Maths with Python Documentation

Release 1.0

Mathematical Sciences, University of Southampton

Oct 13, 2016

Contents

1 First Steps **3**
- 1.1 First steps . . . 3
- 1.2 How to use these notes . . . 3
- 1.3 Python . . . 4
- 1.4 Spyder . . . 6
- 1.5 Reading list . . . 8
- 1.6 Versions . . . 8

2 Python Basics **9**
- 2.1 Python . . . 9
- 2.2 Debugging . . . 14
- 2.3 Exercise: Variables and assignment . . . 17

3 Programs **19**
- 3.1 Programs . . . 19
- 3.2 Using programs and modules . . . 20
- 3.3 Functions . . . 20
- 3.4 Printing and strings . . . 27
- 3.5 Putting it together . . . 29
- 3.6 Exercise: basic functions . . . 30
- 3.7 Exercise: Floating point numbers . . . 31

4 Loops - how to repeat yourself **33**
- 4.1 Loops . . . 33
- 4.2 Containers and Sequences . . . 37
- 4.3 Control flow . . . 41
- 4.4 Debugging . . . 44
- 4.5 Exercise: Prime numbers . . . 46

5 Basic Plotting **49**
- 5.1 Plotting . . . 49
- 5.2 Exercise: Logistic map . . . 53

6 Classes and objects **55**
- 6.1 Classes and Object Oriented Programming . . . 55
- 6.2 Exercise: Equivalence classes . . . 59

7 Scientific Python **61**
- 7.1 Scientific Python . . . 61

7.2 numpy . . . 61
7.3 Plotting . . . 64
7.4 scipy . . . 75
7.5 Exercise: Lorenz attractor . . . 82
7.6 Exercise: Mandelbrot . . . 83
7.7 Exercise: The shortest published Mathematical paper . . . 84

8 Symbolic Python **87**
8.1 Symbolic Python . . . 87
8.2 sympy . . . 87
8.3 Further reading . . . 97
8.4 Exercise : systematic ODE solving . . . 97

9 Statistics **99**
9.1 Statistics . . . 99
9.2 Getting data in . . . 99
9.3 Basic statistical functions . . . 100
9.4 Categorical data . . . 104
9.5 Regression . . . 105
9.6 Random numbers . . . 106
9.7 Exercise: Anscombe's quartet . . . 109

10 Exceptions and Testing **111**
10.1 Exceptions and Testing . . . 111
10.2 Exceptions . . . 111
10.3 Testing . . . 117

11 Iterators and Generators **127**
11.1 Iterators and Generators . . . 127
11.2 Exercise : twin primes . . . 130
11.3 Exercise : a basis for the polynomials . . . 131

12 Classes and OOP **133**
12.1 Classes and Object Oriented Programming . . . 133
12.2 Exercise: Equivalence classes . . . 138
12.3 Exercise: Rational numbers . . . 139

13 Indices and tables **141**

This is material for an introductory Python course for first year undergraduate Mathematics students at the University of Southampton.

CHAPTER 1

First Steps

1.1 First steps

Programming is about getting the computer to do the calculation for you. This is needed when the calculation is long and has many repetitive steps. It does *not* mean that you can get the computer to understand things for you: usually you need to understand the steps before telling the computer what to do!

Using a computer, particularly for mathematical or scientific purposes, involves a lot more than programming. There is also

- *Algorithmic thinking*: understanding how to convert the solution to a problem into a sequence of steps that can be followed without further explanation.
- *Efficient implementation* and *complexity*: there are many ways to solve a given problem, which will give equivalent answers in principle. In reality, some solutions will solve some problems to a reasonable accuracy in reasonable time, and it can be important to be able to check which solutions work in which cases.
- *Effective implementation*: solving a problem on a computer *once* is great. Being able to re-use your solution on many problems is much better. Being able to give your code to anybody else, and it working for them, or saying *why* it won't work, without further input from you, is best.
- *Reproducible science*: in principle, any scientific result should be able to be checked by somebody else. With complex scientific code, presenting and communicating its contents so that others can reproduce results is important and not always easy.

First, we will get the computer to do *something*, and later worry about doing it efficiently and effectively. Your time is more valuable than the computer's (literally: compare the hourly cost of computer time through eg Amazon, typically *much* less than $5 per hour, against the minimum wage). We want the computer doing the work, and only when that wastes your time should you worry about the speed of the calculation.

1.2 How to use these notes

1.2.1 The material

The four essential sections are on the basics, programs, loops and flow control, and basic plotting. You should work through the notes by typing in the commands as they appear, ensuring that you understand what's going on, and seeing where you make mistakes. At the end of each section, try the exercises that you think you can tackle. Also look back at previous exercises and see if you can solve them more straightforwardly with your additional knowledge.

The section on classes should be read before reading the other sections: the details of creating your own classes won't be needed for later sections, but some understanding is important. The section on scientific Python is then the most important and should be explored in detail. At this point you should be able to tackle most of the exercises.

The sections on symbolic Python and statistics should then be covered to get an overview of how Python can be used in these areas. The section on LaTeX is not directly related to programming but is essential for writing mathematical documents. Further sections are useful as your codes get more complex, but initially are less important.

1.2.2 How to work when coding

When working on code it is often very useful to work in pairs, or groups. Talk about what you're doing, and why you're doing it. When something goes wrong, check with other people, or explain to them what you're trying to do (rubber duck debugging). When working on exercises, use pair programming techniques. If there's more than one way of doing something, try them all and see which you think is best, and discuss why.

There is no "one right way" to code, but well documented, easy to understand, clearly written code that someone else can follow as well is always a good start.

1.3 Python

To introduce programming we will use the Python programming language. It's a good general purpose language with lots of tools and libraries available, and it's free. It's a solid choice for learning programming, and for testing new code.

1.3.1 Using Python on University machines

A number of Python tools are available on a standard university desktop machine. We will mostly be using Python through spyder, which allows us to write, run, test and debug python code in one place. To launch spyder, either type `spyder` in the search bar, or go to `Start`, then `All Programs`, then `Programming Languages`, then `Anaconda`, then choose `spyder`.

1.3.2 Using Python on your own machine

As Python is free you can install and run it on any machine (or tablet, or phone) you like. In fact, many will have Python already installed, for the use of other software. However, for programming, it is best to have an installation that all works together, which you can easily experiment with, and which won't break other programs if you change something. For these reasons, we recommend you install the anaconda distribution.

Anaconda

If you have enough bandwidth and time (you will be downloading about 1G of software) then you can use the Anaconda graphical installer. There are two versions of Python: a Python `2.X` and a Python `3.X`. There are small differences between the two. Everything we show here will work on either version. We will be using the `3.X` version.

The Anaconda package installs both the essential Python package and a large amount of useful Python software. It will put a launcher icon on your desktop. Clicking on the launcher will bring up a window listing a number of applications: we will be using `spyder` as seen below.

miniconda

If you do not want to download all the Python packages, but only the essential ones, there is a smaller version of Anaconda, called miniconda. First, download the miniconda package for your computer. Again, we will be using the `3.X` version.

The miniconda package installs the basic Python and little else. There are a number of useful packages that we will use. You can install those using the `conda` app (either via the launcher, or via the command line). But before doing that, it is best to create an environment to install them in, which you can modify without causing problems.

Environments

Packages may rely on other packages, and may rely on *specific versions* of other packages in order to work. This can lead to "dependency hell", when you need (for different purposes) package `A` and package `B` which rely on conflicting versions of package `C`.

The answer to this is *environments*, which allow you to organize your different packages to minimize conflicts. Environments are like folders, and you have one for each project you are working on. That way, you ensure that updating or installing packages for one project does not cause problems for a different project.

To get the most out of environments, we need to use the command line, or terminal.

Terminals

A *terminal*, or a *command prompt window*, is a window where commands can be typed in to directly run commands or affect files. On Windows you select the `Command Prompt` from the Accessories menu. On Mac or Linux system you open a terminal or an XTerm. Inside the terminal you can change directories using the `cd` command, and run commands associated with Anaconda or miniconda using the `conda` command.

Creating the environment

We will create a single environment called `labs`. If you are running on a Mac or on Linux, open a terminal. If on Windows, use a command prompt. Then type

```
conda create -n labs python=3
```

This creates the new environment, and installs the basic `python` package in the `python 3.X` flavour. It does not activate the environment. In order to work within this environment, if using Windows type

```
activate labs
```

If using Mac or Linux type

```
source activate labs
```

Then any command launched from the terminal or command prompt will use the packages in this environment.

Packages

After creating the environment, and activating it, the key packages that need installing (if using miniconda; they are all installed with the full Anaconda) are:

- `ipython`

- numpy
- matplotlib
- scipy
- spyder
- spyder-app
- sympy

Other packages that will be useful are

- jupyter
- nose
- numba
- pandas

The command to install new packages is `conda install`. So, to install the packages above type (or copy and paste) first

```
activate labs
```

if on Windows, or

```
source activate labs
```

if on Mac or Linux, and then type

```
conda install ipython numpy matplotlib scipy spyder spyder-app sympy \
 jupyter nose numba pandas
```

Note: the '\' backslash character should continue an overly long line: if you are typing and not copying and pasting this should be unnecessary.

This will download and install a lot of additional packages that are needed; just agree and continue.

1.4 Spyder

There are many ways of writing and running Python code. If you open a terminal (on Mac or Linux; on Windows, this would be a Command Prompt) you can type `python` or `ipython` to launch a very bare bones *console*. This allows you to enter code which it will then run.

More helpful alternatives are *Integrated Development Environments* (IDEs) or the *notebook*. The Jupyter notebook (previously called the IPython notebook) is browser based and very powerful for exploratory computing. To run, type `jupyter notebook` in the terminal prompt.

However, when just getting started, or when writing large codes, IDEs are a better alternative. A simple IDE is `spyder` which we will use here. To launch, either select `spyder` from the appropriate menu, or type `spyder` at the terminal prompt.

You should then see a screen something like the figure (without the annotations).

The four essential parts of the screen are outlined.

1. The console (bottom right, marked in blue). You can work *interactively* here. Code run, either interactively or from the editor, will output any results here. Error messages will be reported here. There are two types of

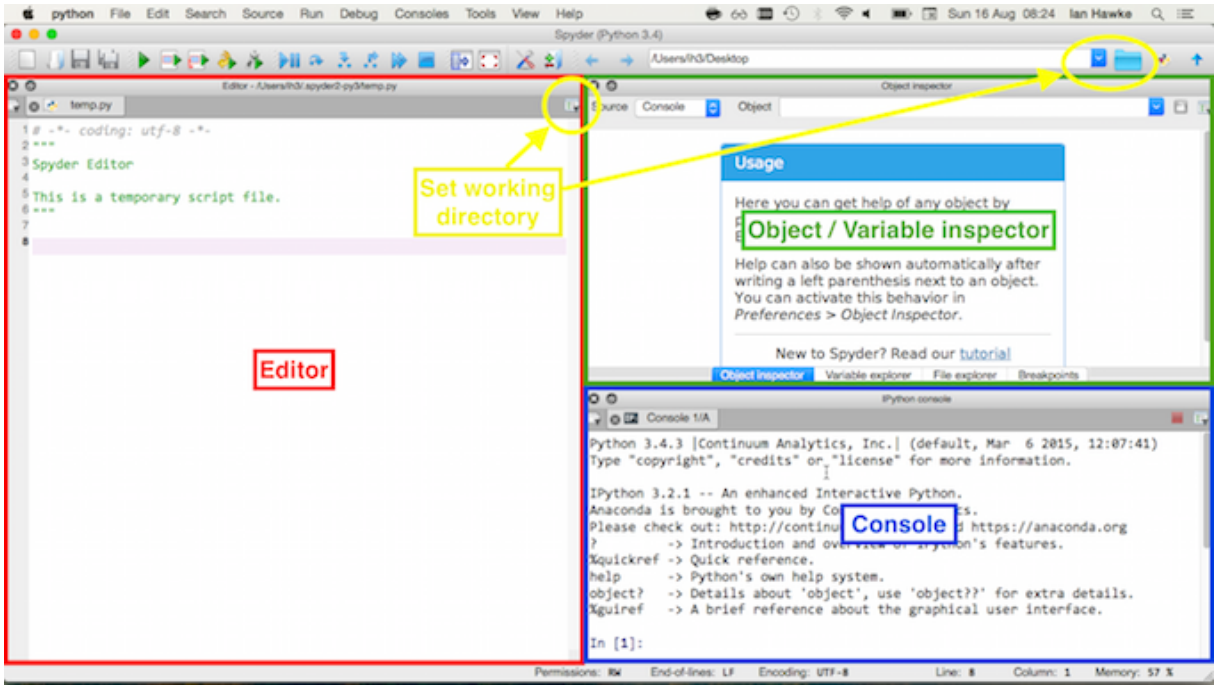


Fig. 1.1: The spyder editor, with the key areas outlined and coloured.

console: a Python console, and an IPython console. Both will run Python code, but the IPython console is easier to use.

2. The editor (left, marked in red). You can write code to be saved to file or run here. This will suggest problems with syntax and has features to help debug and give additional information.
3. The inspector (top right, marked in green). Can display detailed help on specific objects (or functions, or...) - the *Object inspector*, or can display detailed information on the variables that are currently defined - the *Variable inspector*. Extremely useful when debugging.
4. The working directory (marked in yellow). When running a code this is the first place that `spyder` looks. You should ensure that the working directory is set to the location where the file is.

For further information on using and setting up `spyder`, see this tutorial.

1.4.1 Tab completion

A crucial feature of IPython and spyder that saves time and reduces errors is *tab completion*. When typing anything, try pressing the tab key. This will either automatically complete the name of the variable (or function, or class), or will present a list of options. This is one way of finding out what functions are available - press tab and it will list them all! By typing the first few characters and then pressing tab, you can rapidly narrow down the options.

1.4.2 Help

There are many ways of getting help. The most useful are:

- Type `help(<thing>)`. Works in the console.
- Type `<thing>?` or `<thing>??`. Works in the console.
- Type the name in the Object Inspector. Works in spyder only.

- Google it. Pay particular attention to the online documentation and sites such as stackoverflow.

1.5 Reading list

There's a lot of material related to Python online and in the library. None are essential, but lots may be useful. The particular books recommended as a first look are

- Langtangen, *A Primer on Scientific Programming with Python*. Detailed, aimed more towards mathematicians than many others.
- Newman, *Computational Physics*. Really aimed at teaching numerical algorithms rather than programming, but there's lots of useful examples at a good level.
- Scopatz & Huff, *Effective Computation in Physics*. Covers a lot more material than just Python, not exactly aimed at mathematics, but essential background for computational research in the sciences.
- Saha, *Doing Math with Python*. Covers more symbolic mathematics and assumes more Python background, but has lots of excellent exercises at the right level.

1.6 Versions

These notes have been constructed using the following versions of Python and related packages:

```
In [1]: %load_ext watermark
        %watermark -v -m -g -p numpy,scipy,matplotlib,sympy

CPython 3.4.4
IPython 4.1.1

numpy 1.10.4
scipy 0.17.0
matplotlib 1.5.1
sympy 0.7.6.1

compiler   : GCC 4.2.1 (Apple Inc. build 5577)
system     : Darwin
release    : 14.5.0
machine    : x86_64
processor  : i386
CPU cores  : 4
interpreter: 64bit
Git hash   : c8886df0783a20e915cec64c793a4f8962b1c889
```

CHAPTER 2

Python Basics

2.1 Python

2.1.1 Notation

When writing computer commands that you can type, the font will change to `command`. For example, `x=2**3.4/5` is a command that can be typed in.

When we talk about a general command, or some piece of missing data that you should complete, we will use angle brackets as in `<command>` or `<variable>`. For example, `<location>` could mean `"Southampton"`.

When showing actual commands as typed into Python, they will start with `In [<number>]:`. This is the notation used by the IPython console. The `<number>` allows you to refer to previous commands more easily. The output associated with that command will start with `Out [<number>]:`.

When displaying code, certain commands will appear in different colours. The colours are not necessary. They highlight different types of command or variable. When using the spyder editor you may find the colours match up and are useful: if not, either ignore them or switch them off.

2.1.2 The console - Python as calculator

Start with using Python as a calculator. Look at the *console* in the bottom right part of spyder. Here we can type commands and see a result. Simple arithmetic gives the expected results:

```
In [1]: 2+2
```

```
Out[1]: 4
```

```
In [2]: (13.5*2.6-1.4)/10.2
```

```
Out[2]: 3.3039215686274517
```

If we want to raise a number to a power, say 2^4, the notation is `**`:

```
In [3]: 2**4
```

```
Out[3]: 16
```

There is an issue with division. If we divide an integer by an integer, Python `3.X` will do *real* (floating point) division, so that:

```
In [4]: 5/2
```

```
Out[4]: 2.5
```

However, Python 2.X will do division in the integers:

```
In []: 5/2
Out []: 2
```

If you are using Python 2.X and want the division operator to behave in this way, start by using the command

```
In [5]: from __future__ import division
```

Then:

```
In [6]: 5/2
```

```
Out[6]: 2.5
```

If you really want to do integer division, the command is //:

```
In [7]: 5//2
```

```
Out[7]: 2
```

Further mathematical commands, even things as simple as log or sin, are not included as part of basic Python:

```
In [8]: log(2.3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-83d4dd34e7bf> in <module>()
----> 1 log(2.3)

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

```
In [9]: sin(1.4)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-6dd12df070cd> in <module>()
----> 1 sin(1.4)

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

First, note the way that errors are reported: we'll see this a lot, and there's useful information there to understand. It's telling us

1. Where the problem occurred
2. What the problem is

The language Python uses takes some getting used to, but it's worth the effort to read these messages, and think about what it's trying to say. Here it's pointing to the line where the problem happened, and saying "I don't understand what this command is!".

