It may not be immediately obvious why, but it turns out to be really useful to be able to create NumPy arrays consisting entirely of 0s or 1s.

```
print(np.zeros(5))
print(np.ones(5))
print(np.zeros([3,4]))
print(np.ones([2,6]))
```

```
[0. 0. 0. 0. 0.]
[1. 1. 1. 1. 1.]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[[1. 1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1. 1.]]
```

7.1.3 Array operators

We've already seen that if two arrays are the same shape, it's possible to add them, and they'll add component-by-component. Well, it turns out it's also possible to use all the other arithmetic operators on arrays as well: `-`, `*`, `/`, `//` and `**`.

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
arr2 = np.array([[5, 3, 2], [1, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
[[6 5 7]
 [1 7 5]]
[[-4 -1  3]
 [-1  1  1]]
```

```
[[ 5  6 10]
 [ 0 12  6]]
[[0.2        0.66666667 2.5       ]
 [0.         1.33333333 1.5       ]]
[[0 0 2]
 [0 1 1]]
[[ 1  8 25]
 [ 0 64  9]]
```

Again, the operations are all carried out component-by-component.

It's also possible to use all the arithmetical operations on a NumPy array and a scalar:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
print(arr1 + 2)
print(arr1 - 2)
print(arr1 * 2)
print(arr1 / 2)
print(arr1 // 2)
print(arr1 ** 2)
```

```
[[3 4 7]
 [2 6 5]]
[[-1  0  3]
 [-2  2  1]]
[[ 2  4 10]
 [ 0  8  6]]
[[0.5 1.  2.5]
 [0.  2.  1.5]]
[[0 1 2]
 [0 2 1]]
[[ 1  4 25]
 [ 0 16  9]]
```

So: you can add (or multiply, or subtract, etc) two arrays of exactly the same shape, and you can add (or multiply, or subtract, etc) an array and a scalar. Can you do anything else? Well, yes, actually. The following works, for example:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3]])
```

```
arr2 = np.array([[5, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
[[6 5 7]
 [5 7 5]]
[[-4 -1  3]
 [-5  1  1]]
[[ 5  6 10]
 [ 0 12  6]]
[[0.2        0.66666667 2.5       ]
 [0.         1.33333333 1.5       ]]
[[0 0 2]
 [0 1 1]]
[[ 1  8 25]
 [ 0 64  9]]
```

On the other hand, the following doesn't:

```
arr1 = np.array([[1, 2, 5], [0, 4, 3], [-2, 2, -3]])
arr2 = np.array([[5, 3, 2], [1, 3, 2]])
print(arr1 + arr2)
print(arr1 - arr2)
print(arr1 * arr2)
print(arr1 / arr2)
print(arr1 // arr2)
print(arr1 ** arr2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-20-b161a505ab0d> in <module>()
      1 arr1 = np.array([[1, 2, 5], [0, 4, 3], [-2, 2, -3]])
      2 arr2 = np.array([[5, 3, 2], [1, 3, 2]])
----> 3 print(arr1 + arr2)
      4 print(arr1 - arr2)
      5 print(arr1 * arr2)

ValueError: operands could not be broadcast together with shapes (3,3) (2,3)
```

More about what exactly the rules are, what exactly happens, and what exactly Python means by "broadcast", in the exercises.

7.1.4 Mathematical functions

The NumPy module comes with a complete set of mathematical functions, largely paralleling those in the `math` and `cmath` modules:

```
print(np.sin(np.pi/6))
print(np.cos(np.pi/6))
print(np.exp(np.log(2)))
```

```
0.49999999999999994
0.8660254037844387
2.0
```

However, there's a difference, which is that NumPy's functions map automatically across arrays:

```
angles = np.arange(0,np.pi/2+0.01,np.pi/6)
sines = np.sin(angles)
print(sines)
```

```
[0.        0.5       0.8660254 1.       ]
```

This is fantastically useful. Consider, for example, the problem of plotting $\cos x$ against x. Here's how we'd have to do that without NumPy.

```
from math import pi, cos
x_values = [i * pi/200 for i in range(201)]
y_values = [cos(x) for x in x_values]
plt.plot(x_values, y_values)
```

With NumPy, this is simply

```
import numpy as np
x_values = np.linspace(0, 2*np.pi, 201)
y_values = np.cos(x_values)
plt.plot(x_values, y_values)
```

No need to use comprehensions at all! I think this is simpler and easier. And, as we'll see later, this "vectorized" way of working can also be fundamentally much more *efficient* than using comprehensions or loops, meaning that NumPy can offer strategies for speeding up certain programs.

7.1.5 Arrays as matrices and vectors

NumPy supports a comprehensive selection of linear algebra functions and methods. 1D arrays can be used to represent vectors ...

```
vec1 = np.array([-1,2,2])
vec2 = np.array([2,-1,2])
print(np.dot(vec1, vec2))
print(np.cross(vec1, vec2))
```

```
0
[ 6  6 -3]
```

... and 2D arrays can ve used to represent matrices ...

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
mat2 = np.array([[1, 3], [-1, 0], [2, -1]])
print(np.dot(mat1, mat2))
```

```
[[-5  8]
 [ 6  3]
 [ 1 -8]]
```

Notice that the same function, `dot`, is used for the scalar product of two vectors and for matrix multiplication.

7.1.6 The `linalg` submodule

The basic linear algebra functions `dot` and `cross` live in the top level of NumPy; for anything even a bit more specialised, you need the `linalg` submodule. First let's import it; then we can use it to calculate the determinant, and the inverse, of our square matrix `mat1`.

```
import numpy.linalg
print(np.linalg.det(mat1))
print(np.linalg.inv(mat1))
```

```
-8.000000000000002
[[ 1.25   1.     1.25 ]
 [ 0.25  -0.     0.25 ]
 [ 1.125  1.     1.625]]
```

7.1.7 Polynomials

NumPy has a collection of functions for dealing with the algebra of polynomials, which are represented by a special kind of array called `poly1d`. These arrays consist of the coefficients in the polynomial, in descending power order. Here, for example, are the polynomials $x^2-3\,x+2$ and $x^3 - 2\,x^2 - 5\,x + 6$:

```
poly1 = np.poly1d([1, -3, 2])
poly2 = np.poly1d([1, -2, -5, 6])
```

We can then add or multiply...

```
print(poly1 + poly2)
print(poly1 * poly2)
```

```
   3     2
1 x - 1 x - 8 x + 8
   5     4     3      2
1 x - 5 x + 3 x + 17 x - 28 x + 12
```

...calculate roots, and subtsitute in values...

```
print(np.roots(poly2))
print(np.polyval(poly1, [1, 3, 5, 7]))
```

```
[-2.  3.  1.]
[ 0  2 12 30]
```

...differentiate and integrate ...

```
print(np.polyder(poly2))
print(np.polyint(poly1))
```

```
   2
3 x - 4 x - 5
        3       2
0.3333 x - 1.5 x + 2 x
```

. . . and so on..

7.2 Programming using NumPy

7.2.1 Indexing, slicing, appending and changing

1D NumPy arrays use the same indexing and slicing conventions as lists and tuples:

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
print(arr3[3])
print(arr3[2:5])
```

```
1
[2 1 9]
```

Multi-dimensional arrays can be indexed using a comma notation:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
print(mat1[0,1])
print(mat1[2,0:2])
print(mat1[:,0])
```

```
3
[-2  1]
[ 2  1 -2]
```

(Note the last example: the zeroth column of `mat1` is returned, as a 1D vector.)

Arrays are **mutable**, like lists, as opposed to being **immutable**, like tuples. You can change individual elements:

```
arr3[2] = 100
print(arr3)
```

```
[  3   5 100   1   9   4]
```

It's also possible to append elements to arrays, though this works differently from how it works with lists:

```
print(np.append(arr3, 77))
print(arr3)
```

```
[  3   5 100   1   9   4  77]
[  3   5 100   1   9   4]
```

Notice those differences. We use a *function* caled `append` from the NumPy module, rather than the *method* called append from core Python; it's not

`arr3.append(77)`

This function *returns as output* the list with the additional element appended; it doesn't have the side effect of changing the value of `arr3`. If we wanted to do that, we'd have to reassign the new value to the variable `arr3`, as follows:

```
arr3 = np.append(arr3, 77)
print(arr3)
```

```
[  3   5 100   1   9   4  77]
```

7.2.2 Iterating over arrays

Just like lists and tuples, arrays can be iterated over, in `for` loops ...

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
for n in arr3:
    print('n = {}; 2n+1 = {}'.format(n,2*n+1))
```

```
n = 3; 2n+1 = 7
n = 5; 2n+1 = 11
n = 2; 2n+1 = 5
n = 1; 2n+1 = 3
n = 9; 2n+1 = 19
n = 4; 2n+1 = 9
```

. . . and in comprehensions . . .

```
arr3 = np.array([3, 5, 2, 1, 9, 4])
print([(n, 2*n+1) for n in arr3])
```

```
[(3, 7), (5, 11), (2, 5), (1, 3), (9, 19), (4, 9)]
```

We can even iterate across multidimensional arrays, using a NumPy function called `nditer`:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
for n in np.nditer(mat1):
    print('n = {}; 2n+1 = {}'.format(n,2*n+1))
```

```
n = 2; 2n+1 = 5
n = 3; 2n+1 = 7
n = -2; 2n+1 = -3
n = 1; 2n+1 = 3
n = -5; 2n+1 = -9
n = 0; 2n+1 = 1
n = -2; 2n+1 = -3
n = 1; 2n+1 = 3
n = 2; 2n+1 = 5
```

Note that in this case, Python has iterated down the rows one by one. In fact, that's not absolutely reliable; and in any case, you might want to iterate column by column. There's a way of taking charge of the iteration order, and I invite you to explore that in the exercises.

Finally, its even possible to use `nditer` to iterate across two arrays at the same time:

```
mat1 = np.array([[2, 3, -2], [1, -5, 0], [-2, 1, 2]])
mat3 = np.array([[6, 6, 6], [11, -4, 7], [7, 5, 9]])
for m, n in np.nditer((mat3, mat1)):
    if m % (n+1) == 0:
        print('{} is a multiple of ({}+1)'.format(m, n))
    else:
        print('{} is not a multiple of ({}+1)'.format(m, n))
```

```
6 is a multiple of 2+1
6 is not a multiple of 3+1
6 is a multiple of -2+1
11 is not a multiple of 1+1
-4 is a multiple of -5+1
7 is a multiple of 0+1
7 is a multiple of -2+1
5 is not a multiple of 1+1
9 is a multiple of 2+1
```

7.2.3 Programmatic generation of arrays

There's a NumPy function called `fromfunction` that allows you to generate an n-dimensional array of floats using a function of n variables. The function will take as its arguments the row and column indexes. In the following example, for instance, each entry of the matrix is equal to twice the row index plus the column index (as a float):

```
def twoxplusy(x, y):
    return 2*x+y
np.fromfunction(twoxplusy, (4, 3))
```

```
array([[0., 1., 2.],
       [2., 3., 4.],
       [4., 5., 6.],
       [6., 7., 8.]])
```

Actually, the easiest way to do that kind of thing is probably to use a **lambda-expression**, which is a way of referring to a function not via its name but via a description of what it does:

```
np.fromfunction(lambda x, y: 2*x+y, (4, 3))
```

```
array([[0., 1., 2.],
       [2., 3., 4.],
       [4., 5., 6.],
       [6., 7., 8.]])
```

In this, you read

```
lambda x, y: 2*x+y
```

as "the function whose arguments are x and y and whose outputs are $2\,x + y$". Lambda-expressions are incredibly useful, and we'll meet them more and more.

7.2.4 Vectorizing your algorithms

NumPy arrays are nice things: you can use them to represent banks of data, or vectors and matrices, or polynomials. But the biggest impact of NumPy on your life as a Python user may well be the way it allows you to write much more efficient code. This makes use of the fact that we can do things in one go using NumPy arrays that in core Python would require a comprehension or a loop.

Challenge 1: create a one-dimensional data structure containing all the squares of the integers between 0 and 9.

If we wanted to do this using only core Python data structures and functions, we'd either use a loop . . .

```
squares = []
for n in range(10):
    squares.append(n**2)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

. . . or a comprehension . . .

```
print([n**2 for n in range(10)])
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Here's how we could do it using NumPy arrays:

```
print(np.arange(10)**2)
```

```
[ 0  1  4  9 16 25 36 49 64 81]
```

If we want to square every element of a NumPy array, we can simply square the array. This is arguably neater (though this may be a matter of preference); what's undeniable is that for long programming tasks this **vectorized** approach tends to be much, much more efficient.

Challenge 2: write and test a function called `pi_sum`, which takes as its argument a number `n`, assumed to be a non-negative integer, and returns the value of

$$\sum_{r=0}^{n} \frac{4 \times (-1)^r}{2r+1}.$$

Here's how we did that using a loop:

```
def pi_sum1(n):
    total = 0.0
    for r in range(n+1):
        total += (4*(-1)**r)/(2*r+1)
    return total
```

Here's how we might do it using NumPy arrays:

```
def pi_sum2(n):
    from numpy import arange
    r_array = arange(n+1)
    return sum((4*(-1)**r_array)/(2*r_array+1))
```

The latter executes quite a lot more quickly for large n, despite the fact that it creates a transparent data structure whereas the first version uses an opaque one. In the following tests, with n equal to ten million, we use the `clock` function from the `time` module to time the operations; the time in seconds is the second printed value.

```
from time import clock
start = clock()
print(pi_sum1(10000000))
print(clock()-start)
```

```
3.1415927535897814
6.411463699904994
```

```
from time import clock
start = clock()
print(pi_sum2(10000000))
print(clock()-start)
```

```
3.1415927535897814
1.8668125593228129
```

Challenge 3: write and test a function called `trapezium_rule`, which takes as its arguments a function `func`, floats (or ints) `xmin` and `xmax` and a positive int `n`, and returns the trapezium rule estimate for the integral of `func` with respect to x between `xmin` and `xmax`, using `n` subintervals.

The formula will be familiar to you all:

$$\int_a^b y\,dx \approx \frac{h}{2}\left(y_0 + 2\,y_1 + 2\,y_2 + \cdots + 2\,y_{n-1} + y_n\right),$$

where h is the interval width.

Here's an implementation based on loops:

```
def trapezium_rule1(func, xmin, xmax, n):
    """
    Returns the trapezium rule approximation to the integral of func
    between xmin and xmax; uses n intervals
    """

    # interval width
    h = (xmax - xmin)/n

    # init total
    total = func(xmin) + func(xmax)

    # loop
    for r in range(1, n):
        total += 2*func(xmin + r*h)

    # return
    return h/2 * total
```

Here's a vectorized one based on NumPy arrays; the key difference is that if `func` is the right kind of function, we can apply it to all the elements in an array in one go:

```
def trapezium_rule2(func, xmin, xmax, n):
    """
    Returns the trapezium rule approximation to the integral of func
    between xmin and xmax; uses n intervals
    """
    from numpy import array, arange

    # interval width
    h = (xmax - xmin)/n

    # end and middle values of x
    ends = array([xmin, xmax])
    mids = arange(xmin+h,xmax,h)

    # return
    return h/2 * (sum(func(ends)) + 2*sum(func(mids)))
```

Then the following both work:

```
print(trapezium_rule1(lambda x: x**5 - x**4, 0, 1, 10))
print(trapezium_rule2(lambda x: x**5 - x**4, 0, 1, 10))
```

```
-0.032505
-0.032505
```

The following also both work

```
import numpy as np
print(trapezium_rule1(np.sin, 0, np.pi/2, 10))
print(trapezium_rule2(np.sin, 0, np.pi/2, 10))
```

```
0.9979429863543572
0.9979429863543572
```

However, only `trapezium_rule1` works in the following case:

```
import math
print(trapezium_rule1(math.sin, 0, math.pi/2, 10))
print(trapezium_rule2(math.sin, 0, math.pi/2, 10))
```

```
0.9979429863543572
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-d1d64aa73f60> in <module>()
      2 import math
      3 print(trapezium_rule1(math.sin, 0, math.pi/2, 10))
----> 4 print(trapezium_rule2(math.sin, 0, math.pi/2, 10))

<ipython-input-42-21a208578962> in trapezium_rule2(func, xmin, xmax, n)
     14
     15     # return
---> 16     return h/2 * (sum(func(ends)) + 2*sum(func(mids)))

TypeError: only size-1 arrays can be converted to Python scalars
```

The error arises because the second implementation tries to apply `func` to and array, and if `func` is a `math` function that won't work.

The second implementation, though, works much more quickly for large values of n; try it!

So, what kinds of tasks can be vectorized in this way? Certainly not everything. There's no vectorized implementation of "Iterate e^{-x} ten times starting at x=0", for example, because that's a task that is in its very nature iterative: each value of x depends on the previous one. Vectorization only works when you want to do the same thing to every element of a data structure: square it, or substitute it into the expression

$$\frac{4 \times (-1)^n}{2\,n + 1},$$

or use it as the argument of some `func`.

Each stage in the task, in other words, can be done independently of the others, in any order; it's a little like the set of circumstances in which you can **parallelize** an algorithm. That's not a bad way to think about vectorization, in fact; as a kind of quasi-parallelization. In fact, vectorized algorithms don't make much use of parallelization, but they're the same sort of thing fundamentally; the subtasks can happen "all in one go" instead of having to be thought of sequentially.

7.2.5 Plotting using arrays

The various plotting functions in `matplotlib` all work just as well with NumPy arrays as with lists, tuples or `range` objects. The ease and efficiency with which arrays can be created

using vectorized approaches can make this substantially preferable.

Challenge 4: create a vectorized, array-based implementation of the `plot_trochoid` function from lectures.

Here's a listing of the original implementation of that function, this time with a docstring and comments in the proper way:

```
def plot_trochoid(outer_radius=11, inner_radius=5, rho=5, *,
                  npoints=200, show_circles=False):
    """
    Draws the trochoid curve corresponding to a certain set of
    parameters
    """
    from math import sin, cos, pi
    import matplotlib.pyplot as plt

    # theta from 0 to 2*pi*inner_radius
    theta_values = [inner_radius*2*r*pi/(npoints-1)
                    for r in range(npoints)]

    # difference between radii
    rdiff = (outer_radius-inner_radius)

    # x- and y-values for curve
    x_values = [rdiff*cos(theta) + rho*cos(rdiff*theta/inner_radius)
                for theta in theta_values]
    y_values = [rdiff*sin(theta) - rho*sin(rdiff*theta/inner_radius)
                for theta in theta_values]

    # size plot and set axes
    plt.figure(figsize = (7,7))
    plt.axes().set_aspect(aspect='equal')

    # plot
    plt.plot(x_values, y_values, 'r')

    # show outer and inner circles if flag is set
    if show_circles:
        # outer circle
        theta_values2 = [2*r*pi/(192) for r in range(193)]
        x_values2 = [outer_radius*cos(theta)
```

```
                    for theta in theta_values2]
        y_values2 = [outer_radius*sin(theta)
                    for theta in theta_values2]
        plt.plot(x_values2, y_values2, 'k')
        # inner circle
        x_values3 = [rdiff+inner_radius*cos(theta)
                    for theta in theta_values2]
        y_values3 = [inner_radius*sin(theta)
                    for theta in theta_values2]
        plt.plot(x_values3, y_values3, 'b')
        # pen position
        plt.plot([rdiff+rho],[0],'g.',markersize=30)
```

Here's the implementation based on arrays:

```
def plot_trochoid2(outer_radius=11, inner_radius=5, rho=5, *,
                   npoints=200, show_circles=False):
    """
    Draws the
trochoid curve corresponding to a certain set of
    parameters
    """
    from numpy import sin, cos, pi, linspace
    import matplotlib.pyplot as plt

    # theta from 0 to 2*pi*inner_radius
    theta_values = linspace(0, 2*pi*inner_radius, npoints)

    # difference between radii
    rdiff = (outer_radius-inner_radius)

    # x- and y-values for curve
    x_values = (rdiff*cos(theta_values) +
                rho*cos(rdiff*theta_values/inner_radius))
    y_values = (rdiff*sin(theta_values) -
                rho*sin(rdiff*theta_values/inner_radius))

    # size plot and set axes
    plt.figure(figsize = (7,7))
    plt.axes().set_aspect(aspect='equal')

    # plot
```

```
    plt.plot(x_values, y_values, 'r')

    # show outer and inner circles if flag is set
    if show_circles:
        # outer circle
        theta_values2 = linspace(0, 2*pi, 192)
        x_values2 = outer_radius*cos(theta_values2)
        y_values2 = outer_radius*sin(theta_values2)
        plt.plot(x_values2, y_values2, 'k')
        # inner circle
        x_values3 = rdiff+inner_radius*cos(theta_values2)
        y_values3 = inner_radius*sin(theta_values2)
        plt.plot(x_values3, y_values3, 'b')
        # pen position
        plt.plot([rdiff+rho],[0],'g.',markersize=30)
```

Both produce the same kinds of diagram; for example

```
%matplotlib inline
plot_trochoid2(rho=3, show_circles=True)
```

produces a diagram as in Figure 1.

