Introduction to Python

Etienne Roesch (e.b.roesch@reading.ac.uk)
Original notes by Chris Thomas and Guy Griffiths.

March 2019

1 Introduction

In this series of sessions, you will be learning the basics of how to program using the python programming language. Programming is an increasingly important skill in sciences in general, and at least a basic knowledge of programming is essential for meteorology, where the vast majority of work is performed on computers.

1.1 What is programming?

Programming essentially involves getting a computer to do exactly what you want it to do. Off-the-shelf programs such as Excel are great at doing certain tasks, but in science often what we want to do is something which would be very repetitive if we're using an existing program. Often there isn't even a program which can do what we want.

There are many programming languages in existence which try to ease the task of translating your intentions into something which a computer will obey. The process of programming involves translating the general concept of what you want to do into a number of small discrete steps, and then translating those steps into the programming language you're working in.

In many ways, the first step is the most important. Once you are able to program, learning a new programming language is often fairly straightforward — the hardest bit is learning to program in the first place. Although we're teaching you how to use python, our main aim is to teach you how to program — that is, how to break down problems into small chunks which can be processed by a computer.

1.2 What is python?

For this course we'll be using a programming language called python. Python is used extensively in science (and other fields). Python is a nice language for beginners because it is very 'high level' — this means we don't end up getting bogged down in irrelevant details, and can focus more on the problem we're trying to solve. Python is also free, and you can get a copy for yourself online.

1.3 How this course is structured

This course will take the form of four practical classes, each of which will include a short lecture. When the lecture has finished the rest of the class will be spent working your way through the notes individually and completing exercises. The lecturer and demonstrators will be on hand to help with any questions or difficulties you may have. All of the helpers are adept at programming in general and using python in particular, so you can call on them whenever you need a hand. You should also feel free to discuss problems with your peers and look up help on the internet — this is how you will program in the real world so it's good to get used to it in class!

1.4 What you can expect to know by the end of the course

By the end of this course you will be familiar with:

- How to use python as a powerful interactive calculator
- How to use python to write scripts which can be run
- Using variables to store data values
- Using lists and arrays to store multiple related data values
- How to use loops to perform the same action repeatedly on multiple different data values
- How to use conditional statements to modify the flow of a program
- How to write functions to encapsulate a piece of functionality to re-use
- Importing third party python code to use in your own scripts
- Using the internet to find help on third party python code
- Using the `matplotlib` module to generate graphs in python
- Reading data from files on disk
- Writing data to files on disk

2 First steps and programming basics

2.1 Running python interactively

There are two ways of using python. You can write a program in a text editor and then run it. This is called a *script*. This is the most common way of working, and what you'll be doing for all the assignments in this course. You can also use it *interactively*, typing and running one line at a time. This is a bit like using a fancy calculator and is a really good way of experimenting with code before you put it in your script.

So let's try that. Run the Python(x,y) program from the Windows start menu. You should get a window like the one in Figure 1. From the dropdown menu named 'Interactive consoles:'

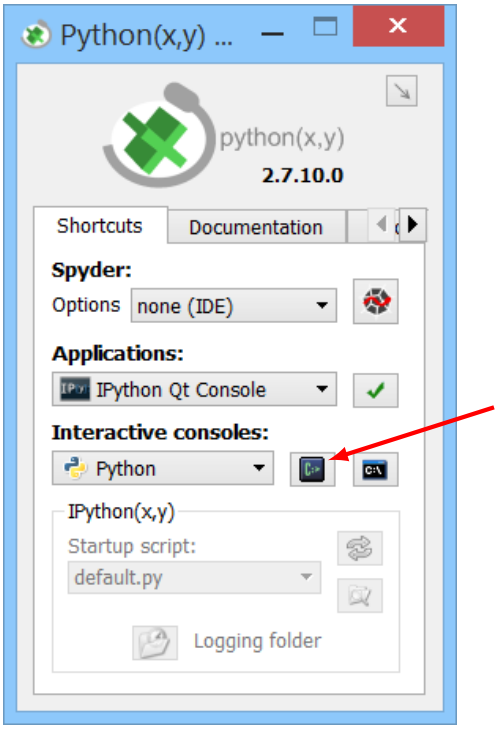

Figure 1: The Python(x,y) start screen. Select 'Python' from the 'Interactive consoles:' dropdown box, then click the button indicated with a red arrow.

pick the one just named 'Python' and click the button just to the right (indicated with a red arrow on in Figure 1). A window should appear with some text in it. Exactly what the text says may vary a little, but it should be something like:

```
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)]
   on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The cursor should start flashing to the right of the `>>>` symbol. Try using python as a calculator: type in a calculation (such as `17 + 32`) and press enter. The answer will appear below the line you just typed. If you want to multiply numbers, use the `*` symbol. For division, use the `/` symbol. And to calculate exponents, use `**` (e.g. `3**2` is equivalent to 3^2). Finally, if you're not sure which operations will happen in which order, use brackets!

Exercises (answers in section 2.8)

1. Try a few simple calculations. Do they give the answers you expect, or is anything weird happening (hint: try some simple divisions)?

2. Can you find a way to stop these weird things happening?

3. Use the python console to calculate $640320^{3/2}/(12 \times 13591409)$.

4. Are you able to do more complex operations (e.g. trigonometry)?

2.1.1 Integer division

You will have noticed that if you divide one integer by another integer, you will always get an integer result (rounded down). This problem is not unique to python, and is one of the main stumbling blocks which beginner programming students trip over. The reason for this behaviour is that when you enter an expression like `3/2`, python tries to figure out what you mean. It sees that 3 is an integer, and that 2 is also an integer. Since you're only dealing with integers, it assumes that you want the result as an integer. The way to get around it is to always write your calculations as decimals (e.g. `3.0/2.0`). That way python can detect that you want the answer as a decimal — it's not important how many decimal places you specify, just that it is a decimal. There is also a function named `float` which converts an integer to a real number. You can use it like so: `float(3) / 2`

In real-world programming this can be a source of bugs, but it is rarer than you might think. Because most real-world problems involve reading data from a file we don't often end up writing actual numbers in calculations. Be aware of it, and if you're using python as a calculator, always write your numbers as decimals.

2.1.2 Mathematical functions

If you tried to calculate the sine or cosine of a number, you'll have spotted that it doesn't work. And when it fails, it gives a slightly unintelligible error:

```
>>> sin(1.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
>>>
```

The reason for this is that mathematical functions are not a core part of python. But that's OK! Most things you'll want to do in python are not a core part of python — they need to be *imported* from *modules*. A module is simply some code which someone has written which you can use. Importing a module just means telling python that you want to use a particular bit of someone else's code.

Common mathematical functions are contained in the `math` module. To start using the math module, enter the command `import math`. Now you can use mathematical functions like so:

```
>>> import math
>>> math.sin(1.0)
```

```
0.8414709848078965
>>> math.cos(1.0)
0.5403023058681398
>>> math.cos(math.pi)
-1.0
>>>
```

To find out what functions (and mathematical constants, such as π and e) you can use, enter the command `help(math)`. You can page through the list by pressing the space bar, or exit the help by pressing '`q`'.

Note that to use the functions in the math module, you will need to write `math.` before each function's name (e.g. `math.cos(0.5)`).

Exercises Using the math module, calculate the following (hint: use the `help(math)` command to find out how):

5. The square root of 10
6. 10! (i.e. 10 factorial)
7. The natural logarithm of e
8. $2^{\frac{1}{2}}$

2.2 Variables

When you're performing calculations it's normally desirable to store the results to use sometime later. In programming languages, this is done through the use of *variables*. A variable in programming can be thought of as somewhere to store some information to be referred to later. Variables have names which consist of alphanumeric characters. In python it is a very strong convention for variables to consist of all lower-case letters, with underscores to separate words.

Once defined, variables can be used in place of numbers. Variables are defined and used as follows (type these examples into your interactive console — they may seem pretty simple, but typing things in yourself helps fix them in your memory):

```
>>> a=math.sin(1.0)
>>> b=math.cos(1.0)
>>> a+b
1.381773290676036
>>> a**2+b**2
1.0
>>> result = a**2 + b**2
>>> result
1.0
```

A few things to note:

- Variables are defined by *`variablename = calculation`*
- You can put spaces around the equals sign and between the operators
- When you assign a variable in interactive mode, it doesn't print the result. To print the result, just type the variable name and press enter

Exercises

9. You're going to calculate the two roots of the quadratic equation $2x^2 - 3x - 2 = 0$. The roots are found by solving:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

First set three variables named `a`, `b`, and `c` to the values 2, -3, and -2 respectively. Reminder: it is good practice to use floats rather than integers.

Now put the three variables into the equation: $\frac{-b+\sqrt{b^2-4ac}}{2a}$.

Then put the variables into the equation: $\frac{-b-\sqrt{b^2-4ac}}{2a}$.

2.3 Lists

One of the most common tasks in programming (particularly scientific programming) is dealing with *lists* of values. For example, if we wanted to plot a graph of wind speed against time, we would use a list of times and a corresponding list of wind speeds. The concept of lists is such a useful one that you would struggle to find a programming language which doesn't have them as a central part of the language in one form or another. In python, a list is denoted using square brackets around some comma separated values. So to define some lists and store them in variables:

```
>>> list_a = [0,1,2,3,4,5]
>>> list_b = [5,2,5,2,7,9,2,1,7,9]
>>> this_list_is_empty = []
>>>
```

We can now use these variable names (`list_a`, `list_b`, `this_list_is_empty`) to refer to the entire lists of numbers. If we want to get at a particular *element* of a list, we use the variable name followed by square brackets with the number of the element within the list (called the list *index*), *counting from zero.*

For example:

```
>>> list_a[0]
0
>>> list_a[1]
1
>>> list_a[2]
2
>>> list_a[3]
3
>>> list_a[4]
4
>>> list_a[5]
5
>>> list_b[0]
5
>>> list_b[1]
2
>>> list_b[2]
5
>>> list_b[3]
2
>>> this_list_is_empty[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> list_a[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> list_b[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

If you try to access a list element which doesn't exist (because the list is not long enough), you get an error. The important bit of the error is 'IndexError: list index out of range' — this tells you that you have tried to access an element of a list which does not exist.

Exercises

10. Create a list with at least five numbers in it and assign it to a variable named `my_list`.

11. Access the third element of the list and assign it to another variable named `third_element`.

12. What do you think will happen if you try to access some elements of your list using negative indices (e.g. `my_list[-1]`)? Try it — is it what you expected? If not, can you figure out what is happening?

13. Access your list using `my_list[0:3]`. What does this do? Play with the numbers — does it behave as you expect?

14. Remember, anywhere you can use a number, you can also use a variable. Access the third element of your list using a variable for the index instead of a number.

15. Change the third element of `my_list` to have the value 100 (hint: you don't have to type in the whole list again)

You should now be reasonably comfortable with creating lists of numbers and accessing them one element at a time. We're going to be using lists a lot throughout this course, so if you're uncomfortable with using them play around more. Look at the answers to the exercise and type them in. Change things. Play with the code. Try and make it give you an error — then try and understand *why* it gave you an error.

2.4 Writing scripts using Spyder

So far we've used python by typing in commands one line at a time. This is all very well and good for doing some experimentation, but when we actually want to do some real work it will be useful if we can gather all of the commands we want into a single script which we can run as many times as we want.

To write our scripts, we're going to use a program named 'Spyder'. Firstly, close the console you have been using so far. Then using the main Python(x,y) window, start Spyder — leave the options set as 'none (IDE)' and click the button to the right (see Figure 2).

Once Spyder starts up (be patient — it takes a while) it will look something like Figure 3. Spyder is what's known as an IDE — an Interactive Development Environment. IDEs are at heart a fancy text editor for programming with a bunch of other useful tools built in. There are three areas in the default Spyder layout: the editor on the left where you can type your programs; the object inspector/variable explorer/file explorer area on the top right; and the console on the bottom right — this is a python console just like the one you've already been using. The nice thing about this combination is that you can program using the editor, and then when you need to experiment with some code to see what it does, you have a python console ready at hand.