Going back to the mathematics, we obviously want to be able to compute more mathematical functions. For this we need a *module* or *package*.

2.1.3 Importing modules and packages

Anything that isn't provided by the base Python can be provided by modules and packages. A module is a file containing functions and definitions that we can include and use in our code. A package is a collection of modules. They're not included by default, to reduce overhead. They're easy to write - we will write our own later - and easy to include.

To use a package we must *import* it. Let's look at the `math` package.

```
In [10]: import math

In [11]: math.log(2.3)

Out[11]: 0.8329091229351039

In [12]: math.sin(1.2)

Out[12]: 0.9320390859672263
```

To use the package we've typed `import <package>`, where in this case `<package>` is `math`. Then we can use functions from that package by typing `<package>.<function>`, as we have here when `<function>` is either `log` or `sin`.

The "dot" notation may seem annoying, and can be avoided by specifying what functions and constants we want to use. For example, we could just get the `log` function and use that:

```
In [13]: from math import log

In [14]: log(2.3)

Out[14]: 0.8329091229351039
```

However, the "dot" notation is useful, as we often find the same symbol or name being used for many different purposes. For example, the `math` package contains the mathematical constant e as `math.e`:

```
In [15]: math.e

Out[15]: 2.718281828459045
```

But there is also the electron charge, usually denoted e, which is in the `scipy.constants` package:

```
In [16]: import scipy.constants

In [17]: scipy.constants.e

Out[17]: 1.6021766208e-19
```

To avoid these name clashes we can import something *as* a different name:

```
In [18]: from math import e

In [19]: from scipy.constants import e as charge_e

In [20]: e

Out[20]: 2.718281828459045

In [21]: charge_e

Out[21]: 1.6021766208e-19
```

You will often see this method used to shorten the names of imported modules or functions. For example, standard examples often used are:

```
In [22]: import numpy as np

In [23]: import matplotlib.pyplot as plt
```

The commands can then be used by typing `np.<function>`, or `plt.<function>`, which saves typing. We would encourage you *not* to do this as it can make your code less clear.

2.1.4 Variables

A *variable* is an object with a name and a value:

```
In [24]: x = 2
```

In standard programming all variables must have a value (although that value may be a placeholder to say "this variable doesn't have a reasonable value *yet*"). Only symbolic packages can mirror the analytical method of having a variable with no specific value. However, code can be written *as if* the variables had no specific value.

For example, we cannot write

```
In [25]: x = y**2

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-9736a48890b4> in <module>()
----> 1 x = y**2

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

as `y` does not have a specific value yet. However, we can write

```
In [26]: y = 3.14159627

In [27]: x = y**2

In [28]: print(x)

9.869627123677912
```

and get a sensible result, even though we have written the exact same line of code, as now `y` has a specific value.

Warning

Note that we have defined the variable `x` twice in rapid succession: first as an *integer* (`x=2`) and next as a *floating point number*, or *float* (the computer's implementation of a real number, using `x=y**2`, where `y` is a float). Not all programming languages allow you to do this. In a *statically typed* language you have to say whether a variable will be an integer, or a float, or another type, before you define it, and then it cannot change. Python is *dynamically typed*, so any variable can have any type, which can be changed as we go.

2.1.5 Variable names

A variable is an object with a name, but not just any name will do. Python has rules which *must* be followed, and conventions that *should* be followed, with a few gray areas.

Variables *must*

- not contain spaces
- not start with a number
- not contain a special character (such as `!@#$%^&*()\|`)

So the following are valid:

```
In [29]: half = 1.0/2.0

In [30]: one_half = 1.0/2.0
```

but the following are not:

```
In [31]: one half = 1.0/2.0
  File "<ipython-input-31-3d1440172542>", line 1
    one half = 1.0/2.0
            ^
SyntaxError: invalid syntax

In [32]: 1_half = 1.0/2.0
  File "<ipython-input-32-7867c2b402d6>", line 1
    1_half = 1.0/2.0
         ^
SyntaxError: invalid syntax

In [33]: one!half = 1.0/2.0
  File "<ipython-input-33-d585fc045f28>", line 1
    one!half = 1.0/2.0
       ^
SyntaxError: invalid syntax
```

Variables *should*

- be descriptive, ie say what their purpose is in the code
- be written entirely in lower case
- separate different words in the variable name using underscores

More detail can be found in PEP8.

Variables *may* contain some unicode characters, depending on Python version and operating system. In Python `3` you can include accents or extended character sets in variable names:

```
rôle = 5
π = math.pi
```

However, these tricks are not always portable between different Python versions (they aren't guaranteed to work in Python `2`), or different operating systems, or even different machines. To ensure that your code works as widely as possible, and that the methods you use will carry over to other programming languages, it is recommended that variables do not use any extended characters, but only the basic latin characters, numbers, and underscores.

2.1.6 Equality and variable assignment

One thing that may seem odd, or just plain *wrong* to a mathematician, is two statements like

```
In [34]: x = 2

In [35]: x = 3
```

How can `x` equal *both* 2 and 3? Mathematically, this is nonsense.

The point is that, in nearly every programming language, the = symbol is not mathematical equality. It is the *assignment* operation: "set the value of the variable (on the left hand side) equal to the result of the operation (on the right hand side)". This implies another difference from the mathematical equality: we cannot flip the two sides and the line of code mean the same. For example,

```
In [36]: 3 = x
```

```
  File "<ipython-input-36-6f12c1f282a0>", line 1
    3 = x
      ^
SyntaxError: can't assign to literal
```

immediately fails as `3` is not a variable but a fixed quantity (a *literal*), which cannot be assigned to. Mathematically there is no difference between $x = 3$ and $3 = x$; in programming there is a huge difference between `x=3` and `3=x` as the meaning of = is not the mathematical meaning.

To get closer to the standard mathematical equality, Python has the == operator. This compares the left and right hand sides and says whether or not their values are equal:

```
In [37]: x == 3.0
Out[37]: True
```

However, this may not be exactly what we want. Note that we have assigned `x` to be the *integer* 3, but have compared its value to the *float* 3.0. If we want to check equality of value and type, Python has the `type` function:

```
In [38]: type(x)
Out[38]: int
In [39]: type(x) == type(3.0)
Out[39]: False
In [40]: type(x) == type(3)
Out[40]: True
```

Direct comparisons of equality are often avoided for floating point numbers, due to inherent numerical inaccuracies:

```
In [41]: p = 2.01
In [42]: p**2-4.0401
Out[42]: -8.881784197001252e-16
In [43]: p**2 == 4.0401
Out[43]: False
```

We will return to this later.

2.2 Debugging

Making mistakes and fixing them is an essential part of both mathematics and programming. When trying to fix problems in your code this process is called *debugging*. As you've been following along with the code you will have made "mistakes" as there are intentionally broken commands above to show, for example, why `one!half` is not a valid variable (and you may have made unintentional mistakes as well).

There are a number of techniques and strategies to make debugging less painful. There's more detail in later chapters, but for now let's look at the crucial first method.

2.2.1 Reading the error message

When you make a mistake Python tells you, and in some detail. When using IPython in particular, there is much information you can get from the error message, and you should read it in detail. Let's look at examples.

A *syntax error* is when Python cannot interpret the command you have entered.

```
In [44]: one half = 1.0/2.0

  File "<ipython-input-44-3d1440172542>", line 1
    one half = 1.0/2.0
             ^
SyntaxError: invalid syntax
```

This example we saw above. The `SyntaxError` is because of the space - `one half` is not a valid variable name. The error message says `invalid syntax` because Python can't interpret the command on the left of the equals sign. The use of the carat ^ points to the particular "variable" that Python can't understand.

```
In [45]: x = 1.0 / ( 2.0 + (3.0 * 4.5)

  File "<ipython-input-45-da9f857ed183>", line 1
    x = 1.0 / ( 2.0 + (3.0 * 4.5)
                                  ^
SyntaxError: unexpected EOF while parsing
```

This example is still a `SyntaxError`, but the pointer (^) is indicating the end of the line, and the statement is saying something new. In this statement `unexpected EOF while parsing`, "`EOF`" stands for "end of file". This can be reworded as "Python was reading your command and got to the end before it expected".

This usually means something is missing. In this case a bracket is missing - there are two left brackets but only one right bracket.

A similar example would be

```
In [46]: name = "This string should end here...

  File "<ipython-input-46-e75a715c38f0>", line 1
    name = "This string should end here...
                                          ^
SyntaxError: EOL while scanning string literal
```

In this case the error message includes "`EOL`", standing for "end of line". So we can reword this message as "Python was reading your command and got to the end of the line before the string finished". We fix this by adding a `"` at the end of the line.

Finally, what happens if the error is buried in many lines of code? This isn't the way we'd enter code in the console, but is how we'd deal with code in the notebook, or in a script or file:

```
In [47]: x = 1.0
         y = 2.3
         z = 4.5 * x y + y
         a = x + y + z
         b = 3.4 * a

  File "<ipython-input-47-8b0bd2354b36>", line 3
    z = 4.5 * x y + y
              ^
SyntaxError: invalid syntax
```

We see that the error message points to the *specific line* that it can't interpret, *and* it says (at first) which line this is.

This is where Python doesn't know the variable you're trying to refer to:

```
In [48]: my_variable = 6
         my_variable_2 = 10
         x = my_variable + my_variable_3

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-48-06bcebc035d2> in <module>()
      1 my_variable = 6
      2 my_variable_2 = 10
----> 3 x = my_variable + my_variable_3

NameError: name 'my_variable_3' is not defined
```

It usually means you've made a typo, or have referred to a variable before defining it (sometimes by cutting and pasting code around). Using tab completion is a good way of minimizing these errors.

Note that there is a difference between syntax errors and other errors. Syntax errors are spotted by the code before it is run; other errors require running the code. This means the format of the output is different. Rather than giving the line number and using a carat ("^") to point to the error, instead it points to the code line using "----->" and adds the line number before the line of code.

A range of "incorrect" mathematical operations will give errors. For example

```
In [49]: 1.0/0.0

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-49-fd44f356366a> in <module>()
----> 1 1.0/0.0

ZeroDivisionError: float division by zero
```

Obviously dividing by zero is a bad thing to try and do within the reals. Note that the definition of floating point numbers *does* include the concept of infinity (via "`inf`"), but this is painful to use and should be avoided where possible.

```
In [50]: math.log(0.0)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-50-7a32bd6f7959> in <module>()
----> 1 math.log(0.0)

ValueError: math domain error
```

A `ValueError` generally means that you're trying to call a function with a value that just doesn't make sense to it: in this case, $\log(x)$ just can't be evaluated when $x = 0$.

```
In [51]: 10.0**(10.0**(10.0**10.0))

---------------------------------------------------------------------------
OverflowError                             Traceback (most recent call last)
<ipython-input-51-c25c175740ef> in <module>()
----> 1 10.0**(10.0**(10.0**10.0))

OverflowError: (34, 'Result too large')
```

An `OverflowError` is when the number gets too large to be represented as a floating point number.

Having read the error message, find the line in your code that it indicates. If the error is clear, then fix it. If not, try splitting the line into multiple simpler commands and see where the problem lies. If using spyder, enter the commands

into the editor and see if the syntax highlighting helps spot errors.

2.3 Exercise: Variables and assignment

2.3.1 Exercise 1

Remember that $n! = n \times (n-1) \times \cdots \times 2 \times 1$. Compute $15!$, assigning the result to a sensible variable name.

2.3.2 Exercise 2

Using the `math` module, check your result for 15 factorial. You should explore the help for the `math` library and its functions, using eg tab-completion, the spyder inspector, or online sources.

2.3.3 Exercise 3

Stirling's approximation gives that, for large enough n,

$$n! \simeq S = \sqrt{2\pi} n^{n+1/2} e^{-n}.$$

Using functions and constants from the `math` library, compare the results of $n!$ and Stirling's approximation for $n = 5, 10, 15, 20$. In what sense does the approximation improve (investigate the absolute error $|n! - S|$ and the relative error $|n! - S|/n!$)?

CHAPTER 3

Programs

3.1 Programs

Using the Python console to type in commands works fine, but has serious drawbacks. It doesn't save the work for the future. It doesn't allow the work to be re-used. It's frustrating to edit when you make a mistake, or want to make a small change. Instead, we want to write a program.

A program is a text file containing Python commands. It can be written in any text editor. Something like the editor in spyder is ideal: it has additional features for helping you write code. However, any plain text editor will work. A program like Microsoft Word will *not* work, as it will try and save additional information with the file.

Let us use a simple pair of Python commands:

```
In [1]: import math
        x = math.sin(1.2)
```

Go to the editor in spyder and enter those commands in a file:

```
import math
x = math.sin(1.2)
```

Save this file in a suitable location and with a suitable name, such as `lab1_basic.py` (the rules and conventions for filenames are similar to those for variable names laid out above: descriptive, lower case names without spaces). The file extension should be `.py`: spyder should add this automatically.

To run this program, either

- press the green "play" button in the toolbar;
- press the function key `F5`;
- select "Run" from the "Run" menu.

In the console you should see a line like

```
runfile('/Users/ih3/PythonLabs/lab1_basic.py',wdir='/Users/ih3/PythonLabs')
```

appear, and nothing else. To check that the program has worked, check the value of `x`. In the console just type `x`:

```
In [2]: x
```

```
Out[2]: 0.9320390859672263
```

Also, in the top right of the spyder window, select the "Variable explorer" tab. It shows the variables that it currently knows, which should include `x`, its type (float) and its value.

If there are many variables known, you may worry that your earlier tests had already set the value for `x` and that the program did not actually do anything. To get back to a clean state, type `%reset` in the console to delete all variables - you will need to confirm that you want to do this. You can then re-run the program to test that it worked.

3.2 Using programs and modules

In previous sections we have imported and used standard Python libraries, packages or modules, such as `math`. This is one way of using a program, or code, that someone else has written. To do this for ourselves, we use exactly the same syntax.

Suppose we have the file `lab1_basic.py` exactly as above. Write a second file containing the lines

```
import lab1_basic
print(lab1_basic.x)
```

Save this file, in the **same directory** as `lab1_basic.py`, say as `lab1_import.py`. When we run this program, the console should show something like

```
runfile('/Users/ih3/PythonLabs/lab1_import.py', wdir='/Users/ih3/PythonLabs')
0.9320390859672263
```

This shows what the `import` statement is doing. All the library imports, definitions and operations in the imported program (`lab1_basic`) are performed. The results are then available to us, using the dot notation, via `lab1_basic.<variable>`, or `lab1_basic.<function>`.

To build up a program, we write Python commands into plain text files. When we want to use, or re-use, those definitions or results, we use `import` on the name of the file to recover their values.

3.2.1 Note

We saved both files - the original `lab1_basic.py`, and the program that imported `lab1_basic.py`, in the same directory. If they were in different directories then Python would not know where to find the file it was trying to import, and would give an error. The solution to this is to create a *package*, which is rather more work.

3.3 Functions

We have already seen and used some functions, such as the `log` and `sin` functions from the `math` package. However, in programming, a function is more general; it is any set of commands that acts on some *input parameters* and *returns some output*.

Functions are central to effective programming, as they stop you from having to repeat yourself and reduce the chances of making a mistake. Defining and using your own functions is the next step.

Let us write a function that converts angles from degrees to radians. The formula is

$$\theta_r = \frac{\pi}{180}\theta_d,$$

where θ_r is the angle in radians, and θ_d is the angle in degrees. If we wanted to do this for, eg, $\theta_d = 30°$, we could use the commands

```
In [3]: from math import pi
        theta_d = 30.0
        theta_r = pi / 180.0 * theta_d
```

```
In [4]: print(theta_r)
```

```
0.5235987755982988
```

This is effective for a single angle. If we want to repeat this for many angles, we could copy and paste the code. However, this is dangerous. We could make a mistake in editing the code. We could find a mistake in our original code, and then have to remember to modify *every* location where we copied it to. Instead we want to have a *single* piece of code that performs an action, and use that piece of code without modification whenever needed.

This is summarized in the "DRY" principle: do not repeat yourself. Instead, convert the code into a function and use the function.

We will define the function and show that it works, then discuss how:

```
In [5]: from math import pi

        def degrees_to_radians(theta_d):
            """
            Convert an angle from degrees to radians.

            Parameters
            ----------

            theta_d : float
                The angle in degrees.