However, I think the code for the second implementation is easier to write and to read, and it will also be quicker to execute (though execution time isn't a major problem here).

Time for one last vectorization programing challenge. The **logistic map**,

$$f_k(x) = k\,x\,(1-x),$$

exhibits lots of different types of behaviour for different values of the parameter k. One way to get a picture of those behaviours, and how they vary with k, is to create a **bifurcation diagram**. To do that, you choose a value of k, iterate the map m times, then retain the next n iterates (several hundred in each case for the best effect). This creates $(n+1)$ coordinate pairs, the horizontal coordinates being k in each case, and the vertical coordinates the iterates $x_m, x_{m+1}, x_{m+1}, \ldots, x_{m+n}$. You then do this for a different value of k, and so on.

Challenge 5: write and test a function called `bifurcation_diagram`. It should take as its arguments:

- a positive int `resolution`;

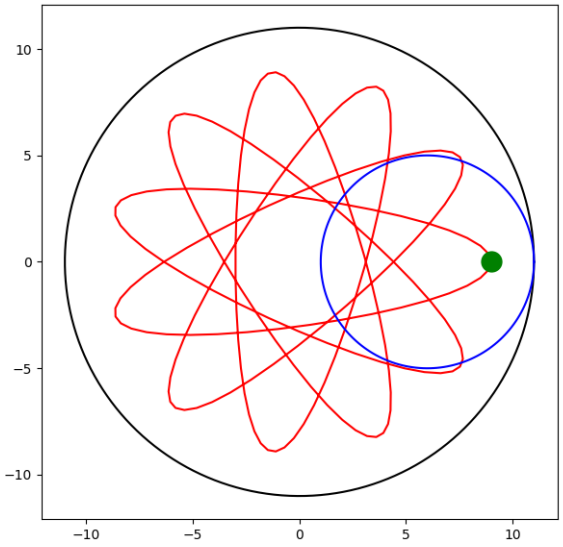


Figure 1: A trochoid curve

- a positive int `nskip`;
- a positive int `niterate`;
- a keyword-only argument, `mksize`, by default equal to 0.5.

It should then create a bifurcation diagram. The parameter k should range from 1 to 4, with a step size equal to the reciprocal of the resolution. The map should first be iterated `nskip` times, with the iterates discarded, and then `niterate` times, with the iterates retained. The bifurcation diagram should then take the form of a point plot, with the `markersize` option set to `mksize`. No data should be returned.

Here's an implementation based on lists, which uses comprehensions:

```
def logistic_map(k, x):
    return k * x * (1 - x)

def bifurcation_diagram1(resolution, nskip, niterate, *, mksize=0.5):
    """
    Plots a bifurcation diagram for the logistic map.
    """
```

```
    # step size
    h = 1/resolution

    # values of k
    k_list = [h * r for r in range(resolution,4*resolution)]

    # initial values of x, 0.3 throughout
    x_list = [0.3 for r in range(3*resolution)]

    # iterate and skip
    for i in range(nskip):
        # update x_list
        x_list = [logistic_map(k_list[j], x_list[j])
                  for j in range(3*resolution)]

    # initial lists of k- and x-values
    k_values = k_list
    x_values = x_list

    # iterate and retain
    for i in range(niterate):
        # update x_list
        x_list = [logistic_map(k_list[j], x_list[j])
                  for j in range(3*resolution)]
        # update k_values and x_values
        k_values = k_values + k_list
        x_values = x_values + x_list

    # plot
    import matplotlib.pyplot as plt
    plt.figure(figsize=(10,7))
    plt.plot(k_values,x_values, '.k', markersize=mksize)
```

Here's one based on arrays:

```
def logistic_map(k, x):
    return k * x * (1 - x)

def bifurcation_diagram2(resolution, nskip, niterate, *, mksize=0.5):
    """
    Plots a bifurcation diagram for the logistic map.
    """
```

```
    from numpy import linspace, ones, concatenate

    # values of k
    k_array = linspace(1, 4, 3*resolution+1)

    # initial values of x, 0.3 throughout
    x_array = 0.3 * ones(3*resolution+1)

    # iterate and skip
    for i in range(nskip):
        # update x_array
        x_array = logistic_map(k_array, x_array)

    # initial arrays of k- and x-values
    k_values = k_array
    x_values = x_array

    # for loop
    for i in range(niterate):
        # update x_array
        x_array = logistic_map(k_array, x_array)
        # update k_values and x_values
        k_values = concatenate((k_values, k_array))
        x_values = concatenate((x_values, x_array))

    # plot
    import matplotlib.pyplot as plt
    plt.figure(figsize=(10,7))
    plt.plot(k_values,x_values, '.k', markersize=mksize)
```

Note the use of NumPy's `concatenate` function, as we can't concatenate using the + operator. Notice too how easy it is to update an array of x-values:

```
x_array = logistic_map(k_array, x_array)
```

The contrast with the first implementation is especially clear here, I think.

What's more, if we type

```
from time import clock
start = clock()
bifurcation_diagram1(700, 300, 700, mksize=0.01)
print(clock() - start)
```

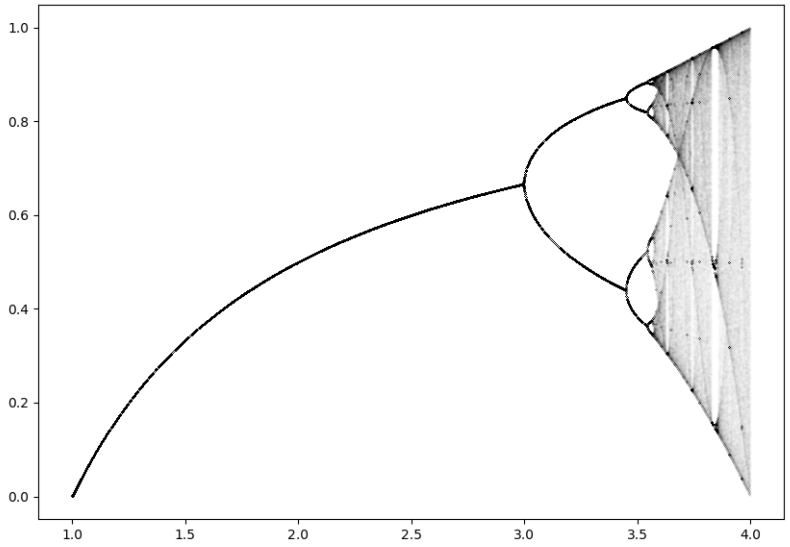


Figure 2: Bifurcation diagram for the logistic map

we get

```
9.604787682899456
```

and if we type

```
from time import clock
start = clock()
bifurcation_diagram2(700, 300, 700, mksize=0.01)
print(clock() - start)
```

we get

```
3.535992631587874
```

In both cases, the diagram looks like figure 2, but the execution time is much better in the vectorized version.

7.2.6 Boolean arrays

We saw in lectures that if you type

```
arr2d1 = np.array([[1, 0, 1], [0, 1, 0], [0, 0, 1]])
arr2d2 = np.array([[2, 1, -1], [-3, 4, 2], [1, 1, -4]])
print(arr2d1 < arr2d2)
```

we get

```
[[ True  True False]
 [False  True  True]
 [ True  True False]]
```

This is what's called a **Boolean array**, and they're very useful things.

Challenge 6: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are factors of 24.

```
ints = np.arange(1,25)
24 % ints == 0
```

```
array([ True,  True,  True,  True, False,  True, False,  True, False,
       False, False,  True, False, False, False, False, False, False,
       False, False, False, False, False,  True])
```

The operators for Boolean *expressions* are, of course, `not`, `and` and `or`. The corresponding operators for Boolean *arrays* are `~`, `&` and `|`.

Challenge 7: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are not factors of 24.

```
ints = np.arange(1,25)
~(24 % ints == 0)
```

```
array([False, False, False, False,  True, False,  True, False,  True,
        True,  True, False,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False])
```

Challenge 8: create a Boolean array corresponding to those integers between 1 and 24 inclusive that are *odd* factors of 24.

```
ints = np.arange(1,25)
(24 % ints == 0) & (ints % 2 > 0)
```

```
array([ True, False,  True, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False])
```

What's really lovely is that you can use Boolean arrays in **indexing**; this offers a neat (and efficient) NumPy alternative to the use of comprehensions for filtering.

Challenge 9: create an array consisting of those integers between 1 and 24 inclusive that are factors of 24.

```
ints = np.arange(1,25)
ints[24 % ints == 0]
```

```
array([ 1,  2,  3,  4,  6,  8, 12, 24])
```

Challenge 10: create an array consisting of those integers between 1 and 24 inclusive that are odd factors of 24.

```
ints = np.arange(1,25)
ints[(24 % ints == 0) & (ints % 2 > 0)]
```

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

By contrast, here's a reminder of how we'd do those tasks with lists and comprehensions:

```
ints = list(range(1, 25))
[n for n in ints if 24 % n == 0]
```

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

```
ints = list(range(1, 25))
[n for n in ints if 24 % n == 0 and n % 2 > 0]
```

```
[1, 3]
```

These Boolean-array indexes look weird when you first meet them, but they're very nice to work with. In the exercises, you use this to create a NumPy implementation of the Sieve of Eratosthenes for calculating lists of primes.

7.3 The `random` module and NumPy's `random` submodule

There are two significant resources for statistical tasks (in the broad sense) in Python: the "Core Python" module `random`, adnd the submodule of the same name that forms part of `numpy`.

The `random` module has one basic function, also called `random`, which takes no arguments, and returns a randomly chosen float (a Core Python float) between 0 and 1:

```
import random as rnd
print(rnd.random())
print(rnd.random())
print(rnd.random())
```

```
0.24853329350505304
0.9876234408428399
0.34253757101753357
```

There are then several subsidiary functions that piggyback on `random`. There's `randint`, which returns a random integer between two values.

```
print(rnd.randint(4,10))
print(rnd.randint(1,1000))
print(rnd.randint(4,4))
```

```
8
708
4
```

Notice that `rand(a, b)` returns a random int n such that $a \leq n \leq b$; this is arguably a slight deviation from the usual Python convention of "less than or equal at one end, strictly less than at the other".

A bit more generally, there's then `randrange`, which returns a randomly chosen int from the object `range(start, stop, step)` (although it doesn't actually create that object).

```
print(rnd.randrange(4,5))
print(rnd.randrange(1,1000,2))
print(rnd.randrange(11))
```

```
4
479
6
```

Notice that this *does* obey the convention "less than or equal at one end, strictly less than at the other"!

More generally still, there's `choice`, which chooses randomly from any given sequence:

```
print(rnd.choice([1,1.0,1+0j,'one']))
print(rnd.choice(range(1,1000,2)))
print(rnd.choice('abcdefg'))
```

```
1.0
217
d
```

Then there's `choices`, which generates lists of such randomly picked members:

```
print(rnd.choices([1,1.0,1+0j,'one'], k=10))
print(rnd.choices(range(1,1000,2), k=20))
print(rnd.choices('abcdefg', k=5))
```

```
['one', 1.0, 'one', 1, 'one', 1, 1.0, 1.0, (1+0j), 'one']
[783, 673, 975, 253, 431, 797, 617, 247, 279, 227, 791,
139, 43, 243, 759, 285, 967, 817, 931, 391]
['a', 'e', 'f', 'e', 'g']
```

The `choices` function allows the user to specify weights, which cause options to be picked with different probabilities, proportional to the weights:

```
print(rnd.choices([1,1.0,1+0j,'one'], weights=[1,1,1,4], k=10))
print(rnd.choices('abcdefg', weights=[10,1,1,1,1,1,1], k=10))
```

```
[1.0, 1, 1.0, 'one', (1+0j), 1, 'one', 'one', 'one', 'one']
['a', 'a', 'a', 'e', 'a', 'b', 'a', 'a', 'e', 'a']
```

Related are the functions `sample`, which draws from a sequence without replacement, and `shuffle`, which shuffles the elements of a sequence in place. Note that `sample` generates output, whereas `shuffle` returns `None` and performs its shuffle as a side-effect. Because of this, you can't use it on immutable data structures like strings or tuples, and must instead make use of `sample`:

```
print(rnd.sample([1,1.0,1+0j,'one'], 3))
print(rnd.sample('abcdefg', 4))
all_ones = [1,1.0,1+0j,'one']; rnd.shuffle(all_ones); print(all_ones)
```

```
[1.0, (1+0j), 1]
['g', 'd', 'f', 'a']
[(1+0j), 1.0, 'one', 1]
```

Note that `sample` generates output, whereas `shuffle` returns `None` and performs its shuffle as a side-effect. Because of this, you can't use it on immutable data structures like strings or tuples, and must instead make creative use of `sample`:

```
alphabet = 'abcdef'
alphabet = ''.join(rnd.sample(alphabet,len(alphabet)))
print(alphabet)
```

```
cadbef
```

Finally, there's a slightly limited range of **probability distributions** implemented in `random`, which you can sample:

```
print(rnd.uniform(5,6))
print(rnd.normalvariate(100,15))
```

```
5.128437075573844
110.6353290359451
```

Note that the parameters for the uniform distribution are the lower and upper bounds of the range, and those for the Normal the mean and standard deviation (*not* the variance).

The NumPy version of all this stuff is (a) array-compatible and (b) a bit turbocharged, as you'd predict. The function called `random` in this submodule works a little differently from the non-NumPy one; yes, you can get a single (NumPy) float between 0 and 1, but you can also get an array of them, of any dimension you like:

```
import numpy.random as nrnd
print(nrnd.random())
print(nrnd.random(5))
print(nrnd.random([3,2]))
print(nrnd.random(size=[3,2]))
```

```
0.971264235704511
[0.90662282 0.78890151 0.11711386 0.33967627 0.56968404]
[[0.73228846 0.04429268]
 [0.72977398 0.72109446]
 [0.02274657 0.09315491]]
[[0.64902511 0.11407875]
 [0.99828324 0.10626859]
 [0.18550121 0.46765575]]
```

NumPy's version of `randint` obeys the Python convention for inequalities; it too is capable of returning arrays:

```
print(nrnd.randint(1,5))
print(nrnd.randint(1,5,[4,4]))
```

```
4
[[1 1 2 1]
 [2 2 1 4]
 [3 2 3 4]
 [4 1 2 2]]
```

Note that the returned values are **NumPy** ints, which means they're not allowed to be of arbitrary size. So the first of these works but the second generates an error message:

```
print(rnd.randint(10**100,10**101))
print(nrnd.randint(10**100,10**101))
```

```
485897882103994313523262388127171218236032248292972
969466551600913789647470024695095324631662491356 95
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-41-f266798a9735> in <module>()
      1 print(rnd.randint(10**100,10**101))
----> 2 print(nrnd.randint(10**100,10**101))

mtrand.pyx in mtrand.RandomState.randint()

ValueError: high is out of bounds for int32
```

In the core `random` module, the `choices` function does repeated selection with replacement, and the `sample` function does it without. In NumPy's `random` submodule, both are handled by the function called `choice`:

```
import numpy as np
print(nrnd.choice(np.array([2, 3, 5, 7, 11])))
print(nrnd.choice(np.array([2, 3, 5, 7, 11]),5))
print(nrnd.choice(np.array([2, 3, 5, 7, 11]),5,replace=False))
```

```
3
[3 7 2 2 2]
[ 3  7  5  2 11]
```

There's a `permutation` function for generating a shuffled version of an array (as output), and a `shuffle` function for shuffling in place (as a side-effect):

```
print(nrnd.permutation(np.array([2, 3, 5, 7, 11])))

primes = np.array([2, 3, 5, 7, 11]); nrnd.shuffle(primes)
print(primes)
```

```
[ 7  2 11  5  3]
[ 3  7 11  5  2]
```

Finally, there's much more **probability distribution** functionality:

```
print(nrnd.normal(100,15))
print(nrnd.normal(100,15,5))
print(nrnd.binomial(10,0.3,5))
print(nrnd.poisson(3,[2,3]))
```

```
100.74505758683581
[112.45045275 128.03507447  82.97648018 116.62469513 109.40454701]
[4 5 3 3 3]
[[3 2 1]
 [5 7 6]]
```

And in fact loads, loads more; we've barely scratched the surface here!

7.4 `matplotlib` and `pyplot`

You're already somewhat familiar with the "workhorse" plotting function from the `pyplot` submodule of `matplotlib`, which is simply called `plot`. It's used for generating line plots, and point plots, of data in the form of x and y coordinates (the data can come in the form of 1D arrays, or lists, or tuples, or ranges). Let's make use of what we know about random numbers to do something a bit ambitious:

Challenge 1: write a function called `three_points_step` which takes as its input a 1D array of length 2 representing a pair of coordinates (x, y). It should then choose a random integer 0, 1 or 2. If it's equal to 0, it should return $(x/2, y/2)$; if 1, $(x/2 + 1/2, y/2)$; if 2, $(x/2 + 1/4, y/2 + \sqrt{3}/4)$.
Iterate this function 100 times, discarding the iterates. Then iterate it a further 10000 times, keeping the iterates in an array. Finally, create a dot plot of these iterates you've retained.

```
%matplotlib inline

def three_points_step(coords):
    from random import randint
    from numpy import sqrt, array
    r = randint(0,2)
    if r==0:
        return coords/2
    elif r==1:
        return (coords+array([1.0,0.0]))/2
```

```
    else:
        return (coords+array([0.5,sqrt(3)/2]))/2

import numpy as np
coords = np.array([0.5, 0.5])

# iterate and discard
for n in range(100):
    coords = three_points_step(coords)

# initialize array of coordinates
coords_array = np.array([coords])

# iterate and retain
for n in range(10000):
    coords = three_points_step(coords)
    coords_array = np.concatenate((coords_array,np.array([coords])))

# plot
import matplotlib.pyplot as plt
plt.figure(figsize=[7,7])
plt.axis('equal')
plt.plot(coords_array[:,0],coords_array[:,1],'k.',markersize=0.5)
```

The image we get forms Figure 3. Things to notice on the NumPy side of things include the way we used `concatenate` to build up the array of coordinate pairs; it's very easy to get the number of parentheses wrong! Things to notice on the plotting side of things include:

- the way we've set the size of the figure;
- how to make the scales equal on both axes;
- the way we've picked out columns 0 and 1 of the coordinate array for plotting using the `:` notation (actually, this is sort of a NumPy observation too);
- the use of the optional keyword-only argument `markersize`.