Important: there are several versions of Spyder installed on the teaching computers. Make sure your version of Spyder is running python version 2.7, not 3.5. You can check the version using `Help → About Sypder...` If the text at the bottom shows python 3.5, close this version of Spyder and try another one.

Let's create a simple script and run it. Firstly, create a folder named 'mt12c' in your home directory to store the code for this course. Click `File → New file...`. This will create a file called something like `untitled.py`. Then click `File → Save as...` and save the file with the name 'week1.py' in the 'mt12c' folder.

The new file won't be completely blank — it should look something like this:

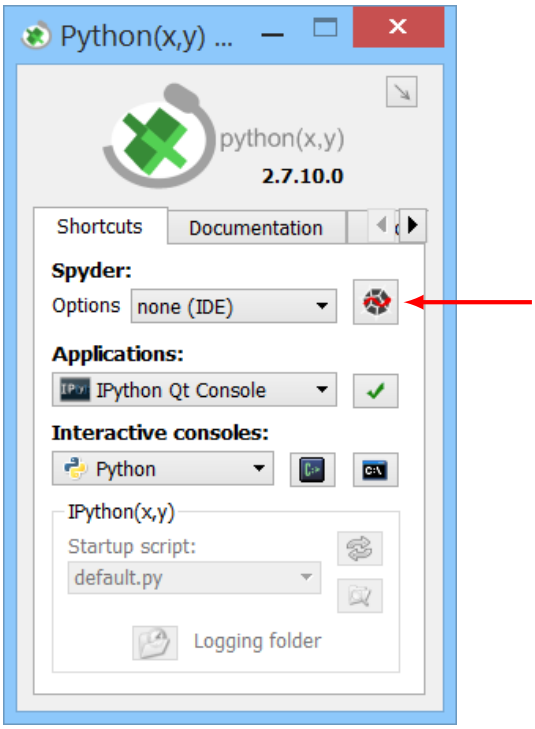

Figure 2: The Python(x,y) start screen. The Spyder button is indicated with a red arrow.

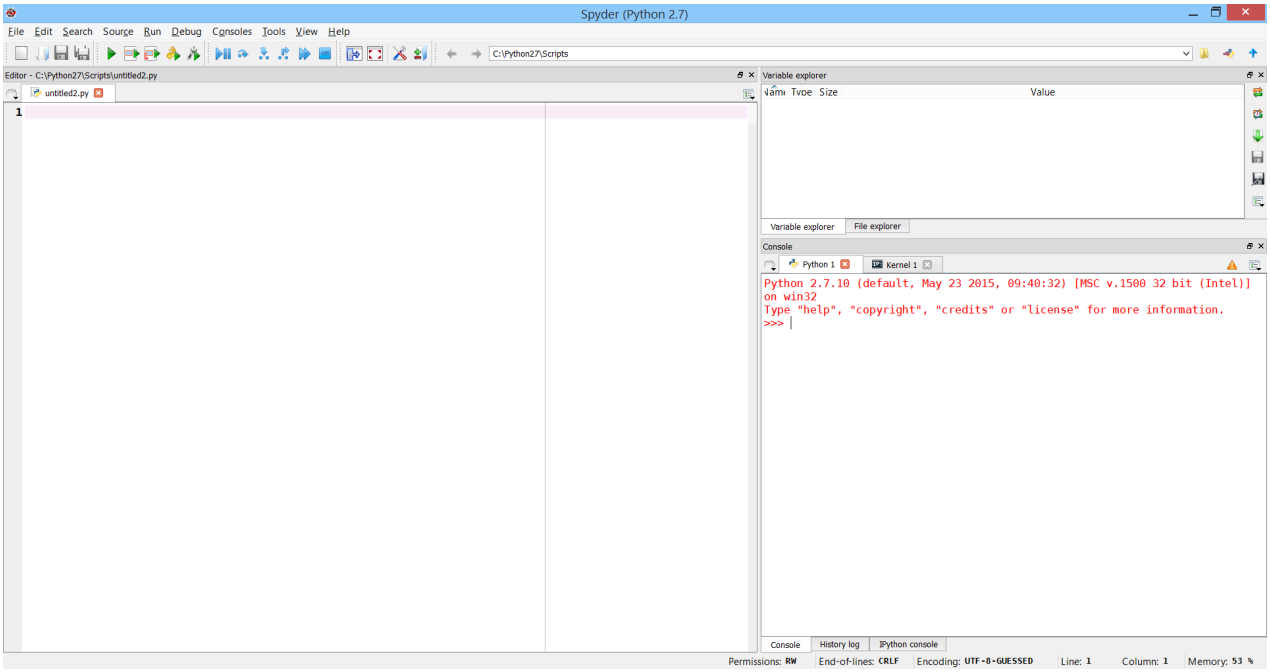

Figure 3: The Spyder window immediately after starting it up.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 28 11:41:42 2013

@author: Bob
"""
```

The line beginning with `#` is a comment, and is ignored when the program is run, and everything between the lines beginning `"""` is documentation, which is also ignored when the program is run. The script therefore doesn't do anything when it is first created. We'll talk more about comments and documentation a bit later on.

So let's try typing in and running a script. After the documentation string, type in the following code:

```
import math

a = 3.0
b = 4.0
c = math.sqrt(a**2 + b**2)

print c
```

and save the file (`Ctrl+s` or `File → Save`). Now run the file by either pressing `F5`, or by clicking the button with a green play symbol. You will get a window asking for the run settings for your file. In the 'Interpreter' section, choose the option 'Execute in a new dedicated Python interpreter'. Leave everything else on the default value, and click 'Run'. In the bottom-right of Spyder, a new tab will appear named 'week1.py' (or whatever you named your file). In that tab, you should see the text '`5.0`'. Hopefully it's clear what's happening here, but it's worth noting that we've introduced something new: the `print` statement.

2.4.1 The `print` statement

One key difference between using python interactively and running scripts is that in interactive mode the result of a calculation is printed to the screen immediately. In script-mode, this doesn't happen — generally when we're running scripts we only want to print output to the screen at certain key moments. The command in python to do this is `print`. To use `print`, we type `print` followed by the things we want to print, separated by commas. If we want to print a literal string of words, we enclose it in quotation marks.

So, since you've just learned a little about the `print` statement, it would be a good idea to experiment with using it. Remember Spyder has an interactive console ready for just this sort of thing. Using the interactive console in Spyder try these exercises:

Exercises

16. Create a variable with the name `x` storing a number (any number!). Now print it.

17. Using the `print` statement, print out 'The value of `x` is' followed by the value of `x`.

`print` may not be very complex or difficult to use, but it's one of the most useful functions in python you'll meet.

So we are now able to write scripts which perform a sequence of python commands one after another. When we run a script, it will have the same effect as:

- Starting an interactive python console
- Typing in all the code in the script one line at a time
- Exiting the interactive python console

This means that if we run a script which has `import math` at the top and then run another one which doesn't, the second script will not be able to access any of the functions in the math module. This is a good thing — you don't want a script which only runs correctly if a different script has been run first!

Writing scripts is the normal way of working in python. In exercise 9 you used python interactively to find the roots of the equation $x^2 - 3x - 4 = 0$. Let's put that in a script instead:

Exercises

18. Create a new python script/module named 'quadratic_roots.py'. Using your answer from exercise 9, write some code to find the roots of $2x^2 - 3x - 2 = 0$ and print out both answers to the screen in the form '`The roots are X and Y`'. As before, it is good practice to use floats rather than integers.

2.4.2 Finding code errors

Spyder has a useful feature: it can spot errors in your code when you are editing a script. These errors are indicated by symbols in the left-hand margin of the script editor. Moving the mouse over each symbols pops up an overlay showing the problem. See Figure 4 for an example.

2.5 Comments

Earlier we mentioned that lines beginning with `#` are treated as comments and are ignored by python when a script is run. So what's the point of them?

Comments are one of the most important parts of programming. The purpose of comments is to explain what the code does. You will often hear that comments are important so that other people can easily understand what your code does. This is true, but it's not the main

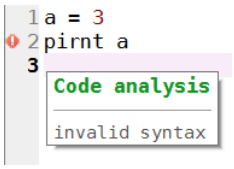

Figure 4: An example of a coding error (misspelling '`print`'). The error is indicated by a red diamond in the left-hand margin. Moving the mouse over it reveals the error.

reason to use comments. The main reason to use comments is so that when you need to change some code which you wrote 6 months ago, *you* understand it. Comments can go anywhere in your code - normally an explanatory comment will go just before the code you are explaining. For example:

```
import math

# Assign values to a and b
a = 3.0
b = 4.0
# Use Pythagoras' theorem
c = math.sqrt(a**2 + b**2)

print c
```

You don't need to comment every single line of a program, but you should have enough comments in a program so that you can see at a glance what's going on.

Exercises

19. Modify your 'quadratic_roots.py' script so that it has comments.

2.6 Dictionaries

When you use a dictionary in real life, you look up the word you're interested in and then read the definition of that word. You can create a 'dictionary' in python which works exactly like this. Python dictionaries consist of pairs of data. The first element in each pair is called the *key*, and the second element is called the *value*. Going back to our real-world example, we would say the word is the key, and the definition is the value.

Unlike lists, which are declared with square brackets (`[]`), dictionaries are declared with curly brackets (`{}`). Key–value pairs are declared inside the brackets as `key: value` and separated by commas. Let's take a look at an example:

```
>>> colours = {1: "red", 2: "orange", 3: "yellow", 4: "green", 5: "blue", \
    6: "indigo", 7: "violet"}
```

(If you're wondering about the \ symbol, this tells python to treat the second line of text as if it's a continuation of the first. This is a convenient way to break long lines into two, but you don't have to use it.)

The dictionary is called `colours`, the keys are the numbers 1–7, and the values are the colours assigned to each key. We can access a particular entry in the dictionary as follows:

```
>>> print colours[2]
orange
```

Note that this is *not* the same as accessing the entries of a list. In this case, `2` is a particular key in the dictionary, and not the index of that entry. If you try to get the first entry of the dictionary, as you would do for a list, the following error occurs:

```
>>> print colours[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

In fact, dictionaries are unordered; they don't have 'first' or 'last' entries. Dictionaries are particularly useful to store data when you want to be able to look up a particular value easily, but you don't care about the precise order in which the data are stored.

Although in the example given all of the keys are numbers and all of the values are strings, the keys and values could in fact be either type. In other words, we can define the following dictionary:

```
>>> things = {"dog": 3, "cheese": "cheddar", 4: 5}
```

Try creating a few dictionaries with different types of entry until you're familiar with how they work.

It is easy to add and remove elements from a dictionary. To add an element, you can simply give the dictionary a new key–value pair:

```
>>> colours[8] = "octarine"
>>> print colours
{1: 'red', 2: 'orange', 3: 'yellow', 4: 'green', 5: 'blue', 6: 'indigo', 7:
    'violet', 8: 'octarine'}
```

Elements of a dictionary can be deleted using the built-in `del` command:

```
>>> del colours[4]
>>> del colours[7]
>>> print colours
{1: 'red', 2: 'orange', 3: 'yellow', 5: 'blue', 6: 'indigo', 8: 'octarine'}
```

Existing elements can also be reassigned as follows:

```
>>> colours[6] = "black"
>>> print colours
{1: 'red', 2: 'orange', 3: 'yellow', 5: 'blue', 6: 'black', 8: 'octarine'}
```

Exercises

20. Create a dictionary called `my_dictionary` which consists of six key–value pairs. Make the first three pairs be string–number, and the second three number–string. Try accessing a few of the elements of the dictionary.