            Returns
            -------

            theta_r : float
                The angle in radians.
            """
            theta_r = pi / 180.0 * theta_d
            return theta_r
```

We check that it works by printing the result for multiple angles:

```
In [6]: print(degrees_to_radians(30.0))
        print(degrees_to_radians(60.0))
        print(degrees_to_radians(90.0))
```

```
0.5235987755982988
1.0471975511965976
1.5707963267948966
```

How does the function definition work?

First we need to use the `def` command:

```
def degrees_to_radians(theta_d):
```

This command effectively says "what follows is a function". The first word after `def` will be the name of the function, which can be used to call it later. This follows similar rules and conventions to variables and files (no spaces, lower case, words separated by underscores, etc.).

After the function name, inside brackets, is the list of input parameters. If there are no input parameters the brackets still need to be there. If there is more than one parameter, they should be separated by commas.

After the bracket there is a colon `:`. The use of colons to denote special "blocks" of code happens frequently in Python code, and we will see it again later.

After the colon, all the code is indented by four spaces or one tab. Most helpful text editors, such as the spyder editor, will automatically indent the code after a function is defined. If not, use the tab key to ensure the indentation is correct. In Python, whitespace and indentation is essential: it defines where blocks of code (such as functions) start and end. In other languages special keywords or characters may be used, but in Python the indentation of the code statements is the key.

The statement on the next few lines is the *function documentation*, or *docstring*.

```
"""
Convert an angle from degrees to radians.

...
"""
```

This is *in principle* optional: it's not needed to make the code run. However, documentation is extremely useful for the next user of the code. As the next user is likely to be you in a week (or a month), when you'll have forgotten the details of what you did, documentation helps you first. In reality, you should **always** include documentation.

The docstring can be any string within quotes. Using "triple quotes" allows the string to go across multiple lines. The docstring can be rapidly printed using the `help` function:

```
In [7]: help(degrees_to_radians)
```

```
Help on function degrees_to_radians in module __main__:

degrees_to_radians(theta_d)
    Convert an angle from degrees to radians.

    Parameters
    ----------

    theta_d : float
        The angle in degrees.

    Returns
    -------

    theta_r : float
        The angle in radians.
```

This allows you to quickly use code correctly without having to look at the code. We can do the same with functions from packages, such as

```
In [8]: help(math.sin)
```

```
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

You can put whatever you like in the docstring. The format used above in the `degrees_to_radians` function follows the numpydoc convention, but there are other conventions that work well. One reason for following this convention can be seen in spyder. Copy the function `degrees_to_radians` into the console, if you have not done so already. Then, in the top right part of the window, select the "Object inspector" tab. Ensure that the "Source"

is "Console". Type `degrees_to_radians` into the "Object" box. You should see the help above displayed, but nicely formatted.

You can put additional comments in your code - anything after a "#" character is a comment. The advantage of the docstring is how it can be easily displayed and built upon by other programs and bits of code, and the conventions that make them easier to write and understand.

Going back to the function itself. After the comment, the code to convert from degrees to radians starts. Compare it to the original code typed directly into the console. In the console we had

```
from math import pi
theta_d = 30.0
theta_r = pi / 180.0 * theta_d
```

In the function we have

```
theta_r = pi / 180.0 * theta_d
return theta_r
```

The line

```
from math import pi
```

is in the function file, but outside the definition of the function itself.

There are four differences.

1. The function code is indented by four spaces, or one tab.
2. The *input parameter* `theta_d` must be defined in the console, but not in the function. When the function is called the value of `theta_d` is given, but inside the function itself it is not: the function knows that the *specific* value of `theta_d` will be given as input.
3. The output of the function `theta_r` is explicitly returned, using the `return` statement.
4. The `import` statement is moved outside the function definition - this is the convention recommended by PEP8.

Aside from these points, the code is identical. A function, like a program, is a collection of Python statements exactly as you would type into a console. The first three differences above are the essential differences to keep in mind: the first is specific to Python (other programming languages have something similar), whilst the other differences are common to most programming languages.

Names used internally by the function are not visible externally. Also, the name used for the output of the function need not be used externally. To see an example of this, start with a clean slate by typing `%reset` into the console.

```
In [9]: %reset -f
```

Then copy and paste the function definition again:

```
In [10]: from math import pi

         def degrees_to_radians(theta_d):
             """
             Convert an angle from degrees to radians.

             Parameters
             ----------

             theta_d : float
                 The angle in degrees.
```

```
    Returns
    -------

    theta_r : float
        The angle in radians.
    """
    theta_r = pi / 180.0 * theta_d
    return theta_r
```

(Alternatively you can use the history in the console by pressing the up arrow until the definition of the function you previously entered appears. Then click at the end of the function and press Return). Now call the function as

```
In [11]: angle = degrees_to_radians(45.0)

In [12]: print(angle)

0.7853981633974483
```

But the variables used internally, `theta_d` and `theta_r`, are not known outside the function:

```
In [13]: theta_d
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-13-a78047e2dfdf> in <module>()
----> 1 theta_d

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

This is an example of *scope*: the existence of variables, and their values, is restricted inside functions (and files).

You may note that above, we had a value of `theta_d` outside the function (from when we were working in the console), and a value of `theta_d` inside the function (as the input parameter). These do not have to match. If a variable is assigned a value inside the function then Python will take this "local" value. If not, Python will look outside the function. Two examples will illustrate this:

```
In [14]: x1 = 1.1

         def print_x1():
             print(x1)

         print(x1)
         print_x1()

1.1
1.1

In [15]: x2 = 1.2

         def print_x2():
             x2 = 2.3
             print(x2)

         print(x2)
         print_x2()

1.2
2.3
```

In the first (`x1`) example, the variable `x1` was not defined within the function, but it was used. When `x1` is printed, Python has to look for the definition outside of the scope of the function, which it does successfully.

In the second (`x2`) example, the variable `x2` *is* defined within the function. The value of `x2` does *not* match the value of the variable with the same name defined outside the function, but that does not matter: within the function, its *local* value is used. When printed outside the function, the value of `x2` uses the external definition, as the value defined inside the function is not known (it is "not in scope").

Some care is needed with using scope in this way, as Python reads the whole function at the time it is *defined* when deciding scope. As an example:

```
In [16]: x3 = 1.3

         def print_x3():
             print(x3)
             x3 = 2.4

         print(x3)
         print_x3()
```

```
1.3
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-16-ded2d55c6779> in <module>()
      6
      7 print(x3)
----> 8 print_x3()

<ipython-input-16-ded2d55c6779> in print_x3()
      2
      3 def print_x3():
----> 4     print(x3)
      5     x3 = 2.4
      6

UnboundLocalError: local variable 'x3' referenced before assignment
```

The only significant change from the second example is the order of the `print` statement and the assignment to `x3` inside the function. Because `x3` is assigned inside the function, Python wants to use the *local* value within the function, and will ignore the value defined outside the function. However, the `print` function is called before `x3` has been set within the function, leading to an error.

Our original function `degrees_to_radians` only had one argument, the angle to be converted `theta_d`. Many functions will take more than one argument, and sometimes the function will take arguments that we don't always want to set. Python can make life easier in these cases.

Suppose we wanted to know how long it takes an object released from a height h, in a gravitational field of strength g, with initial vertical speed v, to hit the ground. The answer is

$$t = \frac{1}{g}\left(v + \sqrt{v^2 + 2hg}\right).$$

We can write this as a function:

```
In [17]: from math import sqrt

         def drop_time(height, speed, gravity):
             """
             Return how long it takes an object released from a height h,
             in a gravitational field of strength g, with initial vertical speed v,
             to hit the ground.
```

```
    Parameters
    ----------

    height : float
        Initial height h
    speed : float
        Initial vertical speed v
    gravity : float
        Gravitional field strength g

    Returns
    -------

    t : float
        Time the object hits the ground
    """

    return (speed + sqrt(speed**2 + 2.0*height*gravity)) / gravity
```

But when we start using it, it can be a bit confusing:

```
In [18]: print(drop_time(10.0, 0.0, 9.8))
         print(drop_time(10.0, 1.0, 9.8))
         print(drop_time(100.0, 9.8, 15.0))
```

```
1.4285714285714284
1.5342519232263467
4.362804694292706
```

Is that last case correct? Did we *really* want to change the gravitational field, whilst at the same time using an initial velocity of exactly the value we expect for g?

A far clearer use of the function comes from using *keyword* arguments. This is where we *explicitly* use the name of the function arguments. For example:

```
In [19]: print(drop_time(height=10.0, speed=0.0, gravity=9.8))
```

```
1.4285714285714284
```

The result is exactly the same, but now it's explicitly clear what we're doing.

Even more useful: when using keyword arguments, we don't have to ensure that the order we use matches the order of the function definition:

```
In [20]: print(drop_time(height=100.0, gravity=9.8, speed=15.0))
```

```
6.300406486742956
```

This is the same as the confusing case above, but now there is no ambiguity. Whilst it is good practice to match the order of the arguments to the function definition, it is only *needed* when you don't use the keywords. Using the keywords is *always* useful.

What if we said that we were going to assume that the gravitational field strength g is nearly always going to be that of Earth, 9.8ms^{-2}? We can re-define our function using a *default* argument:

```
In [21]: def drop_time(height, speed, gravity=9.8):
             """
             Return how long it takes an object released from a height h,
             in a gravitational field of strength g, with initial vertical speed v,
             to hit the ground.
```

```
    Parameters
    ----------

    height : float
        Initial height h
    speed : float
        Initial vertical speed v
    gravity : float
        Gravitional field strength g

    Returns
    -------

    t : float
        Time the object hits the ground
    """

    return (speed + sqrt(speed**2 + 2.0*height*gravity)) / gravity
```

Note that there is *only one* difference here, in the very first line: we state that `gravity=9.8`. What this means is that if this function is called and the value of `gravity` is *not* specified, then it takes the value `9.8`.

For example:

```
In [22]: print(drop_time(10.0, 0.0))
         print(drop_time(height=50.0, speed=1.0))
         print(drop_time(gravity=15.0, height=50.0, speed=1.0))
```

```
1.4285714285714284
3.2980530129318018
2.649516083708069
```

So, we can still give a specific value for `gravity` when we don't want to use the value `9.8`, but it isn't needed if we're happy for it to take the default value of `9.8`. This works both if we use keyword arguments and if not, with certain restrictions.

Some things to keep in mind.

- Default arguments can only be used without specifying the keyword if they come after arguments without defaults. It is a **very strong** convention that arguments with a default come at the end of the argument list.
- The value of default arguments can be pretty much anything, but care should be taken to get the behaviour you expect. In particular, it is **strongly discouraged** to allow the default value to be anything that might change, as this can lead to odd behaviour that is hard to find. In particular, allowing a default value to be a container such as a list (seen below) can lead to unexpected behaviour. See, for example, this discussion, pointing out why, and that the value of the default argument is fixed when the function is defined, not when it's called.

3.4 Printing and strings

We have already seen the `print` function used multiple times. It displays its argument(s) to the screen when called, either from the console or from within a program. It prints some representation of what it is given in the form of a *string*: it converts simple numbers and other objects to strings that can be shown on the screen. For example:

```
In [23]: import math
         x = 1.2
         name = "Alice"
```

```
         print("Hello")
         print(6)
         print(name)
         print(x)
         print(math.pi)
         print(math.sin(x))
         print(math.sin)
         print(math)
```

```
Hello
6
Alice
1.2
3.141592653589793
0.9320390859672263
<built-in function sin>
<module 'math' from '/Users/ih3/anaconda/envs/labs/lib/python3.5/lib-dynload/math.so'>
```

We see that variables are converted to their values (such as `name` and `math.pi`) and functions are called to get values (such as `math.sin(x)`), which are then converted to strings displayed on screen. However, functions (`math.sin`) and modules (`math`) are also "printed", in that a string saying what they are, and where they come from, is displayed.

Often we want to display useful information to the screen, which means building a message that is readable and `print`ing that. There are many ways of doing this: here we will just look at the `format` command. Here is an example:

```
In [24]: print("Hello {}. We set x={}.".format(name, x))
```

```
Hello Alice. We set x=1.2.
```

The `format` command takes the string (here `"Hello {}. We set x={}."`) and replaces the `{}` with the values of the variables (here `name` and `x` in order).

We can use the `format` command in this way for anything that has a string representation. For example:

```
In [25]: print ("The function {} applied to x={} gives {}".format(math.sin, x, math.sin(x)))
```

```
The function <built-in function sin> applied to x=1.2 gives 0.9320390859672263
```

There are many more ways to use the `format` command which can be helpful.

We note that `format` is a function, but a function applied to the string before the dot. This type of function is called a *method*, and we shall return to them later.

We have just printed a lot of strings out, but it is useful to briefly talk about what a string *is*.

In Python a string is not just a sequence of characters. It is a Python object that contains additional information that "lives on it". If this information is a constant property it is called an *attribute*. If it is a function it is called a *method*. We can access this information (using the "dot" notation as above) to tell us things about the string, and to manipulate it.

Here are some basic string methods:

```
In [26]: name = "Alice"
         number = "13"
         sentence = "   a  b  c d     e  "
         print(name.upper())
         print(name.lower())
         print(name.isdigit())
         print(number.isdigit())
```

```
        print(sentence.strip())
        print(sentence.split())
```

```
ALICE
alice
False
True
a  b  c d     e
['a', 'b', 'c', 'd', 'e']
```

The use of the "dot" notation appears here. We saw this with accessing functions in modules and packages above; now we see it with accessing attributes and methods. It appears repeatedly in Python. The `format` method used above is particularly important for our purposes, but there are a lot of methods available.

There are other ways of manipulating strings.

We can join two strings using the + operator.

```
In [27]: print("Hello" + "Alice")
```

```
HelloAlice
```

We can repeat strings using the * operator.

```
In [28]: print("Hello" * 3)
```

```
HelloHelloHello
```

We can convert numbers to strings using the `str` function.

```
In [29]: print(str(3.4))
```

```
3.4
```

We can also access individual characters (starting from 0!), or a range of characters:

```
In [30]: print("Hello"[0])
         print("Hello"[2])
         print("Hello"[1:3])
```

```
H
l
el
```

We will come back to this notation when discussing lists and slicing.

3.4.1 Note

There are big differences between how Python deals with strings in Python `2.X` and Python `3.X`. Whilst most of the commands above will produce identical output, string handling is one of the major reasons why Python `2.X` doesn't always work in Python `3.X`. The ways strings are handled in Python `3.X` is much better than in `2.X`.

3.5 Putting it together

We can now combine the introduction of programs with functions. First, create a file called `lab1_function.py` containing the code

```
from math import pi

def degrees_to_radians(theta_d):
    """
    Convert an angle from degrees to radians.

    Parameters
    ----------

    theta_d : float
        The angle in degrees.