The **plot** function is very useful, but it's not the only thing out there. For example, there's also `imshow`, which takes as its argument an array of values, and visualises that array using coloured squares:

```
mat1 = np.array([[0,1,2],[1,2,3],[3,4,5]])
```

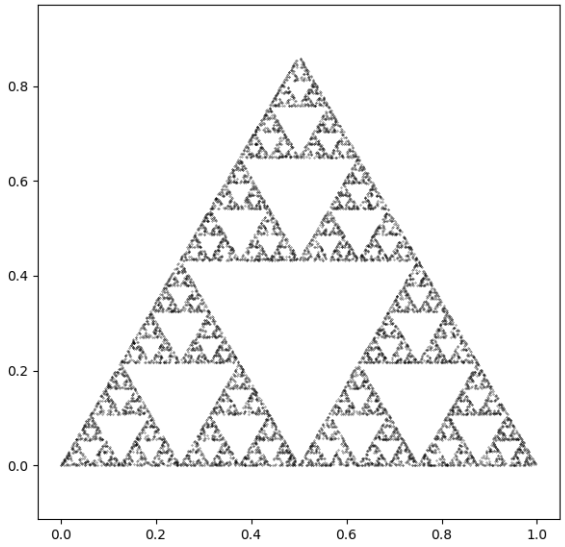


Figure 3: The Sierpinski triangle

```
plt.imshow(mat1)
```

The image is shown as Figure 4. The `imshow` function gives us one way of visualizing a function of two variables. The tactic is to create values of $u(x, y)$ for values of x and y lying on a lattice. The most flexible way to do that uses a NumPy function called meshgrid For example, to create the matrix we used in the last example, we could do this:

```
x_values = np.arange(3)
y_values = np.arange(3)
x, y = np.meshgrid(x_values, y_values)
mat1 = x+y
print(x)
print(y)
print(mat1)
```

```
[[0 1 2]
 [0 1 2]
 [0 1 2]]
```

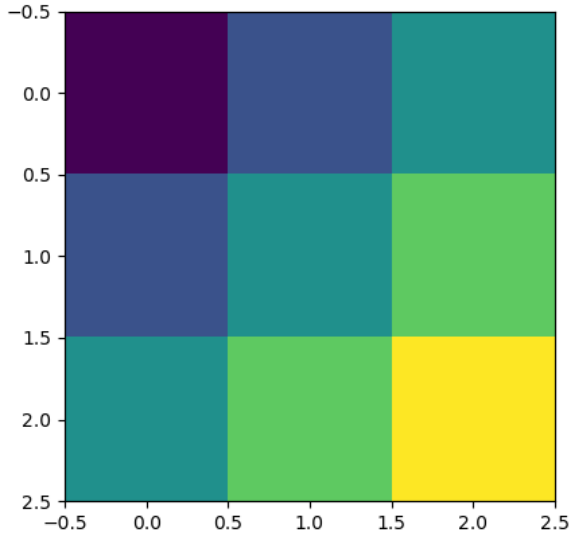


Figure 4: Visualisation of a (3 by 3) matrix using `imshow`

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

Look what `meshgrid` does; it allows us to create arrays `x` and `y`, standing for the x- and y-coordinates respectively of nine points on a lattice. The matrix then represents the values on that lattice of $u(x, y) = x + y$, and the diagram visualises that function on that lattice.

(In this case, of course, we oluld also have used `fromfunction`, but going via `meshgrid` allows us to use non-integer ranges, so it's much more flexible.)

Challenge 2: use `imshow` to create a visualisation of the function

$$u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$$

for 101 equally-spaced values of x and y, each between -2.0 and 2.0

```
x_values = np.linspace(-2.0,2.0,101)
```

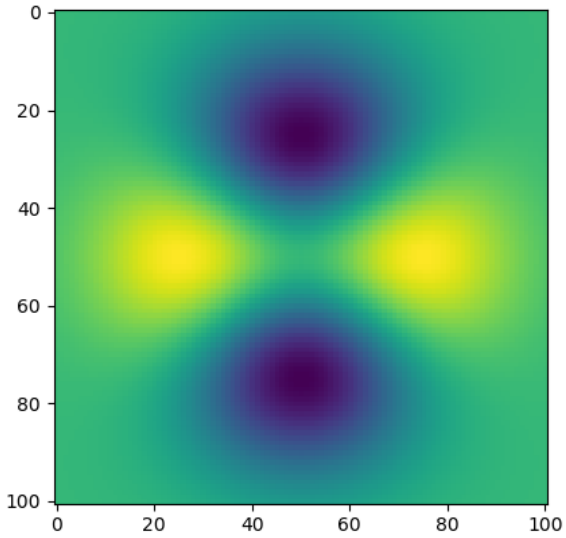


Figure 5: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using `imshow`

```
y_values = np.linspace(-2.0,2.0,101)
x, y = np.meshgrid(x_values, y_values)

u = (x**2 - 2*y**2)*np.exp(-x**2-y**2)
plt.imshow(u)
```

The image is shown as Figure 5. There are alternative ways to visualise a function of two variables. In two dimensions, the main one is a **contour plot**. By default, `matplotlib`'s `contour` function shows contours in a variety of colours; we can override this using the `colors` option.

```
fig, (ax1, ax2) = plt.subplots(1,2)
ax1.axis('equal')
ax2.axis('equal')
ax1.contour(x, y, u)
ax2.contour(x, y, u, colors='red')
```

The image is shown as Figure 6. Observe, in passing, how we set up two subplots in the same grid; in this case, a 2 by 1 grid.

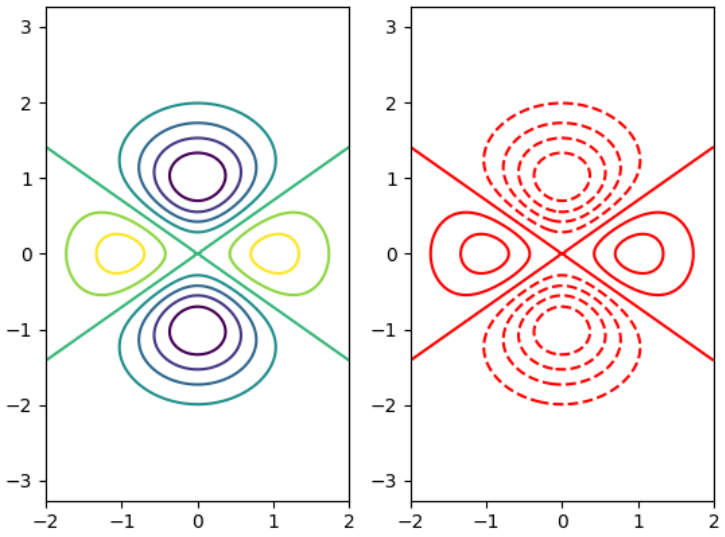


Figure 6: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using `contour`

There are also three important 3D visualisation tools for functions of two variables: `plot_wireframe`, `plot_surface` and the 3D version of `contour`. The way we use them, though, is a little different from in 2D.

```
from mpl_toolkits.mplot3d import Axes3D
ax = plt.axes(projection='3d')
ax.plot_wireframe(x, y, u)
```

```
ax = plt.axes(projection='3d')
ax.plot_surface(x, y, u)
```

```
ax = plt.axes(projection='3d')
ax1contour(x, y, u, colors='red')
```

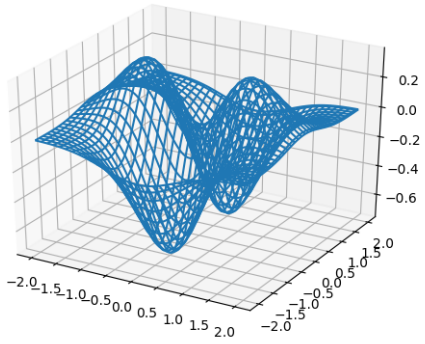


Figure 7: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a wireframe

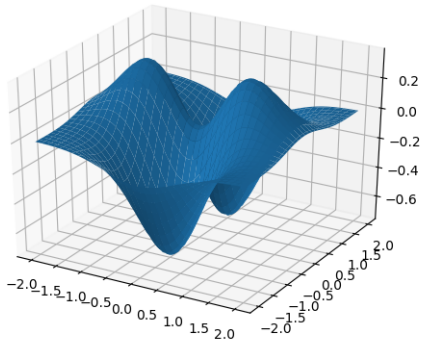


Figure 8: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a surface plot

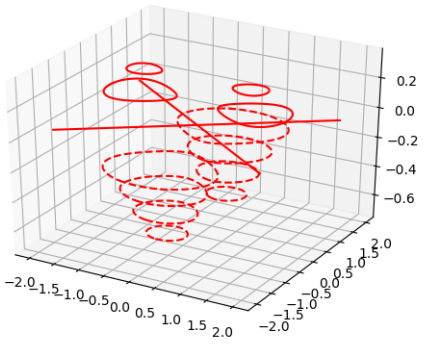


Figure 9: Visualisation of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$ using a 3D contour plot

Another important 2D diagram is the **vector field plot**, sometimes called the **quiver plot**; this consists of a field of arrows representing a vector field $\mathbf{f}(x, y)$, in which each pair of coordinates is associated with a vector. The `matplotlib` function we use for this is called `quiver`; it takes four arguments: values of x and y specifying the positions of the arrows, and the two components of $\mathbf{f}(x, y)$, specifying the corresponding vectors. Here, for example, is a plot of the two partial derivatives of our function

$$u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$$

with respect to x and y respectively We've used a 21 by 21 lattice of points (in general, you should use a coarser mesh for quiver plots than for, say, contour plots).

```
X_values = np.linspace(-2.0,2.0,21)
Y_values = np.linspace(-2.0,2.0,21)
X, Y = np.meshgrid(X_values, Y_values)

dudx = -2*X*(-1+X**2-2*Y**2)*np.exp(-X**2-Y**2)
dudy = -2*Y*(2+X**2-2*Y**2)*np.exp(-X**2-Y**2)
plt.axis('equal')
plt.quiver(X, Y, dudx, dudy)
```

The image appears as Figure 10. The quiver plot looks interesting superimposed on a contour plot of the original function; this is shown in Figure 11.

Finally, `pyplot` supports a variety of **statistical diagrams**: bar charts, histograms, pie charts, box and whisker plots, scatter diagrams and many others.

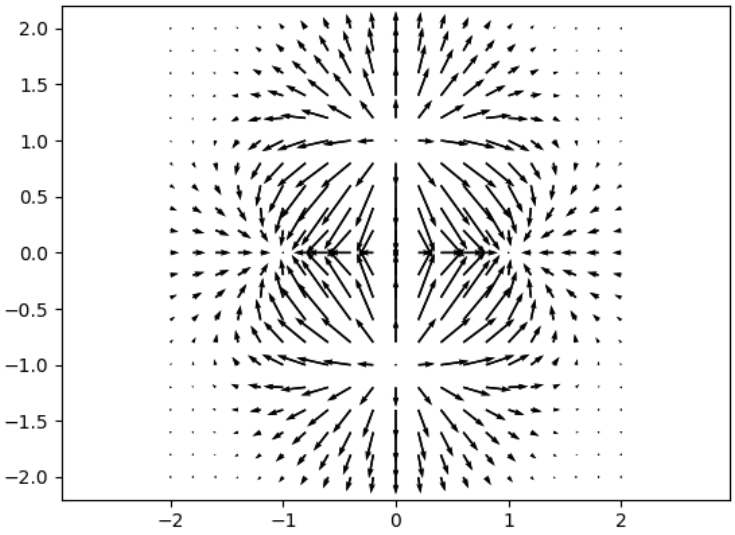


Figure 10: Quiver plot of the partial derivatives of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$

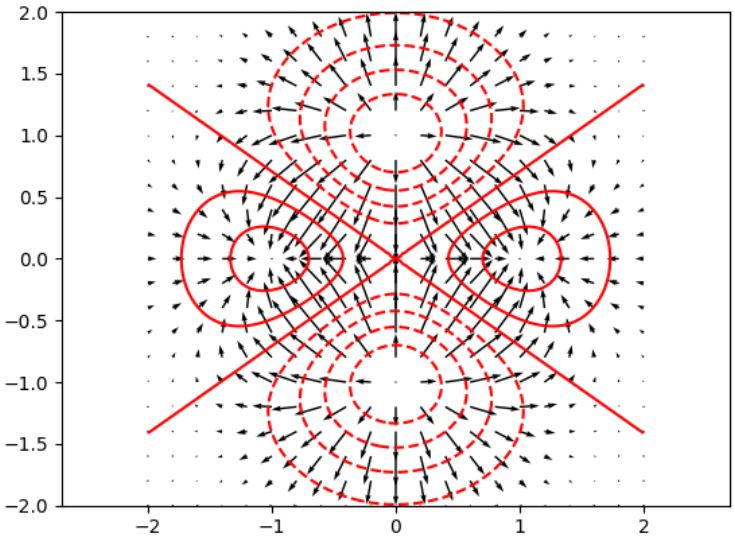


Figure 11: Quiver plot of the partial derivatives of $u(x, y) = (x^2 - 2\,y^2)\,e^{-x^2-y^2}$, superimposed on contour plot of u

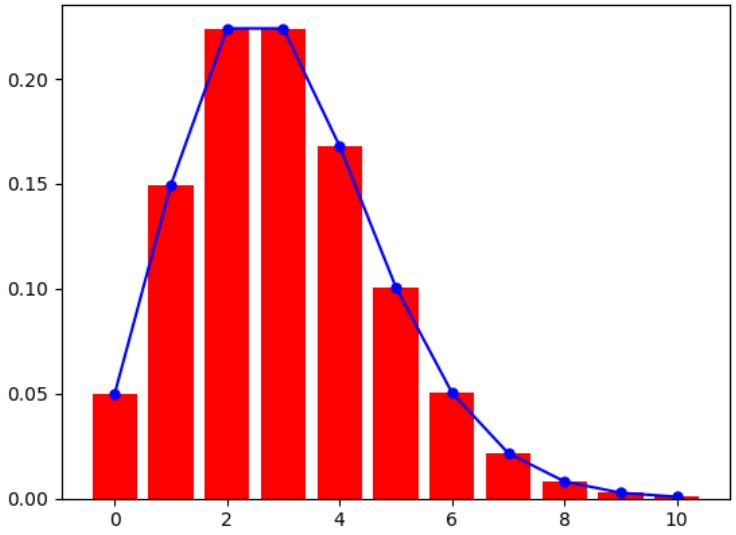


Figure 12: Visualisation of a Poisson distribution

Challenge 3: calculate, as a 1D array, the values of $r!$ for r from 0 to 10, and hence the values of the pdf for those values of a random variable that is Poisson distributed with parameter 3.0. Show these, on the same pair of axes, as a line plot, a point plot and a bar chart.

```
r = np.arange(11)
rfac = np.concatenate((np.array([1]),np.cumprod(r[1:len(r)])))
lamb = 3.0

poissprob = np.exp(-lamb)*lamb**r/rfac

plt.plot(r, poissprob, 'b')
plt.plot(r, poissprob, '.b', markersize=10)
plt.bar(r, poissprob, color='red')
plt.imshow(u)
```

The image is shown as Figure 12. Note the use of NumPy's `cumprod` function.

By default, **histograms** (as opposed to bar charts) are scaled so that the total area is the total size of the data set. However, they can, by setting `normed` to `True`, be scaled so

that the total area is 1, meaning that the height is the **relative frequency density**. Bin boundaries are by default set automatically, but you can override that.

Challenge 4: sample the Poisson distribution with parameter 3.0 (a) 100 times and (b) 1000 times, and show your results on normed histograms on which line and point plots of the underlying distribution also appear.

```
import numpy.random as nrnd

ndata1 = 100
data1 = nrnd.poisson(3.0, ndata1)
ndata2 = 1000
data2 = nrnd.poisson(3.0, ndata2)

fig, (ax1, ax2) = plt.subplots(1,2)
ax1.plot(r, poissprob, 'b')
ax1.plot(r, poissprob, '.b', markersize=10)
ax1.hist(data1, bins=np.arange(-0.5,11.5),color='red',normed=True);
ax2.plot(r, poissprob, 'b')
ax2.plot(r, poissprob, '.b', markersize=10)
ax2.hist(data2, bins=np.arange(-0.5,11.5),color='red',normed=True);
```

The image is shown as Figure 13.

7.5 The SymPy module

Traditionally, the key contribution of computing to mathematics and science lay in "number-crunching": essentially, in doing calculations with floating-point numbers. For many, many decades now, though, computers have also been able to do **symbolic** calculations: that is, algebraic manipulation, calculus etc. Systems that can do this include Maple, Wolfram Mathematica and, a bit clunkily, via its Symbolic Math Toolbox, Matlab; Python does this (and does it for free!) using the SymPy module (short for "symbolic Python").

7.5.1 Functions, constants, rationals and surds

We start by importing the module (and let's also import NumPy and `math` for purposes of comparison):

```
import sympy as sp
import numpy as np
```

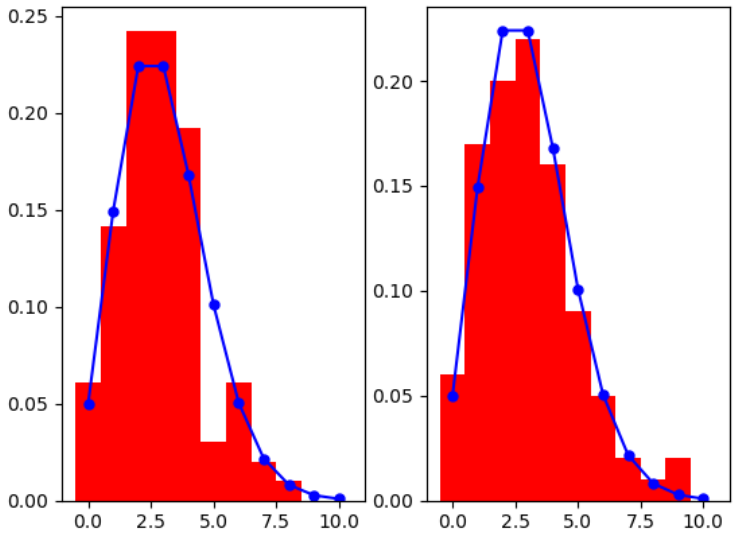


Figure 13: Normed histograms of samples of a Poisson distribution; sample sizes 100 and 1000

```
import math
```

The module contains a nice feature called pretty-printing; in Jupyter notebooks, this makes symbolic expressions appear as LaTeX-formatted 2D maths, which is nice. Up to you whether you switch it on, but if you want to, you do it like this:

```
sp.init_printing()
```

Note that pretty-printing doesn't work with the `print` function, so we'll aim to use that as little as possible.