21. Delete two entries in the dictionary. Use `print` to look at the values that remain.

22. Create an empty dictionary and then add an element to it.

2.7 Summary

You should now be familiar with the following things:

- What programming is
- What python is
- How to use python interactively to perform calculations
- How to assign values to variables and use them again later
- How to import the `math` module to use mathematical functions
- How to store values in a list, and how to access individual elements of the list
- How to start Spyder, write a script and run it
- How to print values of your variables from a script
- How to comment your code
- How to use dictionaries

If you're having trouble with anything on that list, go through the notes again, ask the supervisors, ask other students, look it up on the internet, and most importantly of all try things out. Whatever you need to do to get familiar with these concepts, do it — once you're comfortable with the basics of programming, you'll find the rest of this course goes much more smoothly.

Once you're comfortable with the material you've covered, log in to blackboard, and download and complete the first assignment.

Python(x,y) is installed on a number of PCs on campus. If you want to use python on your own machine, you can download it from `http://python-xy.github.io/`.

2.8 Answers to exercises

Answers

1. You should find that calculations like 3/2 don't give the right answer - instead they round *down* to an integer.
2. One way to stop this happening is to specify the numbers as decimals: e.g. 3.0/2.0
3. You can perform the calculation as follows:

```
>>> 640320.0**(3.0/2.0) / (12.0 * 13591409.0)
3.1415926535897345
```

4. Python doesn't have functions such as `sin`, `cos`, $\sqrt{}$ built into the language.
5. `math.sqrt(10.0)`
6. `math.factorial(10)`
7. `math.log(math.e)` (Note that `math.exp()` is a function which allows you to do e^x, whereas `math.e` is just the constant, e. Also note that `math.log()` gives the natural logarithm, unlike on most calculators, which use `ln`)
8. `math.pow(2, 0.5)` - you could have used `2**0.5` to get the same result, but that wouldn't be using the math module.
9. To find the roots:

```
>>> a = 2.0
>>> b = -3.0
>>> c = -2.0
>>> (-b + math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)
2.0
>>> (-b - math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)
-0.5
>>>
```

10. `my_list = [7,2,4,9,6]` (for example)
11. `third_element = my_list[2]`. Remember - list indexing *starts at zero*, so the third element is `my_list[2]`.
12. Negative indices count from the end of the list. So in the above example:

```
>>> my_list[-1]
6
>>> my_list[-2]
9
>>> my_list[-3]
4
>>> my_list[-4]
2
>>> my_list[-5]
7
>>> my_list[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

13. This notation creates a sub-list from element number 0 up to (but not including) element number 3.

14. To access a list element using a variable:

```
>>> element_index = 2
>>> my_list[element_index]
4
>>>
```

15. `my_list[2] = 100`. You can change the values of individual elements of a list like this. To check that it worked:

```
>>> my_list[2] = 100
>>> my_list
[7, 2, 100, 9, 6]
>>>
```

16. Using the `print` statement:

```
>>> x=1.23
>>> print x
1.23
>>>
```

17. Printing multiple items:

```
>>> print 'The value of x is', x
The value of x is 1.23
>>>
```

18. Finding the roots in a script:

```
import math

a = 2.0
b = -3.0
c = -2.0

root1 = (-b + math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)
root2 = (-b - math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)

print 'The roots are', root1, 'and', root2
```

19. Commented version:

```
import math

# Define our coefficients
a = 2.0
b = -3.0
c = -2.0

# Calculate both roots using the quadratic formula
root1 = (-b + math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)
root2 = (-b - math.sqrt(b**2 - 4.0*a*c) )/(2.0*a)

# Print the result to screen
print 'The roots are', root1, 'and', root2
```

20. An example of a dictionary is:

```
>>> my_dictionary = {"a": 1, "b": 2, "c": 3, 4: "d", 5: "e", 6:"f"}
>>> print my_dictionary["a"]
1
>>> print my_dictionary[5]
e
```

21. To delete two elements of the dictionary:

```
>>> del my_dictionary[4]
>>> del my_dictionary["c"]
>>> print my_dictionary
{'a': 1, 'b': 2, 5: 'e', 6: 'f'}
```

22. An empty dictionary can be created as follows:

```
>>> empty_dictionary = {}
```

To add an entry to it:

```
>>> empty_dictionary["not so empty"] = "any more"
>>> print empty_dictionary
{'not so empty': 'any more'}
```

3 Looping, branching, and functions

So far we've learnt to use python as a calculator and to write simple scripts. To write a script, we wrote a bunch of python commands one after another, and when we ran scripts, python went from the top of the script down to the bottom, running each command in order.

This is absolutely fine for very simple cases, but as we start to write more complex programs we will run into cases where it's not very efficient. For example:

- We want to do the same thing to each one of a group of different numbers (e.g. find the square of them all)
- We want to do different things depending on some condition (e.g. find the square root of a number if it's positive, but print an error to the screen if it is negative)
- Write some code which we can use with different *inputs* (e.g. find the roots of a quadratic equation — we want to give our code the values for a, b, and c and have give back the roots of the equation)

In these types of case, we need to change the *flow* of the code, and there are a few ways to do this when programming. We're going to investigate them all in this session.

3.1 Loops

A *loop* in programming can be summarised as doing the following:

- Set up the loop
- Run a series of commands which use some variables to do some calculations
- Change one or more of the variables
- Run *the exact same* commands with the new values of the variables
- Change one or more of the variables
- Run *the exact same* commands with the new values of the variables
- Keep repeating until the loop is finished.

3.1.1 Basic `for` loops

Example 1: Open Spyder, create a new python script, *type* the following into it (don't just copy and paste it) and run it.

```
values = [0,1,2,3,4,5,6,7,8,9,10]
for i in values:
    print i, 'squared is', i**2
```

Let's look at what this does in detail:

- It creates a list of numbers and stores it in a variable named `values`
- `for i in values:` This sets up a loop. What this says is 'take each of the elements in the variable `values` in turn, name it `i` and run the code below'. Each time the code in the loop runs, a new variable named `i` is created with the value of the next element in the list.
- Take the variable `i` and print its value and its square (with a bit of text in the middle).

Note that whilst `i` is a very common variable name to use in a loop, we could have chosen any valid variable name. Some other (equally good) suggestions could be:

```
values = [0,1,2,3,4,5,6,7,8,9,10]
for value in values:
    print value, 'squared is', value**2
```

```
values = [0,1,2,3,4,5,6,7,8,9,10]
for v in values:
    print v, 'squared is', v**2
```

In this example, we've only got one line of code inside the loop, but in practice you can have as many as you want. This type of loop is defined as starting with `for` and ending when the indentation stops. For example:

```
values = [2,4,6,8,10]
for i in values:
    print i, 'squared is', i**2
    print 'This statement is still in the loop'
    print 'So is this - you will see them many times'
print 'The loop has ended, so this will only be printed once'
```

So that's all well and good, but what use is it? In general you should use a loop when **you want to do the same thing several times, with only a minor variation**. Let's look at some more examples

Example 2:

```
values = [1,3,5,7,9,11,13]
sum_of_values = 0.0
for i in values:
    sum_of_values = sum_of_values + i
print 'The sum of the values', values, 'is', sum_of_values
```

This is a *very* common pattern in programming. What we are doing here is

- Setting the variable `sum_of_values` to 0.0 (if we set it to 0, we may find we run into integer division problems (see 2.1.1)).
- Taking each item in the list `values` in turn and adding it to the sum

The line `sum_of_values = sum_of_values + i` may look a little strange at first, because it has the same variable on both sides of the equation. But not only is that fine, it's incredibly common. If you recall, when we looked at variables we said that variables are defined by *`variablename = calculation`*. The calculation on the right-hand side can involve numbers, functions, and variables. The result of the calculation is then stored in a variable. You can read the above code as 'add the value of `sum_of_values`, to the value of `i`, and update `sum_of_values` to this new value'.

Exercises (answers in section 3.6)

1. Write some code which loops over a list of numbers and prints each number, twice that number, and three times that number to the screen.
2. Write some code to find the mean value of a list of numbers and print it to the screen. Some things to bear in mind:
   - You will need to know the number of items of your list. There is a function available to calculate this for you — use Google to find it.
   - There is also a function which could calculate the sum for you — **don't use this** (the aim of the exercise is to practice loops)
   - If it doesn't work, have another look at section 2.1.1
   - Once it works, you should try it with a few different lists of numbers to make sure that it *really* works. By storing your list of numbers in a variable, you will only need to change that variable.

3.1.2 Indexed `for` loops

If you come into contact with older code, or code written in another language, there is a very common pattern you will see in `for` loops. Instead of looping over the elements of a list, we create a loop over a range of numbers and use those numbers to *index* the list.

Try typing in and running the following example:

```
values = [10,20,30,40,50,60,70,80,90,100]
for i in range(len(values)):
    val = values[i]
    print val, 'squared is', val**2
```

This is very similar to the first example we saw above and behaves the same way. However, we've got a couple of new things here; built-in python functions named `len` and `range`. `len` is a function which calculates the length of a list:

```
>>> values = [10,20,30,40,50,60,70,80,90,100]
>>> len(values)
10
```

You should have discovered the existence of this function when writing code to calculate the mean value of a list.

Exercises

3. How can you print the final element of a list if you don't know its length off the top of your head?

`range` is a function which generates a list of numbers. In its simplest form, it takes one *argument* and generates a list of numbers ranging from 0 up to, but not including, that argument (this is so that if you use `range(5)` it creates a list with 5 items, rather than 6):

```
>>> range(4)
[0, 1, 2, 3]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> n = 15
>>> range(n)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> range(n-1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
>>> range(n-2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

`range` can also take some optional arguments. If you give it two arguments, it will create a list of numbers ranging from the first argument up to, but not including, the second argument:

```
>>> range(5,10)
[5, 6, 7, 8, 9]
>>> range(100,110)
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
>>> range(-5,6)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
>>> n=10
>>> range(n-5,n)
[5, 6, 7, 8, 9]
```

So, in our above example, we are creating a loop where `i` varies from 0 to 9. `i` is then being used to access the *elements* of the list `values`. We would also use `range()` in a loop if we did not care about using the values, but merely wanted to do something a certain number of times. For example:

```
for i in range(10):
    print 'All work and no play makes Jack a dull boy'
```

Exercises

4. Rewrite exercise 1 using an indexed `for` loop.

5. Write an indexed `for` loop to calculate the sum of all of the numbers in a list

6. This pattern is very common when you want to loop over almost all of a loop. By adjusting the arguments to the `range` function, you can pick which parts of the list to include. Bearing this in mind, write some code to calculate the sum of all of the numbers in a list *apart from the final one.*

7. Write some code to calculate the sum of all of the numbers in a list *apart from the first and final elements.*

3.1.3 A worked example: the Fibonacci sequence

We are now going to look at an example of code which uses a loop to calculate the first n Fibonacci numbers and print them to the screen. This will break down the stages we need to write an algorithm, and will introduce the idea of using temporary variables which get reused over and over.

When solving a problem using programming, there are two main stages: breaking the problem down into small steps which a computer can solve, and converting these small steps into the programming language you're using. The first step should be done on paper — you should never just start coding until you've got a clear idea of what you're going to do.