    Returns
    -------

    theta_r : float
        The angle in radians.
    """
    theta_r = pi / 180.0 * theta_d
    return theta_r
```

This is almost exactly the function as defined above.

Next, write a second file `lab1_use_function.py` containing

```
from lab1_function import degrees_to_radians

print(degrees_to_radians(15.0))
print(degrees_to_radians(30.0))
print(degrees_to_radians(45.0))
print(degrees_to_radians(60.0))
print(degrees_to_radians(75.0))
print(degrees_to_radians(90.0))
```

This function uses our own function to convert from degrees to radians. To save typing we have used the `from <module> import <function>` notation. We could have instead written `import lab1_function`, but then every function call would need to use `lab1_function.degrees_to_radians`.

This program, when run, will print to the screen the angles $(n\pi)/12$ for $n = 1, 2, \ldots, 6$.

3.6 Exercise: basic functions

Write a function to calculate the volume of a cuboid with edge lengths a, b, c. Test your code on sample values such as

1. $a = 1, b = 1, c = 1$ (result should be 1);
2. $a = 1, b = 2, c = 3.5$ (result should be 7.0);
3. $a = 0, b = 1, c = 1$ (result should be 0);
4. $a = 2, b = -1, c = 1$ (what do you think the result should be?).

Write a function to compute the time (in seconds) taken for an object to fall from a height H (in metres) to the ground, using the formula

$$h(t) = \frac{1}{2}gt^2.$$

Use the value of the acceleration due to gravity g from `scipy.constants.g`. Test your code on sample values such as

1. $H = 1$m (result should be ≈ 0.452s);
2. $H = 10$m (result should be ≈ 1.428s);
3. $H = 0$m (result should be 0s);
4. $H = -1$m (what do you think the result should be?).

Write a function that computes the area of a triangle with edge lengths a, b, c. You may use the formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \qquad s = \frac{a+b+c}{2}.$$

Construct your own test cases to cover a range of possibilities.

3.7 Exercise: Floating point numbers

Computers cannot, in principle, represent real numbers perfectly. This can lead to problems of accuracy. For example, if

$$x = 1, \qquad y = 1 + 10^{-14}\sqrt{3}$$

then it *should* be true that

$$10^{14}(y - x) = \sqrt{3}.$$

Check how accurately this equation holds in Python and see what this implies about the accuracy of subtracting two numbers that are close together.

3.7.1 Note

The standard floating point number holds the first 16 significant digits of a real.

Exercise 2

3.7.2 Note: no coding required

The standard quadratic formula gives the solutions to

$$ax^2 + bx + c = 0$$

as

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

Show that, if $a = 10^{-n} = c$ and $b = 10^n$ then

$$x = \frac{10^{2n}}{2}\left(-1 \pm \sqrt{1 - 4 \times 10^{-4n}}\right).$$

Using the expansion (from Taylor's theorem)

$$\sqrt{1 - 4 \times 10^{-4n}} \simeq 1 - 2 \times 10^{-4n} + \ldots, \qquad n \gg 1,$$

show that

$$x \simeq -10^{2n} + 10^{-2n} \quad \text{and} \quad -10^{-2n}, \qquad n \gg 1.$$

Exercise 3

3.7.3 Note: no coding required

By multiplying and dividing by $-b \mp \sqrt{b^2 - 4ac}$, check that we can also write the solutions to the quadratic equation as

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}.$$

Exercise 4

Using Python, calculate both solutions to the quadratic equation

$$10^{-n}x^2 + 10^n x + 10^{-n} = 0$$

for $n = 3$ and $n = 4$ using both formulas. What do you see? How has floating point accuracy caused problems here?

Exercise 5

The standard definition of the derivative of a function is

$$\left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x=X} = \lim_{\delta \to 0} \frac{f(X+\delta) - f(X)}{\delta}.$$

We can *approximate* this by computing the result for a *finite* value of δ:

$$g(x, \delta) = \frac{f(x+\delta) - f(x)}{\delta}.$$

Write a function that takes as inputs a function of one variable, $f(x)$, a location X, and a step length δ, and returns the approximation to the derivative given by g.

Exercise 6

The function $f_1(x) = e^x$ has derivative with the exact value 1 at $x = 0$. Compute the approximate derivative using your function above, for $\delta = 10^{-2n}$ with $n = 1, \ldots, 7$. You should see the results initially improve, then get worse. Why is this?

CHAPTER 4

Loops - how to repeat yourself

4.1 Loops

4.1.1 while loops

The program we wrote at the end of the last part performs a series of repetitive operations, but did it by copying and pasting the call to our function and editing the input. As noted above, this is a likely source of errors. Instead we want to write a formula, algorithm or abstraction of our repeated operation, and reproduce that in code.

First we reproduce that function:

```
In [1]: from math import pi

        def degrees_to_radians(theta_d):
            """
            Convert an angle from degrees to radians.

            Parameters
            ----------

            theta_d : float
                The angle in degrees.

            Returns
            -------

            theta_r : float
                The angle in radians.
            """
            theta_r = pi / 180.0 * theta_d
            return theta_r
```

We showed above how to use this code to print the angles $(n\pi)/12$ for $n = 1, 2, \ldots, 6$. We did this by calling the `degrees_to_radians` function on the angles $15n$ degrees for $n = 1, 2, \ldots, 6$. So this is the formula we want to reproduce in code. To do that we write a *loop*.

Here is a standard way to do it:

```
In [2]: theta_d = 0.0
        while theta_d <= 90.0:
            print(degrees_to_radians(theta_d))
            theta_d = theta_d + 15.0
```

```
0.0
0.2617993877991494
0.5235987755982988
0.7853981633974483
1.0471975511965976
1.3089969389957472
1.5707963267948966
```

Let's examine this line by line. The first line defines the angle in degrees, `theta_d`. We start from $\theta_d = 0$.

The next line defines the loop. This has similarities to our definition of a function. We use the keyword `while` to say that what follows is going to be a loop. We then define a *logical condition* that will be either `True` or `False`. Whilst the condition is `True`, the statements in the loop will be executed. The colon `:` at the end of the line ends the logical condition and says that what follows will be the statements inside the loop. As with the function, the code block with the statements inside the loop is indented by four spaces or one tab.

The code block contains two lines, both of which will be executed. The first prints the converted angle to the screen. The second increases the angle in degrees by 15. At the end of the code block, Python will check the logical condition `theta_d <= 90.0` again. If it is `True` the statements inside the loop are executed again. If it is `False`, the code moves on to the next line after the loop.

There is another way to write a loop that we can use:

4.1.2 for loops

```
In [3]: steps = 1, 2, 3, 4, 5, 6
        for n in steps:
            print(degrees_to_radians(15*n))
```

```
0.2617993877991494
0.5235987755982988
0.7853981633974483
1.0471975511965976
1.3089969389957472
1.5707963267948966
```

Let's examine this code line by line. It first defines a set of numbers, `steps`, which contains the integers from 1 to 6 (we will make this more precise later when we discuss lists and tuples). We then define the loop using the `for` command. This looks at the set of numbers `steps` and picks an entry out one at a time, setting the variable `n` to be the value of that member of the set. So, the first time through the loop `n=1`. The next, `n=2`. Once it has iterated through all members of the set `steps`, it stops.

The colon `:` at the end of the line defines the code block that each iteration of the loop should perform. Exactly as when we defined a function, the code block is indented by four spaces or one tab. In each iteration through the loop, the commands indented by this amount will be run. In this case, only one line (the `print...` line) will be run. On each iteration the value of `n` changes, leading to the different angle.

Writing out a long list of integers is a bad idea. A better approach is the use the `range` function. This compresses the code to:

```
In [4]: for n in range(1,7):
            print(degrees_to_radians(15*n))
```

```
0.2617993877991494
0.5235987755982988
0.7853981633974483
1.0471975511965976
```

```
1.3089969389957472
1.5707963267948966
```

The `range` function takes the input arguments `<start>` and `<end>`, and the optional input argument `<step>`, to produce the integers from the start up to, but not including, the end. If the `<start>` is not given it defaults to `0`, and if the `<step>` is not given it defaults to `1`.

(Strictly, `range` does not return the full list in one go. It generates the results one at a time. This is much faster and more efficient. In the `for` loop this is all you need. To actually view what `range` generates all together, convert it to a `list`, as `list(range(...))`.

Check this against examples such as:

```
In [5]: print(list(range(4)))
        print(list(range(-1,3)))
        print(list(range(1,10,2)))
```

```
[0, 1, 2, 3]
[-1, 0, 1, 2]
[1, 3, 5, 7, 9]
```

In some programming languages this is where the discussion of a `for` loop would end: the "loop counter" must be an integer. In Python, a loop is just iterating over a set of values, and these can be much more general. An alternative way (using floats) to do the same loop would be

```
In [6]: angles = 15.0, 30.0, 45.0, 60.0, 75.0, 90.0
        for angle in angles:
            print(degrees_to_radians(angle))
```

```
0.2617993877991494
0.5235987755982988
0.7853981633974483
1.0471975511965976
1.3089969389957472
1.5707963267948966
```

But we can get much more general than that. The different things in the set don't have to have the same type:

```
In [7]: things = 1, 2.3, True, degrees_to_radians
        for thing in things:
            print(thing)
```

```
1
2.3
True
<function degrees_to_radians at 0x104b4fbf8>
```

This can be used to write very efficient code, but is a feature that isn't always available in other programming languages.

When should we use `for` loops and when `while` loops? In most cases either will work. Different algorithms have different conventions, so where possible follow the convention. The advantage of the `for` loop is that it is clearer how much work will be done by the loop (as, in principle, we know how many times the loop block will be executed). However, sometimes you need to perform a repetitive operation but don't know in advance how often you'll need to do it. For example, to the nearest degree, what is the largest angle θ_d that when converted to radians is less than 4? (For the picky, we're restricting θ_d so that $0° \leq \theta_d < 360°$).

Rather than doing the sensible thing and doing the analytic calculation, we can do the following:

1. Set $\theta_d = 0°$.
2. Calculate the angle in radians θ_r.

3. If $\theta_r < 4$:
    - Increase θ_d by $1°$;
    - Repeat from step 2.

We can reproduce this algorithm using a `while` loop:

```
In [8]: theta_d = 0.0

        while degrees_to_radians(theta_d) < 4.0:
            theta_d = theta_d + 1.0
```

This could be done in a `for` loop, but not so straightforwardly.

To summarize:

The structure of the `while` loop is similar to the `for` loop. The loop is defined by a keyword (`while` or `for`) and the end of the line defining the loop condition is given by a colon. With each iteration of the loop the indented code is executed. The difference is in how the code decides when to stop looping, and what changes with each iteration.

In the `for` loop the code iterates over the objects in a set, and some variable is modified with each iteration based on the new object. Once all objects in the set have been iterated over, the loop stops.

In a `while` loop some condition is checked; while it is true the loop continues, and as soon as it is false the loop stops. Here we are checking if θ_r, given by `degrees_to_radians(theta_d)`, is still less than 4. However, nothing in the definition of the loop actually changes: it is the statement *within* the loop that actually changes the angle θ_d.

We quickly check that the answer given makes sense:

```
In [9]: print(theta_d - 1.0)
        print(theta_d)
        print(degrees_to_radians(theta_d-1.0) / 4.0)
        print(degrees_to_radians(theta_d) / 4.0)
```

```
229.0
230.0
0.9992009967667537
1.0035643198967394
```

We see that the answer is $229°$.

4.1.3 Logical statements

We noted above that whether or not a `while` loop executes depends on the truth (or not) of a particular statement. In programming these logical statements take *Boolean values* (either true or false). In Python, the values of a Boolean statement are either `True` or `False`, which are the keywords used to refer to them. Multiple statements can be chained together using the logical operators `and`, `or`, `not`. For example:

```
In [10]: print(True)
         print(6 < 7 and 10 > 9)
         print(1 < 2 or 1 < 0)
         print(not (6 < 7) and 10 > 9)
         print(6 < 7 < 8)
```

```
True
True
True
False
True
```

The last example is particularly important, as this *chained* example (`6 < 7 < 8`) is equivalent to `(6 < 7) and (7 < 8)`. This checks *both* inequalities - checking that $6 < x < 8$ when $x = 7$, by checking that *both* $6 < x$ and $x < 8$ is true when $x = 7$, which is true (and mathematically what you would expect). *However*, many programming languages would not interpret it this way, but would instead interpret it as `(6 < 7) < 8`, which is equivalent to `True < 8`, which is nonsense. Chaining operations in this way is useful in Python, but don't expect it to always work in other languages.

4.2 Containers and Sequences

When talking about loops we informally introduced the collection of objects `1,2,3,4,5,6`, and assigned it to a single variable `steps`. This is one of many types of *container*: a single object that contains other objects. If the objects in the container have an *order* then the container is often called a *sequence*: an object that contains an ordered sequence of other objects. These sorts of objects are everywhere in mathematics: sets, groups, vectors, matrices, equivalence classes, categories, ... Programming languages also implement a large number of them. Python has four essential containers, the most important of which for our purposes are *lists*, *tuples*, and *dictionaries*.

4.2.1 Lists

A list is an ordered collection of objects. For example:

```
In [11]: list1 = [1, 2, 3, 4, 5, 6]
         list2 = [15.0, 30.0, 45.0, 60.0, 75.0, 90.0]
         list3 = [1, 2.3, True, degrees_to_radians]
         list4 = ["hello", list1, False]
         list5 = []
```

Lists are defined by square brackets, `[]`. Objects in the list are separated by commas. A list can be empty (`list5` above). A list can contain other lists (`list4` above). The objects in the list don't have to have the same type (`list3` and `list4` above).

We can access a member of a list by giving its name, square brackets, and the index of the member (starting from 0!):

```
In [12]: list1[0]
```

```
Out[12]: 1
```

```
In [13]: list2[3]
```

```
Out[13]: 60.0
```

Note

There is a big divide between programming languages that index containers (or vectors, or lists) starting from 0 and those that index starting from 1. There is no consensus on which is better, so as you move between languages, get used to checking which is used.

Entries in a list can be modified:

```
In [14]: list4[1] = "goodbye"
         list4
```

```
Out[14]: ['hello', 'goodbye', False]
```

Additional entries can be appended onto the end of a list:

```
In [15]: list4.append('end')
         list4
```

```
Out[15]: ['hello', 'goodbye', False, 'end']
```

Entries can be removed (popped) from the end of a list:

```
In [16]: entry = list4.pop()
         print(entry)
         list4
```

```
end
```

```
Out[16]: ['hello', 'goodbye', False]
```

The length of a list can be found:

```
In [17]: len(list4)
```

```
Out[17]: 3
```

Lists are probably the most used container, but there's a closely related container that we've already used: the tuple.

4.2.2 Tuples

Tuples are ordered collections of objects that, once created, cannot be modified. For example:

```
In [18]: tuple1 = 1, 2, 3, 4, 5, 6
         tuple2 = (15.0, 30.0, 45.0, 60.0, 75.0, 90.0)
         tuple3 = (1, 2.3, True, degrees_to_radians)
         tuple4 = ("hello", list1, False)
         tuple5 = ()
         tuple6 = (5,)
```

Tuples are defined by the commas separating the entries. The round brackets `()` surrounding the entries are conventional, useful for clarity, and for grouping. If you want to create an empty tuple (`tuple5`) the round brackets are necessary. A tuple containing a single entry (`tuple6`) must have a trailing comma.

Tuples can be accessed in the same ways as lists, and their length found with `len` in the same way. But they cannot be modified, so we cannot add additional entries, or remove them, or alter any:

```
In [19]: tuple1[0]
```

```
Out[19]: 1
```

```
In [20]: tuple4[1] = "goodbye"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-20-954f9f41259d> in <module>()
----> 1 tuple4[1] = "goodbye"

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

However, if a member of a tuple can itself be modified (for example, it's a list, as `tuple4[1]` is), then *that entry* can be modified:

```
In [21]: print(tuple4[1])
         tuple4[1][1] = 33
         print(tuple4[1])
```

```
[1, 2, 3, 4, 5, 6]
[1, 33, 3, 4, 5, 6]
```

Tuples appear a lot when using functions, either when passing in parameters, or when returning results. They can often be treated like lists, and there are functions that convert lists to tuples and vice versa:

```
In [22]: converted_list1 = list(tuple1)
         converted_tuple1 = tuple(list1)
```

4.2.3 Slicing

Accessing and manipulating multiple entries of a list at once is an efficient and effective way of coding: it shows up a lot in, for example, linear algebra. This is where *slicing* comes in.

We have seen that we can access a single element of a list using square brackets and an index. We can use similar notation to access multiple elements:

```
In [23]: list1 = [1, 2, 3, 4, 5, 6]
         print(list1[0])
         print(list1[1:3])
         print(list1[2:])
         print(list1[:4])
```

```
1
[2, 3]
[3, 4, 5, 6]
[1, 2, 3, 4]
```

The slicing notation `[<start>:<end>]` returns all entries from the `<start>` to the entry *before* the `<end>`. So:

- `list1[1:3]` returns all entries from the *second* (index 1) to the third (the entry before index 3).
- if `<end>` is not given all entries up to the end are returned, so `list1[2:]` returns all entries from the third (index 2) to the end.
- if `<start>` is not given all entries from the start are returned, so `list1[:4]` returns all entries from the start until the fourth (the entry before index 4).

There are a number of other ways that slicing can be used. First, we can specify the step:

```
In [24]: print(list1[0:6:2])
         print(list1[1::3])
         print(list1[4:1:-1])
```

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

By using a negative step we can reverse the order (as shown in the final example), but then we need to be careful with the `<start>` and `<end>`.

This `<start>:<end>:<step>` notation varies between programming languages: some use `<start>:<step>:<end>`.

Second, we can give an index that counts from the end, where the final entry is `-1`:

```
In [25]: print(list1[-1])
         print(list1[-2])
         print(list1[2:-2])
         print(list1[-4:-2])
```

```
6
5
[3, 4]
[3, 4]
```

4.2.4 Unpacking

Slicing is often seen as part of assignment. For example

```
In [26]: list_slice = [0, 0, 0, 0, 0, 0, 0, 0]
         list_slice[1:4] = list1[3:]
         print(list_slice)
```

```
[0, 4, 5, 6, 0, 0, 0, 0]
```

This is related to a very useful Python feature: unpacking. Normally we have assigned a single variable to a single value (although that value might be a container such as a list). However, we can assign multiple values in one go:

```
In [27]: a, b, c = list1[3:]
         print(a)
         print(b)
         print(c)
```

```
4
5
6
```

This can be used to directly swap two variables, for example:

```
In [28]: a, b = b, a
         print(a)
         print(b)
```

```
5
4
```

The number of entries on both sides must match.

4.2.5 Dictionaries

All the containers we have seen so far have had an order - lists and tuples are sequences, and we access the objects within them using `list[0]` or `tuple[3]`, for example. A *dictionary* is our first *unordered* container. These are useful for collections of objects with meaningful names, but where the order of the objects has no importance.

Dictionaries are defined using curly braces. The "name" of each entry is given first (usually called its *key*), followed by a `:`, and then after the colon comes its *value*. Multiple entries are separated by commas. For example:

```
In [29]: from math import sin, cos, exp, log

         functions = {"sine" : sin,
                      "cosine" : cos,
                      "exponential" : exp,
                      "logarithm" : log}

         print(functions)
```

```
'sine': <built-in function sin>, 'exponential': <built-in function exp>, 'logarithm': <bui
```

Note that the order it prints out need not match the order we entered the values in. In fact, the order could change if we used a different machine, or entered the values again. This emphasizes the unordered nature of dictionaries.