The SymPy module has its own versions of most of the mathematical functions, except that its behaviour is a little different. Compare

```
np.sqrt(8)
```

```
2.8284271247461903
```

and

```
math.sqrt(8)
```

```
2.8284271247461903
```

with

```
sp.sqrt(8)
```

$$2\sqrt{2}$$

The last one is an **exact** quantity; in this case a surd. We can do calculations with it ...

```
x = sp.sqrt(8)

(x/2)**7
```

$2\sqrt{2}$ ... we can convert it to a core Python float (this is called **casting**)...

```
x = sp.sqrt(8)

float(x)
```

```
2.8284271247461903
```

...or we can use a method called `evalf` to evaluate it as a float to arbitrary precision:

```
x = sp.sqrt(8)

x.evalf(1000)
```

2.828427124746190097603377448419396157139343750753896146353359475981464956924214077700775068655283145470027692461824594049849672111701474425288242994199871662826445331855011185511599901002305564121142940219119943211940549069193724029457034837281778397219101

46584609686174286429016795252072559905028159793745067930926636176592812412305167047901094915005755199234596711504406750637140227087492068169976943207737999413980096300610880555806329084956461369858738372431611569262231933374260260312371379744744705770185297224989954308436668408571372120293649441542871709748311314139355307440452970894031717603241516949845314452004171168933042916797787887418531836006227764929363141652602011897174080063729606843897945565812820901452737626274797105122346440804901824554004538822551472545609914762179350080367397367369014515987294581215259938827609513096474579943606536049488412585382497181043620089196843011822404988826834570629562116072067421546183657386294203422233678331634537788395174331643042564590369769 4

Notice what happens if we type

```
x = sp.sqrt(8)

x**4/6
```

$$\frac{32}{3}$$

We don't get `10.666666666666666`, which is what `32/3` would give us, or `10`, which we'd get from `10//3`; instead, we get an exact **rational**. If we want to create this rational without going to the trouble of creating a surd first, we've two options:

```
y = sp.Rational(32, 3)

y
```

$$\frac{32}{3}$$ or

```
num = sp.Integer(32)
den = sp.Integer(3)
y = num/den

y
```

$$\frac{32}{3}$$

Rationals behave sensibly under the main arithmetic operations:

```
a = sp.Rational(2,3)
b = sp.Rational(1,6)

(a + b, a - b, a*b, a/b, a**2)
```

$$\left(\frac{5}{6}, \frac{1}{2}, \frac{1}{9}, 4, \frac{4}{9}\right)$$

SymPy has its own version of most of the mathematical functions and constants you'd fine in NumPy or `math`; however, by default, most of them return **exact** values.

```
(sp.sin(sp.pi/3), sp.atan(1), sp.exp(sp.log(7)))
```

$$\left(\frac{\sqrt{3}}{2}, \frac{\pi}{4}, 7\right)$$

The SymPy module contains its own implementation of complex numbers:

```
z1 = 3 + 4*sp.I

z1
```

$$3 + 4i$$

Notice that the square root of minus one is represented not by `1j` but by the SymPy constant `I`. We can do calculations:

```
z2 = sp.expand(z1**2)

z2
```

$$-7 + 24i$$

```
z3 = sp.exp(sp.log(2)+sp.I*sp.pi/2)

z3
```

$2i$

7.5.2 Symbols and expressions

We really start seeing what's special about SymPy when we start doing algebraic manipulation and calculus with it. We need first to set up some symbolic variables:

```
x, y = sp.symbols('x y')
```

What this means is "Let `x` and `y` be variables whose values are the symbols x and y." Because Python now has values for these variables, we can type something like

```
expr = x + 2*y**2

expr
```

$x + 2y^2$

We call `x` and `y` **symbols**; `expr` is a **symbolic expression**. SymPy allows us to perform a wide variety of algebra and calculus operations on such symbolic objects. First some manipulation:

```
expr = x + 2*y**2

expr - x + y**2
```

$3y^2$

```
expr = x + 2*y**2
```

```
sp.expand(expr**3)
```

$$x^3 + 6x^2y^2 + 12xy^4 + 8y^6$$

```
expr = x**3 + 6*x**2*y**2 + 12*x*y**4 + 8*y**6

sp.factor(expr)
```

$$\left(x + 2y^2\right)^3$$

```
expr = 1/((x+2)*(x+1))

sp.apart(expr)
```

$$-\frac{1}{x+2} + \frac{1}{x+1}$$

(Notice that the `apart` function resolves into partial fractions.)

There's an overarching manipulation function called `simplify`, which tries, not always very successfully, to express anything you give it in the simplest form possible:

```
expr = (x+1)**4 - (x-1)**4

sp.simplify(expr)
```

$$8\,x\left(x^2 + 1\right)$$

Challenge 1: the **Chebyshev polynomials of the first kind** are defined by the recurrence relation

$$\begin{aligned} T_0 &= 1, \\ T_1 &= x, \\ T_n &= 2\,x\,T_{n-1}(x) - T_{n-2}(x). \end{aligned}$$

Write and test a function that takes as its argument a non-negative integer `n` and a variable `x` and returns the nth Chebyshev polynomial as a fully expanded symbolic expression in `x`.

First an iterative implementation:

```
def chebyshevT1(n, x):
    """
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """
    # import from sympy
    from sympy import Integer, expand
    # special case
    if n==0:
        return Integer(1)
    else:
        # initialize
        t_old, t_new = Integer(1), x

        # for loop
        for i in range(2,n+1):
            # update using recurrence relation
            t_old, t_new = t_new, expand(2*x*t_new - t_old)

        # return last one
        return t_new
```

Testing:

```
x, t = sp.symbols('x t')
(chebyshevT1(0, x), chebyshevT1(5, x),
 chebyshevT1(7, t), chebyshevT1(7, sp.Rational(1,2)))
```

$$\left(1, \quad 16x^5 - 20x^3 + 5x, \quad 64t^7 - 112t^5 + 56t^3 - 7t, \quad \frac{1}{2}\right)$$

Now a recursive one, using the "inner-and-outer" trick to avoid a combinatorial explosion in execution time:

```
def chebyshevTpair(n, x):
    # import from sympy
    from sympy import Integer, expand
    # base case
    if n==1:
        return (Integer(1), x)
    # iteration step
    else:
        tpair = chebyshevTpair(n-1, x)
        return (tpair[1], expand(2*x*tpair[1] - tpair[0]))

def chebyshevT2(n, x):
    """
    Returns the nth Chebyshev polynomial of the first kind
    as a symbolic expression in x
    """
    # import from sympy
    from sympy import Integer
    # special case
    if n==0:
        return Integer(1)
    else:
        return chebyshevTpair(n, x)[1]
```

Testing produces the same results.

7.5.3 The subs method and the lambdify function

Suppose I have an expression in x, such as:

```
x = sp.symbols('x')
expr = 16*x**5 - 20*x**3+5*x

expr
```

$$16x^5 - 20x^3 + 5x$$

Suppose I now want to substitute in the value $x = 2$. You might think that this would work:

```
x = 2

expr
```

However, it doesn't; it just produces $16x^5 - 20x^3 + 5x$ again. That's because although the value of the variable `x` has been changed, the value of the variable `expr` has not, and it's still defined in terms of the **symbol** `x`. (I realise this is a bit confusing).

There are two main ways of doing this. One is to use the `subs` method:

```
expr.subs(x, 2)
```

$$362$$

The alternative is to convert this symbolic expression into a function; actually, into a lambda-expression. SymPy has a function called `lambdify` that does this:

```
f = sp.lambdify(x, expr)

f(2)
```

$$362$$

Using both approaches, we can perform a symbolic substitution:

```
y = sp.symbols('y')

expr.subs(x, y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

```
y = sp.symbols('y')

f(y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

The second approach may seem a little long-winded, but it's in some ways quite a lot more flexible. There is, for example, an optional third argument to `lambdify` which, if we set it to `'numpy'`, means our new lambda-expression works on arrays:

```
import numpy as np
fn = sp.lambdify(x, expr, 'numpy')

fn(np.arange(-3,4))
```

```
array([-3363,  -362,    -1,     0,     1,   362,  3363])
```

Best of all, our NumPy-compatible lambda-expression still works with symbols!

```
fn(y**2)
```

$$16y^{10} - 20y^6 + 5y^2$$

7.5.4 Four kinds of equality

SymPy offers three ways of *testing* equality, with varying levels of strictness, and a third idea of equality used for a different purpose.

When you use it with SymPy symbolic expressions, the operator == is very strict: it only returns `True` if two expressions are, once they've undergone a standard rearrangement, exactly the same.

```
expr1 = (x+1)**4 - (x-1)**4
expr2 = -(x-1)**4 + (x+1)**4
expr3 = 8*x*(x**2+1)

(expr1==expr2, expr==expr3)
```

```
(True, False)
```

A more generous test of whether two expressions are equivalent is to see if their difference simplifies to give zero:

```
expr1 = (x+1)**4 - (x-1)**4
expr2 = -(x-1)**4 + (x+1)**4
expr3 = 8*x*(x**2+1)

(sp.simplify(expr1-expr2)==0,sp.simplify(expr1-expr3)==0)
```

```
(True, True)
```

However, this doesn't always work. The following expressions are exactly equivalent, for example, but `simplify` doesn't pick this up:

```
expr1 = sp.sqrt(13+4*sp.sqrt(3))
expr2 = (1+2*sp.sqrt(3))

sp.simplify(expr1-expr2)==0
```

```
False
```

In the exercises, you explore an approach to complicated surds that uses a function called `nthroot` instead of `sqrt`. But more generally, if you think two things are equal and neither a straight comparison with == nor simplifying their difference seems to agree, there's something called the `equals` method:

```
expr1 = sp.sqrt(13+4*sp.sqrt(3))
expr2 = (1+2*sp.sqrt(3))

expr1.equals(expr2)
```

```
True
```

Warning: this method is (a) quite crude (it just compares values for a few chosen values of any variables you use) and (b) I think a bit buggy; there are times when it should return `True` but instead returns `None`.

The fourth kind of equality is quite distinct, and arises when you don't want to *test* whether two quantities are equal, but simply to set up an equation, perhaps in order to solve it. For that we use the SymPy function `Eq`. Here's how it works, for example, with the very useful `solve` function:

```
expr1 = (x+1)**4 - (x-1)**4

sp.solve(sp.Eq(expr1,0), x)
```

$[0, i, -i]$

Challenge 2: write and test a function that takes as its arguments a symbolic expression `expr`, a symbolic variable `x` and a value `a`, and finds the value of the derivative of `expr` with respect to `x` at $x = a$.

```
def deriv_val(expr, x, a):
    from sympy import diff
    dexpr = diff(expr, x)
    return dexpr.subs(x, a)
```

Testing:

```
deriv_val((x+1)**4 - (x-1)**4, x, 3)
```

224

Challenge 3: find the values of the derivative of $(x+1)^4 - (x-1)^4$ at integer values of x between -3 and 3 inclusive.

The trouble here is that our `deriv_val` function doesn't work on arrays. We could use a comprehension:

```
[deriv_val((x+1)**4 - (x-1)**4, x, a) for a in range(-3, 4)]
```

```
[224,104,32,8,32,104,224]
```

Or there's a NumPy function called `vectorize` that makes any given function into one that works with arrays:

```
import numpy as np
deriv_val_vec = np.vectorize(deriv_val)

deriv_val_vec((x+1)**4 - (x-1)**4, x, np.arange(-3,4))
```

```
array([224, 104, 32, 8, 32, 104, 224], dtype=object)
```

(Notice the weird "`dtype=object`"; this signifies that these are SymPy integers rather than NumPy ones.)

Perhaps best, though, would be to write a version of `deriv_val` that instead uses `lambdify`:

```
def deriv_val2(expr, x, a):
    from sympy import diff, lambdify
    dexpr = diff(expr, x)
    dexpr_f = lambdify(x, dexpr, 'numpy')
    return dexpr_f(a)
```

Then it just works over arrays:

```
deriv_val2((x+1)**4 - (x-1)**4, x, np.arange(-3,4))
```

```
array([224, 104, 32, 8, 32, 104, 224], dtype=int32)
```

7.5.5 Linear algebra

The SymPy module has its own linear algebra functions nd methods:

```
m = sp.Matrix([[1, 2], [2, 2]])
(m * m, m ** 2, m.det(), m.inv())
```

$$\left(\begin{bmatrix}5 & 6\\ 6 & 8\end{bmatrix}, \quad \begin{bmatrix}5 & 6\\ 6 & 8\end{bmatrix}, \quad -2, \quad \begin{bmatrix}-1 & 1\\ 1 & -\frac{1}{2}\end{bmatrix}\right)$$

Notice that unlike NumPy, SymPy does use * and ** to stand, respectively, for matrix multiplication and matrix exponentiation.

```
m.eigenvals()
```

$$\left\{\frac{3}{2} + \frac{\sqrt{17}}{2} : 1, \quad -\frac{\sqrt{17}}{2} + \frac{3}{2} : 1\right\}$$

```
m.eigenvects()
```

$$\left[\left(\frac{3}{2} + \frac{\sqrt{17}}{2}, \quad 1, \quad \left[\begin{bmatrix} -\frac{2}{-\frac{\sqrt{17}}{2} - \frac{1}{2}} \\ 1 \end{bmatrix}\right]\right), \quad \left(-\frac{\sqrt{17}}{2} + \frac{3}{2}, \quad 1, \quad \left[\begin{bmatrix} -\frac{2}{-\frac{1}{2} + \frac{\sqrt{17}}{2}} \\ 1 \end{bmatrix}\right]\right)\right]$$

7.5.6 Plotting

Finally, SymPy incorporates a set of plotting functions, which allow us, unlike the ones in `matplotlib.pyplot`, to plot functions directly without having to use them to make data sets. The basic plotting function is called `plot`, just like the one in `pyplot`; however, it works quite differently. As a reminder, here's how we'd create a plot of $y = \sin x$ in `pyplot`:

Compare and contrast: `pyplot`

```
import matplotlib.pyplot as plt
import numpy as np
x_values = np.linspace(0, 2*np.pi, 97)
y_values = np.sin(x_values)
plt.plot(x_values, y_values)
```

This is shown in Figure **??**.

Here's how we'd do it in SymPy:

```
import sympy as sp
sp.plot(sp.sin(x), (x, 0, 2*sp.pi))
```

This is shown in Figure **??**.

Good news, eh? There are even axes shown; we would need to add them to the `pyplot` version using the functions `axhline` and `axvline`. But there are few pieces of not-so-good

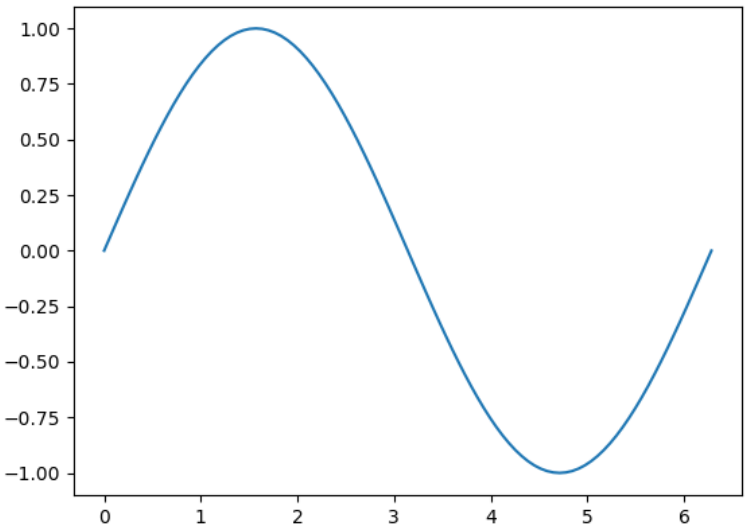


Figure 14: Sine function plotted using `pyplot`

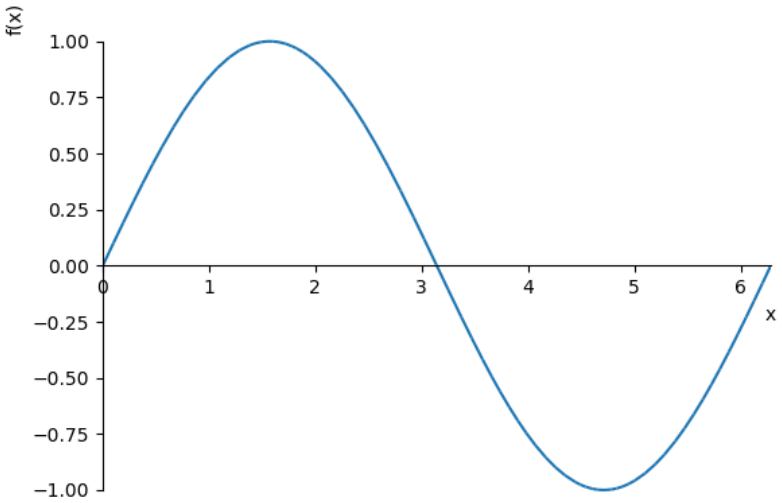


Figure 15: Sine function plotted using `sympy`

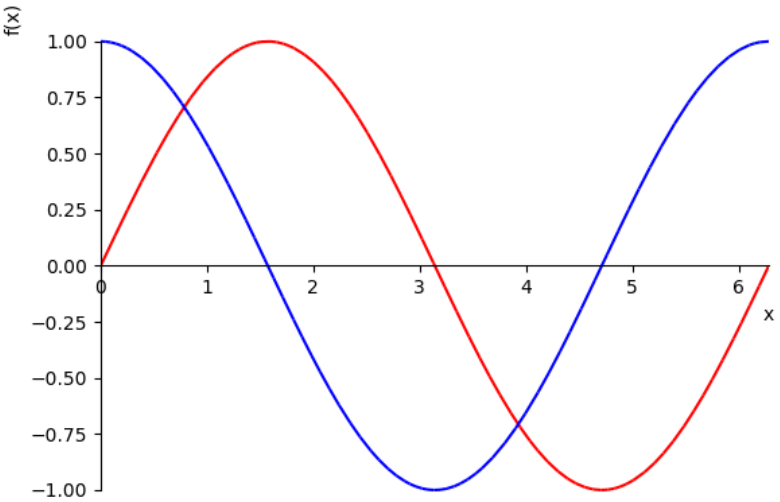


Figure 16: Sine and cosine functions plotted using `sympy`

news. The first is that superimposing two plots on the same pair of axes is a bit of a palaver:

```
p = sp.plot(sp.sin(x), sp.cos(x), (x, 0, 2*sp.pi), show=False)
p[0].line_color = 'red'
p[1].line_color = 'blue'
p.show()
```

(Figure 16).

The second is that whereas in `pyplot`, as in Matlab, you only really need one 2d plotting function, in SymPy you need one for every kind of plot. For example, suppose we want to plot the parametric curve $x = \cos 3t$, $y = \sin 5t$. Here's how we'd do it in `pyplot`:

Compare and contrast: `pyplot`

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

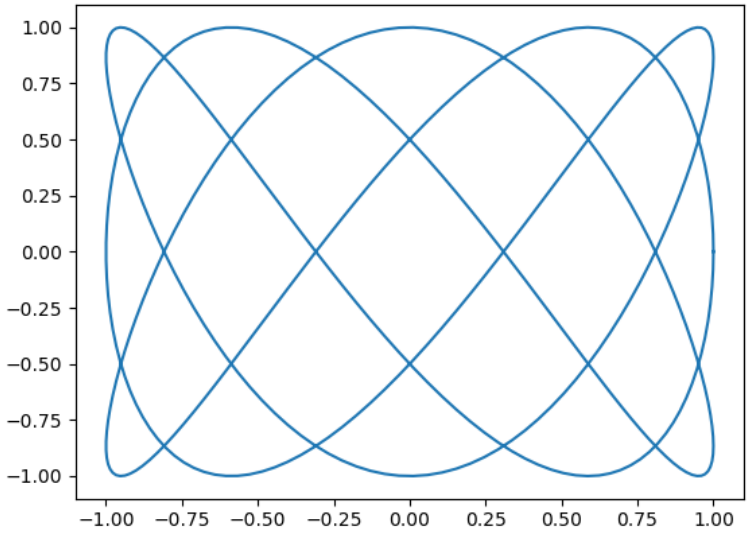


Figure 17: Lissajous figure plotted using `pyplot`

```
t_values = np.linspace(0, 2*math.pi, 400)
x_values = np.cos(3*t_values)
y_values = np.sin(5*t_values)
plt.plot(x_values, y_values)
```

(Figure 17.)