The Fibonacci sequence is defined as follows:

- The first two Fibonacci numbers are both 1
- A Fibonacci number is the sum of the previous two Fibonacci numbers

To calculate the Fibonacci sequence we have the following basic steps to the algorithm:

- Start with the first two numbers (f_1 and f_2), stored in variables
- Add the two numbers together to get the next Fibonacci sequence
- Print the the result to the screen
- Update the two variables so that we're storing f_2 and f_3
- Add the two numbers together to get the next Fibonacci sequence
- Print the the result to the screen
- Update the two variables so that we're storing f_3 and f_4
- Add the two numbers together to get the next Fibonacci sequence

- Print the the result to the screen
- Update the two variables so that we're storing f_4 and f_5
- Keep going until we've printed n Fibonacci numbers to the screen

This looks very similar to the pattern we saw at the very beginning of section 3.1. We are doing the same thing over and over with just a minor change. Let's break this down into programming tasks, one of which will be using a loop:

- Start by setting up two variables to store the previous two Fibonacci numbers. I'll call these `f_previous` and `f_2previous` to emphasize the fact that they are the 2 previous Fibonacci numbers (i.e. the *current* Fibonacci number is the one we're calculating)
- Set up another variable `n` which stores how many Fibonacci numbers we want to calculate.
- Set up a loop to go from 1 to n (*The* ***`range`*** *function will be useful here*)
- In the loop:
  - Add the two numbers together and store in a variable `f_current`
  - Print the result to the screen
  - We now want to overwrite the oldest Fibonacci number (i.e. `f_2previous`) with the next oldest one (`f_previous`)
  - We also want to overwrite `f_previous` with our current number, `f_current`.
  - Now when we go back to the start of the loop, our variables `f_previous` and `f_2previous` will contain the correct values for calculating the next Fibonacci number

Now that we have a basic outline for our program, we can start to code. We're going to create this code in a script in Spyder. When writing code, it is usually helpful to write the basic structure first, perhaps with a few initial variables defined, and comments to say what you're going to do where. Then when you're happy with the structure, write the actual code under the comments. My first pass at the Fibonacci program looks like this:

```
# Set up f_2previous and f_previous
f_2previous = 1
f_previous = 1

# Print the first 2 Fibonacci numbers to the screen
print f_2previous
print f_previous

# Define how many Fibonacci numbers to calculate
n = 20

# Loop from 1 to n
```

```
for i in range(n):
    # Add the numbers together and store in f_current

    # Print to the screen

    # Now update the old values, ready to loop again
    # Copy the value of f_previous to f_2previous

    # Copy the value of f_current to f_previous

# End of loop, nothing more to do
```

Now filling in the details:

```
# Set up f_2previous and f_previous
f_2previous = 1
f_previous = 1

# Print the first 2 Fibonacci numbers to the screen
print f_2previous
print f_previous

# Define how many Fibonacci numbers to calculate
n = 20

# Loop from 1 to n
for i in range(n):
    # Add the numbers together and store in f_current
    f_current = f_previous + f_2previous

    # Print to the screen
    print f_current

    # Now update the old values, ready to loop again
    # Copy the value of f_previous to f_2previous
    f_2previous = f_previous

    # Copy the value of f_current to f_previous
    f_previous = f_current
```

Although it seems like a lot of work at first, you should always follow a similar procedure to this when programming. **Most of the work is done before you actually start coding**. To summarise the steps you should be taking:

- Write down the steps of the problem
- Break down the problem whilst thinking about the tools you have available (i.e. loops, variables, etc)
- Once you're happy that the method is broken down into the basic steps, write the structure of code in your code editor (for python this is going to be Spyder), with comments explaining what you're doing

- Review your code and once you're sure it matches the method you wrote down, fill in the details of the code.

Exercises

8. A Fibonacci number can be defined as $f_n = f_{n-1} + f_{n-2}$, with $f_1 = 1$ and $f_2 = 1$. We're going to define a new type of number called a Super-Fibonacci number, s_n, as $s_n = s_{n-1} + s_{n-2} + s_{n-3}$, , with $s_1 = 1$, $s_2 = 2$, and $s_3 = 3$. Using the same process as outlined above, write code to print out the next 10 Super-Fibonacci numbers.

3.2 Conditionals and Branching

The next way to change the flow of your code is by *branching*. Essentially this means running different bits of code under different conditions. For example, calculating the square root of a number if it is positive, and printing an error message if it is negative. To do this, we need to introduce a couple of things:

- A way to represent the concepts 'this thing is true' and 'this thing is false'
- A way to test things to see whether they are true or false.

In python we have two special values, `True` and `False`. These are called *boolean* values. You can assign these values to variables, and you can generate them using comparison operators. In python, the following comparison operators are available:

Operator	Description
`<`	less than
`>`	greater than
`==`	equal
`!=`	not equal
`<=`	less than or equal
`>=`	greater than or equal

Notice that the operator for comparing if two values are equal is `==`. This is to distinguish it from the single equals sign used to assign values to variables.

You can use these operators to check whether a comparison is `True` or `False`. For example:

```
>>> 1 > 4
False
>>> 5 >= 5
True
>>> 1 == 1
True
>>> 1 == 2
False
```

```
>>> 1 != 1
False
>>> 1 != 2
True
>>> a = 3
>>> b = 7
>>> a > b
False
>>> a < b
True
>>> a == b
False
>>> a != b
True
```

You can also check for multiple conditions using the operators:

Operator	Description
`and`	true if both operands are true
`or`	true if one operand is true
`not`	true if the operand is false

For example:

```
>>> 4 > 5 or 7 < 10
True
>>> 4 > 5 and 7 < 10
False
>>> not 4 > 5
True
>>> not 4 < 5
False
>>> a = 10
>>> b = 20
>>> a == 5 or a == 10
True
>>> a == 10 and b == 10
False
>>> a == 10 and b == 20
True
```

To reiterate, ==, not =, must be used to test equality. If you accidentally write

```
>>> a == 10 and b = 20
```

you will find that python returns an error message:

```
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: can't assign to operator
```

3.2.1 The `if` statement

So, we can now use these tests to change the flow of the code, using the `if` statement. Type the following example into a script and run it:

```
numbers = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

for i in numbers:
    if i < 0:
        print i, 'is a negative number'
```

The `if` statement starts with the keyword `if` followed by a condition, then a colon. It forms the start of an indented block, and all of the code in the block will run *if* the condition is true.

Exercises

9. Modify the code above so that:
   - The `range` function is used to generate the list of numbers. Make it vary from -10 up to 10
   - Instead of printing negative numbers, the code only prints out numbers which are *non*-negative.

3.2.2 `else` and `elif`

Very often you will need to do something if a condition is `True` and something else if it is `False`. This is done by using the `else` statement:

```
if condition:
    do something
else:
    do something different
```

If the first condition is met, the first block of code will be run. If the first condition is not met, then the second block of code will be run. (Please note that in real code, the italic text will be replaced with python commands.)

Exercises

10. Modify the last bit of code above so that if prints one string if a number is negative, and a different string if it is non-negative (e.g. '`-10 is a negative number`' and '`5 is a non-negative number`')

Although it's less common, you can also check if several conditions are true, and run a different bit of code when each is true. This uses the `elif` statement, which is short for 'else if':

```
if condition:
    action
elif other condition:
    other action
elif yet another condition:
    yet another action
    ...
    ...
else:
    still another action
```

Exercises

11. Modify the last bit of code above so that if prints one string if a number is negative, a different string if it is non-negative, and yet another string if the number is zero.

12. In python, how can you check if a number is even? And how can you check if it is a multiple of 3? (*Hint: You may not already know this. But you're sat at a computer with access to the internet...*)

13. Write some code which creates a list of numbers from 0 to 99, and checks two things:

    - Whether it is an even number. If so print '⟨*number*⟩ `is even`' (where ⟨*number*⟩ is the number...)
    - Whether it is a multiple of 3. If so print '⟨*number*⟩ `is a multiple of 3`'

14. Repeat the previous question with the following difference:

    - If the number is even, print '⟨*number*⟩ `is even`' (same as before)
    - If the number is *not* even, but it *is* a multiple of 3, print '⟨*number*⟩ `is an odd multiple of 3`'

3.3 Functions

We've already used several functions in python. For example:

- `math.sin`: we gave it a number (e.g. `math.sin(1.23)`) and it gave us back the sine of that number.
- `math.cos`: we gave it a number (e.g. `math.cos(1.23)`) and it gave us back the cosine of that number.

- `math.factorial`: we gave it a number (e.g. `math.factorial(20)`) and it gave us back the factorial of that number.
- `range`: we gave it a number, or several numbers (e.g. `range(10)`, `range(5,10)`) and it gave us back a list of a range of numbers.
- `len`: we gave it a list (e.g. `len([1,2,3,4,5])`) and it gave us back the length of that list.

A *function* is a piece of code which you can give values to (called *arguments*) and which will *return* a value back to you. We *define* functions with the keyword `def` — this starts a new indented block which defines the function. For example:

```
def double_number(i):
    return 2 * i
```

This code creates a function which, *when used*, will double a number. Using functions you have written works exactly the same way as using built-in or imported functions: write its name where is appropriate in your code and give it the arguments it requires.

```
# Defines the function
def double_number(i):
    return 2 * i

# Uses the function
print double_number(10)

# Uses the function by passing it variables
a = 15
b = double_number(a)
print b
```

You can only use a function *after* it has been defined. The following code will not work:

```
# Uses a function that has not been defined yet!
print double_number(3)

# Defines the function
def double_number(i):
    return 2 * i
```

It you try to run this code python will return the following error message:

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

Let's go through how to define a function in more depth. The basic template to follow is:

```
def function_name(arguments):
    action
    another action
    ...
    return value
```

Breaking this down into its various components:

- The keyword `def`, to start the function.
- The name of the function: this should preferably be all lower case with underscores used to separate words.
- Brackets containing the *arguments* to the function, followed by a colon. When you use the function, you will give it values (or variables containing values) in place of the arguments inside the brackets. The values are then stored in a variable with the name given inside the brackets. If your function does not require any arguments, give it a pair of empty brackets: `()`.
- An indented block containing the code which makes up the function. This will always use the *arguments* supplied to the function (if it doesn't, they don't need to be there!).
- A `return` statement. This will end the running of the function **as soon as it is reached**, and give back the value which follows `return`.

The variables defined in the argument list are *only* available within your function block. In the example above, the variable `i` is only available inside the indented function definition. If you try and use it elsewhere you will get an error.

Exercises

15. Write a function called `square_it` which takes an argument and returns its square. Also write some code to use the function to print the squares of 5, 10, and 100.

16. What will the following code do?

```
def func(sequence):
    for i in sequence:
        return 2 * i

a = func([1,2,3,4,5])
print a
```

    (a) Print `[2,4,6,8,10]` to the screen
    (b) Print `2` to the screen
    (c) Print `10` to the screen
    (d) Print `[2,2,3,4,5]` to the screen
    (e) Print `[1,2,3,4,10]` to the screen

17. **Write a function** to calculate n Fibonacci numbers, and return them all as a list. This exercise is a little bit more involved than previous ones. When doing it, bear in mind the following:

- Your function will need to take one argument, `n`
- Where previously you have just printed the numbers to screen, now you will need to store them in a list. To do this you should create an empty list before the loop, (e.g. `fib_list = []`) and then add items to it using the `append` method. You may need to search for help on how to use this.
- When you've stored `n` numbers in your list, you need to `return` it.

Once you have written the function, use it with `n = 10`.

3.4 The `while` statement

Another way of looping over some items is to use the `while` statement. Unlike `for`, which loops over a pre-specified number of elements, `while` keeps on looping until a particular condition is met. You can use `while` as follows:

```
while test:
    action
```

As long as `test` is `True`, `action` will be performed. As soon as `test` is `False` the loop will finish.

For an example of when this is useful, consider a function that takes a number (`n`) and multiplies it by 2 until it is larger than 1000. It would be difficult to use a `for` loop to do this because you don't know how many multiplications are required; for example, if the input number is 3, nine multiplications are needed, but if the input number is 501, only one is needed. The function can be written as follows:

```
def double_until_1000(n):
    while n < 1000:
        n = n * 2
    return n
```

Let's say we have `double_until_1000(3)` somewhere in our program. Consider the first iteration of the `while` loop. At this point `n` is equal to 3, and since $3 < 1000$, the multiplication by 2 is executed. `n` is now 6, which is also less than 1000, etc., etc. Eventually, however, `n` is larger than 1000 and no more multiplications occur. At that point `n` is returned by the function. This value can then be printed to the screen or used in other parts of the code.

Exercises

18. Modify the above code to return the number of multiplications required to make `n` larger than 1000. Call the function and print out the number it returns. (Hint: you will need to use another variable inside the function.)