To access an individual value, we use its key:

```
In [30]: print(functions["exponential"])
```

```
<built-in function exp>
```

To find all the keys or values we can use dictionary methods:

```
In [31]: print(functions.keys())
         print(functions.values())
```

```
dict_keys(['sine', 'exponential', 'logarithm', 'cosine'])
dict_values([<built-in function sin>, <built-in function exp>, <built-in function log>, <bu
```

Depending on the version of Python you are using, this might either give a list or an iterator.

When iterating over a dictionary (`for k in dict:`) the *key* is returned, as if we had said `for key in dict.keys():`. This is most useful in a loop, such as the following. Think carefully about this code, and make sure you understand what is happening!

```
In [32]: for name in functions:
             print("The result of {}(1) is {}.".format(name, functions[name](1.0)))
```

```
The result of sine(1) is 0.8414709848078965.
The result of exponential(1) is 2.718281828459045.
The result of logarithm(1) is 0.0.
The result of cosine(1) is 0.5403023058681398.
```

To explain: The first line says that we are going to iterate over each entry in the dictionary by assigning the value of the key (which is the name of the function in `functions`) to the variable `name`. The next line extracts the function associated with that name using `functions[name]` and applies that function to the value 1 using `functions[name](1.0)`. The full line prints out the result in a human readable form.

But Python has other ways of iterating over dictionaries that can make life even easier, such as the `items` function:

```
In [33]: for name, function in functions.items():
             print("The result of {}(1) is {}.".format(name, function(1.0)))
```

```
The result of sine(1) is 0.8414709848078965.
The result of exponential(1) is 2.718281828459045.
The result of logarithm(1) is 0.0.
The result of cosine(1) is 0.5403023058681398.
```

So this does exactly the same thing as the previous loop, and most of the code is the same. However, rather than accessing the dictionary each time (using `functions[name]`), the value in the dictionary has been returned at the start. What is happening is that the `items` function is returning *both* the key *and* the value as a tuple on each iteration through the loop. The `name,function` notation then uses unpacking to appropriately set the variables. This form is "more pythonic" (ie, is shorter, clearer to many people, and faster).

Above we have always set the key to be a string. This is not necessary - it can be an integer, or a float, or any *constant* object. We have also set the values to have the same type. As with lists this is not necessary.

Dictionaries are very useful for adding simple structure to your code, and allow you to pass around complex sets of parameters easily.

4.3 Control flow

Not every algorithm can be expressed as a single mathematical formula in the manner used so far. Alternatively, it may make the algorithm appear considerably simpler if it isn't expressed in one complex formula but in multiple simpler forms. This is where the computer has to be able to make choices, to control *when* and *if* a particular formula is used.

As a simple example, if we used our `degrees_to_radians` calculation as previously given, then if the angle θ_d is outside the standard $[0, 360]^\circ$ interval then the converted angle in radians θ_r will be outside the $[0, 2\pi]$ interval.

Suppose we want to "normalize" all our angles to lie within the $[0, 2\pi]$ interval. We could use modular arithmetic using the `%` operator:

```
In [34]: theta_d = 5134.6
         theta_d_normalized = theta_d % 360.0
         print(theta_d_normalized)
```

```
94.60000000000036
```

But it might be that the input is just wrong: there's a typo, and the caller should be warned, and maybe the input completely rejected. We can write a different function to check that.

```
In [35]: from math import pi

         def check_angle_normalized(theta_d):
             """
             Check that an angle lies within [0, 360] degrees.

             Parameters
             ----------

             theta_d : float
                 The angle in degrees.

             Returns
             -------

             normalized : Boolean
                 Whether the angle lies within the range
             """

             normalized = True
             if theta_d > 360.0:
                 normalized = False
                 print("Input angle greater than 360 degrees. Did you mean this?")
             if theta_d < 0.0:
                 normalized = False
                 print("Input angle less than 0 degrees. Did you mean this?")
             return normalized
```

```
In [36]: theta_d = 5134.6
         print(check_angle_normalized(theta_d))
         theta_d = -52.3
         print(check_angle_normalized(theta_d))
```

```
Input angle greater than 360 degrees. Did you mean this?
False
Input angle less than 0 degrees. Did you mean this?
False
```

The control flow here uses the `if` statement. As with loops such as the `for` and `while` loops we have a condition which is checked which, if satisfied, leads to the indented code block after the colon being executed. The logical statements `theta_d > 360.0` and `theta_d < 0.0` are evaluated and return either `True` or `False` (which is how Python represents *boolean* values). If `True`, then the statement is executed.

We could use only a single logical statement to check if θ_d lies in an acceptable range by using logical relations. For example, we could replace the two `if` statements by the single statement

```
if (theta_d > 360.0) or (theta_d < 0.0):
    normalized = False
    print("Input angle outside [0, 360] degrees. Did you mean this?")
```

The logical statement `(theta_d > 360.0) or (theta_d < 0.0)` is either `True` or `False` as above. In addition to the logical `or` statement, Python also has the logical `and` and logical `not` statements, from which more complex statements can be generated.

Often we want to do one thing if a condition is true, and another if the condition is false. A full example of this would be to rewrite the whole function as:

```
In [37]: from math import pi

         def check_angle_normalized(theta_d):
             """
             Check that an angle lies within [0, 360] degrees.

             Parameters
             ----------

             theta_d : float
                 The angle in degrees.

             Returns
             -------

             normalized : Boolean
                 Whether the angle lies within the range
             """

             normalized = True
             if theta_d > 360.0:
                 normalized = False
                 print("Input angle greater than 360 degrees. Did you mean this?")
             elif theta_d < 0.0:
                 normalized = False
                 print("Input angle less than 0 degrees. Did you mean this?")
             else:
                 print("Input angle in range [0, 360] degrees. Good.")
             return normalized
```

The `elif` statement allows another condition to be checked - it is how Python represents "else if", or "all previous checks have been false; let's check this statement as well". Multiple `elif` blocks can be included to check more conditions. The `else` statement contains no logical check: this code block will always be executed if all previous statements were false.

For example:

```
In [38]: theta_d = 543.2
         print(check_angle_normalized(theta_d))
         theta_d = -123.4
         print(check_angle_normalized(theta_d))
         theta_d = 89.12
         print(check_angle_normalized(theta_d))
```

```
Input angle greater than 360 degrees. Did you mean this?
False
Input angle less than 0 degrees. Did you mean this?
False
```

```
Input angle in range [0, 360] degrees. Good.
True
```

We can *nest* statements as deep as we like, nesting loops and control flow statements as we go. We have to ensure that the indentation level is consistent. Here is a silly example.

```
In [39]: angles = [-123.4, 543.2, 89.12, 0.67, 5143.6, 30.0, 270.0]

         # We run through all the angles, but only print those that are
         # - in the range [0, 360], and
         # - if sin^2(angle) < 0.5

         from math import sin

         for angle in angles:
             print("Input angle in degrees:", angle)
             if (check_angle_normalized(angle)):
                 angle_r = degrees_to_radians(angle)
                 if (sin(angle_r)**2 < 0.5):
                     print("Valid angle in radians:", angle_r)
```

```
Input angle in degrees: -123.4
Input angle less than 0 degrees. Did you mean this?
Input angle in degrees: 543.2
Input angle greater than 360 degrees. Did you mean this?
Input angle in degrees: 89.12
Input angle in range [0, 360] degrees. Good.
Input angle in degrees: 0.67
Input angle in range [0, 360] degrees. Good.
Valid angle in radians: 0.011693705988362009
Input angle in degrees: 5143.6
Input angle greater than 360 degrees. Did you mean this?
Input angle in degrees: 30.0
Input angle in range [0, 360] degrees. Good.
Valid angle in radians: 0.5235987755982988
Input angle in degrees: 270.0
Input angle in range [0, 360] degrees. Good.
```

4.4 Debugging

Earlier we saw how to read error messages to debug single statements. When we start including loops and functions it may be more complex and the information from the error message alone, whilst useful, may not be enough.

In these more complex cases the reason for the error depends on the calculations inside the code, and the steps through the code need inspecting in detail. This is where a *debugger* is useful. It allows you to run the code, pause at specific points or conditions, step through it as it runs line-by-line, and inspect all the values as you go. There are a number of Python debuggers - `pdb` and `ipdb` being the most basic. However, `spyder` has a debugger built in, and learning to use it will make your life considerably easier.

4.4.1 Breakpoints

The main use of the debugger is to inspect the internal state of a code whilst it is running. To do that we have to stop the execution of the code somewhere. This is typically done using *breakpoints*.

Copy the following function into a file named `breakpoints.py`:

```
def test_sequence(N):
    """
    Compute the infinite sum of 2^{-n} starting from n = 0, truncating
    at n = N, returning the value of 2^{-n} and the truncated sum.

    Parameters
    ----------

    N : int
        Positive integer, giving the number of terms in the sum

    Returns
    -------

    limit : float
        The value of 2^{-N}
    partial_sum : float
        The value of the truncated sum

    Notes
    -----

    The limiting value should be zero, and the value of the sum should
    converge to 2.
    """

    # Start sum from zero, so give zeroth term
    limit = 1.0
    partial_sum = 1.0

    # At each step, increment sum and change summand
    for n in range(1, N+1):
        partial_sum = partial_sum + limit
        limit = limit / 2.0

    return limit, partial_sum

if __name__ == '__main__':
    print(test_sequence(50))
```

This computes the value 2^{-N} and the partial sum $\sum_{n=0}^{N} 2^{-n}$. The limit as $N \to \infty$ of the value should be zero, and of the sum should be two.

The final two lines ensures that, if the file is run as a python script, the function will be called (with `N=50`). The `if` statement is a standard Python convention: if you have code that you want executed *only if* the file is run as a script, and not if the file is imported as a module.

If we run the function we find it does not work as expected:

```
In [40]: import breakpoints
         print(breakpoints.test_sequence(10))
         print(breakpoints.test_sequence(100))
         print(breakpoints.test_sequence(1000))
```

```
(0.0009765625, 2.998046875)
(7.888609052210118e-31, 3.0)
(9.332636185032189e-302, 3.0)
```

The value is tending to the right limit; the sum is not. The test (that we should have written formally) has failed. It may be obvious what the problem is, but we illustrate how to find it using the debugger.

First, we must know what to expect. The terms in the partial sum (which is where the problem lies) should be

$$\sum_{n=0}^{N} 2^{-n} = 1 + \frac{1}{2} + \frac{1}{4} + \dots$$

We want to inspect what the partial sum actually does. This is performed in the code within the `for` loop, which starts on line 32 of the code.

1. Open the code in `spyder`.
2. Ensure that the working directory is set to the directory the file is in (shortcut: the top right corner of the editor part of the window contains a drop down "Options" menu: the second option "Set console working directory" will do what is needed.)
3. Set a breakpoint on line 32. To do this, double-click on the "32" on the left hand side of the editor screen. A small red circle will appear next to it. (To remove a breakpoint, double click the number again)
4. Start debugging by going to the "Debug" menu and clicking "Debug". The console will indicate that
   (a) The code has been started
   (b) The code has paused at line 32.
5. In the top right of the `spyder` window, click on the "Variable explorer" tab. This shows the values of all the variables at this point. Note that line 32 has not yet been executed. The value of the partial sum, given by `partial_sum`, is currently `1` (as the `N=0` term is set outside of the sum).
6. Work out which toolbar button "Runs" the current line. You should see
   (a) The "active line" marker in the editor and console move forward to line 33;
   (b) The value of the loop counter `n` appear in the Variable explorer, set to 1.
7. "Run" the current line again (which will now be line 33). We see that it adds the current value of `limit`, which is meant to represent 2^{-n}, to the partial sum `partial_sum`. However, this value is 1, so `partial_sum` becomes 2. According to the sum written out above, this $n = 1$ term should be $1/2$. So this is a bug.

To fix this bug, the simplest thing to do is to ensure that `limit` has the value corresponding to the `n`th part of the sum *before* it is used, by swapping lines 33 and 34.

You should experiment with adding breakpoints and stepping through some of your own codes. In particular, you should note that "Run"ning the current line of code will skip over any function calls, including calls to functions you have yourself defined. If you want to follow the code as it executes functions, you need to use the "Step into" button instead.

4.5 Exercise: Prime numbers

4.5.1 Exercise 1

Write a function that tests if a number is prime. Test it by writing out all prime numbers less than 50.

Hint: if `b` divides `a` then `a % b == 0` is `True`.

4.5.2 Exercise 2

500 years ago some believed that the number $2^n - 1$ was prime for *all* primes n. Use your function to find the first prime n for which this is not true.

4.5.3 Exercise 3

The *Mersenne* primes are those that have the form $2^n - 1$, where n is prime. Use your previous solutions to generate all the $n < 40$ that give Mersenne primes.

4.5.4 Exercise 4

Write a function to compute all prime factors of an integer n, including their multiplicities. Test it by printing the prime factors (without multiplicities) of $n = 17, \ldots, 20$ and the multiplicities (without factors) of $n = 48$.

Note

One effective solution is to return a *dictionary*, where the keys are the factors and the values are the multiplicities.

4.5.5 Exercise 5

Write a function to generate all the integer divisors, including 1, but not including n itself, of an integer n. Test it on $n = 16, \ldots, 20$.

Note

You could use the prime factorization from the previous exercise, or you could do it directly.

4.5.6 Exercise 6

A *perfect* number n is one where the divisors sum to n. For example, 6 has divisors 1, 2, and 3, which sum to 6. Use your previous solution to find all perfect numbers $n < 10,000$ (there are only four!).

4.5.7 Exercise 7

Using your previous functions, check that all perfect numbers $n < 10,000$ can be written as $2^{k-1} \times (2^k - 1)$, where $2^k - 1$ is a Mersenne prime.

4.5.8 Exercise 8 (bonus)

Investigate the `timeit` function in Python or IPython. Use this to measure how long your function takes to check that, if k on the Mersenne list then $n = 2^{k-1} \times (2^k - 1)$ is a perfect number, using your functions. Stop increasing k when the time takes too long!

Note

You could waste considerable time on this, and on optimizing the functions above to work efficiently. It is *not* worth it, other than to show how rapidly the computation time can grow!

CHAPTER 5

Basic Plotting

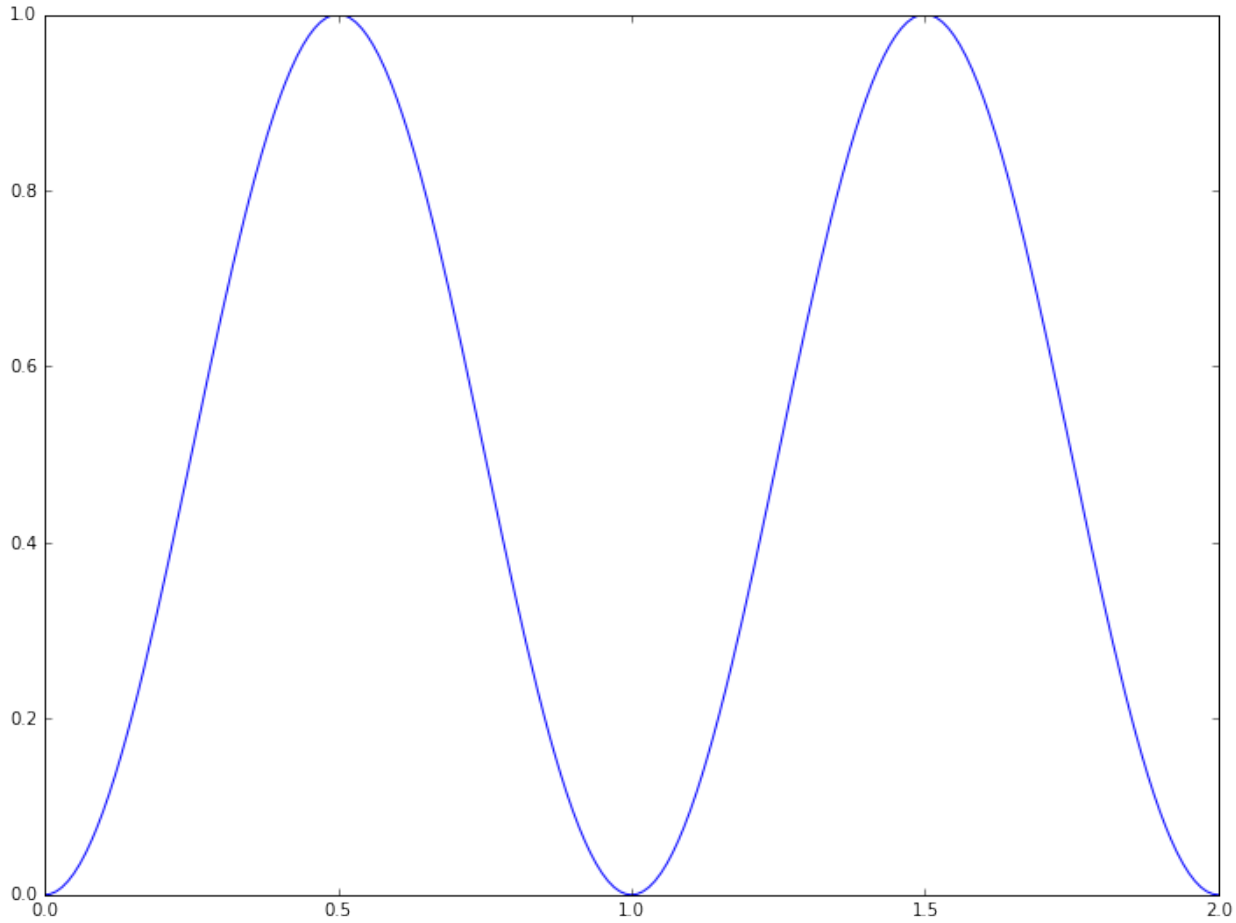
5.1 Plotting

There are many Python plotting libraries depending on your purpose. However, the standard general-purpose library is `matplotlib`. This is often used through its `pyplot` interface.