We use `pyplot`'s `plot` function; it's all pretty easy to remember. But `sympy`'s `plot` function, by contrast, only does *explicit Cartesian* plots of the form $y = f(x)$; if we want a parametric plot, like this one, we need a different plotting function. The one we want, like most of the plotting functions, lies in a submodule called `plotting`, and is called `plot_parametric`. Here's how it all works.

```
import sympy as sp
import sympy.plotting as splt
t = sp.symbols('t')
splt.plot_parametric(sp.cos(3*t), sp.sin(5*t), (t, 0, 2*sp.pi))
```

(Figure 18.)

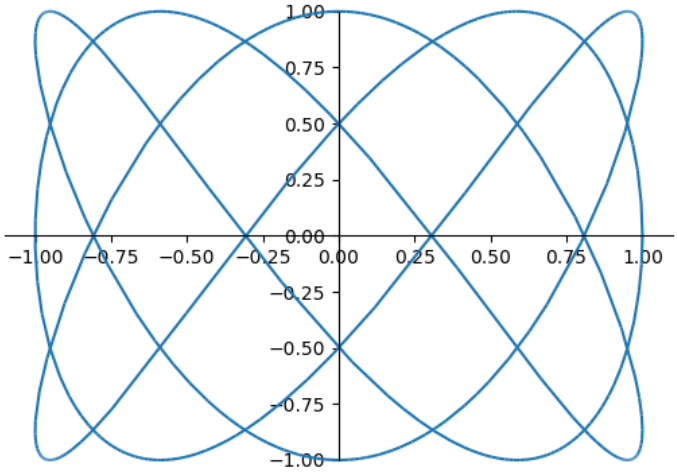


Figure 18: Lissajous figure plotted using sympy

There are several other plotting functions for contour plots, 3D plots, etc; you get to explore these in the exercises.

The third piece of bad news concerns what we have to do if we want to superimpose two plots from different plotting functions. Again, this is a bit of a bother to do:

```
import sympy
import sympy.plotting as splt
x, t = sp.symbols('x t')
# Cartesian plot
p1 = sp.plot(x**3-3*x, (x, -2, 2), line_color = 'blue', show=False)
# parametric plot
p2 = splt.plot_parametric(t**3 - 3*t, t, (t, -2, 2), \
                    line_color = 'red', show=False)
# join plots
p1.extend(p2)
# show
p1.show()
```

(Figure 19).

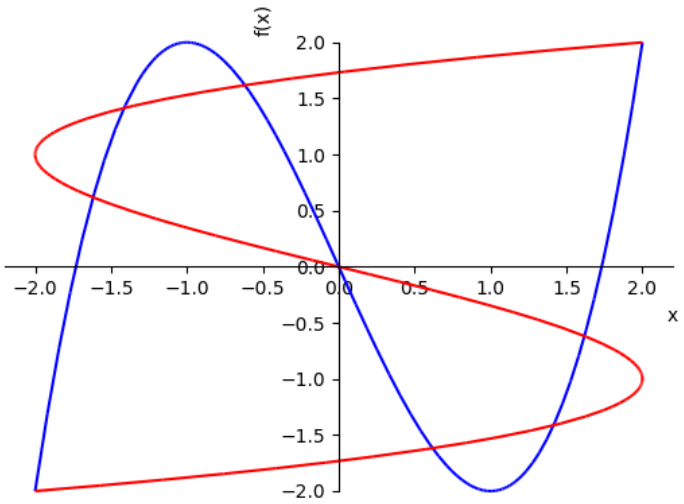


Figure 19: Superimposition of two plots using `sympy`

MATH40006: An Introduction To Computation
Course notes, Volume 4

These notes, together with all the other resources you'll need, are available on Blackboard, at

`https://bb.imperial.ac.uk/`

---

8 Principles of programming

8.1 Algorithms and complexity

8.1.1 Algorithms

What is a computer program? One one level, it's a set of instructions written in a particular language, telling a particular computer to do a particular thing. But computer programs are embodiments of something more abstract and general: an **algorithm**.

An algorithm is a finite sequence of instructions which, if followed, will solve a particular problem in finite time. An algorithm is something that doesn't depend on a particular choice of programming language: a concrete computer program, written in a particular language, may be said to **implement** an algorithm.

For example, consider the problem of searching a data structure to see whether it contains a particular data item. Here, shorn of all docstrings and comments, is how we might implement that in Python:

```
def search(data, element):

    for x in data:

        if x == element:
            return True

    return False
```

In Matlab, we might write:

```
function isThere = search(data, element)

for x = data
    disp(x)
    if x == element
        isThere = 1;
        return
    end
end

isThere = 0;
end
```

In Maple:

```
search := proc(data, element)

    local x;

    for x in data do

        if x = element then
            return True;
        end if

    end do;

    return False;

end proc
```

In the Wolfram Language:

```
search[data_, element_] :=
 Module[{isThere = False},
  Do[If[data[[i]] == element, isThere = True; Break[]], {i, 1,
    Length[data]}]; isThere]
```

and so on. Widely differing programming languages; just one algorithm. That algorithm:

search through `data` term by term, and if you encounter `element` give the output `True`; if you never encounter `element`, return `False`.

Notice that this process must terminate in finite time: no collection of data is infinite, and at some point we will either encounter what we're looking for or use up all the data. This is one of the things that makes it an algorithm.

8.1.2 Pseudocode

It's useful to have a way of describing an algorithm that doesn't depend on the particular programming language we're working with. Natural language (as above) is one such way, but it has the drawback of being a bit imprecise. Another is **pseudocode**: a way of laying out the parts of an algorithm that's informal, but not quite so informal as natural language.

Pseudocode is meant to be human-readable, and has no strict rules. Our seach algorithm, in a typical pseudocode, might look something like this:

```
algorithm search
    input:  1D array data, particular data item element
    output:  boolean value True or False

for each x in data do
        if x = element
            return True, then stop

    otherwise return False
```

For all that algorithms are independent of choice of language, this is definitely "Python-style" pseudocode, with indenting being used to delineate blocks, and so on; people tend to write pseudocode that looks a bit like the code they're used to. However, it tends to be less bogged down in the details of the implementation, and (to really take the pressure off) it doesn't have to work as such; it only needs to make sense to the reader.

8.1.3 Counting operations: big O, big Ω

Let's consider again the problem of searching a list for a particular data item, and now let's make the assumption that our data is **sorted**: that it's in ascending order. We could still use our `search` function, but in some ways we'd be silly to. It would be better to use that information to speed up the search. Here's some pseudocode for a function to do that:

```
algorithm binary_search
    input:  1D array data, particular data item element
    output:  boolean value True or False
```

```
if no data remaining
      return False
   else
      middle_item := item at midpoint of data
      if middle_item = element
         return True
      else if middle_item > element
         return binary_search(left half, element)
      else
         return binary_search(right half, element)
```

Notice that is is a **recursive** implementation; this feels like a natural for such an approach.

Here's what it looks like as a Python function:

```
def binary_search(data, element):
    """
    Searches data for element, using binary search
    data is assumed to be in ascending order
    """

    # number of data items
    ndata = len(data)

    # no data remaining
    if ndata == 0:
        return False
    else:
        # find halfway point
        half_length = ndata//2
        # have we found element?
        if data[half_length] == element:
            return True
        # recursively search left half
        elif data[half_length] > element:
            return binary_search(data[0:half_length], element)
        # recursively search right half
        else:
            return binary_search(data[half_length+1:ndata], element)
```

Now, this certainly feels like a much better way of searching for data in a sorted list than `search`. But can we make that intuition more precise?

Both of our algorithms are capable, by pure dumb luck, of hitting upon the element we're searching for first go. So they have identical **best case** behaviour. But things look very different in the **worst case**. In the case of `search`, our luck might be out: either the element might not be in the list, or it might be last ("It's always the last place you look," as people say.) In that case the number of iterations would be the length of `data`.

Now, what about `binary_search`? Well, suppose the data starts off consisting of 255 items. After one inspection of the midpoint, either we've found `element` or we haven't; we're thinking about the worst case, so let's assume we're out of luck. We then throw away the midpoint, and half of the rest of the list, leaving us with a list of length 127. After another inspection of the midpoint, we're down to 63 (assuming we haven't found it). Then we get 31, then 15, then 7, then 3, then 1; one final inspection and we're done. That was a total of 8; `search` would, in the worst-case assumption, have taken 255 iterations.

You'll have noticed that if the list starts off of length $2^m - 1$, then after one inspection of the midpoint with no luck, it's down to $2^{m-1} - 1$, then $2^{m-2} - 1$ and so on. The precise number of inspections of the midpoint necessary for a list of length $2^m - 1$, under "worst case" assumptions, is m.

So if $n = 2^m - 1$, the number of iterations necessary is $\log_2(n + 1)$. This will work as a decent approximation for other values of n as well.

Now, as n gets large, the difference between $\log_2(n + 1)$ and $\log_2 n$ becomes negligible. Moreover, $\log_2 n$ is just a constant multiple of $\log_e n$ or $\log_{10} n$; so the base of the logarithm doesn't matter. We say that the worst-case complexity of this algorithm is logarithmic, and we use the notation $O(\log n)$.

More generally, if $f(n)$ is a function so that, under worst case assumptions, the number of operations in a task of size n is asymptotically equal to a multiple of $f(n)$, we write that the algorithm is $O(f(n))$. This is, unsurprisingly, known as **big O notation**.

By contrast, then, `search` is $O(n)$, which is much worse than $O(\log n)$. Though that doesn't alter the fact that if the data isn't sorted, we're going to have to use it!

What about the best case? Well, here out two algorithms are on all fours. In either case, we may just be lucky, and hit our `element` first time we look. The best-case complexity is represented using **big Ω notation**: we say that both `search` and `binary_search` are $\Omega(1)$.

Just to see this in action, here are versions of our functions with global variables that keep a count of the number of comparisons each method makes:

```
def search(data, element):
    """
    Searches data for element, term by term
    """

    global inspection_count

    # for loop
    for x in data:
```

```
        inspection_count += 1
        # have we found element?
        if x == element:
            return True

    return False

def binary_search(data, element):
    """
    Searches data for element, using binary search
    data is assumed to be in ascending order
    """

    global inspection_count

    # number of data items
    ndata = len(data)

    # no data remaining
    if ndata == 0:
        return False
    else:
        inspection_count += 1
        # find halfway point
        half_length = ndata//2
        # have we found element?
        if data[half_length] == element:
            return True
        # recursively search left half
        elif data[half_length] > element:
            return binary_search(data[0:half_length], element)
        # recursively search right half
        else:
            return binary_search(data[half_length+1:ndata], element)
```

Then define a data set in which our target element is unlikely to appear:

```
from random import randint
data = [randint(1,10**5) for r in range(255)]
```

Then:

```
inspection_count = 0
print(search(data, 11))
print(inspection_count)
```

```
False
255
```

```
inspection_count = 0
print(binary_search(data, 11))
print(inspection_count)
```

```
False
8
```

This matters little when the data set is as small is 255, but for much, much larger data sets there will be an appreciable difference in execution time, becoming more severe as the data lists get longer.

8.1.4 Big Θ

Let's consider another task: **sorting** a list of data. Now, you take a deep dive into this on this week's problem sheet, but for now, let's look at a very simple sorting algorithm; essentially, the one you implemented yourselves a few weeks ago.

This is called **selection sort**, and it works like this, for a data list of length 16:

- Find the minimum of all the items, and place that in position 0, swapping positions with the item that's there now.
- Find the minimum of the items in positions 1, 2, 3 etc, and place that in position 1.
- Find the minimum of the items in positions 2, 3, 4 etc, and place that in position 2.
- Continue until you've placed an item in position 14; the final item will then automatically be in its correct position, namely 15.

Now, the first search for a minimum involves 15 comparisons between data items: you start by comparing item 0 with item 1, then whichever is the smallest of those with item 2, then the mininum of those 3 with item 4, and so on.

The second search for a minimum only involves 14 comparisons, though: you leave item 0 where it is, and compare starting with item 1.

Then there are 13 comparisons, and so on.

All in all, there are $15 + 14 + 13 + \cdots + 1 = 120$ comparisons. More generally, with a data list of length n, there are $\dfrac{n\,(n-1)}{2}$.

(You can view animations of this search algorithm on Blackboard.)

Suppose we decide to measure the complexity of a search method by the number of comparisons of data items (there's room for debate about whether we should, but we generally do). Then what's the worst-case complexity, and what's the best-case complexity?

The answer is the same in both cases. Asymptotically, $\dfrac{n\,(n-1)}{2}$ is a multiple of n^2; for large n, the quadratic term dominates. So this algorithm is both $O(n^2)$ and $\Omega(n^2)$; when that's the case, we say it's $\Theta(n^2)$; this is **big Θ notation**.

Note that the best-case and worst-case performances don't have to be the same, as they are here; for a Θ-complexity to be well defined, they simply have to be, asymptotically as n gets large, multiples of the same function of n. So an algorithm that required $40000\,n^2 + 500000n$ operations in the worst case, and $(n^2 - n)/2$ in the best, would be both $O(n^2)$ and $\Omega(n^2)$, and hence $\Theta(n^2)$.

Here's a listing of an implementation of selection sort, complete with a global variable called `comparison_count`.

```
def selection_sort(data):
    """
    Sorts data in place, using selection sort
    """
    global comparison_count

    # start position goes from 1 to len(data) - 2
    for start_position in range(len(data)-1):
        # search for minimum and record its position
        min_position = start_position
        for index in range(start_position+1, len(data)):
            comparison_count += 1
            # compare minimum so far with next data item
            if data[min_position] > data[index]:
                min_position = index
        # swap positions of minimum and starting elements
        data[start_position], data[min_position] = \
        data[min_position], data[start_position]
```

Then

```
data = [3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
```

```
comparison_count = 0
selection_sort(data)
print(data)
print(comparison_count)
```

```
[3, 5, 7, 12, 17, 18, 19, 21, 22, 24, 26, 27, 29, 31, 34, 40]
120
```

Notice that `data` is sorted **in place**; our function doesn't return any output, but instead sorts `data` as a side-effect, changing its value. For that reason, when writing sort algorithms, it's often convenient to have a "master" data list, from which one makes a copy, but which doesn't itself undergo any change.

Now, you might thing the following would work for that:

```
masterData = \
[3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
data = masterData
```

But in fact it doesn't; as you may remember, in Python, when we set one list equal to another like that, they end up pointing to the same object, meaning that if one changes so does the other. The way to get round that is to use a function called `copy`, which lives in the `copy` module.

```
masterData = \
[3, 34, 12, 22, 27, 17, 31, 29, 40, 24, 21, 19, 7, 18, 26, 5]
from copy import copy
data = copy(masterData)
```

Now we can make as many changes as we like to `data`, and `masterData` will remain unaltered.

8.1.5 A cool Maths example

Challenge 1: write your own version of the `pow` function, for calculating the **modular exponent**: that is, the residue of a^b modulo c.

Here's a first go at that:

```
def mypow1(a, b, c):
    """
    Calculates (a**b) % c by repeated multiplication
    """

    # initialize output
    d = 1

    # loop b times
    for r in range(b):
        # multiply d by a
        d = (d*a) % c

    return d
```

And, I mean, that works fine, as the following example shows:

```
print(pow(123, 456, 789))
print(mypow1(123, 456, 789))
```

```
699
699
```

However, the number of iterations is always exactly b, making this method $\Theta(n)$. We can do a lot better than that!

This implementation works, in essence, by representing b as a binary number: that is, as a sum of powers of 2. So, for example

$$456 = 256 + 128 + 64 + 8.$$

Our plan is this.

- Set $d = 1$.
- Start with a^1. But 1 doesn't appear in b's binary expansion, so do nothing yet.
- Square this number to get a^2, reducing mod c. 2 doesn't appear either, so sit tight.
- Square again to get a^4, reducing mod c. Still no need to do anything.
- Square a third time to get a^8, reducing mod c. This time,as 8 does appear in b's binary expansion, multiply d by this number, reducing mod c.
- Keep squaring till you get to a^{64} and multiply d by that, always reducing mod c.
- Square to get a^{128} and multiply d by that.

- Square to get a^{256} and multiply d by that.

At the end of the process, we'll have multiplied d by $a^8 \times a^{64} \times a^{128} \times a^{256} = a^{456}$, but we've only used a few iterations to do so. This "repeated squaring" algorithm is tamer of large numbers.

Here's an implementation:

```
def mypow2(a, b, c):
    """
    Calculates (a ** b) % c by repeated squaring
    """

    # initialize d
    d = 1

    # while loop
    while b > 0:
        # last binary digit of b
        digit = b % 2

        # if it's 1, multiply d by a
        # otherwise, do nothing
        if digit == 1:
            d = (d*a) % c

        # square a
        a = (a**2) % c

        # halve b using integer divsion
        # to get rid of final binary digit
        b //= 2

    return d
```

Then it works . . .

```
print(pow(123, 456, 789))
print(mypow1(123, 456, 789))
print(mypow2(123, 456, 789))
```

```
699
```

```
699
699
```

. . . , but it's way, way quicker for large values of b; in fact, it's almost as fast as `pow` itself (and you can bet this is how `pow` is implemented).

```
from random import randint
a = randint(10**99,10**100)
b = randint(10**99,10**100)
c = randint(10**99,10**100)
print(a)
print(b)
print(c)
print(pow(a,b,c))
print(mypow2(a,b,c))
```

```
65747536772064001083941071512280362408965691447664185594081691029815679905521968465350991
87386707965931861835426662601898685610202160807469191665027109736741963342704620037403606
79129013959580129784335711019653429753068421862170661450928414621411964109915150395616511
26760265388233163272475055429871890071628127635698429981726373922561413241213384723500932
26760265388233163272475055429871890071628127635698429981726373922561413241213384723500932
```

Just what is its complexity? Well, the number of iterations is equal to the number of times we can integer-divide b by 2 until we hit zero, which is $\log_2 b$. This iteration count doesn't vary, meaning that this algorithm is $\Theta(\log n)$.

9 Intermediate Python: Data, Files and Objects

9.1 Sets, dictionaries and frozensets

We've already met the data structures **list**, **tuple** and **string**. Now for three we didn't get to look at back then.