19. How can you make the function return *both* the final value of `n`, and the number of multiplications required? Print both of these values.

20. If you pass the function a negative number, or zero, it will never stop running and you will have to terminate it manually. How can you protect against this happening?

3.5 Summary

In this section we have covered loops, conditionals, branching, and functions. Together with the different types of variables we covered in section 2 these make up the building blocks of programming, no matter what programming language you learn.

It's very important that you understand these concepts, since they will underpin every program you write. Make sure that you have understood all of the exercises, and try them again if you had any difficulties (take a day off first though — often these ideas sink in slowly and coming back to them after a bit of a rest can work wonders).

The next section will focus on using other people's code, and creating graphs in python, and will contain exercises which reinforce the use of loops, branching, and functions.

3.6 Answers to exercises

Answers

1.
```
numbers = [1,5,9,28,101,10]

for num in numbers:
    print num, 2*num, 3*num
```

2.
```
numbers = [10,9,8,1,2,3]
sum = 0.0

for num in numbers:
    sum = sum + num

# We could have used float(sum) here if we had set it to 0 instead
   of 0.0
print sum / len(numbers)
```

3. This is where `len` is useful. It can be used to find the length of a list, which can then be used to access the final element of that list. Note that because indices start at 0, we need to subtract 1 from the length, as follows:

```
print my_list[len(my_list) - 1]
```

This won't work if the list is empty!

4.
```
numbers = [1,5,9,28,101,10]

for i in range(len(numbers)):
    num = numbers[i]
    print num, 2*num, 3*num
```

5.
```
numbers = [10,9,8,1,2,3]
sum = 0.0

for i in range(len(numbers)):
    sum = sum + numbers[i]

print sum
```

6.
```
numbers = [10,9,8,1,2,3]
sum = 0.0

for i in range(len(numbers)-1):
    sum = sum + numbers[i]

print sum
```

7.
```
numbers = [10,9,8,1,2,3]
sum = 0.0

for i in range(1, len(numbers)-1):
    sum = sum + numbers[i]

print sum
```

8.
```
# Set up s_3previous, s_2previous and s_previous
s_3previous = 1
s_2previous = 2
s_previous = 3

# Define how many Super-Fibonacci numbers to calculate
n = 10

# Loop from 1 to n
for i in range(n):
    # Add the numbers together and store in s_current
    s_current = s_previous + s_2previous + s_3previous
```

```
    # Print to the screen
    print s_current

    # Now update the old values, ready to loop again
    # Copy the value of s_2previous to s_3previous
    s_3previous = s_2previous

    # Copy the value of s_previous to s_2previous
    s_2previous = s_previous

    # Copy the value of s_current to s_previous
    s_previous = s_current
```

9.
```
numbers = range(-10, 11)

for i in numbers:
    if i >= 0:
        print i, 'is a non-negative number'
```

10.
```
numbers = range(-10, 11)

for i in numbers:
    if i >= 0:
        print i, 'is a non-negative number'
    else:
        print i, 'is a negative number'
```

11.
```
numbers = range(-10, 11)

for i in numbers:
    if i > 0:
        print i, 'is a positive number'
    elif i < 0:
        print i, 'is a negative number'
    else:
        print i, 'is zero'
```

12. You can use the modulo operator (%). This give the remainder of division between two numbers. So if the remainder of dividing a number by 2 is zero, it is even. If the remainder of dividing a number by 3 is zero, it is divisible by 3:

```
n % 2 == 0
n % 3 == 0
```

13.
```
numbers = range(100)

for num in numbers:
    if num % 2 == 0:
        print num, 'is even'
    if num % 3 == 0:
        print num, 'is a multiple of 3'
```

14.
```
for num in numbers:
    if num % 2 == 0:
        print num, 'is even'
    elif num % 3 == 0:
        print num, 'is an odd multiple of 3'
```

15.
```
def square_it(n):
    return n * n

#Use the function with a variety of inputs
print square_it(5)
print square_it(10)
print square_it(100)
```

16. (b) Print 2 to the screen. The first time the code goes through the loop, `i` will have the value 1. The function will get to the line `return 2*i` (i.e. 2) and the function will finish running, returning the value 2.

17.
```
def fibonacci(n):
    fib_numbers = []

    f_previous = 1
    f_2_previous = 1

    # We'll add the first two Fibonacci numbers to the list first
    fib_numbers.append(f_2previous)
    fib_numbers.append(f_previous)

    for i in range(n):
        # Calculate the current Fibonacci number
        f_current = f_previous + f_2previous
        # Add it to the list
        fib_numbers.append(f_current)
        # Update the values of f_previous and f_2previous
        f_2previous = f_previous
        f_previous = f_current
    return fib_numbers

#Use the function with n = 10
```

```
fibonacci(10)
```

18. You can define a variable called `count` to determine how many multiplications were required:

```
def double_until_1000(n):
  count = 0
  while n < 1000:
    n = n * 2
    count = count + 1
  return count

#Use the function
print double_until_1000(3)
```

19. The `return` statement can be used to pass back multiple values at once:

```
def double_until_1000(n):
  count = 0
  while n < 1000:
    n = n * 2
    count = count + 1
  return n, count

#Use the function
print double_until_1000(3)
```

20. To protect against zero or negative numbers, you can add another condition to the `while` statement as follows:

```
def double_until_1000(n):
  count = 0
  #Only proceed if n is positive
  while n > 0 and n < 1000:
    n = n * 2
    count = count + 1
  return n, count
```

If `n` is negative or zero, the `while` statement will not activate. In that case, the `return` statement will pass back the original `n` and the initial value of `count`, which is zero.

Alternatively, an `if` statement could be used:

```
def double_until_1000(n):
  count = 0
  #Only proceed if n is positive
  if n > 0:
```

```
    while n < 1000:
      n = n * 2
      count = count + 1
  return n, count
```

4 Using external modules and basic plotting

In section 3 we came across a few of python's built-in functions (e.g. `len` and `range`). Checking on the website there are around 75 of them. That's quite a few, and they're certainly useful, but there aren't enough to cover the many things a programmer might want to do. As we've already seen in section 2, there aren't even `sin` and `cos` functions. So if there's something we'd like to do that doesn't have a built-in function, we either need to write it ourselves (which is hard work and less preferable) or use some code from a *module*.

We've already touched on using modules in section 2 when we used the `math` module to do some mathematical calculations for us. We *imported* the `math` module and then we used the *functions* in it. The key stages of using modules are:

- Wanting some functionality from our code
- Thinking "Someone must have programmed this already"
- Finding the module name which contains the code which someone else wrote
- *Importing* the module so that it's available in our code
- Reading the help about how to use the functions we want
- Using the functions in our code

4.1 Finding the module you need

Shortly we're going to learn about basic plotting in python. To do this, we're going to need a plotting module of some sort.

Exercises (answers in section 4.9)

1. Without reading ahead in the notes (but using any other method you like), find a plotting module to use.

Finding the correct module to use is an important skill in programming. In many cases it's an easy task; python has the philosophy "There should be one — and preferably only one — obvious way to do it", and a quick Google search will tell you what you need to know. However, sometimes there may be more than one good option and you'll need to read up on a few different modules to see which one is best for your needs.

4.2 Importing a module

To import a module in python, it must first be installed on your system. The good news is that python already comes with a very large number of modules available. The other good news is

that installing new python modules is pretty easy, particularly when you're using Python(x,y). Even better, all the modules we're going to use in this course are already available on the lab machines, so you don't need to worry about this part at all for the moment.

Once a module is present on your system, you can make it available to use in the following way:

```
import math
import datetime
import random
```

This code makes 3 modules available in our code *from that point on*: `math`, `datetime`, and `random`. You can put `import` statements wherever you like in your script, but it's normal to put them all together at the top of the script. You should stick to that convention until you have a good reason not to. During this course you are very unlikely to have a good reason not to do this — we expect to see all `import` statements at the top of your scripts.

4.3 Finding functions within a module

Once you've imported the module you want, it's often useful to see the details of the functions it holds, and how to use them. There are a few different ways to do this. I'd recommend one of these:

- Using an interactive python console, type `help(`*`modulename`*`)`. You must have imported the module first.
- Search on the internet. Something like 'python *modulename* documentation' or 'python *modulename* api' will almost always find you what you want (API stands for Application Programming Interface, and is the acronym often used for this type of documentation).

Both methods will end up giving you the same information, including a list of all of the functions in the module, and how to use them (i.e. how many *arguments* they take, and what each one means).

Exercises

2. Using an interactive console, import the `math` module and use the `help` function to find out the functions it contains. Find two methods of calculating the base 10 logarithm of a number. (Note that square brackets around a function argument indicate that it is optional.)

4.4 Using functions within a module

As you may remember from section 2.1.2, to use the functions in a module, you must prefix them with the module name followed by a dot. For example, in the `math` module:

```
import math

x = 3.14
y = math.sin(x)
z = math.cos(x)
```

The functions we have looked at so far all have one *argument* inside the brackets. A function can take any number of arguments (including zero). The arguments are comma-separated quantities which the function needs to perform the calculation. The arguments can be either literal values (such as numbers, True/False, or strings) or variables which have been defined elsewhere. If you give the function a variable, the name of that variable doesn't matter; it's the value of the variable that is important. So all of the below lines are valid (and give the same answer):

```
import math

x = 3.14
a_long_variable_name = 3.14
this_var_is_nearly_pi = 3.14

print math.sin(x)
print math.sin(a_long_variable_name)
print math.sin(this_var_is_nearly_pi)
print math.sin(3.14)
```

If you give the wrong number of arguments to a function, it will complain when you try and run it:

```
>>> math.sin(1,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sin() takes exactly one argument (2 given)
>>>
```

The error message you get here is a clear one: the function `math.sin()` was expecting one argument, but you gave it two.

In the last exercise, we touched on the `math.log()` function. This is a function which has an optional argument. If you look at the documentation (you can do this in an interactive console by typing `help(math.log)` or just `help(math)` and scrolling down), you'll see that it specifies the arguments as:

```
log(x[, base])
```

Here, square brackets indicate an optional argument. The documentation should specify what happens in cases where the optional arguments are present and when they are not. Indeed:

```
log(x[, base])

Return the logarithm of x to the given base.
```

```
If the base not specified, returns the natural logarithm (base e) of x.
```

4.5 Importing a module, revisited

Now we have covered the simple cases of importing and using a module we will look at other useful ways of importing and using a module. They all perform the same task — making other people's code available for us to use — but they have some advantages over the method we've seen already.

4.5.1 Giving a module a shorter name

Soon we will be looking at plotting and we're going to use the `matplotlib` module. But one thing we haven't mentioned yet is this: as well as functions, modules can contain other modules. For example, `matplotlib` has a sub-module named `pyplot`. We can import and use this submodule as follows:

```
import matplotlib.pyplot
```

Now when we want to use the functions in this module, we just have to find their names, and add `matplotlib.pyplot.` to the front of each one. If we're writing a script to generate a lot of plots, that might add up to a lot of typing. Perhaps you really enjoy repetitive tasks, but the idea of typing out `matplotlib.pyplot` 50 times doesn't fill me with joy. It would be nice if we could just refer to `matplotlib.pyplot` by a shorter name that we choose. And by a happy coincidence, that's exactly what we can do:

```
import matplotlib.pyplot as plt
```

Now, instead of typing things like `matplotlib.pyplot.plot(x,y)`, we can simply type `plt.plot(x,y)`. Much more concise! And it works with any import statement. Taking an example from earlier:

```
import math as m

x = 3.14
a_long_variable_name = 3.14
this_var_is_nearly_pi = 3.14

print m.sin(x)
print m.sin(a_long_variable_name)
print m.sin(this_var_is_nearly_pi)
print m.sin(3.14)
```

4.5.2 Importing some functions from a module

Let's say we're going to write a script which does a lot of trigonometrical operations. You want to use the `sin` and `cos` functions from the `math` module, but nothing else. Having paid

diligent attention to the previous bit, you're planning on renaming `math` to `m` to save on some typing. But wait! There's an even better trick you can pull.