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

```
In [2]: %matplotlib inline
        from matplotlib import rcParams
        rcParams['figure.figsize']=(12,9)
```

```
In [3]: from math import sin, pi

        x = []
        y = []
        for i in range(201):
            x_point = 0.01*i
            x.append(x_point)
            y.append(sin(pi*x_point)**2)

        pyplot.plot(x, y)
        pyplot.show()
```



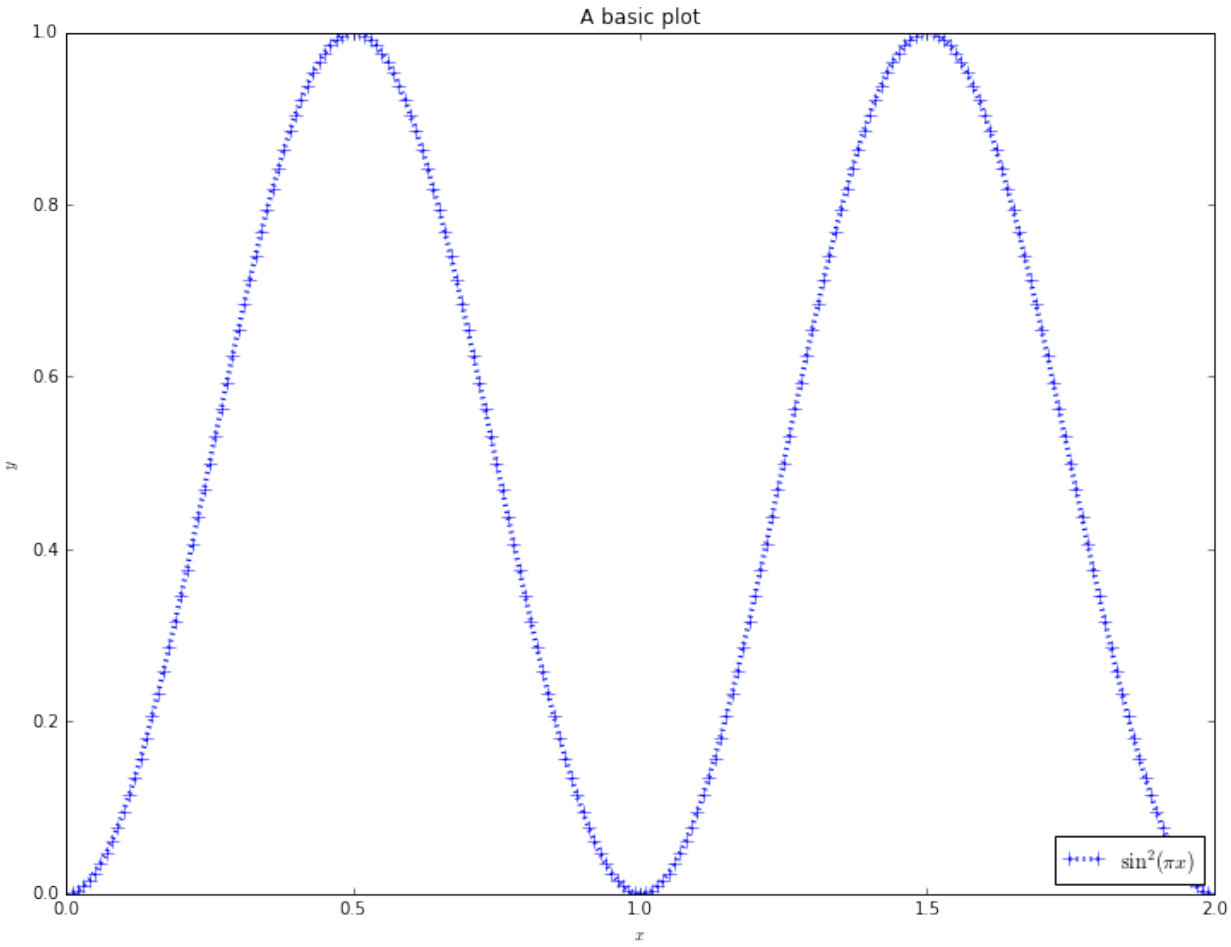
We have defined two sequences - in this case lists, but tuples would also work. One contains the x-axis coordinates, the other the data points to appear on the y-axis. A basic plot is produced using the `plot` command of `pyplot`. However, this plot will not automatically appear on the screen, as after plotting the data you may wish to add additional information. Nothing will actually happen until you either save the figure to a file (using `pyplot.savefig(<filename>)`) or explicitly ask for it to be displayed (with the `show` command). When the plot is displayed the program will typically pause until you dismiss the plot.

If using the notebook you can include the command `%matplotlib inline` or `%matplotlib notebook` before plotting to make the plots appear automatically inside the notebook. If code is included in a program which is run inside `spyder` through an IPython console, the figures may appear in the console automatically. Either way, it is good practice to always include the `show` command to explicitly display the plot.

This plotting interface is straightforward, but the results are not particularly nice. The following commands illustrate some of the ways of improving the plot:

```
In [4]: from math import sin, pi

        x = []
        y = []
        for i in range(201):
            x_point = 0.01*i
            x.append(x_point)
            y.append(sin(pi*x_point)**2)

        pyplot.plot(x, y, marker='+', markersize=8, linestyle=':',
```

```
                linewidth=3, color='b', label=r'$\sin^2(\pi x)$')
pyplot.legend(loc='lower right')
pyplot.xlabel(r'$x$')
pyplot.ylabel(r'$y$')
pyplot.title('A basic plot')
pyplot.show()
```



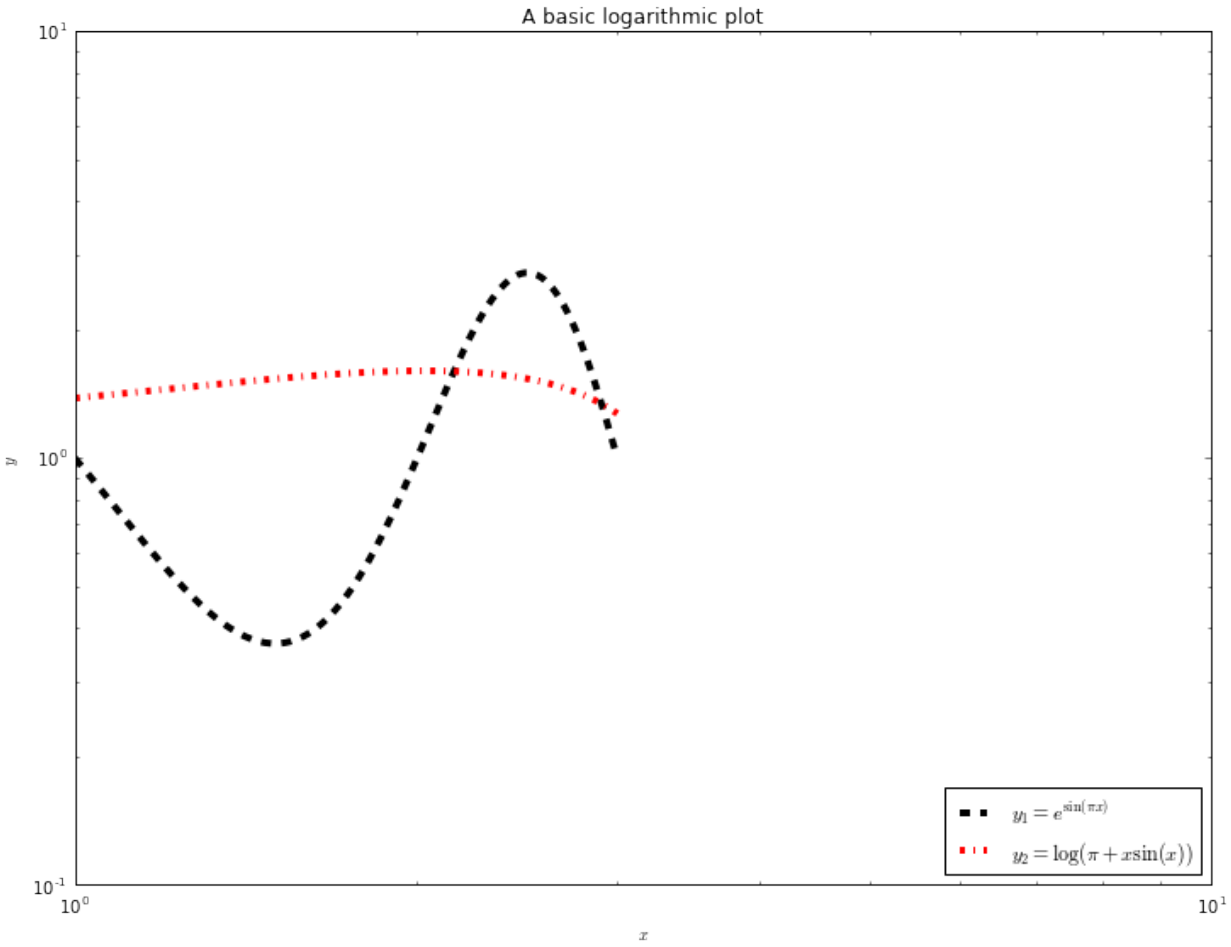
Whilst most of the commands are self-explanatory, a note should be made of the strings line `r'$x$'`. These strings are in LaTeX format, which is *the* standard typesetting method for professional-level mathematics. The `$` symbols surround mathematics. The `r` before the definition of the string is Python notation, not LaTeX. It says that the following string will be "raw": that backslash characters should be left alone. Then, special LaTeX commands have a backslash in front of them: here we use `\pi` and `\sin`. Most basic symbols can be easily guessed (eg `\theta` or `\int`), but there are useful lists of symbols, and a reverse search site available. We can also use `^` to denote superscripts (used here), _ to denote subscripts, and use `{}` to group terms.

By combining these basic commands with other plotting types (`semilogx` and `loglog`, for example), most simple plots can be produced quickly.

Here are some more examples:

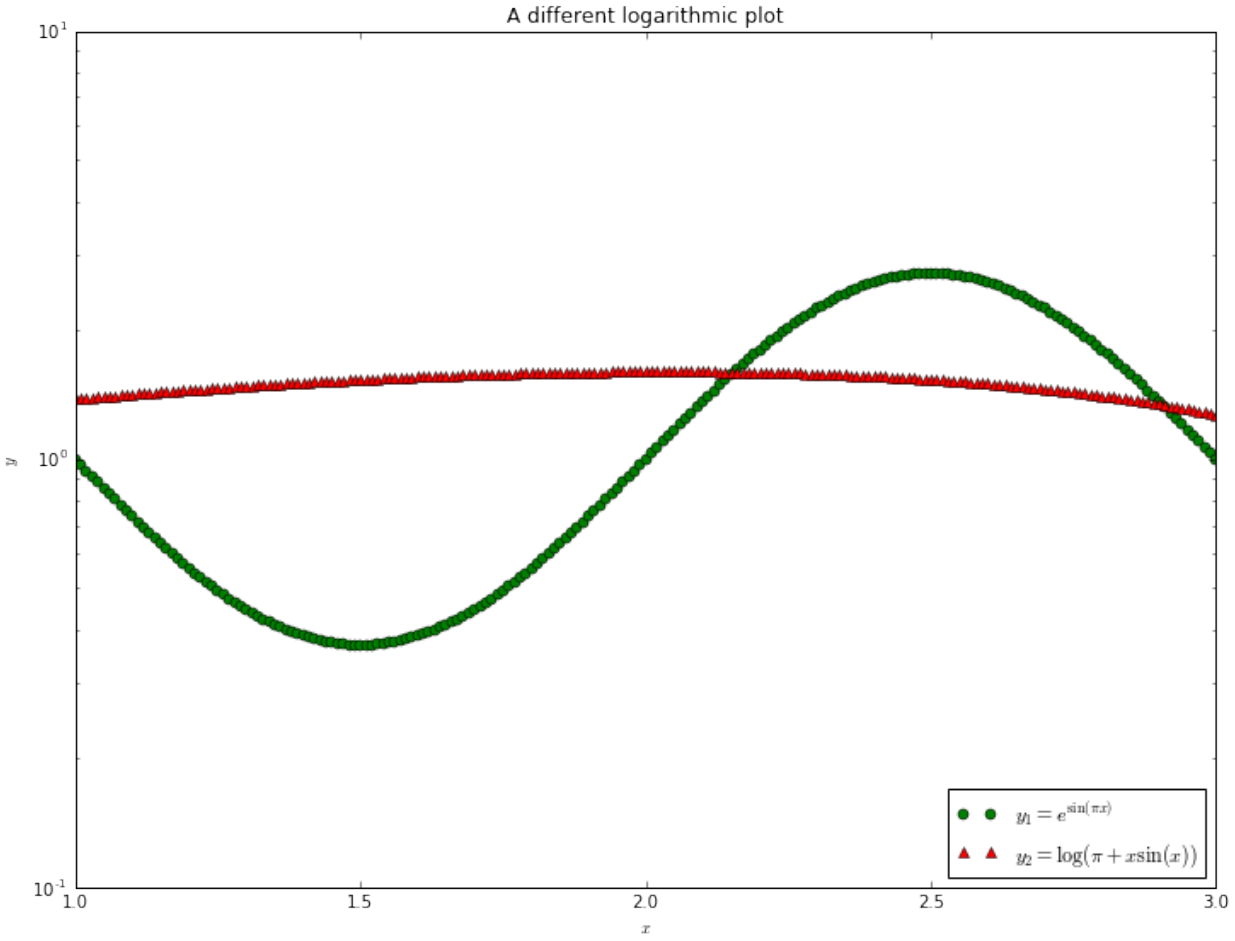
```
In [5]: from math import sin, pi, exp, log

        x = []
        y1 = []
        y2 = []
```

```
for i in range(201):
    x_point = 1.0 + 0.01*i
    x.append(x_point)
    y1.append(exp(sin(pi*x_point)))
    y2.append(log(pi+x_point*sin(x_point)))

pyplot.loglog(x, y1, linestyle='--', linewidth=4,
              color='k', label=r'$y_1=e^{\sin(\pi x)}$')
pyplot.loglog(x, y2, linestyle='-.', linewidth=4,
              color='r', label=r'$y_2=\log(\pi+x\sin(x))$')
pyplot.legend(loc='lower right')
pyplot.xlabel(r'$x$')
pyplot.ylabel(r'$y$')
pyplot.title('A basic logarithmic plot')
pyplot.show()
```




```
In [6]: from math import sin, pi, exp, log

        x = []
        y1 = []
        y2 = []
        for i in range(201):
            x_point = 1.0 + 0.01*i
            x.append(x_point)
```

```
        y1.append(exp(sin(pi*x_point)))
        y2.append(log(pi+x_point*sin(x_point)))

    pyplot.semilogy(x, y1, linestyle='None', marker='o',
                    color='g', label=r'$y_1=e^{\sin(\pi x)}$')
    pyplot.semilogy(x, y2, linestyle='None', marker='^',
                    color='r', label=r'$y_2=\log(\pi+x\sin(x))$')
    pyplot.legend(loc='lower right')
    pyplot.xlabel(r'$x$')
    pyplot.ylabel(r'$y$')
    pyplot.title('A different logarithmic plot')
    pyplot.show()
```



We will look at more complex plots later, but the matplotlib documentation contains a lot of details, and the gallery contains a lot of examples that can be adapted to fit. There is also an extremely useful document as part of Johansson's lectures on scientific Python, and an introduction by Nicolas Rougier.

5.2 Exercise: Logistic map

The logistic map builds a sequence of numbers $\{x_n\}$ using the relation

$$x_{n+1} = rx_n\left(1 - x_n\right),$$

where $0 \leq x_0 \leq 1$.

5.2.1 Exercise 1

Write a program that calculates the first N members of the sequence, given as input x_0 and r (and, of course, N).