A **set** is a data structure that ignores (a) order and (b) multiplicity. We create sets in Python by using curly brackets, or by wrapping the word `set` around a list or tuple or string:

```
set1 = {5, 5, 3, 1, 3, 7, 9, 1, 5, 3, 3, 7, 9, 1, 1, 7}
set2 = set([3, 5, 7, 9, 1, 3, 5, 7, 9])
set3 = set((5, 5, 5, 5, 7, 7, 7, 7, 3, 3, 3, 3, 1, 1, 1, 1, 9, 9, 9, 9))
```

```
set4 = set('the quick brown fox jumps over the lazy dog')

print(set1)
print(set2)
print(set3)
print(set4)
```

```
{1, 3, 5, 7, 9}
{1, 3, 5, 7, 9}
{1, 3, 5, 7, 9}
{'p', 'b', 'x', 'q', 'i', ' ', 'w', 'm', 'y', 'c', 'v', 'e', 'o',
 'l', 'r', 'g', 'd', 'f', 's', 'n', 'u', 't', 'j', 'z', 'k', 'a', 'h'}
```

Notice that all multiplicities have been suppressed, and the set of characters seems to be in an entirely arbitrary order. This is by design; order and multiplicity don't matter woth sets. In fact, you could say that the only thing that matters about a set is whether a certain piece of data is an element of it or not; not where it appears or how often.

```
print(set1 == set2)
print(set2 == set3)
print(set3 == set1)
```

```
True
True
True
```

The set-theoretic operations of union and intersection are represented by, respectively, `|` and `&`:

```
primes = {2, 3, 5, 7, 11, 13, 17}
odds = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
print(primes | odds)
print(primes & odds)
```

```
{1, 2, 3, 5, 7, 9, 11, 13, 15, 17, 19}
{3, 5, 7, 11, 13, 17}
```

The command `a - b` gives those elements that are in `a` but not `b`:

```
print(primes - odds)
print(odds - primes)
```

```
{2}
{1, 19, 9, 15}
```

The command `a ^ b` gives those elements that are in `a` or `b`, but not both:

```
print(primes ^ odds)
```

```
{1, 2, 9, 15, 19}
```

To check whether something is an element of a set, use `in` (this also works with lists and tuples, of course):

```
print(2 in primes)
print(2 in odds)
```

```
True
False
```

The `add` method allows you to place additional elements in a set; it's the rough equivalent of `append` for lists.

```
primes.add(23)
print(primes)
```

```
{2, 3, 5, 7, 11, 13, 17, 23}
```

The `remove` method deletes a specific element:

```
primes.remove(23)
print(primes)
```

```
{2, 3, 5, 7, 11, 13, 17}
```

]If you try to `remove` an element that isn't there, an error message is thrown:

```
odds.remove(18)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-4c89fd943bbf> in <module>()
----> 1 odds.remove(18)

KeyError: 18
```

The `discard` method acts like `remove`, except that it throws no error if it fails to find the target element:

```
odds.discard(19)
print(odds)
odds.discard(18)
print(odds)
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17}
{1, 3, 5, 7, 9, 11, 13, 15, 17}
```

The `add`, `remove` and `discard` methods are all pure side-effects; none of them return a value (or rather, they all return the value `None`). By contrast, the `pop` method works a bit like its counterpart for lists; it both removes and returns an element. The difference is that if the case of lists, the element returned is always the last in the list, whereas with sets, it's arbitrary and unpredictable (sometimes, this doesn't matter).

```
print(set4)
print(set4.pop())
print(set4)
```

```
{'p', 'b', 'x', 'q', 'i', ' ', 'w', 'm', 'y', 'c', 'v', 'e', 'o',
'l', 'r', 'g', 'd', 'f', 's', 'n', 'u', 't', 'j', 'z', 'k', 'a', 'h'}
p
{'b', 'x', 'q', 'i', ' ', 'w', 'm', 'y', 'c', 'v', 'e', 'o', 'l',
'r', 'g', 'd', 'f', 's', 'n', 'u', 't', 'j', 'z', 'k', 'a', 'h'}
```

The elements of a set can be iterated across:

```
for n in odds:
    print('({} + 1) / 2 is equal to {}'.format(n, (n+1)//2))
```

```
(1 + 1) / 2 is equal to 1
(3 + 1) / 2 is equal to 2
(5 + 1) / 2 is equal to 3
(7 + 1) / 2 is equal to 4
(9 + 1) / 2 is equal to 5
(11 + 1) / 2 is equal to 6
(13 + 1) / 2 is equal to 7
(15 + 1) / 2 is equal to 8
(17 + 1) / 2 is equal to 9
```

Sets, just like lists, can form the output of a comprehension:

```
new_odds = {2*n-1 for n in range(1,11)}
print(new_odds)
```

```
{1, 3, 5, 7, 9, 11, 13, 15, 17, 19}
```

Notice that lists and sets share this property of being able to form the output of a comprehension, whereas tuples lack it.

A **dictionary** in Python is a data structure that is indexed not by a range of numbers but by a set of *keys* (which can be any kind of Python data, including strings). For example:

```
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}
print(polyhedra['archimedean'])
print(polyhedra['catalan'])
```

```
13
13
```

To access the keys:

```
print(polyhedra.keys())
```

```
dict_keys(['platonic', 'archimedean', 'catalan',
'johnson (simple)', 'johnson', 'kepler-poinsot'])
```

To access the associated values:

```
print(polyhedra.values())
```

```
dict_values([5, 13, 13, 28, 92, 4])
```

To access both:

```
print(polyhedra.items())
```

```
dict_items([('platonic', 5), ('archimedean', 13), ('catalan', 13),
 ('johnson (simple)', 28), ('johnson', 92), ('kepler-poinsot', 4)])
```

The keys, values and items of a dictionary can be iterated across:

```
for poly in polyhedra.keys():
    print('What are the properties of {} polyhedra?'.format(poly))
```

```
What are the properties of platonic polyhedra?
What are the properties of archimedean polyhedra?
What are the properties of catalan polyhedra?
What are the properties of johnson (simple) polyhedra?
What are the properties of johnson polyhedra?
What are the properties of kepler-poinsot polyhedra?
```

```
for n in polyhedra.values():
    print('There are {} of a certain kind of polyhedron.'.format(n))
```

```
There are 5 of a certain kind of polyhedron.
There are 13 of a certain kind of polyhedron.
There are 13 of a certain kind of polyhedron.
There are 28 of a certain kind of polyhedron.
There are 92 of a certain kind of polyhedron.
There are 4 of a certain kind of polyhedron.
```

```
for poly, n in polyhedra.items():
    print('There are {} {} polyhedra.'.format(n, poly))
```

```
There are 5 platonic polyhedra.
There are 13 archimedean polyhedra.
There are 13 catalan polyhedra.
There are 28 johnson (simple) polyhedra.
There are 92 johnson polyhedra.
There are 4 kepler-poinsot polyhedra.
```

9.2 More about mutability

We've met, in the course so far, two types of data structure. Lists and sets are what we call **mutable**: you can change them piecemeal. For example, lists support the operation of *appending*, and sets that of `adding`.

By contrast, tuples and strings are **immutable**. If you assign an immutable value to a variable, the only way to change it is to completely redefine it:

```
tuple1 = tuple(range(1, 16, 2))
print(tuple1)

tuple1 += tuple([17])
print(tuple1)
```

```
(1, 3, 5, 7, 9, 11, 13, 15)
(1, 3, 5, 7, 9, 11, 13, 15, 17)
```

Notice that the command

```
tuple1 += tuple([17])
```

is short for

```
tuple1 = tuple1 + tuple([17])
```

That is, we're assigning an entirely fresh value to the variable `tuple1`.

Individual pieces of data, such as ints, longs or floats, are also immutable in this sense (kind of by default, really, as they don't have separate components that can be changed one by one).

You may be wondering, given that sets are mutable, whether there's a set-like version of a tuple: that is, an immutable data structure that ignores multiplicity and order. There is, though it's a bit unwieldy; it's known as a **frozenset**.

```
frozen_odds = frozenset(new_odds)
print(frozen_odds)
```

```
frozenset({1, 3, 5, 7, 9, 11, 13, 15, 17, 19})
```

You may remember that when we first met this distinction between mutable and immutable data, I was rather vague about why the latter was necessary. Now it can be revealed: we need immutable data because mutable data is simply too unstable to serve as dictionary keys. The keys to any dictionary don't have to be of any particular type, but they must all be immutable.

Mutable data (lists and sets) can't form the elements of a set or a frozenset either (so though we can have a set of frozensets, we can never have a set of sets, or a frozenset of sets). Neither can it serve as dictionary keys. Immutable data can be used in both these ways.

We've noted before an interesting effect when we change the value of mutable data. Check out the following:

```
list1 = list(range(1, 16, 2))
list2 = list1
list3 = list(range(1, 16, 2))
print(list1)
print(list2)
print(list3)
```

```
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 5, 7, 9, 11, 13, 15]
```

Then:

```
list1 = list(range(1, 16, 2))
list2 = list1
list3 = list(range(1, 16, 2))
print(list1)
print(list2)
print(list3)
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17]
[1, 3, 5, 7, 9, 11, 13, 15, 17]
[1, 3, 5, 7, 9, 11, 13, 15]
```

The way to explain this is that the variable names `list1` and `list2` *actually refer to the same piece of data*, whereas `list3`, which was defined separately, refers to a different piece of data that happens, at the moment, to have the same *value*. So when we change the value of `list1` using mutability, the value of `list2` also changes, but that of `list3` doesn't.

Does that mean that if we want to create a list with the same value as another list, but consisting of separate data, we have to assign the value separately like this? That could be pretty laborious. Fortunately, the answer is no. The following works just as well:

```
from copy import copy
list1 = list(range(1, 16, 2))
list2 = list1
list3 = copy(list1)
```

9.3 File input/output

9.3.1 Reading and writing strings

It's quite easy to write strings to, and read them from, external files. Try the following:

```
example_str = "Python and its file I/O make me feel dumbstruck etc."

fo = open("fileio_test.txt", "w")
fo.write(example_str)

fo.close()
```

(Note, `"w"` stands for "write".)

Now, locate the file `fileio_test.txt`, which should be somewhere on your computer (exactly where will depend on your Python set-up). If you read it, the string should be there.

To read the string in again, try:

```
fo = open("fileio_test.txt", "r")
new_str = fo.read()

fo.close()

print(new_str)
```

```
Python and its file I/O make me feel dumbstruck etc.
```

9.3.2 Using the eval function

OK, so that's OK for strings. But suppose we want to write to a file, and read from a file, some other kind of Python object, like a list, or a tuple, or a set, or a dictionary? One way to do that is to convert it to a string before we write it, and convert it back from a string after we're read it.

Converting a Python object into a string is often pretty easy: we can just wrap a `str()` around it. So for example:

```
# define dictionary
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}
# convert into string
polyhedra_str = str(polyhedra)

# open a new file and write to it
fo = open("fileio_test2.txt", "w")
fo.write(polyhedra_str)

# close the file
fo.close()
```

If you check the computer, this file should now have appeared, with the coirrect string in it.

When we read the string back in, we have to convert it back into a dictionary again. There's a function that takes any string and converts it (if it can) into a Python object, and that function is called `eval`. Here's how that works:

```
# open the file and read from it
fo = open("fileio_test2.txt", "r")
new_dict_str = fo.read()

# close file
fo.close()

# convert to dictionary
new_dict = eval(new_dict_str)

# testing:
print(new_dict['platonic'])
```

5

9.3.3 The pickle module

Sometimes using `eval` like that really is the best way to do things (an example might be when you're taking Python data from a URL). But there are several drawbacks to it. One is that the file can end up bigger than it needs to be. The amount of data in our dictionary is quite small, but the amount of memory needed to store it *as a string* might be much greater. (This doesn't matter much for our little toy case here, but if you were storing large amounts of data in a Python dictionary, it might.)

Another drawback is that some Python objects can't, in any case, readily be described using strings (we'll meet some of these later in the course).

There's a module called `pickle` that allows us to write Python objects directly to files, and read them in again. This requires Python to convert the object to and from a readable/writeable form. If you're an old computing hand, you may have met the general idea before, and heard it called *serializing* or *marshalling*; in Python, it usually gets called **pickling**.

Here's how it works for our dictionary example above. First writing:

```
import pickle

# define dictionary
polyhedra = {'platonic' : 5, 'archimedean' : 13, 'catalan' : 13,
        'johnson (simple)': 28, 'johnson' : 92, 'kepler-poinsot': 4}

# open a new file and write a pickled version of the dictionary to it
po = open("fileio_test3.pickle", "wb")
pickle.dump(polyhedra, po)

# close the file
po.close()
```

(Note that the second input we give to the `open` function is not `"w"` but `"wb"`; the 'b' stands for "bytes".)

Now reading:

```
# open the file and read from it
po = open("fileio_test3.pickle", "rb")
new_dict = pickle.load(po)
```

```
# close file
po.close()

# testing:
print(new_dict['kepler-poinsot'])
```

```
4
```

The `dump` and `load` functions in the `pickle` module are the rough `pickle` equivalents of the `write` and `read` functions from core Python. Again, notice it's not `"r"` but `"rb"`.

In the exercises, you take a deeper dive into pickling; you"ll find that just about anything Pythonic can be pickled: not just data like our polyhedra dictionary, but also functions, not to mention the thing you're about to meet, **classes** (more on them later).

9.3.4 Using the exec function

It's even possible to read, and run, Python code. For that we need a way of taking a string that represents Python code, and getting Python to run it as code. You might think that would be a job for `eval`, but in fact there are strict limits on what `eval` will do; essentially, it will *evaluate* strings representing Python *expressions*, but it won't *execute* strings representing Python *instructions*. For that we need something stronger, namely the function called `exec`.

Here's an example. Let's start by writing a code string to an external file (note the use of the triple-quote to break the long string over several lines, and the use of the special character `\n` to represent a line break *in the code*):

```
code_string = """feeling_good = input('Are you feeling good? Y/N ')\n
if feeling_good=='Y': print('Glad to hear it!')\n
else: print('Oh dear!')"""

fo = open("fileio_test4.txt", "w")
fo.write(code_string)

fo.close()
```

Now to read it and run it:

```
fo = open("fileio_test4.txt", "r")
new_code_string = fo.read()
```

```
fo.close()

exec(new_code_string)
```

This has the same effect as running the small script

```
feeling_good = input('Are you feeling good? Y/N ')
if feeling_good=='Y':
    print('Glad to hear it!')
else:
    print('Oh dear!')
```

Notice, in passing, the use of the function `input` to set up a dialog with the user.

9.4 Objects and classes: putting data and programs in the same place

This course has mostly been about the traditional form of programming, in which there is *data*, and in which there are *programs* that operate on data, and the two are entirely separate. But Python also supports what's called **object-oriented programming** (OOP), in which data, and the things that operate on data, are combined into a single entity.

If you carry on with computing next year, there'll be much, much more about OOP. But let's have a taster now.

In one way, an **object** can be thought of a bit like a dictionary, in that it consists in part of data indexed non-numerically. But an object also consists of functions that work with this data. The pieces of data are known in Python as **attributes** and the functions are called **methods**.

To create an object is a two-stage process: first one defines what's called a **class**, and then one creates an **instance** of that class.

Let's look at an example. Our aim here is to build a kind of toy SymPy: a way of representing and processing symbolic mathematical expressions. For that we'll need a new type of Python data, and some ways of doing things with them.

Challenge 1: set up a new type of Python data called `OperatorExpression`, which is capable of representing expressions like `x + 2` or `5 ** 7`.

The new type of data we'll be writing is called a **class**, and the associated representation processes are called **methods**. Here's the code: Here's a first go at that:

```
class OperatorExpression(object):
    """Expression of the form a <operator> b, where <operator> is
    +, -, *, /, //, ** or %"""
```

```
    def __init__(self, root, left, right):
        self.contents = (root, left, right)
```

Lots to unpack here. First, notice the syntax of the first line. The keyword `class` lets Python know to expect a class, and we've specified its name, `OperatorExpression`. (The convention with classes is to use "camel case", in which every word in the class name starts with a capital letter; we don't normally use underscores to separate words, though it's perfectly possible to do so.)

Then there's the word `object` in brackets. `object`. This lets Python know that this new class is an extension of the most general Python class, the `object`.

Next we have something that every class must have: an `__init__` method, which tells Python what to do when we **instantiate** our class; that is, create data of this new type. This particular `__init__` method shows that our class carries around one piece of data, called `contents`: a tuple containing a root, a left and a right. The root will represent our operation, whereas the left and right will represent the two terms we're operating on. so that `x + 2` could be represented by the tuple `('plus', 'x', 2)`. This piece of data attached to the class is called an **attribute**.

Let's test our class by setting up an **instance** of it:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.contents)
```

```
('plus', 'x', 2)
```

Notice that although our `__init__` method seems to have four arguments, `self`, `root`, `left` and `right`, we only include the values of `root`, `left` and `right` when we set up an instance of our class. This will be a theme: the `self` never appears as an explicit argument; instead, it signals to Python that this is a method rather than an ordinary **function**. More on that later.

Now, this is all very well, but so far not very impressive. We'd like to be able to **do** things with our new class of Python object. Let's start by representing it in a more human-readable way.

Challenge 2: write a method called `prefixForm`, in which `x + 2` is represented as the string `'plus(x, 2)'` and `5 ** 7` is represented as `'power(5, 7)'`.

We add the following to our code:

```
    def prefixForm(self):
        root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
        return root+'('+str(left)+', '+str(right)+')'
```

Then the following works:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.prefixForm())
```

```
plus(x, 2)
```

The main thing to notice here is the syntax of calling our new method. As with `__init__`, the `self` is ignored; this method appears, when we call it, to have no arguments. Also, we type not `prefixForm(expr1)` but `expr1.prefixForm()`; this, as you know, is the typical syntax for a method rather than a function.

Let's now give users an alternative, even more human-readable representation:

Challenge 3: write a method called `infixForm`, in which `x + 2` is represented as the string `'(x) + (2)'` and `5 ** 7` is represented as `'(5) ** (7)'`.

We add the following to our code:

```
    def infixOperator(self):
        root = self.contents[0]
        if root == 'plus':
            return '+'
        elif root == 'subtract':
            return '-'
        elif root == 'times':
            return '*'
        elif root == 'divide':
            return '/'
        elif root == 'intdivide':
            return '//'
        elif root == 'power':
            return '**'
        elif root == 'mod':
```

```
            return '%'

    def infixForm(self):
        root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
        return '('+str(left)+') '+self.infixOperator()+' ('+str(right)+')'
```

Then the folliowing works:

```
expr1 = OperatorExpression('plus', 'x', 2)
print(expr1.infixForm())
```

```
(x) + (2)
```

Those parentheses are a bit annoying, but getting rid of them in a robust way is a bit of a can of worms, so let's live with them.

Let's get more ambitious

Challenge 4: write methods called `subs`, in which you can substitute a value for a variable, and `evaluate`, which allows you to take numerical expressions like `5 ** 7` and find their values.

We add the following to our code:

```
    def subs(self, var, val):
        root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
        if left == var and right == var:
            return OperatorExpression(root, val, val)
        elif left == var:
            return OperatorExpression(root, val, right)
        elif right == var:
            return OperatorExpression(root, left, val)
        else:
            return self

    def evaluate(self):
        root, left, right = self.contents[0], self.contents[1], \
```

```
self.contents[2]
        if root == 'plus':
            return left + right
        elif root == 'subtract':
            return left - right
        elif root == 'times':
            return left * right
        elif root == 'divide':
            return left / right
        elif root == 'intdivide':
            return left // right
        elif root == 'power':
            return left ** right
        elif root == 'mod':
            return left % right
```

Then the following work:

```
expr1 = OperatorExpression('plus', 'x', 2)
expr2 = expr1.subs('x', 5)
print(expr2.infixForm())
print(expr2.evaluate())
```

```
(5) + (2)
7
```

Notice the way the `subs` method seems to have three arguments, `self`, `var` and `val`, but has only two, `var` and `val`, when we call it; as always, the `self` argument is never actually used, but serves only to badge this as a method rather than a function.

Let's be *really* ambitious now.