Once you know which functions you want to use from a module, you can import them individually. If you just want to use `sin`, you can import it thus:

```
from math import sin

x = 3.14
a_long_variable_name = 3.14
this_var_is_nearly_pi = 3.14

print sin(x)
print sin(a_long_variable_name)
print sin(this_var_is_nearly_pi)
print sin(3.14)
```

The pattern to do this is `from` *`modulename`* `import` *`function_name`*. You can import multiple functions at the same time by separating them with commas:

```
from math import sin, cos

x = 3.14

print sin(x)
print cos(x)
```

You should use this method when a module contains a few functions which you need to use a lot. Normally you should just import the module and name it something short. This helps make it clear to someone reading the code (which may well be yourself in 6 months' time!) which bits of the code are coming from where. Of course, they should already know that from the comments (hint, hint...).

4.5.3 I'm telling you this so that you know never to do it

What I'm about to tell you is a bad idea. It may not seem like a very bad idea, and it saves you typing, but it is. If you get into the habit of it, people won't like to read your code, and it may just cause a problem which is *really* annoying to try and debug.

You should know about this so that if you see it in someone else's code, you'll know what it does. If you use it in your own scripts during this course, you will get marked down for it.

```
from math import *

print sin(pi)
print cos(pi)
print tan(pi)
print abs(pi)
print pow(pi, 2)
```

What's happened here is that everything (`*` commonly means 'everything' in programming/computing) in the `math` module has been imported *with its own name*. Great, right? Less typing, and no need for `math.` in front of everything! So what's the problem?

The problem is that in reality, scripts can contain many variables, many functions, and many imported modules. Some of these things may have the same names. Normally that's fine, because you know which module each function is from. When you use `from` *`modulename`* `import *` it's not easy to see which functions have come from where, and it's not even necessarily obvious which function you're using. Unless you know exactly what is in each module, it's a risky thing to be doing.

For example, if we have 2 modules, named `module1` and `module2`, each of which contains a function named `useful_func`, what does the following code do?

```
from module1 import *
from module2 import *

useful_func()
```

1. It runs `useful_func()` from `module1`.
2. It runs `useful_func()` from `module2`.
3. It doesn't matter, because I promise I won't ever do that.

The correct answer is 3 (however, if you see it in someone else's code, the correct answer is 2).

4.5.4 Modules don't just have to contain functions

As well as functions, modules can contain other things. We've come across modules which can contain other modules, but a module can also contain variables, and *classes*. Classes are beyond the scope of this course, and you already know about variables. In fact, we've already had an example using a module with a variable in.

Exercises

3. What was the module which contained a variable? What was the variable?
4. Is importing a variable from a module different to importing a function?

4.6 The `matplotlib` plotting module

`matplotlib` is a plotting library for python (the term 'library' is used to refer either to individual modules or collections of modules). It is capable of generating almost any sort of 2D plot you can think of. There is a gallery of plots online (`http://matplotlib.org/gallery.html`),

each of which comes with the python code you need to generate it. If you ever have an unusual type of plot to generate, the quickest thing to do is usually to go the gallery, find a similar example, and copy then modify the code.

Exercises

5. Go to the `matplotlib` gallery, choose a plot you like, copy the code into Spyder and run it, so that you generate the plot yourself. See how much of the code you understand.

6. Modify your code so that something is different (it doesn't matter what).

7. Look at some other examples in the gallery. What two lines do they all have in common?

4.6.1 Basic 2D plots

In the last set of exercises you copied and pasted some code and ran it. You may not have understood what every single line did, but you should have been able to tell roughly what each section of the code did (from the comments and the function names). And that's before we've actually learned anything about matplotlib!

You should also have found that each example contained two things: the `import` statement at the start, and `plt.show()` at the end. All of the examples used the same import statement: `import matplotlib.pyplot as plt`. This is a very common convention when dealing with `matplotlib`, and we're going to stick to it. That way, when other people read your code, it should be immediately obvious to them that you're using `matplotlib`, even if they have skimmed over the `import` statement. As for `plt.show()`, it will come as no real surprise that this shows the plots you've drawn. This comes at the end of your script, and once `plt.show()` has been called, you will get an interactive pop-up window where you can zoom in and out, pan around, and save your figure.

So in between these two statements is where the actual plotting work goes. To start with, we're going to generate simple 2D plots of y against x. The function we're going to use is `plt.plot()`. In its simplest form, `plt.plot()` takes two lists of the same length as arguments, uses the values as the x and y values, and generates a plot of y vs x with the points joined by a blue line.

Exercises

8. Create lists named `x` and `y`, each of which contains five numbers, and then call the `plot` function using `x` and `y` as arguments. Finally, display the plot on the screen.

All well and good, but perhaps we don't want to plot a solid line — maybe we want to see exactly where the points we've plotted are. `plt.plot()` takes an optional third argument

which specifies what style to plot in. This is a string (so it goes between inverted commas) and it consists of two possible parts: a letter representing the colour, one or two characters representing the line style, and another character representing the point style. It doesn't matter which order you put these things in, and you don't have to specify all of them. For example, to plot a red (`'r'`) dashed line (`'--'`) with circular markers (`'o'`), the style string is: `'r--o'` (although you could also use `'ro--'`, `'--or'`, `'--ro'`, `'o--r'`, `'or--'` since they are all equivalent).

Exercises

9. What are the style strings for the following styles:
    - Black dots for the points, no line
    - Green triangles pointing to right joined by a solid line
    - Yellow pentagons for the points, joined by a dash-dot line
    - Magenta stars for the points, joined by a dotted line
10. Modify your script from the previous exercise so that the line is plotted in a style of your choice.

As soon as you call `plt.plot` in a program, a new figure is created and a line plotted on it. After that, every call to `plt.plot` will result in a new line being added to the same figure. If you want a to create two separate plots in the same program, you need the command `plt.figure()`, which creates a new figure. The procedure is then (for example):

- Use `plt.plot()` to create a plot
- Use `plt.plot()` to add lines to the same plot
- Use `plt.figure()` to create a second plot
- Use `plt.plot()` to add lines to the second plot
- Use `plt.figure()` to create a third plot
- Use `plt.plot()` to add lines to the third plot
- Continue until you have all of the graphs you want
- Use `plt.show()` to display all of the graphs

4.6.2 Aside: more on python help

When you got help on the `plt.plot` function in previous exercises, you should have seen something like this:

```
plot(*args, **kwargs)
    Plot lines and/or markers to the
    :class:`~matplotlib.axes.Axes`.  *args* is a variable length
    argument, allowing for multiple *x*, *y* pairs with an
    optional format string.  For example, each of the following is
    legal::

        plot(x, y)         # plot x and y using default line style and
   color
        plot(x, y, 'bo')   # plot x and y using blue circle markers
        plot(y)            # plot y using x as index array 0..N-1
        plot(y, 'r+')      # ditto, but with red plusses

    If *x* and/or *y* is 2-dimensional, then the corresponding columns
    will be plotted.

    An arbitrary number of *x*, *y*, *fmt* groups can be
    specified, as in::

        a.plot(x1, y1, 'g^', x2, y2, 'g-')

    Return value is a list of lines that were added.
```

What does `plot(*args, **kwargs)` mean? The `*args` bit means that this function can take different numbers of arguments. That makes sense, since we've seen that we can use `plt.plot()` in two ways: with a style string and without one (which require three and two arguments respectively). The `**kwargs` bit means that we can also have *named*, or *keyword* ('kw') arguments as well. If you look a bit further down the documentation, you'll see the following examples:

```
plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
plot([1,2,3], [1,4,9], 'rs',  label='line 2')
```

In these examples, `label` and `linewidth` are the named arguments. In python, all of the named arguments must be given to the function after the other arguments. `label` is a useful one, since it allows us to give each line a label, which can be used in a legend. That can be very useful! Let's look at that in a bit more detail.

4.6.3 Labelling your plot

Exercises

11. Modify your script so that it now plots two different lines on the same graph, and give them each a label. Did the plot you produced look as expected?

12. Make the plot appear in the way you expect.

Having completed the above exercise, you've found out how to label plots and generate a legend. There are two other essential parts of generating a plot: labelling the axes, and giving it a title. `matplotlib` makes this very simple. The commands are:

- `plt.title('The title of your plot')`
- `plt.xlabel('The title of the x-axis')`
- `plt.ylabel('The title of the y-axis')`

Exercises

13. Finally add a title and some axis labels to your plot. Now save it to disk.

4.7 3D plots

So far we've seen how to plot one variable (y) against another (x) using the `plot` function. What about if we want to plot the value of a third variable, z, against both x and y? We might want to do this if we have values of temperature at a set of locations and want to see where's hottest and where's coldest. In this case it would be good if different values of z were represented by different colours. In this section we'll look at two ways to make 3D plots: `plt.scatter` and `plt.imshow`. As you can see they are both contained in the `matplotlib.pyplot` library, which has been imported with the name `plt`.

4.7.1 Plotting with `scatter`

Firstly we'll use the `plt.scatter` function to plot a set of temperatures at different locations. Try this:

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y = [4,1,10,2,3]
T = [50,40,30,20,10]
plt.scatter(x, y, c = T, s = 200, cmap = 'coolwarm')
plt.show()
```

There are a few things going on here. Firstly, temperature is represented by the `T` variable and has values between 10 and 50°C. The values of `x` and `y` represent coordinates at which the temperatures were measured. Finally, instead of the `plt.plot` function, `plt.scatter` has been used. This works similarly to `plot` but takes some new arguments:

- `c = T` uses the values of `T` as the colour,

- `s = 200` makes the markers bigger than they are initially,
- `cmap = 'coolwarm'` uses a particular *colour map*, which in this case goes between blue (low values) and red (high values).

Exercises

14. Find an online list of colour maps and try a few different ones in the code above.
15. Using `plt.scatter`, plot the longitude (`x`), latitude (`y`) and average July temperature (`T`) for four cities of your choice.
16. Draw the same plot but this time make the marker sizes proportional to the city populations. Choose the marker sizes sensibly; I recommend using multiples of 10000.

4.7.2 Plotting with `imshow`

Sometimes you may need to plot a regularly-spaced grid of values stored in a 2D array. This could, for example, be a $1° \times 1°$ map of surface air temperatures across the globe. You could use `plt.scatter` for this, but it would be more convenient to be able to pass the array directly to a plotting routine without having to worry about latitude/longitude. You can use `plt.imshow` to achieve this, for example:

```
import matplotlib.pyplot as plt
T = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
plt.imshow(T, interpolation = 'none')
plt.show()
```

Note that in this case x and y were not specified explicitly. Instead, when using `imshow` it is assumed that they are regularly spaced according to the size of the array to be plotted (which in this case is 3×3).

Exercises

17. Use `imshow` to plot a 4×2 grid of numbers.
18. How can you flip the y-axis so (0, 0) is at the bottom left of the plot?
19. What does `interpolation = 'none'` mean? Search online to find different interpolation options and try one.

4.7.3 Colour bars

So far we've used both `plt.scatter` and `plt.imshow` to make 3D plots with the colour showing the values of z. However, a casual viewer won't know which colour refers to which value. To fix this, we can use a *colour bar* which shows the correspondence between colours and z-values. Let's go ahead and add a colour bar to the scatter plot we made before:

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y = [4,1,10,2,3]
T = [50,40,30,20,10]
plt.scatter(x, y, c = T, s = 200, cmap = 'coolwarm')
plt.colorbar()
plt.show()
```

That's it! You should see a colour bar to the right of your scatter plot. We simply told python to add the colour bar with `plt.colorbar()`. Note that `color` is spelt without a 'u'!

Exercises

20. Add a colour bar to the `imshow` plot you made above.

21. Add a colour bar to the `scatter` plot you made in Exercise 16. Find out how to give the colour bar a suitable title to show what it represents.