5.2.2 Exercise 2

Fix $x_0 = 0.5$. Calculate the first 2,000 members of the sequence for $r = 1.5$ and $r = 3.5$. Plot the last 100 members of the sequence in both cases.

What does this suggest about the long-term behaviour of the sequence?

5.2.3 Exercise 3

Fix $x_0 = 0.5$. For each value of r between 1 and 4, in steps of 0.01, calculate the first 2,000 members of the sequence. Plot the last 1,000 members of the sequence on a plot where the x-axis is the value of r and the y-axis is the values in the sequence. Do not plot lines - just plot markers (e.g., use the `'k.'` plotting style).

5.2.4 Exercise 4

For iterative maps such as the logistic map, one of three things can occur:

1. The sequence settles down to a *fixed point*.
2. The sequence rotates through a finite number of values. This is called a *limit cycle*.
3. The sequence generates an infinite number of values. This is called *deterministic chaos*.

Using just your plot, or new plots from this data, work out approximate values of r for which there is a transition from fixed points to limit cycles, from limit cycles of a given number of values to more values, and the transition to chaos.

CHAPTER 6

Classes and objects

6.1 Classes and Object Oriented Programming

We have looked at functions which take input and return output (or do things to the input). However, sometimes it is useful to think about *objects* first rather than the actions applied to them.

Think about a polynomial, such as the cubic

$$p(x) = 12 - 14x + 2x^3.$$

This is one of the standard forms that we would expect to see for a polynomial. We could imagine representing this in Python using a container containing the coefficients, such as:

```
In [1]: p_normal = (12, -14, 0, 2)
```

The order of the polynomial is given by the number of coefficients (minus one), which is given by `len(p_normal)-1`.

However, there are many other ways it could be written, which are useful in different contexts. For example, we are often interested in the roots of the polynomial, so would want to express it in the form

$$p(x) = 2(x - 1)(x - 2)(x + 3).$$

This allows us to read off the roots directly. We could imagine representing this in Python using a container containing the roots, such as:

```
In [2]: p_roots = (1, 2, -3)
```

combined with a single variable containing the leading term,

```
In [3]: p_leading_term = 2
```

We see that the order of the polynomial is given by the number of roots (and hence by `len(p_roots)`). This form represents the same polynomial but requires two pieces of information (the roots and the leading coefficient).

The different forms are useful for different things. For example, if we want to add two polynomials the standard form makes it straightforward, but the factored form does not. Conversely, multiplying polynomials in the factored form is easy, whilst in the standard form it is not.

But the key point is that the object - the polynomial - is the same: the representation may appear different, but it's the object itself that we really care about. So we want to represent the object in code, and work with that object.

6.1.1 Classes

Python, and other languages that include *object oriented* concepts (which is most modern languages) allow you to define and manipulate your own objects. Here we will define a *polynomial* object step by step.

```
In [4]: class Polynomial(object):
            explanation = "I am a polynomial"

            def explain(self):
                print(self.explanation)
```

We have defined a *class*, which is a single object that will represent a polynomial. We use the keyword `class` in the same way that we use the keyword `def` when defining a function. The definition line ends with a colon, and all the code defining the object is indented by four spaces.

The name of the object - the general class, or type, of the thing that we're defining - is `Polynomial`. The convention is that class names start with capital letters, but this convention is frequently ignored.

The type of object that we are building on appears in brackets after the name of the object. The most basic thing, which is used most often, is the `object` type as here.

Class variables are defined in the usual way, but are only visible inside the class. Variables that are set outside of functions, such as `explanation` above, will be common to all class variables.

Functions are defined inside classes in the usual way (using the `def` keyword, indented by four additional spaces). They work in a special way: they are not called directly, but only when you have a member of the class. This is what the `self` keyword does: it takes the specific *instance* of the class and uses its data. Class functions are often called *methods*.

Let's see how this works on a specific example:

```
In [5]: p = Polynomial()
        print(p.explanation)
        p.explain()
        p.explanation = "I change the string"
        p.explain()
```

```
I am a polynomial
I am a polynomial
I change the string
```

The first line, `p = Polynomial()`, creates an *instance* of the class. That is, it creates a specific `Polynomial`. It is assigned to the variable named `p`. We can access class variables using the "dot" notation, so the string can be printed via `p.explanation`. The method that prints the class variable also uses the "dot" notation, hence `p.explain()`. The `self` variable in the definition of the function is the instance itself, `p`. This is passed through automatically thanks to the dot notation.

Note that we can change class variables in specific instances in the usual way (`p.explanation = ...` above). This only changes the variable for that instance. To check that, let us define two polynomials:

```
In [6]: p = Polynomial()
        p.explanation = "Changed the string again"
        q = Polynomial()
        p.explanation = "Changed the string a third time"
        p.explain()
        q.explain()
```

```
Changed the string a third time
I am a polynomial
```

We can of course make the methods take additional variables. We modify the class (note that we have to completely re-define it each time):

```
In [7]: class Polynomial(object):
            explanation = "I am a polynomial"

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
```

We then use this, remembering that the `self` variable is passed through automatically:

```
In [8]: r = Polynomial()
        r.explain_to("Alice")
```

```
Hello, Alice. I am a polynomial.
```

At the moment the class is not doing anything interesting. To do something interesting we need to store (and manipulate) relevant variables. The first thing to do is to add those variables when the instance is actually created. We do this by adding a special function (method) which changes how the variables of type `Polynomial` are created:

```
In [9]: class Polynomial(object):
            """Representing a polynomial."""
            explanation = "I am a polynomial"

            def __init__(self, roots, leading_term):
                self.roots = roots
                self.leading_term = leading_term
                self.order = len(roots)

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
                print("My roots are {}.".format(self.roots))
```

This `__init__` function is called when a variable is created. There are a number of special class functions, each of which has two underscores before and after the name. This is another Python *convention* that is effectively a rule: functions surrounded by two underscores have special effects, and will be called by other Python functions internally. So now we can create a variable that represents a specific polynomial by storing its roots and the leading term:

```
In [10]: p = Polynomial(p_roots, p_leading_term)
         p.explain_to("Alice")
         q = Polynomial((1,1,0,-2), -1)
         q.explain_to("Bob")
```

```
Hello, Alice. I am a polynomial.
My roots are (1, 2, -3).
Hello, Bob. I am a polynomial.
My roots are (1, 1, 0, -2).
```

It is always useful to have a function that shows what the class represents, and in particular what this particular instance looks like. We can define another method that explicitly `displays` the `Polynomial`:

```
In [11]: class Polynomial(object):
             """Representing a polynomial."""
             explanation = "I am a polynomial"

             def __init__(self, roots, leading_term):
                 self.roots = roots
                 self.leading_term = leading_term
                 self.order = len(roots)
```

```
            def display(self):
                string = str(self.leading_term)
                for root in self.roots:
                    if root == 0:
                        string = string + "x"
                    elif root > 0:
                        string = string + "(x - {})".format(root)
                    else:
                        string = string + "(x + {})".format(-root)
                return string

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
                print("My roots are {}.".format(self.roots))
```

```
In [13]: p = Polynomial(p_roots, p_leading_term)
         print(p.display())
         q = Polynomial((1,1,0,-2), -1)
         print(q.display())
```

```
2(x - 1)(x - 2)(x + 3)
-1(x - 1)(x - 1)x(x + 2)
```

Where classes really come into their own is when we manipulate them as objects in their own right. For example, we can multiply together two polynomials to get another polynomial. We can create a method to do that:

```
In [14]: class Polynomial(object):
             """Representing a polynomial."""
             explanation = "I am a polynomial"

             def __init__(self, roots, leading_term):
                 self.roots = roots
                 self.leading_term = leading_term
                 self.order = len(roots)

             def display(self):
                 string = str(self.leading_term)
                 for root in self.roots:
                     if root == 0:
                         string = string + "x"
                     elif root > 0:
                         string = string + "(x - {})".format(root)
                     else:
                         string = string + "(x + {})".format(-root)
                 return string

             def multiply(self, other):
                 roots = self.roots + other.roots
                 leading_term = self.leading_term * other.leading_term
                 return Polynomial(roots, leading_term)

             def explain_to(self, caller):
                 print("Hello, {}. {}.".format(caller,self.explanation))
                 print("My roots are {}.".format(self.roots))
```

```
In [15]: p = Polynomial(p_roots, p_leading_term)
         q = Polynomial((1,1,0,-2), -1)
         r = p.multiply(q)
         print(r.display())
```

```
-2(x - 1)(x - 2)(x + 3)(x - 1)(x - 1)x(x + 2)
```

We now have a simple class that can represent polynomials and multiply them together, whilst printing out a simple string form representing itself. This can obviously be extended to be much more useful.

6.2 Exercise: Equivalence classes

An *equivalence class* is a relation that groups objects in a set into related subsets. For example, if we think of the integers modulo 7, then 1 is in the same equivalence class as 8 (and 15, and 22, and so on), and 3 is in the same equivalence class as 10. We use the tilde $3 \sim 10$ to denote two objects within the same equivalence class.

Here, we are going to define the positive integers programmatically from equivalent sequences.

6.2.1 Exercise 1

Define a Python class `Eqint`. This should be

1. Initialized by a sequence;
2. Store the sequence;
3. Have a `display` method that returns a string showing the integer length of the sequence;
4. Have an `equals` method that checks if two `Eqints` are equal, which is `True` if, and only if, their sequences have the same length.

6.2.2 Exercise 2

Define a `zero` object from the empty list, and three `one` objects, from a single object list, tuple, and string. For example

```
one_list = Eqint([1])
one_tuple = Eqint((1,))
one_string = Eqint('1')
```

Check that none of the `one` objects equal the zero object, but all equal the other `one` objects. Display each object to check that the representation gives the integer length.

6.2.3 Exercise 3

Redefine the class by including an `add` method that combines the two sequences. That is, if `a` and `b` are `Eqints` then `a.add(b)` should return an `Eqint` defined from combining `a` and `b`s sequences.

Note

Adding two different *types* of sequences (eg, a list to a tuple) does not work, so it is better to either iterate over the sequences, or to convert to a uniform type before adding.

6.2.4 Exercise 4

Check your addition function by adding together all your previous `Eqint` objects (which will need re-defining, as the class has been redefined). Display the resulting object to check you get `3`, and also print its internal sequence.

6.2.5 Exercise 5

We will sketch a construction of the positive integers from *nothing*.

1. Define an empty list `positive_integers`.
2. Define an `Eqint` called `zero` from the empty list. Append it to `positive_integers`.
3. Define an `Eqint` called `next_integer` from the `Eqint` defined by *a copy of* `positive_integers` (ie, use `Eqint(list(positive_integers))`. Append it to `positive_integers`.
4. Repeat step 3 as often as needed.

Use this procedure to define the `Eqint` equivalent to 10. Print it, and its internal sequence, to check.

CHAPTER 7

Scientific Python

7.1 Scientific Python

A lot of computational algorithms are expressed using Linear Algebra terminology - vectors and matrices. This is thanks to the wide range of methods within Linear Algebra for solving the sort of problems that computers are good at solving!

Within Python, our first thought may be to represent a vector as a list. But there is a downside: lists do not *naturally* behave as vectors. For example:

```
In [1]: x = [1, 2, 3]
        y = [4, 9, 16]
        print(x+y)
```

```
[1, 2, 3, 4, 9, 16]
```

Similarly, we cannot apply algebraic operations or functions to lists in a straightforward manner that matches our expectations.

However, there is a Python package - `numpy` - that does give us the behaviour we want.

7.2 numpy

```
In [2]: import numpy
```

`numpy` is used so frequently that in a lot of cases and online explanations you will see it abbreviated, using `import numpy as np`. Here we try to avoid that - auto-completion inside spyder means that the additional typing is trivial, and using the full name is clearer.

`numpy` defines a special type, an `array`, which can represent vectors, matrices, and other higher-rank objects. Unlike standard Python lists, an `array` can only contain objects of a single type. The notation to create these objects is straightforward: one easy way is to start with a list:

```
In [3]: x_numpy = numpy.array(x)
        y_numpy = numpy.array(y)
        print(x_numpy + y_numpy)
        print(x_numpy[0])
        print(y_numpy[1:])
```

```
[ 5 11 19]
1
[ 9 16]
```

We see that the `array` objects behave as we would expect, and accessing elements is exactly the same as for a list. We can also perform other mathematical operations on the *whole vector*:

```
In [4]: print(3*x_numpy)
        print(numpy.log(x_numpy))
        print(x_numpy*y_numpy)
        print((x_numpy-1)**2)
```

```
[3 6 9]
[ 0.          0.69314718  1.09861229]
[ 4 18 48]
[0 1 4]
```

Think about these carefully.

1. The first case is straightforward: all elements of the vector are multiplied by a constant.
2. The second case applies a function to each element separately. `numpy` implements a version of most interesting mathematical functions, which are applied directly to each element.
3. The third case is *elementwise* multiplication of the vectors. The first component of the answer is the product of the first component of `x_numpy` with the first component of `y_numpy`. The second component of the answer is the product of the second component of `x_numpy` with the second component of `y_numpy`. We cannot use the `*` operator to represent matrix multiplication, but must use a function (see below; note that there will be an operator in Python `3.5+`, but using it will mean your code is, for now, not widely useable).
4. The fourth case shows a combination of cases above. The answer is given by elementwise subtraction of 1, then squaring (elementwise) that result.

Defining a matrix can be done by applying the `array` function to a list of lists:

```
In [5]: A_numpy = numpy.array([ [1, 2, 3], [4, 5, 6], [7, 8, 0]])
        print(A_numpy**2)
```

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

We see that for higher rank objects such as matrices, the operations are still performed *elementwise*.

To multiply a matrix by a vector, or a vector by a vector, in the standard linear algebra sense, we use the `numpy.dot` function:

```
In [6]: x_squared = numpy.dot(x_numpy, x_numpy)
        A_times_x = numpy.dot(A_numpy, x_numpy)
        print(x_squared)
        print(A_times_x)
```

```
14
[14 32 23]
```

Note that we have appeared to multiply a matrix by a row vector, and get a row vector back. This is because `numpy` does not distinguish between row and column vectors, so everything appears as a row vector. (You could define a $n \times 1$ `array` instead, but there is no advantage).

To actually check the *shape* and *size* of `numpy` arrays, you can directly check their attributes:

```
In [7]: print(x_numpy.size)
        print(x_numpy.shape)
        print(A_numpy.size)
        print(A_numpy.shape)
```

```
3
(3,)
9
(3, 3)
```

numpy contains a number of very efficient functions for working with arrays, for finding extreme values, and performing linear algebra tasks. Particular functions that are worth knowing, or starting from, are

- arange: constructs an array containing increasing integers
- linspace: constructs a linearly spaced array
- zeros and ones: constructs arrays containing just ones or zeros
- diag: extracts the diagonal of a matrix, or build a matrix with just diagonal entries
- mgrid: constructs matrices from vectors for 3d plots
- random.rand: constructs an array of random numbers.

7.2.1 Linear algebra

numpy also defines a number of linear algebra functions. However, a more comprehensive set of functions, which is better maintained and often more efficient, is given by scipy:

```
In [8]: from scipy import linalg

In [9]: print(linalg.solve(A_numpy, x_numpy))
        print(linalg.det(A_numpy))
```

```
[-0.33333333  0.66666667  0.        ]
27.0
```

In addition to solving linear systems and computing determinants, you can also factorize matrices and generally do most linear algebra operations that you need. The scipy documentation is comprehensive, and has a specific section on Linear Algebra, as well as a section in the tutorial. Johansson also has a tutorial on scipy in general.

7.2.2 Working with files

Often we will want to work with data - constants, parameters, initial conditions, measurements, and so on. numpy provides ways to work with data stored in files - either reading them in or writing them out. A list of "File I/O routines" is available, but the two key routines are `loadtxt <http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html#numpy.loadtxt>`__ and `savetxt <http://docs.scipy.org/doc/numpy/reference/generated/numpy.savetxt.html#numpy.savetxt>`__.