Challenge 5: write a class called `ExpressionTree` that is capable of representing not only expressions like `(x + 1)` but also more complicated expressions like `(x + 1) ** 2 - 5`

An `ExpressionTree` is, in effect, an `OperatorExpression` whose left and right branches are both `ExpressionTrees`. So its natural structure is recursive, meaning that the secret to making it work is making all our *methods* recursive.

We'll also use a trick called **inheritance**, which allows us to set up our `ExpressionTree` class as an extension to our `OperatorExpression` class. It will then inherit any methods

and attributes we don't override. In particular, we won't need to rewrite the `infixOperator` method; our new class can simply inherit it.

When you look at our new class in all its glory, note the way the methods have all become recursive. For example, the `prefixForm` method is now

```
    def prefixForm(self):
        if len(self.contents) == 1:
            return str(self.contents[0])
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return root+'('+left.prefixForm()+', '+right.prefixForm()+')'
```

If you ask for the `prefixForm` of an `ExpressionTree` whose contents are something like `(1,)` or `('x',)`, you just get the string `'1'` or `'x'`. This is the base case. If you ask for the `prefixForm` of anything with a `root`, a `left` and a `right`, the method calls itself on `left` and `right`; this is the recursion step. The other methods have been defined in a similar recursive way.

Here's the whole thing:

```
class ExpressionTree(OperatorExpression):
    """Algebraic expression as a tree"""

    def __init__(self, root, left=None, right=None):
        if left==None or right==None:
            self.contents = (root,)
        else:
            self.contents = (root, left, right)

    def prefixForm(self):
        if len(self.contents) == 1:
            return str(self.contents[0])
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return root+'('+left.prefixForm()+', '+right.prefixForm()+')'

    def infixForm(self):
        if len(self.contents) == 1:
            return str(self.contents[0])
```

```
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return '('+left.infixForm()+') '+self.infixOperator()+' ('+\
right.infixForm()+')'

    def evaluate(self):
        if len(self.contents) == 1:
            if isinstance(self.contents[0], str):
                return eval(self.contents[0])
            else:
                return self.contents[0]
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            if root == 'plus':
                return left.evaluate() + right.evaluate()
            elif root == 'subtract':
                return left.evaluate() - right.evaluate()
            elif root == 'times':
                return left.evaluate() * right.evaluate()
            elif root == 'divide':
                return left.evaluate() / right.evaluate()
            elif root == 'intdivide':
                return left.evaluate() // right.evaluate()
            elif root == 'power':
                return left.evaluate() ** right.evaluate()
            elif root == 'mod':
                return left.evaluate() % right.evaluate()

    def subs(self, var, val):
        if len(self.contents) == 1:
            if self.contents[0] == var:
                return ExpressionTree(val)
            else:
                return self
        else:
            root, left, right = self.contents[0], self.contents[1], \
self.contents[2]
            return ExpressionTree(root, left.subs(var, val), \
right.subs(var, val))
```

Notice that this class supports two kinds of expression tree; ones where the `contents` tuple is of length 1 (representing expressions like `x` or 2), and ones where it's of length 3 (representing anything more complicated with a "root, left, right") structure. It does this by assigning the default values `None` and `None` to the `left` and `right` arguments in the `__init__` method; if these values are not specified by the user, `__init__` simply sets up `contents` as a tuple of length 1.

Here's how we might use it:

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(1)
expr3 = ExpressionTree(2)
expr4 = ExpressionTree('plus', expr1, expr2)
expr5 = ExpressionTree('power', expr4, expr3)
print(expr5.prefixForm())
print(expr5.infixForm())
```

```
power(plus(x, 1), 2)
((x) + (1)) ** (2)
```

Then

```
expr6 = expr5.subs('x', 2)
print(expr6.prefixForm())
print(expr6.infixForm())
print(expr6.evaluate())
```

```
power(plus(2, 1), 2)
((2) + (1)) ** (2)
9
```

This is finally beginning to feel potentially useful!

Now for the icing on the cake:

Challenge 6: introduce some basic simplification to this class, so that `x + 0` becomes just `x`, and so on.

For this we want, in effect, to do some pre-processing on the input, so that if the user sets up `ExpressionTree('plus', 'x', 0)` this is automatically simplified to `ExpressionTree('x')`.

One way to do this is by changing the `__init__` method to incorporate a whole host of simplification rules:

```
    def __init__(self, root, left=None, right=None):
       if left==None or right==None:
           self.contents = (root,)
       else:
           if root == 'plus' and left.contents == (0,):
               self.contents = right.contents
           elif root == 'plus' and right.contents == (0,):
               self.contents = left.contents
           elif root == 'subtract' and right.contents == (0,):
               self.contents = left.contents
           elif root == 'times' and left.contents == (1,):
               self.contents = right.contents
           elif root == 'times' and right.contents == (1,):
               self.contents = left.contents
           elif root == 'divide' and right.contents == (1,):
               self.contents = left.contents
           elif root == 'intdivide' and right.contents == (1,):
               self.contents = left.contents
           elif root == 'power' and right.contents == (1,):
               self.contents = left.contents
           elif root == 'power' and right.contents == (0,):
               self.contents = (1,)
           elif root == 'power' and left.contents == (1,):
               self.contents = (1,)
           else:
               self.contents = (root, left, right)
```

Now the following happens, for example:

```
expr1 = ExpressionTree('x')
expr2 = ExpressionTree(5)
expr3 = ExpressionTree(2)
expr4 = ExpressionTree('plus', expr1, expr2)
expr5 = ExpressionTree('power', expr4, expr3)
print(expr5.prefixForm())
print(expr5.infixForm())
```

```
power(plus(x, 5), 2)
((x) + (5)) ** (2)
```

Then

```
expr6 = expr5.subs('x', 0)
print(expr6.infixForm())
```

now delivers not

```
((0) + (5)) ** (2)
```

but simply

```
(5) ** (2)
```

The expression has undergone some automatic simplification!

In fact, this isn't absolutely the best way to do it; the best way to do it uses something called **properties**, which you'll learn about in next year's course if you choose to take it.

10 Data analysis and pandas

A key use of computing in the modern age is **data analysis**: extracting information from data sets, creating models, etc. For example, this lies at the heart of the current excitement around **machine learning**.

You can do quite a lot of data analysis using just core Python plus NumPy, but Python does have specialised data analysis functionality. The most important data analysis module is called **pandas**.

The pandas module is built on two basic data structures: **Series** and **DataFrame**. (There's another, called **Panel**, but it's on the way out and we'll ignore it.) You can think of a DataFrame as being a bit like a spreadsheet: rows and columns of data. You can think of a Series as being a single column of a spreadsheet.

10.1 The Series structure

A pandas Series is in some respects a little like a dictionary: it consists of **index keys** and **values**. The first thing we notice, though, is that in Jupyter notebooks, a Series will automatically output, or print, in tabulated form. Here's an example:

```
platonic_s = pd.Series([4,6,8,12,20],
                       index=['tetrahedron',
                              'cube',
                              'octahedron',
                              'dodecahedron',
                              'icosahedron']
```

```
                          )
platonic_s
```

```
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
dtype: int64
```

An alternative way to set up the same series is using a dictionary:

```
platonic_dict = {'tetrahedron': 4,
                 'cube': 6,
                 'octahedron': 8,
                 'dodecahedron': 12,
                 'icosahedron': 20}

platonic_s2 = pd.Series(platonic_dict)

platonic_s2
```

```
cube             6
dodecahedron    12
icosahedron     20
octahedron       8
tetrahedron      4
dtype: int64
```

Notice that if we do it like this, the Series is automatically sorted into index order. If we now wish our first series was sorted like that, we can type

```
platonic_s = platonic_s.sort_index()

platonic_s
```

```
cube             6
dodecahedron    12
icosahedron     20
octahedron       8
tetrahedron      4
dtype: int64
```

If, on the other hand, we wish our second Series was in value order, we can type

```
platonic_s2 = platonic_s2.sort_values()

platonic_s2
```

```
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
dtype: int64
```

Something to notice straight away is that the `sort_index` and `sort_values` methods for Series aren't like Python's native `sort` method: they don't sort in place, as a side effect. Instead, they **return** the sorted Series, as a value. This is baked into the design of pandas; pretty much all pandas functions and methods work like that.

The equivalent of the `keys` and `values` methods for dictionaries aren't methods at all in the case of Series; they're implemented as `attributes`:

```
print(platonic_s2.index)
print(platonic_s2.values)
```

```
Index(['tetrahedron', 'cube', 'octahedron', 'dodecahedron',
'icosahedron'], dtype='object')
[ 4  6  8 12 20]
```

Notice that the `index` attribute is returned as a special `Index` object, whereas the `values` attribute is returned as a NumPy array. Pandas is built on top of NumPy, and the two are intimately linked.

10.2 Mutability and homogeneity

So, pandas Series are a bit like dictionaries. But they're not similar in every respect. Dictionaries are extremely flexible things; Series are, by design, less so.

First, a similarity. I can, if I choose to be mathematically incorrect, change one of my values in `platonic_dict`:

```
platonic_dict['cube'] = 600

platonic_dict
```

```
{'tetrahedron': 4,
 'cube': 600,
 'octahedron': 8,
 'dodecahedron': 12,
 'icosahedron': 20}
```

I can do the same with a Series, and the syntax is exactly the same:

```
platonic_s2['cube'] = 600

platonic_s2
```

```
tetrahedron       4
cube            600
octahedron        8
dodecahedron     12
icosahedron      20
dtype: int64
```

Let's quickly change them back before anybody notices:

```
platonic_dict['cube'] = 6
platonic_s2['cube'] = 6
```

Now, a difference. Dictionaries support data of mixed values:

```
platonic_dict['cube'] = 6.0

platonic_dict
```

```
{'tetrahedron': 4,
 'cube': 6.0,
 'octahedron': 8,
 'dodecahedron': 12,
 'icosahedron': 20}
```

Series don't; the data must be **homogeneous**:

```
platonic_s2['cube'] = 6.0

platonic_s2
```

```
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
dtype: int64
```

Series can represent any kind of data (as long as it's all of the same type), but they were designed to represent **time series**: that is, data that represents the change of something over time. Here's some data representing closing prices of the New York stock exchange over a period of a week in 2019:

```
import numpy as np
dates = np.array(['2019-03-11', '2019-03-12', '2019-03-13',
       '2019-03-14',  '2019-03-15'])
closing_prices = np.array([20.889999, 20.370001, 20.1,
       19.950001, 19.690001])

prices_s = pd.Series(closing_prices, index=dates)
prices_s
```

```
2019-03-11    20.889999
2019-03-12    20.370001
2019-03-13    20.100000
2019-03-14    19.950001
2019-03-15    19.690001
dtype: float64
```

We can perform various types of analysis on this data–but this stuff really comes into its own when we talk about DataFrames, so let's defer that discussion till then. Instead, let's content ourselves with a line plot and a bar chart:

```
%matplotlib inline
```

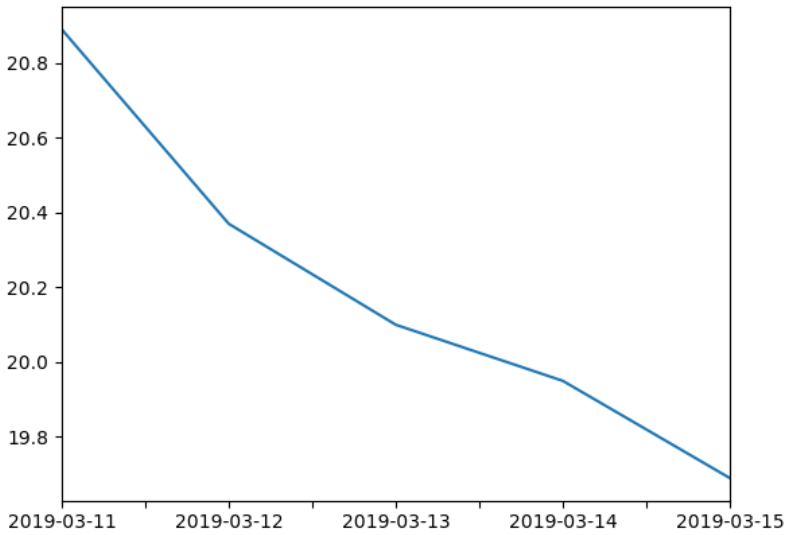


Figure 1: Line plot of closing stock prices

```
prices_s.plot()
```

```
prices_s.plot.bar()
```

Shown in Figures 1 and 2 respectively.

One more thing to note: if we don't specify an index when setting up a series, the default integer indexing will be used:

```
closing_prices = np.array([20.889999, 20.370001, 20.1,
       19.950001, 19.690001])

prices_s2 = pd.Series(closing_prices)
prices_s2
```

```
0    20.889999
1    20.370001
2    20.100000
```

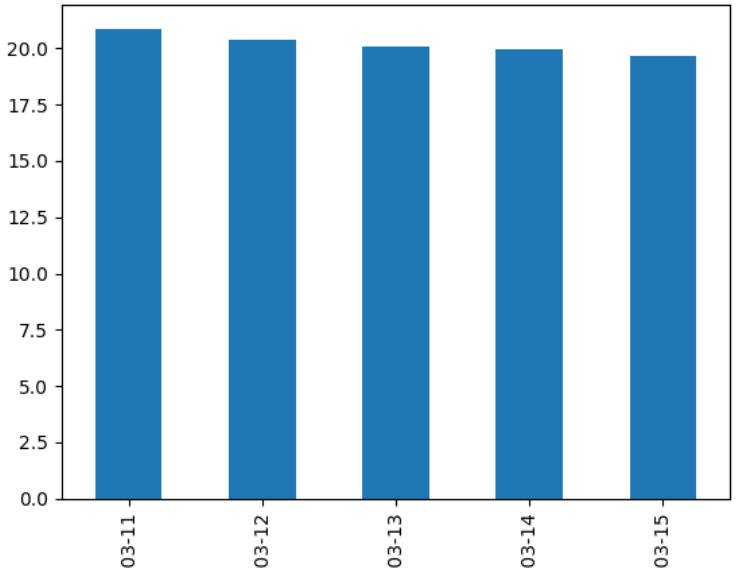


Figure 2: Bar chart of closing stock prices

```
3    19.950001
4    19.690001
dtype: float64
```

10.3 DataFrames

A pandas DataFrame is a collection of pandas Series that share an index. Well, that's one way of thinking about it: another is that it's kind of like a spreadsheet, in which the index provides the row headings and the column headings specify the various series.

Notice that because each column of a DataFrame corresponds to a Series, each column must be homogeneous; however, the data type is perfectly free to vary from column to column.

Let's set one up. One way to do that is using a dictionary, in which the keys are the column headings, and the values come from lists, tuples, arrays or Series:

```
platonic_dict2 = {
    'name': ['tetrahedron',
             'cube',
             'octahedron',
             'dodecahedron',
             'icosahedron'],
```

```
    'faces': [4, 6, 8, 12, 20],
    'vertices': [4, 8, 6, 20, 12],
    'edges': [6, 12, 12, 30, 30],
    'face shape': ['triangle',
                   'square',
                   'triangle',
                   'pentagon',
                   'triangle']
}

platonic_df = pd.DataFrame(platonic_dict2)

platonic_df
```

	edges	face shape	faces	name	vertices
0	6	triangle	4	tetrahedron	4
1	12	square	6	cube	8
2	12	triangle	8	octahedron	6
3	30	pentagon	12	dodecahedron	20
4	30	triangle	20	icosahedron	12

Now, it's automatically sorted the columns, and we may not want that. Here's how we fix it:

```
platonic_df = platonic_df[
    ['name', 'faces', 'vertices', 'edges', 'face shape']
]

platonic_df
```

	name	faces	vertices	edges	face shape
0	tetrahedron	4	4	6	triangle
1	cube	6	8	12	square
2	octahedron	8	6	12	triangle
3	dodecahedron	12	20	30	pentagon
4	icosahedron	20	12	30	triangle

Also, it's used the default indexing. Suppose we want to index by name instead:

```
platonic_df = platonic_df.set_index('name')

platonic_df
```

name	faces	vertices	edges	face shape
tetrahedron	4	4	6	triangle
cube	8	4	20	square
octahedron	8	6	12	triangle
dodecahedron	12	20	30	pentagon
icosahedron	20	12	30	triangle

We can get a particular column in the form of a Series:

```
platonic_df['faces']
```

```
name
tetrahedron      4
cube             6
octahedron       8
dodecahedron    12
icosahedron     20
Name: faces, dtype: int64
```

We can add a column to the DataFrame:

```
platonic_df['Euler check'] = platonic_df['faces'] + \
platonic_df['vertices'] - platonic_df['edges']

platonic_df
```

name	faces	vertices	edges	face shape	Euler check
tetrahedron	4	4	6	triangle	2
cube	8	4	20	square	2
octahedron	8	6	12	triangle	2
dodecahedron	12	20	30	pentagon	2
icosahedron	20	12	30	triangle	2

Notice that when we do calculations with them, Series objects behave just like NumPy arrays: we can add, subtract, multiply or divide them in a wholly vectorized way.

10.4 Reading from a file

It's actually fairly rare, however, that we'd want to set up a DataFrame ourselves from lists like this. More often, we'd want to read it in from an external file. Here's a partial printout of a file called `births_and_deaths.csv`, which is available on Blackboard; it shows births and deaths of males and females in a particular locality (one that seems a bit rigid about gender!)

```
Quarter,Male Live Births,Female Live Births,Male Deaths,Female Deaths
2000Q1,7639,7139,3346,3070
2000Q2,7365,6866,3372,3178
2000Q3,7174,6843,3675,3511
2000Q4,6979,6600,3357,3151
2001Q1,7496,7232,3231,3070
2001Q2,7101,6796,3481,3392
2001Q3,6873,6783,3914,4000
2001Q4,6863,6655,3357,3380
2002Q1,6891,6757,3265,3258
...
```

This is called **comma-separated values** format (or CSV), and it's a common convention for storing data in files.

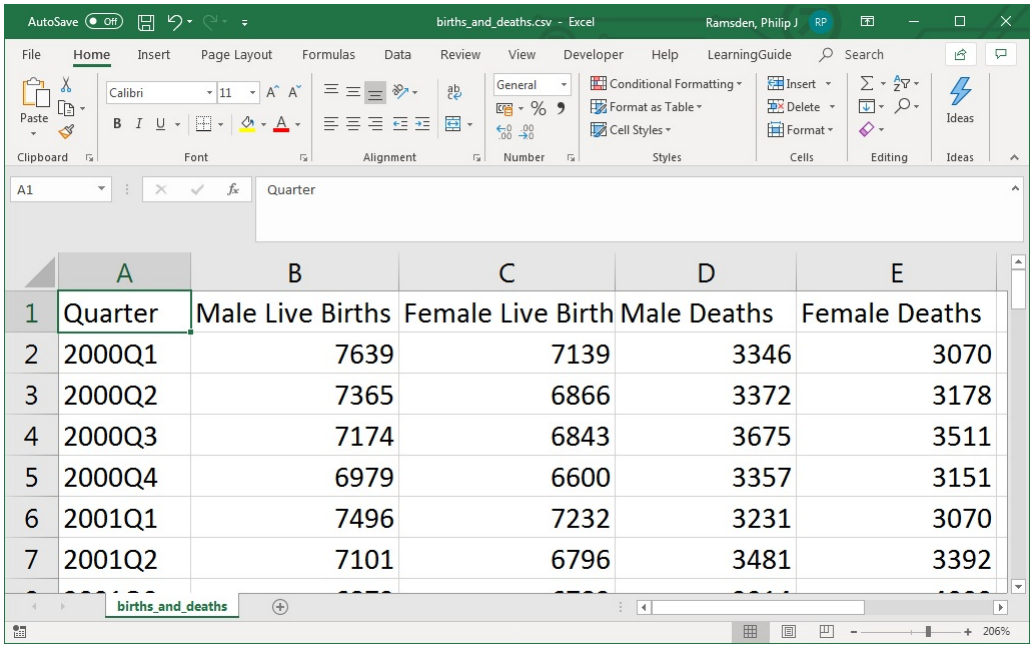
Figure 3 shows what the same file looks like, displayed in Excel (which can read CSV).

And here's how we read it in as a DataFrame; this needs the file to be in the same folder as our notebook.