4.8 Summary

You should now know about the following things:

- How to import modules
- How to rename modules so that you don't need to type so much
- How to import just the bits you need from a module
- How to use the `matplotlib` module to create basic 2D plots with the `plot` command
- How to label your plots appropriately
- How to make 3D plots using `scatter` and `imshow`

Although the `plt.plot()` function is not very complicated to use, it is one of the most useful functions in python for a scientist. Once you're used to using it it's considerably quicker than using Excel to draw a graph, and so is invaluable when you're exploring data. Combined with reading data from files (which we will cover next session), it is one of the simplest yet most powerful tools you'll come across.

4.9 Answers to exercises

Answers

1. We're going to use `matplotlib` for plotting. You should either find this by searching on the internet (Googling 'python plotting module' works pretty well) or by asking someone who has experience of python (if you'd asked me, I'd have recommended `matplotlib`). Don't worry if you chose another one — the point is to emphasise that you should be using the internet to get answers when programming.

2. `log10(x)` will find the base 10 logarithm, but so will `log(x, 10)`.

3. The `math` module contains the variable `math.pi`, representing π.

4. Importing variables and functions works in exactly the same way!

5. This may seem a little pointless — it's an exercise in copying and pasting — but you should get used to the fact that often you will be doing just that. If you understood all of the code, well done!

6. It doesn't matter what change you made — if your code ran successfully and the graph looked different to the last exercise you've done it right!

7. They all have `import matplotlib.pyplot as plt` near the start, and they all end with `plt.show()`.

8. For example:

```
# Import the plotting module
import matplotlib.pyplot as plt

# The variables to plot
x = [1,2,3,4,5]
y = [1,4,9,16,25]

# Create the plot
plt.plot(x,y)

# Display the plot
plt.show()
```

9. By using `help(plt.plot)`:

   - `'k.'`
   - `'g>-'`
   - `'yp-.'`

- `'m*:'`

10. For example:

```
# Import the plotting module
import matplotlib.pyplot as plt

# The variables to plot
x = [1,2,3,4,5]
y = [1,4,9,16,25]

# Create the plot with a cyan dashed line, marking each point with
   a hexagon
plt.plot(x, y, 'ch--')

# Display the plot
plt.show()
```

11. For example:

```
# Import the plotting module
import matplotlib.pyplot as plt

# The variables to plot
x1 = [1,2,3,4,5]
y1 = [1,4,9,16,25]

x2 = [1,2,3,4,5]
y2 = [8,6,4,2,0]

# Create the plots
plt.plot(x1, y1, 'r', label='x-squared')
plt.plot(x2, y2, 'g', label='10 - 2x')

# Display the plot
plt.show()
```

Although the lines have been labelled, the legend has not appeared.

12. Once you have added the lines to your plot, you need to call the function `plt.legend()`.

13. For example:

```
# Import the plotting module
import matplotlib.pyplot as plt

# The variables to plot
x1 = [1,2,3,4,5]
y1 = [1,4,9,16,25]
```

```
x2 = [1,2,3,4,5]
y2 = [8,6,4,2,0]

# Create the plots
plt.plot(x1, y1, 'r', label='x-squared')
plt.plot(x2, y2, 'g', label='10 - 2x')

plt.legend()

plt.title('Some extremely interesting results')
plt.xlabel('x')
plt.ylabel('f(x)')

# Display the plot
plt.show()
```

You can then save the figure using the save button on the interactive window.

14. A list of colour maps can be found here. Different colour maps can be used by changing the word `blah` inside the `cmap = 'blah'` argument. Take a look at `viridis`, `hot` and `gist_earth`.

15. For example:

```
import matplotlib.pyplot as plt

#Cities used are London, New York, Sydney and Johannesburg
x = [0, -74, 151, 28]
y = [51, 40, -33, -26]
T = [19, 25, 13, 11]
plt.scatter(x, y, c = T, s = 200, cmap = 'coolwarm')
plt.show()
```

(Note: there are some plotting routines which can deal with latitude/longitude maps properly, including drawing different map projections and plotting the continents automatically. That's beyond the scope of this course but is something you may want to bear in mind.)

16. For example:

```
import matplotlib.pyplot as plt

#Cities used are London, New York, Sydney and Johannesburg
x = [0, -74, 151, 28]
y = [51, 40, -33, -26]
T = [19, 25, 13, 11]
#population in multiples of 10000
N = [900, 800, 500, 100]
```

```
plt.scatter(x, y, c = T, s = N, cmap = 'coolwarm')
plt.show()
```

17. For example:

```
import matplotlib.pyplot as plt
T = [[1, 2, 3, 4], [5, 6, 7, 8]]
plt.imshow(T, interpolation = 'none')
plt.show()
```

18. You can move the origin in the call to `imshow` as follows:

```
import matplotlib.pyplot as plt
T = [[1, 2, 3, 4], [5, 6, 7, 8]]
plt.imshow(T, interpolation = 'none', origin = 'lower')
plt.show()
```

19. `interpolation = 'none'` draws the plot with sharp boundaries between the edges of each box. It is possible to use smoother interpolation methods, for example:

```
import matplotlib.pyplot as plt
T = [[1, 2, 3, 4], [5, 6, 7, 8]]
plt.imshow(T, interpolation = 'gaussian', origin = 'lower')
plt.show()
```

20. For example:

```
import matplotlib.pyplot as plt
T = [[1, 2, 3, 4], [5, 6, 7, 8]]
plt.imshow(T, interpolation = 'none', origin = 'lower')
plt.colorbar()
plt.show()
```

21. The colour bar can be given a title as follows:

```
import matplotlib.pyplot as plt

#Cities used are London, New York, Sydney and Johannesburg
x = [0, -74, 151, 28]
y = [51, 40, -33, -26]
T = [19, 25, 13, 11]
#population in multiples of 10000
N = [900, 800, 500, 100]
plt.scatter(x, y, c = T, s = N, cmap = 'coolwarm')
cb = plt.colorbar()
cb.set_label("Average July temperature (degrees C)")
plt.show()
```

5 Reading/writing text files and the `numpy` module

In section 4 we looked at using `matplotlib` to plot simple graphs of data which we entered by hand. The ability to create plots is very useful, but it is fairly limited if we have to type in everything. In this section we're going to look at reading data from a file so that we can plot more useful graphs. And since we're covering reading data from files, we're also going to cover writing data to files.

We're also going to look at the basics of the `numpy` (numeric python) module; this is a module which is very widely used in scientific computing, and which you'll come across often throughout your degree.

There are many different formats for storing data which you might come across, but we're only going to cover two in this course: text files and CSV files (which are just text files with a pre-defined layout). A lot of simpler meteorological data is supplied in CSV format (including all of the data from the atmospheric observatory), and it is a very commonly understood data format.

5.1 Reading from text files

To open a file we will use the python built-in function `open`. This takes a single argument, the name of the file to read, and returns a *file object*. This file object can then be used to read the data, either all at once or a line at a time.

Create a new script in Spyder, download `textfile.txt` from blackboard **into the same directory as the script you just created**, and run the following code in it:

```
f = open('textfile.txt')
text = f.readlines()
f.close()

for line in text:
    print line
```

This opens the text file, reads the entire file into a list of strings (each element of the list is a single line in the file), and closes the file. We then loop over the list and print each line to the screen.

You can also use the following method to do the same thing:

```
f = open('textfile.txt')
for line in f:
    print line
f.close()
```

which loops over the lines of text in the file one at a time. Generally the first method is preferable, because it means that we have the contents of entire file in memory, and can access it later in the program. The second method is better if you have a very large file (e.g. $>$ 100MB) because often we only want to scan through the file once, and don't want to fill up the computer's memory with something we're not going to use again. Since we won't be working with very large files, I'd recommend using the first method throughout this course.

However, it's useful to be aware of the second method for when you come across it in other people's code, or when you need to deal with large data files.

5.2 Reading from CSV files

A CSV (comma-separated values) file is a very common way of storing numerical data. It is just a text file with the suffix '.csv' and the following format:

```
column1,column2,column3
1,2,3
4,5,6
7,8,9
10,11,12
...
```

The first row is a list of text descriptions of what the columns contain, each separated by a comma. All of the rows after that are values separated by a comma. This format is understood by Excel, as well as many other data manipulation programs.

We read CSV files in exactly the same way as we read text files. Then by processing each line of text correctly, we can store the individual columns of data in separate lists. Download `temps_jan_2015.csv` from Blackboard. First, open in it a text editor to see what each column contains. You can use Spyder, but you will need to select 'All files (*)' before it shows up in the open dialog. You can also use WordPad or Excel. Once you have done that, try the following code:

```
f = open('temps_jan_2015.csv')
text = f.readlines()
f.close()

days = []
temperatures = []

# We want to skip the first line, because it just has the column names
for line in text[1:]:
    fields = line.split(',')
    days.append(float(fields[0]))
    temperatures.append(float(fields[1]))

print days
print temperatures
```

Note that we have introduced the `split()` function. If we take a variable which contains a string (here `line`) we can call its `split()` function as above. That will split the string into a list of other strings. The argument to `split()` is the character which we should split the string at. So the above code takes the variable `line` and splits it at every comma, giving a list of strings which we've named `fields`. Try adding `print fields` after the line `fields = line.split(',')` in the above code to see what `fields` contains.

Exercises (answers in section 5.5)

1. What is the purpose of the `float()` command in the above code?
2. Modify the above script so that it plots a graph of days against mean temperatures. Make sure to add appropriate axis labels and a title.
3. Download `min_max_temps_reading_2015.csv` from Blackboard. This file contains minimum and maximum temperature records for every day in 2015. Create a plot with two lines — the minimum and maximum temperatures in 2015.
   Hints:
   - Examine the file before you do any coding.
   - To represent the dates, you should use the `datetime` module which you came across in assignment 3
   - You can create the datetimes as you go along — there is no need to store the years, months and days in separate lists first.
4. Using `min_max_temps_reading_2015.csv`, create a graph of the difference between maximum and minimum daily temperatures for 2015.

5.3 Writing to file

Opening a file to write to is a very similar process to opening it to read from. The only difference is that the `open` method takes an additional argument, stating that the file should be able to be written to. If you give the name of a pre-existing file **it will be overwritten**, so be careful! Once you have opened the file for writing, you can use the `write()` function to add lines of text. Let's look at an example:

```
# The 'w' argument says that this file can be written to
f = open('output_test.txt', 'w')

f.write('This text is written to the file')
f.write('Now this text is written to the file')
f.write('So is this text.  Simple, hmm?')
f.write('But wait...  Is this actually doing what we want?')
f.close()
```

Try running the above code. You should see the file `output_test.txt` created on your filesystem, in the same directory as the script. Open it in a text editor and look at the contents. Are they what you expect? (That's a rhetorical question — the answer is no!)

Exercises

5. If you want to add to a file rather than overwrite it, how can you do that?

6. Find out how you can make each line appear on a new line, and modify the above program so that it does that.

Note that in all of these examples, we have used the variable `f` to store the file object. This is a common convention, but of course you can name it whatever you like. In the case where you have multiple files open at the same time, you'll need more than one variable. Give them sensible names! For example, if we want to copy the contents of one file to another in python, we could do this:

```
# The file to read from
in_file = open('textfile.txt')

# The file to write to
out_file = open('output_textfile.txt', 'w')

for line in in_file:
    out_file.write(line)

in_file.close()
out_file.close()
```

A few things to note:

- When we *read* lines from a file, they already contain the newline character at the end
- Here, I have chosen the method of looping over the input file one line at a time and writing it to the output file.

Exercises

7. Change the above script so that it reads the entire contents of `textfile.txt` into a list, closes the input file, and *then* writes it to the output file.