```
births_and_deaths_df = pd.read_csv('births_and_deaths.csv')
```

Let's display it; but not the whole file, which runs to 52 rows. Instead, we'll use a really handy pandas method called `head`, which just displays the first few:

```
births_and_deaths_df.head()
```



	A	B	C	D	E
1	Quarter	Male Live Births	Female Live Birth	Male Deaths	Female Deaths
2	2000Q1	7639	7139	3346	3070
3	2000Q2	7365	6866	3372	3178
4	2000Q3	7174	6843	3675	3511
5	2000Q4	6979	6600	3357	3151
6	2001Q1	7496	7232	3231	3070
7	2001Q2	7101	6796	3481	3392

Figure 3: A CSV file in Excel

	Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
0	2000Q1	7639	7139	3346	3070
1	2000Q2	7365	6866	3372	3178
2	2000Q3	7174	6843	3675	3511
3	2000Q4	6979	6600	3357	3151
4	2001Q1	7496	7232	3231	3070

Again, we'd probably like to override the default indexing, and have the 'Quarter' values form the index. We can actually achieve that at the read-in stage:

```
births_and_deaths_df = pd.read_csv('births_and_deaths.csv',
                                   index_col = 'Quarter')

births_and_deaths_df.head().to_latex()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000Q1	7639	7139	3346	3070
2000Q2	7365	6866	3372	3178
2000Q3	7174	6843	3675	3511
2000Q4	6979	6600	3357	3151
2001Q1	7496	7232	3231	3070

Finally, we can convert the index to the pandas DateTime format, which offers all sorts of benefits:

```
births_and_deaths_df.index = pd.to_datetime(births_and_deaths_df.index)

births_and_deaths_df.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	7365	6866	3372	3178
2000-07-01	7174	6843	3675	3511
2000-10-01	6979	6600	3357	3151
2001-01-01	7496	7232	3231	3070

10.5 Operations on DataFrames

There are now a whole host of things we can do with our data. Suppose we want to know the percentage changes quarter-on-quarter:

```
births_and_deaths_pc = births_and_deaths_df.pct_change()

births_and_deaths_pc.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	NaN	NaN	NaN	NaN
2000-04-01	-0.035869	-0.038241	0.007770	0.035179
2000-07-01	-0.025933	-0.003350	0.089858	0.104783
2000-10-01	-0.027181	-0.035511	-0.086531	-0.102535
2001-01-01	0.074079	0.095758	-0.037534	-0.025706

Those NaNs are a bit annoying; let's "cleanse" the data by converting them to zeros:

```
births_and_deaths_pc = births_and_deaths_pc.fillna(0)

births_and_deaths_pc.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	0.000000	0.000000	0.000000	0.000000
2000-04-01	-0.035869	-0.038241	0.007770	0.035179
2000-07-01	-0.025933	-0.003350	0.089858	0.104783
2000-10-01	-0.027181	-0.035511	-0.086531	-0.102535
2001-01-01	0.074079	0.095758	-0.037534	-0.025706

Let's create a DataFrame with running totals in all four columns:

```
births_and_deaths_tot = births_and_deaths_df.cumsum()

births_and_deaths_tot.head()
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	15004	14005	6718	6248
2000-07-01	22178	20848	10393	9759
2000-10-01	29157	27448	13750	12910
2001-01-01	36653	34680	16981	15980

How about a table of four-quarter rolling averages?

```
births_and_deaths_rolling = births_and_deaths_df.rolling(4).mean()

births_and_deaths_rolling.head(8)
```

Quarter	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-01-01	NaN	NaN	NaN	NaN
2000-04-01	NaN	NaN	NaN	NaN
2000-07-01	NaN	NaN	NaN	NaN
2000-10-01	7289.25	6862.00	3437.50	3227.50
2001-01-01	7253.50	6885.25	3408.75	3227.50
2001-04-01	7187.50	6867.75	3436.00	3281.00
2001-07-01	7112.25	6852.75	3495.75	3403.25
2001-10-01	7083.25	6866.50	3495.75	3460.50

Note the use of the optional argument to the `head` method, giving us some extra rows in the display.

And then a really useful feature, which we can only use because we converted the index to DateTime format. We can **resample** the DataFrame at a different frequency; let's say annually. We need to specify how to combine the data, and there are various ways to do this; in this case it makes sense to add the figures for all four quarters in each year. Here's how we do that:

```
births_and_deaths_resampled = births_and_deaths_df.resample('A').sum()

births_and_deaths_resampled.index = \
```

```
births_and_deaths_resampled.index.rename('Year')

births_and_deaths_resampled
```

Year	Male Live Births	Female Live Births	Male Deaths	Female Deaths
2000-12-31	29157	27448	13750	12910
2001-12-31	28333	27466	13983	13842
2002-12-31	27577	26444	14023	14042
2003-12-31	28820	27314	14020	13990
2004-12-31	29744	28329	14075	14344
2005-12-31	29546	28199	13431	13603
2006-12-31	30240	28953	13924	14321
2007-12-31	33013	31031	14275	14247
2008-12-31	33102	31241	14535	14653
2009-12-31	32112	30431	14480	14484
2010-12-31	32904	30993	14223	14215
2011-12-31	31476	29927	14823	15259
2012-12-31	31243	29935	15056	15043

The 'A' stands for 'annually'. If we'd wanted once every three years, we'd have used '3A'. Alternative **sampling rules** are things like '6M' (for six months), '2H' (for two hours), '20min', '10S' and so on.

Let's suppose we decide the term "Live" is unnecessary. How can we rename our columns? One way is this:

```
births_and_deaths_df = births_and_deaths_df.rename(
    columns = lambda str: str.replace('Live ', ''))

births_and_deaths_df.head()
```

Quarter	Male Births	Female Births	Male Deaths	Female Deaths
2000-01-01	7639	7139	3346	3070
2000-04-01	7365	6866	3372	3178
2000-07-01	7174	6843	3675	3511
2000-10-01	6979	6600	3357	3151
2001-01-01	7496	7232	3231	3070

Finally, let's produce an **aggregate table** showing totals and means:

```
births_and_deaths_agg = births_and_deaths_df.aggregate(['sum','mean'])

births_and_deaths_agg
```

	Male Births	Female Births	Male Deaths	Female Deaths
sum	397267.00	377711.000000	184598.000000	184953.000000
mean	7639.75	7263.673077	3549.961538	3556.788462

10.6 Plots

The pandas module supports a whole host of specialist plotting tools. The general-purpose one is `plot`, which produces a line graph:

```
births_and_deaths_df.plot()
```

A bar chart for the first few rows of our DataFrame:

```
births_and_deaths_df.head().plot.bar()
```

A histogram for the 'Female Births' column:

```
births_and_deaths_df['Female Births'].plot.hist()
```

A histogram for the 'Female Births' and 'Male Births' columns:

```
births_and_deaths_df['Female Births'].plot.hist()
```

A scatter diagram for male and female births:

```
births_and_deaths_df.plot.scatter(x='Male Births',y='Female Births')
```

Box-and-whisker plots for male births and female births:

```
births_and_deaths_df[['Female Births','Male Births']].plot.box()
```

There are many others. These are all shown in Figure 4.

10.7 Descriptive statistics

DataFrames support a wide variety of descriptive statistics, returned in the form of Series:

```
births_and_deaths_df.mean()
```

```
Male Births      7639.750000
Female Births    7263.673077
Male Deaths      3549.961538
Female Deaths    3556.788462
dtype: float64
```

```
births_and_deaths_df.median()
```

```
Male Births      7635.0
Female Births    7307.0
Male Deaths      3498.5
Female Deaths    3487.5
dtype: float64
```

```
births_and_deaths_df.std()
```

```
Male Births      506.576548
Female Births    445.757682
Male Deaths      272.253844
Female Deaths    329.419890
dtype: float64
```

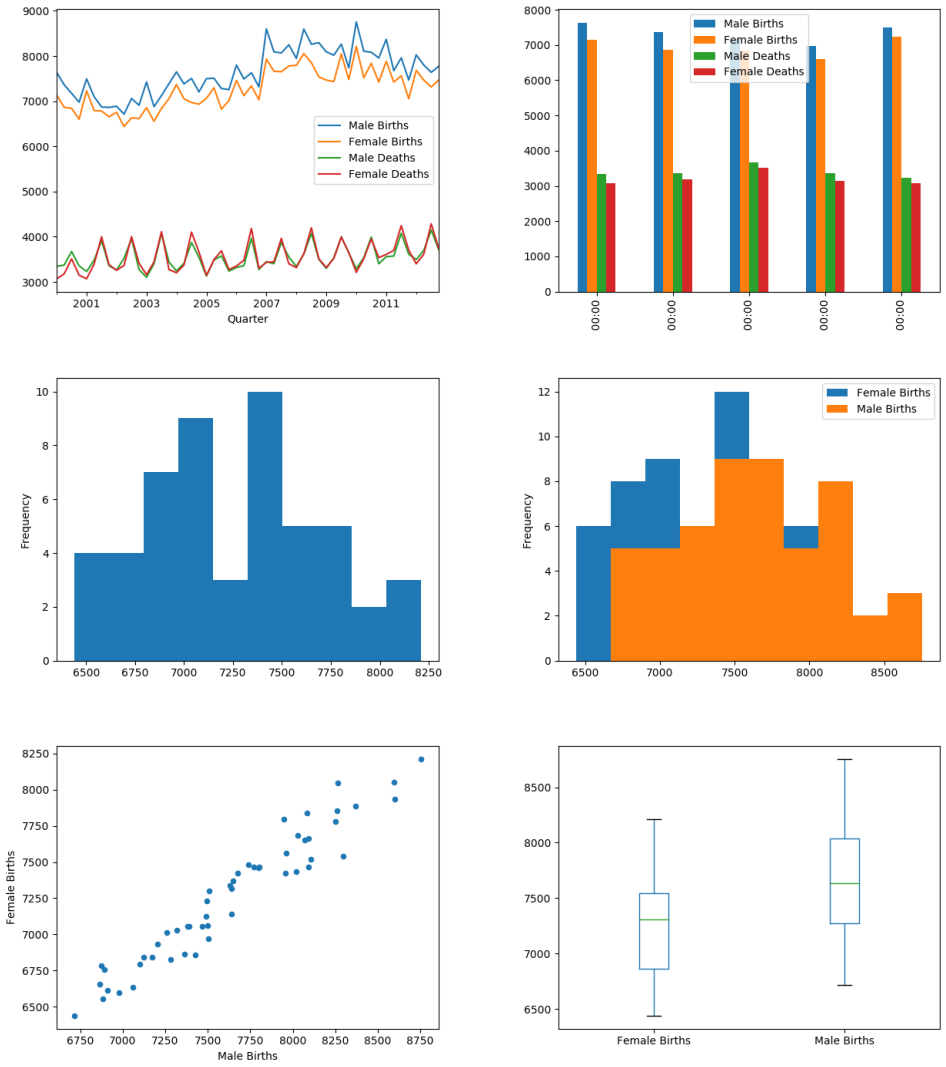


Figure 4: Plots of births and deaths data

```
births_and_deaths_df.quantile(0.75)
```

```
Male Births      8037.25
Female Births    7544.25
Male Deaths      3680.00
Female Deaths    3699.75
Name: 0.75, dtype: float64
```

```
births_and_deaths_df.quantile(0.75)-births_and_deaths_df.quantile(0.25)
```

```
Male Births      762.25
Female Births    680.00
Male Deaths      325.75
Female Deaths    380.50
dtype: float64
```

A DataFrame summarising a lot of this information is returned by the `describe` method:

```
births_and_deaths_df.describe()
```

	Male Births	Female Births	Male Deaths	Female Deaths
count	52.000000	52.000000	52.000000	52.000000
mean	7639.750000	7263.673077	3549.961538	3556.788462
std	506.576548	445.757682	272.253844	329.419890
min	6713.000000	6438.000000	3103.000000	3070.000000
25%	7275.000000	6864.250000	3354.250000	3319.250000
50%	7635.000000	7307.000000	3498.500000	3487.500000
75%	8037.250000	7544.250000	3680.000000	3699.750000
max	8756.000000	8212.000000	4149.000000	4287.000000

10.8 Queries, pivot tables and groupby

Let's load another DataFrame using `read_csv`; this one concerns sales of real estate in California. We'll look at several rows so you can get an idea of what the data shows; it's quite wide, so I've split it into two listings.

```
real_estate_df = pd.read_csv('real_estate.csv', index_col = 'sale_date')

real_estate_df.index = pd.to_datetime(real_estate_df.index)

real_estate_df.head(10)
```

sale_date	street	city	zip
2008-05-21	3526 HIGH ST	SACRAMENTO	95838
2008-05-21	51 OMAHA CT	SACRAMENTO	95823
2008-05-21	2796 BRANCH ST	SACRAMENTO	95815
2008-05-21	2805 JANETTE WAY	SACRAMENTO	95815
2008-05-21	6001 MCMAHON DR	SACRAMENTO	95824
2008-05-21	5828 PEPPERMILL CT	SACRAMENTO	95841
2008-05-21	6048 OGDEN NASH WAY	SACRAMENTO	95842
2008-05-21	2561 19TH AVE	SACRAMENTO	95820
2008-05-21	11150 TRINITY RIVER DR Unit 114	RANCHO CORDOVA	95670
2008-05-21	7325 10TH ST	RIO LINDA	95673

sale_date	beds	baths	sq__ft	type	price
2008-05-21	2	1	836	Residential	59222
2008-05-21	3	1	1167	Residential	68212
2008-05-21	2	1	796	Residential	68880
2008-05-21	2	1	852	Residential	69307
2008-05-21	2	1	797	Residential	81900
2008-05-21	3	1	1122	Condo	89921
2008-05-21	3	2	1104	Residential	90895
2008-05-21	3	1	1177	Residential	91002
2008-05-21	2	2	941	Condo	94905
2008-05-21	3	2	1146	Residential	98937

Let's start by finding all sales over $600000; we'll just show the city, square feet and price

```
real_estate_df.query('price > 600000')[['city', 'sq__ft', 'price']]
```

sale_date	city	sq__ft	price
2008-05-21	EL DORADO HILLS	0	606238
2008-05-21	SACRAMENTO	2325	660000
2008-05-21	EL DORADO HILLS	0	830000
2008-05-20	ROSEVILLE	3838	613401
2008-05-20	ROSEVILLE	0	614000
2008-05-20	FAIR OAKS	2846	680000
2008-05-20	SACRAMENTO	2484	699000
2008-05-20	LOOMIS	1624	839000
2008-05-19	FOLSOM	2660	636000
2008-05-19	CARMICHAEL	3357	668365
2008-05-19	GRANITE BAY	2896	676200
2008-05-19	PLACERVILLE	2025	677048
2008-05-19	WILTON	3788	691659
2008-05-19	GRANITE BAY	3670	760000
2008-05-16	ROSEVILLE	3579	610000
2008-05-16	EL DORADO HILLS	0	622500
2008-05-16	EL DORADO HILLS	0	680000
2008-05-16	EL DORADO HILLS	0	879000
2008-05-16	WILTON	4400	884790

This operation, **querying**, is one of the things we most often want to do with data.

Now let's set up a **pivot table**, summarising key information from the table. Suppose we want to calculate the average price in each city:

```
real_estate_df2.pivot_table(
    values='price',
    index='city',
    aggfunc='mean').head(10)
```

city	price
ANTELOPE	232496.393939
AUBURN	405890.800000
CAMERON PARK	267944.444444
CARMICHAEL	295684.750000
CITRUS HEIGHTS	187114.914286
COOL	300000.000000
DIAMOND SPRINGS	216033.000000
EL DORADO	247000.000000
EL DORADO HILLS	491698.956522
ELK GROVE	271157.692982

Since what interests us is the average, we've used the **aggregation function** `mean`. If we were more interested in the totals. . . :

```
real_estate_df2.pivot_table(
    values='price',
    index='city',
    aggfunc='sum').head(10)
```

city	price
ANTELOPE	7672381
AUBURN	2029454
CAMERON PARK	2411500
CARMICHAEL	5913695
CITRUS HEIGHTS	6549022
COOL	300000
DIAMOND SPRINGS	216033
EL DORADO	494000
EL DORADO HILLS	11309076
ELK GROVE	30911977

What if we wanted to break it down by property type?

```
real_estate_df2.pivot_table(
    values='price',
```

```
    index='city',
    aggfunc='sum').head(10)
```

type city	Condo	Multi-Family	Residential	Unkown
ANTELOPE	115000.0	NaN	7557381.0	NaN
AUBURN	260000.0	285000.0	1484454.0	NaN
CAMERON PARK	119000.0	NaN	2292500.0	NaN
CARMICHAEL	571634.0	NaN	5342061.0	NaN
CITRUS HEIGHTS	185250.0	256054.0	6107718.0	NaN
COOL	NaN	NaN	300000.0	NaN
DIAMOND SPRINGS	NaN	NaN	216033.0	NaN
EL DORADO	NaN	NaN	494000.0	NaN
EL DORADO HILLS	NaN	NaN	11309076.0	NaN
ELK GROVE	688000.0	NaN	30223977.0	NaN

If these were averages, NaN would make sense; but these are totals, so let's make the NaNs into zeros:

```
real_estate_df2.pivot_table(
    values='price',
    index='city',
    aggfunc='sum').fillna(0).head(10)
```

type city	Condo	Multi-Family	Residential	Unkown
ANTELOPE	115000.0	0.0	7557381.0	0.0
AUBURN	260000.0	285000.0	1484454.0	0.0
CAMERON PARK	119000.0	0.0	2292500.0	0.0
CARMICHAEL	571634.0	0.0	5342061.0	0.0
CITRUS HEIGHTS	185250.0	256054.0	6107718.0	0.0
COOL	0.0	0.0	300000.0	0.0
DIAMOND SPRINGS	0.0	0.0	216033.0	0.0
EL DORADO	0.0	0.0	494000.0	0.0
EL DORADO HILLS	0.0	0.0	11309076.0	0.0
ELK GROVE	688000.0	0.0	30223977.0	0.0

As our final task, let's group all the data by city, showing the details of the most recent sale in each case. We'll suppress the 'street' and 'zip' data.

```
real_estate_df.groupby('city')[
    ['beds','baths','sq__ft','type','price']
].last().head(10)
```

city	beds	baths	sq__ft	type	price
ANTELOPE	3	2	1517	Residential	212000
AUBURN	4	3	0	Residential	560000
CAMERON PARK	3	2	0	Residential	224500
CARMICHAEL	4	2	1319	Residential	220000
CITRUS HEIGHTS	3	2	1216	Residential	235000
COOL	3	2	1457	Residential	300000
DIAMOND SPRINGS	3	2	1300	Residential	216033
EL DORADO	2	1	1040	Residential	205000
EL DORADO HILLS	3	2	1362	Residential	235738
ELK GROVE	4	2	1685	Residential	235301

What makes sure we get the most recent transaction is our use of the `last` method; other methods available include `sum`, `mean`, `max` etc.