5.3.1 Writing numbers to file

The `write()` function takes a string and writes it to a file. That's all very well, but we're going to want to write numbers to file. In python, there are two main ways of converting a number to a string:

- `str()`: This function takes a number (i.e. a literal number or a variable containing a number) and converts it to a string. It's simple, and will choose what it thinks is a sensible number of decimal places for you. In most cases, this is the best function to use.

- `format()`: This function is used for specifying precisely how you want your numbers to appear (e.g. how many decimal places, padding the number with leading zeroes etc.). You should be aware of it, but we are not going to use it in this course; `str()` will be sufficient for all our needs.

Exercises

8. Download the file `temps_jan_2016.csv` from Blackboard. This is a slightly different format to `temps_jan_2015.csv` — it contains full dates, and all of the temperatures are in Kelvin. Write a script to read this file, and write a new file which is in the same format, and has the same units, as `temps_jan_2015.csv`.

5.4 The `numpy` module

While python lists are fine for dealing with the sort of data we've been using, they can be a little too slow when dealing with the sort of very large data sets which you often encounter in environmental science. They also behave rather counterintuitively regarding multiplication and other mathematical operations. To overcome these things, the `numpy` module was created. The primary aim of `numpy` is to provide fast multidimensional arrays for use in place of python lists. There are a few important differences between `numpy` arrays and python lists:

- The size of a `numpy` array is fixed — you cannot append new elements to it
- All of the items in a `numpy` array must be the same type (e.g. integers)
- Multiplying a `numpy` array by a number will multiply every element in the array by that number
- If you multiply two `numpy` arrays of the same length together, each pair of elements is multiplied to give a new array

Apart from those differences, `numpy` arrays can be use in much the same way as python lists. You will find that `numpy` will be used throughout your degree courses in place of python lists.

There are a few ways to create `numpy` arrays. The simplest is to create an array from a python list:

```
# We import numpy and name it np.  This is an almost universal convention
import numpy as np

arr = np.array([1.0,2.0,3.0,4.0])

print arr
print 4 * arr
print arr * arr
```

Exercises

9. There are other ways to create `numpy` arrays. Find out how to use the following functions to create arrays:

   - `np.arange()`
   - `np.zeros()`
   - `np.ones()`
   - `np.linspace()`

As well as providing fast arrays, there are a many useful numerical functions available in the `numpy` module. There is a useful `numpy` tutorial here: `https://docs.scipy.org/doc/numpy-1.14.0/user/quickstart.html`. Go ahead and work through that tutorial, up to (not including) the section 'Less Basic'. Then try the exercises below.

Exercises

10. Write a script which generates a linear range of 500 numbers between -2 and 2 using `numpy`. Plot the square, and the cube of those numbers on a graph.

11. Create a `numpy` array, of shape 5×6, which is filled with 1 everywhere. Set all of the elements in a row of your choice to 3, and all of the elements in a column of your choice to -1. Choose a 3×2 block somewhere within the array and set the values in that block to 2. Print the array. Take the sine of the entire array, multiply the result by 6, and print it again. Take the transpose of the array and print the result. 'Flatten' the array so it has only one dimension and print it again. Finally, print the minimum, maximum, mean, and standard deviation of the values in the array.

Using python lists to do the tasks in the previous question would either require more lines of code (e.g. several `for` loops) or horribly convoluted 'list comprehensions'. The `numpy` approach is easier to write and also to easier understand when revisiting the code at a later date. More importantly, when dealing with large matrices (e.g. 1000×1000), using python lists would take a very long time to execute. `numpy` is optimised to deal with this sort of case and will produce results quickly (try it!).

5.5 Answers to exercises

Answers

1. `float()` is necessary because `fields[0]` and `fields[1]` contain *text*, **not** numbers.

2. This can be done using `matplotlib`; see the previous chapter if you need to refresh your memory.

```
import matplotlib.pyplot as plt

f = open('temps_jan_2015.csv')
text = f.readlines()
f.close()

days = []
temperatures = []

# We want to skip the first line, because it just has the column
    names
for line in text[1:]:
    fields = line.split(',')
    days.append(float(fields[0]))
    temperatures.append(float(fields[1]))

plt.plot(days, temperatures)
plt.xlabel('Date')
plt.ylabel('Mean temperature (C)')
plt.title('Mean Temperatures in January 2015')
plt.show()
```

3.

```
import matplotlib.pyplot as plt
from datetime import datetime

f = open('min_max_temps_reading_2015.csv')
text = f.readlines()
f.close()

dates = []
min_temps = []
max_temps = []

# We want to skip the first line, because it just has the column
    names
for line in text[1:]:
    fields = line.split(',')

    year = int(fields[0])
    month = int(fields[1])
    day = int(fields[2])

    date = datetime(year, month, day)
    dates.append(date)
    min_temps.append(float(fields[3]))
```

```
    max_temps.append(float(fields[4]))

plt.plot(dates, min_temps, label='Minimum temperature (C)')
plt.plot(dates, max_temps, label='Maximum temperature (C)')
plt.legend()
plt.xlabel('Date')
plt.ylabel('Temperature (C)')
plt.title('Minimum and Maximum Temperatures in 2015')
plt.show()
```

4.
```
import matplotlib.pyplot as plt
from datetime import datetime

f = open('min_max_temps_reading_2015.csv')
text = f.readlines()
f.close()

dates = []
temp_diffs = []

# We want to skip the first line, because it just has the column
   names
for line in text[1:]:
    fields = line.split(',')

    year = int(fields[0])
    month = int(fields[1])
    day = int(fields[2])

    date = datetime(year, month, day)
    dates.append(date)
    temp_diffs.append(float(fields[4]) - float(fields[3]))


plt.plot(dates, temp_diffs)
plt.xlabel('Date')
plt.ylabel('Temperature (C)')
plt.title('Difference between min/max temperatures in 2015')
plt.show()
```

5. Use `open('outputfile.txt', 'a')` to **append** to the file.

6. Add newline characters (`\n`) to the end of each line:

```
# The 'w' argument says that this file can be written to
f = open('output_test.txt', 'w')

f.write('This text is written to the file\n')
f.write('Now this text is written to the file\n')
```

```
f.write('So is this text.  Simple, hmm?\n')
f.write('But wait...  Is this actually doing what we want?\n')
f.write('Yes, yes it is.\n')
f.close()
```

7.
```
# The file to read from
in_file = open('textfile.txt')
# Read the entire file
lines = in_file.readlines()
# Close the file
in_file.close()

# The file to write to
out_file = open('output_textfile.txt', 'w')

# Write the contents to the output file
for line in lines:
    out_file.write(line)

# Close the output file
out_file.close()
```

8.
```
# The file to read from
in_file = open('temps_jan_2016.csv')
# Read the entire file
lines = in_file.readlines()
# Close the file
in_file.close()

# The file to write to
out_file = open('temps_jan_2016_out.csv', 'w')

# Write the header
out_file.write('day,mean_temp\n')

# Skip the header line, and write data to output file
for line in lines[1:]:
    fields = line.split(',')

    # We don't need to convert the day to a number
    # since we're just going to write it straight out
    day_str = fields[2]

    # Read the temperature in Kelvin as a number
    temperature_k = float(fields[3])
    # Now convert it to degrees Celsius
    temperature_c = temperature_k - 273.15

    # Now write out the day and the temperature
```

```
    out_file.write(day_str+','+str(temperature_c)+'\n')

# Close the output file
out_file.close()
```

9.
- `np.arange()`: This behaves like the `range()` function in python, but returns a `numpy` array instead of a list.
- `np.zeros()`: This takes either a number or a list of numbers. For a single number, it creates a 1D array of zeros with the specified number of elements. For a list of numbers, it creates an nD array (where n is the number of elements in the list) with the given sizes of dimension. For example `np.zeros([3,4])` creates a 3×4 array of zeros.
- `np.ones()`: This is exactly like `np.zeros()`, except it fills the arrays with ones.
- `np.linspace()`: This takes 3 arguments, `start`, `stop`, and `num`, and creates a linearly spaced array of `num` values going from `start` to `stop` inclusive.

10.
```
# Import the modules we're going to use
import numpy as np
import matplotlib.pyplot as plt

# Create the range of x values
x = np.linspace(-2,2,500)

# Because we're using numpy, the square and cube are simple
y1 = x**2
y2 = x**3

# Now plot the data and show it
plt.plot(x,y1)
plt.plot(x,y2)

plt.show()
```

11. For example:

```
import numpy as np

#Create a 5 * 6 numpy array filled with 1 everywhere
a = np.ones(shape = (5, 6))

#Set all values in a chosen row to 3
a[4, :] = 3

#Set all values in a chosen column to -1
a[:, 0] = -1
```

```
#Set values within a chosen 3 * 2 block to 2
a[1:4, 2:4] = 2

print a

#Take the sine of the entire array
b = np.sin(a)

#Multiply the array by 6
b = b * 6

print b

#Take the transpose of the array
c = np.transpose(b)

print c

#Flatten the array
d = np.ravel(c)

print d

#Print the minimum, maximum, mean and standard deviation of the
   values in the array
print np.min(d)
print np.max(d)
print np.mean(d)
print np.std(d)
```

6 Conclusion

Now that you have completed all of the notes along with the exercises and assignments, please go back to section 1.4 and make sure that you are comfortable with all of the points listed. If there are things you are not sure about, go back through the notes, do some exercises again, talk to the lecturer, the demonstrators, and your classmates. Try some programming of your own, or find some python code online and see how much of it you can understand.

Next time you're sat at a computer and you need to perform some calculations, try opening python instead of the default calculator (or Google). And when you need to generate graphs, don't just fire up Excel, load up Python(x,y) and create a script to do it for you.

One of the objectives of this course was to get you familiar with programming in general. In particular, the concept of breaking down a large task into small steps is a key part of programming, regardless of the language you are using. If you can do this effectively it will stand you in good stead when you come across new programming problems in the future.

6.1 Summary

This section contains a summary of the key ideas that have been introduced in the handout.

- The `print` statement is used to write output to the screen.
- It's good practice to use `float` when performing calculations.
- Beware of inadvertent integer division, e.g. `2 / 3` may give an answer of `0`.
- Lists are collections of objects such as numbers (`[1, 2, 3]`) and strings (`["a", "b"]`).
- `for` loops (and the `range` function) are very important for looping over lists, etc.
- `while` loops are useful for doing actions until a particular condition is satisfied.
- `if`/`elif`/`else` are very useful for making decisions and controlling program flow.
- Functions (`def xyz`) are essential for longer programs; they enable you to divide the problem into smaller chunks, which is good programming practice.
- The `math`, `datetime` and `random` modules.
- The `matplotlib` module, and in particular `matplotlib.pyplot` (usually imported as `plt`); the plotting routines `plt.plot()`, `plt.scatter()` and `plt.imshow()`.
- Reading and writing text and CSV files with `open`, `readlines` and `write`.
- `numpy`: fast and efficient multidimensional array manipulation.

7 References

This is a list of useful python/programming references. It collects any links which were in the text, along with additional resources which you may find helpful. If you come across any other resources which you find helpful, please email us and let us know — we will try to include them in the course next year!

- `http://docs.python.org/2/tutorial/`: The official python tutorial.
- `https://docs.python.org/2/library/index.html`: Documentation for all of the standard libraries which come with python, as well as built-in functions.
- `http://python-xy.github.io/`: Python(x,y) — a good python distribution for Windows.
- `http://www.numpy.org/`: `numpy` website. *The* numerical library for scientific programming in python.
- `http://matplotlib.org/`: `matplotlib` website. A top-quality plotting library for python.
- `http://matplotlib.org/gallery.html`: The `matplotlib` gallery. This should be your first stop when you want to draw a particular graph and you're not sure how to go about it.
- `https://developers.google.com/blockly/`: Blockly, a visual programming website. Good for getting to grips with some of the simple programming concepts such as variables and loops.

8 Acknowledgements

Many thanks to Guy Griffiths who wrote the first versions of the notes, and Chris Thomas for updating them and discussing them with me.