As a simple example we take our matrix A_numpy above and save it to a file:

```
In [10]: numpy.savetxt('A_numpy.txt', A_numpy)
```

We can then check the contents of that file (you should open the file on your machine to check):

```
In [11]: !cat A_numpy.txt
```

```
1.00000000000000000e+00 2.00000000000000000e+00 3.00000000000000000e+00
4.00000000000000000e+00 5.00000000000000000e+00 6.00000000000000000e+00
7.00000000000000000e+00 8.00000000000000000e+00 0.00000000000000000e+00
```

Finally, we can read the contents of that file into a new variable and check that it matches:

```
In [12]: A_from_file = numpy.loadtxt('A_numpy.txt')
         print(A_from_file == A_numpy)
```

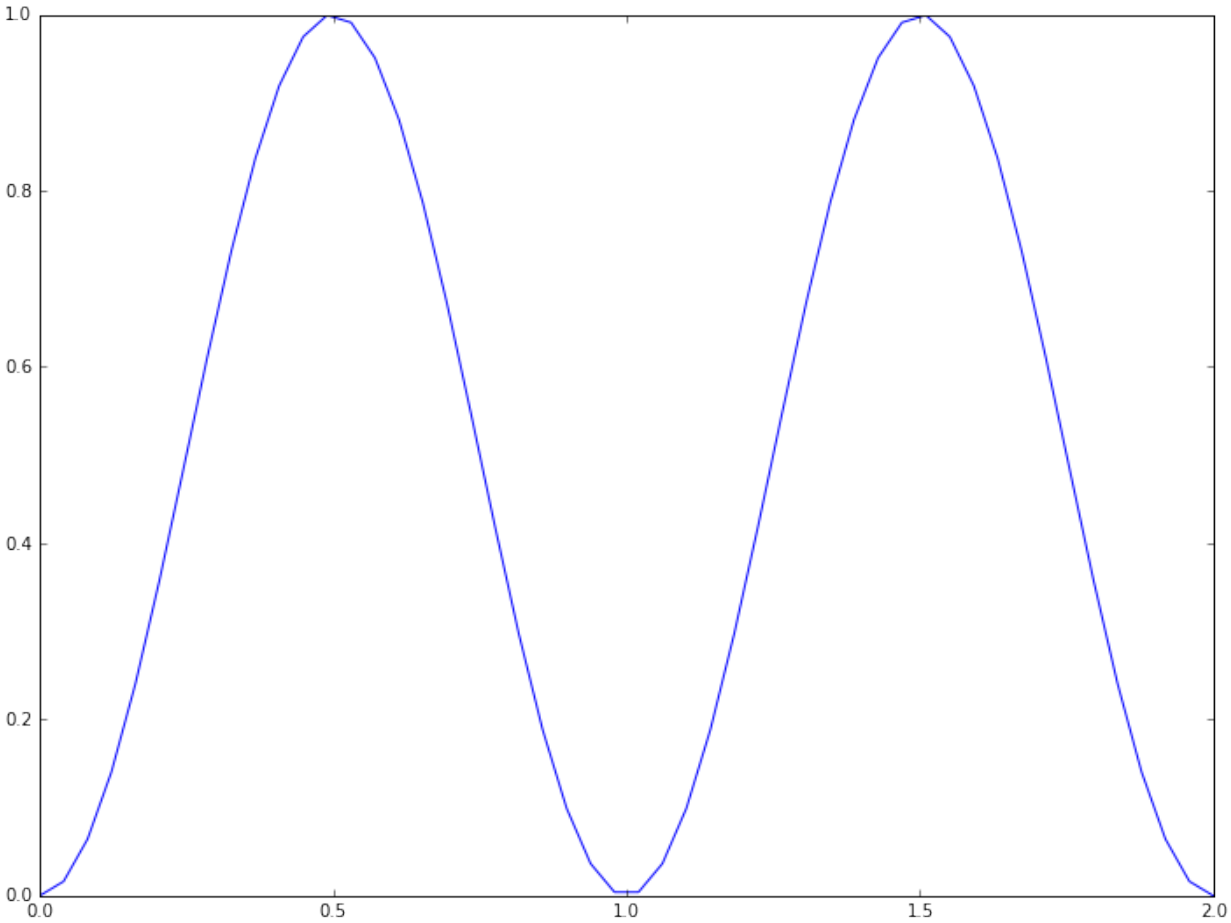
```
[[ True  True  True]
 [ True  True  True]
 [ True  True  True]]
```

7.3 Plotting

There are many Python plotting libraries depending on your purpose. However, the standard general-purpose library is `matplotlib`. This is often used through its `pyplot` interface.

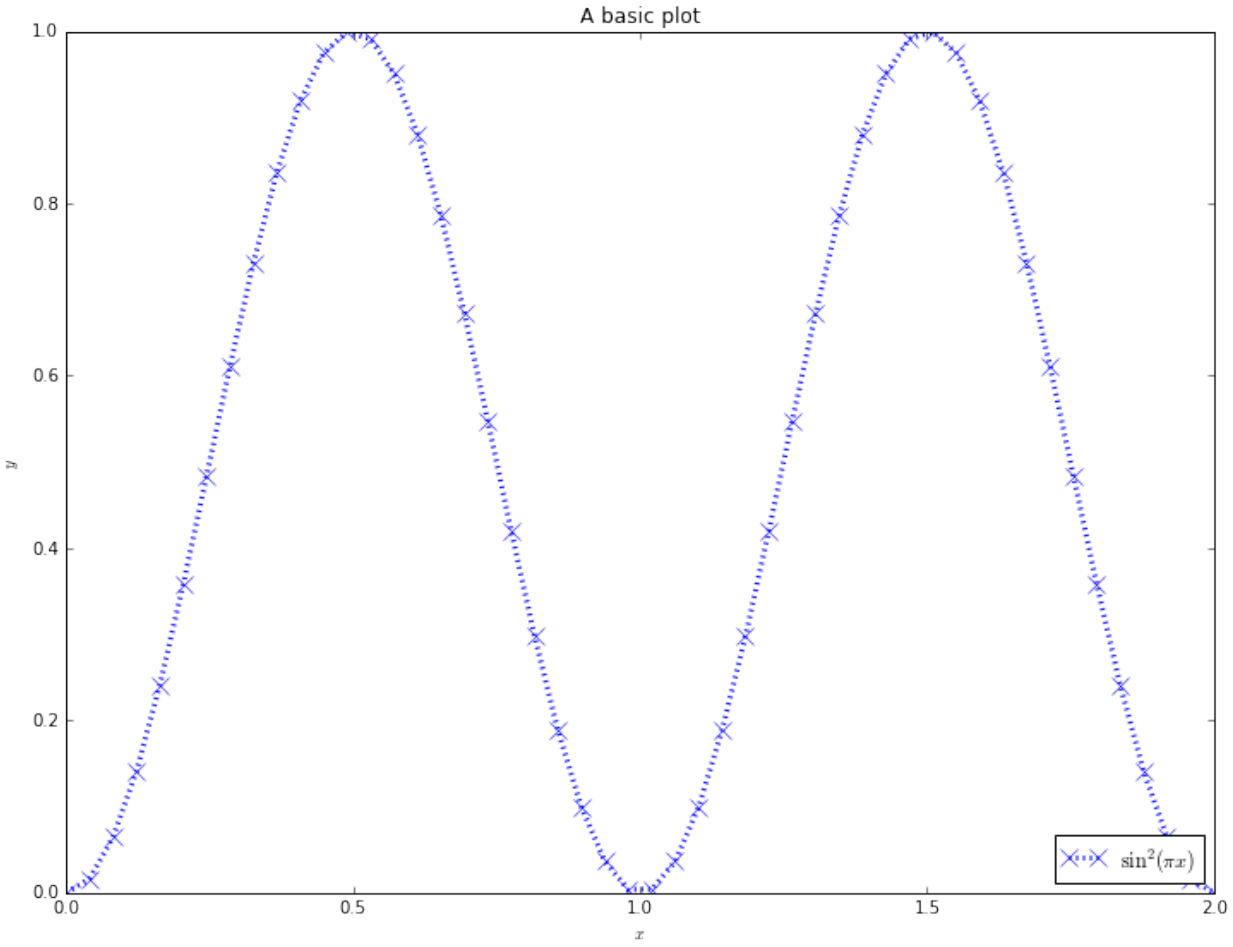
This is a quick recap of the basic plotting commands, but using `numpy` as well.

```
In [13]: from matplotlib import pyplot

In [14]: %matplotlib inline
         from matplotlib import rcParams
         rcParams['figure.figsize']=(12,9)

In [15]: x = numpy.linspace(0, 2.0)
         y = numpy.sin(numpy.pi*x)**2
         pyplot.plot(x, y)
         pyplot.show()
```



This plotting interface is straightforward, but the results are not particularly nice. The following commands illustrate some of the ways of improving the plot:

```
In [16]: x = numpy.linspace(0, 2.0)
         y = numpy.sin(numpy.pi*x)**2
         pyplot.plot(x, y, marker='x', markersize=10, linestyle=':', linewidth=3,
                     color='b', label=r'$\sin^2(\pi x)$')
         pyplot.legend(loc='lower right')
         pyplot.xlabel(r'$x$')
         pyplot.ylabel(r'$y$')
         pyplot.title('A basic plot')
         pyplot.show()
```



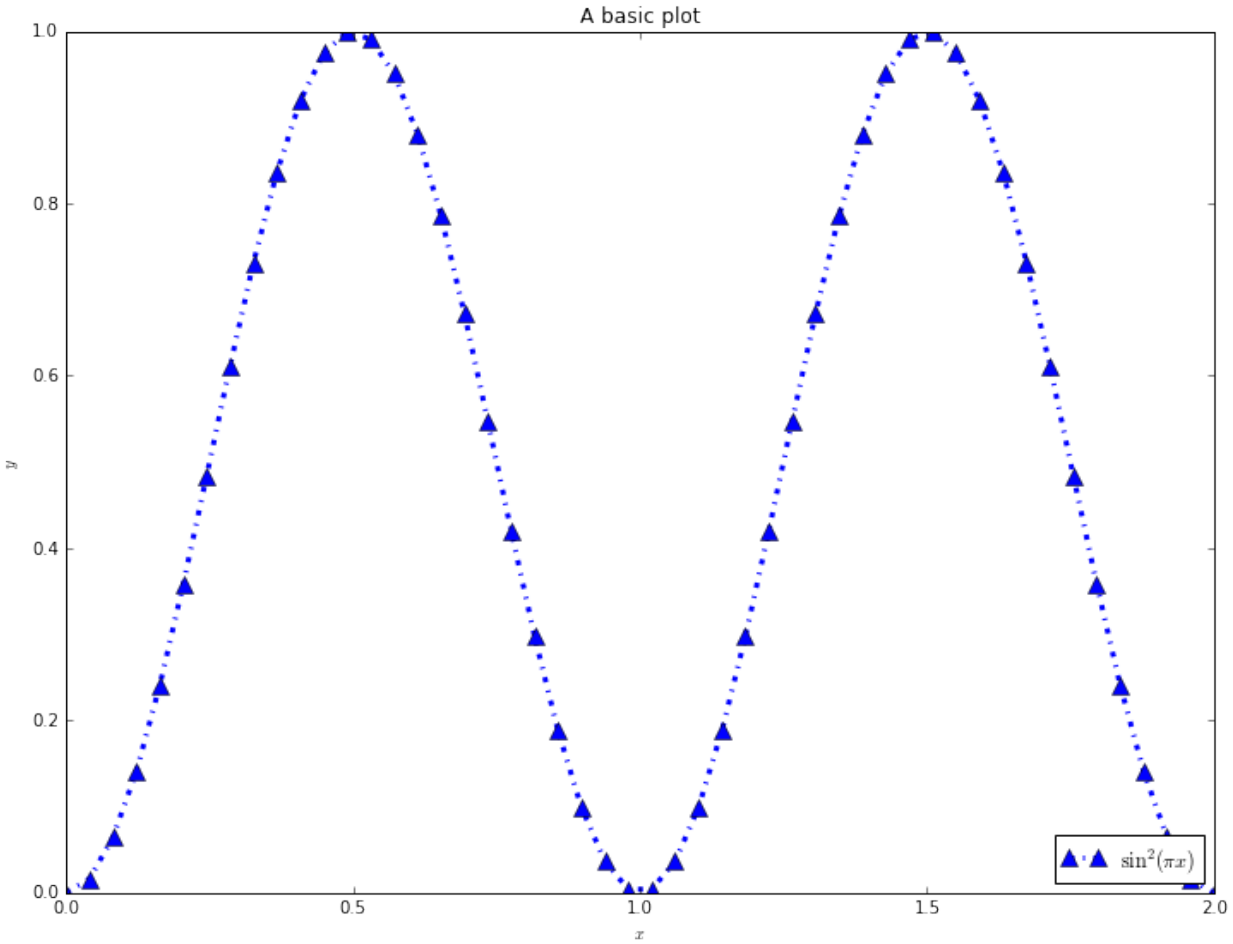
Whilst most of the commands are self-explanatory, a brief note should be made of the strings line `r'$x$'`. These strings are in LaTeX format, which is *the* standard typesetting method for professional-level mathematics. The `$` symbols surround mathematics. The `r` before the definition of the string says that the following string will be "raw": that backslash characters should be left alone. Then, special LaTeX commands have a backslash in front of them: here we use `\pi` and `\sin`. We can also use `^` to denote superscripts (used here), _ to denote subscripts, and use `{}` to group terms.

By combining these basic commands with other plotting types (`semilogx` and `loglog`, for example), most simple plots can be produced quickly.

7.3.1 Saving figures

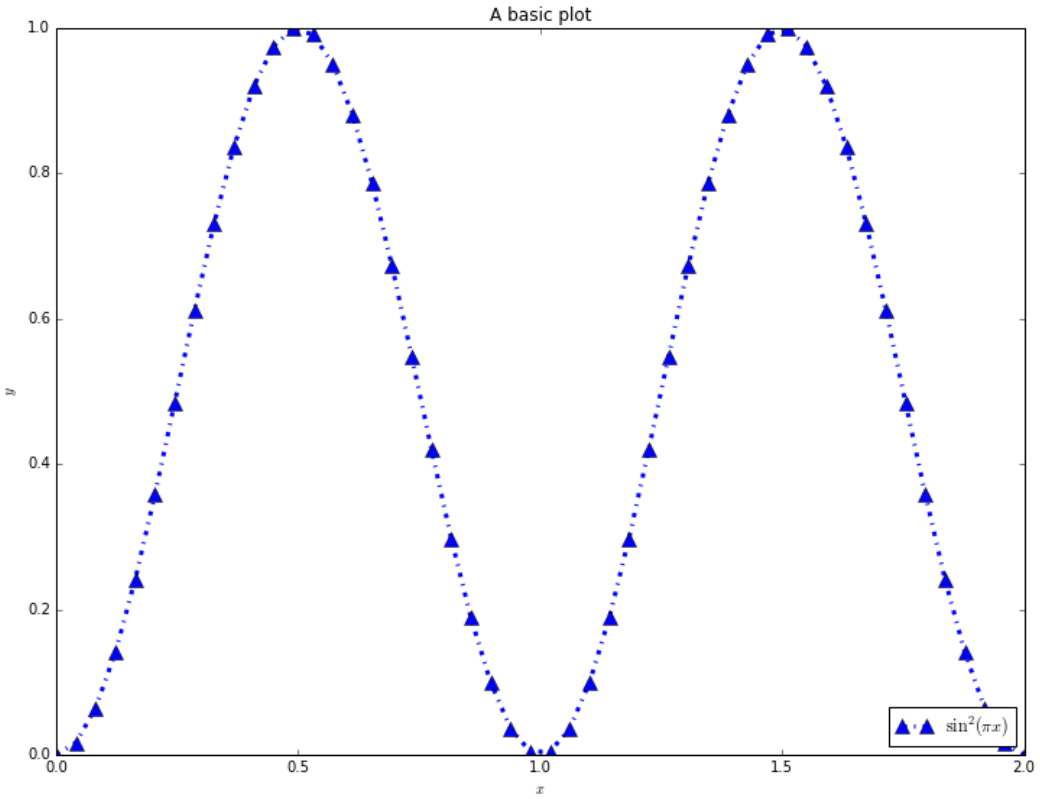
If you want to save the figure to a file, instead of printing it to the screen, use the `savefig <http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.savefig>`__ command instead of the `show` command. For example, try:

```
In [17]: x = numpy.linspace(0, 2.0)
         y = numpy.sin(numpy.pi*x)**2
         pyplot.plot(x, y, marker='^', markersize=10, linestyle='-.', linewidth=3,
                     color='b', label=r'$\sin^2(\pi x)$')
         pyplot.legend(loc='lower right')
         pyplot.xlabel(r'$x$')
         pyplot.ylabel(r'$y$')
         pyplot.title('A basic plot')
         pyplot.savefig('simple_plot.png')
```



We can then check the file on disk (you should open the file on your machine to check):

```
In [18]: from IPython.display import Image
         Image('simple_plot.png')
```



The type of the file is taken from the extension. Here we have used a `png` file, but `svg` and `pdf` output will also work.

7.3.2 Object-based approach

To get a more detailed control over the plot it's better to look at the objects that `matplotlib` is producing. Remember, when we talked about classes we said that it is an object with attributes and methods (functions) that are accessed using dot notation. Here are steps to completely control the plot.

First we define a figure object. We do not have to define the figure class - it is defined within `matplotlib` itself, along with a lot of useful methods. We call the constructor of the figure object in the same way as in the previous section, by calling `pyplot.figure()`. We can and will control its size (the units default to inches) by passing additional arguments to the constructor:

```
In [19]: fig = pyplot.figure(figsize=(12, 9))
```

```
<matplotlib.figure.Figure at 0x10b0e5240>
```

We will then define two axes on this figure. The numbers refers to the positions of the edges of the axes with respect to the figure window (between 0 and 1):

```
In [20]: axis1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height
         axis2 = fig.add_axes([0.4, 0.7, 0.2, 0.15])
```

We will then add data to the both axes:

```
In [21]: axis1.plot(x, y)
         axis2.plot(x, y)
```

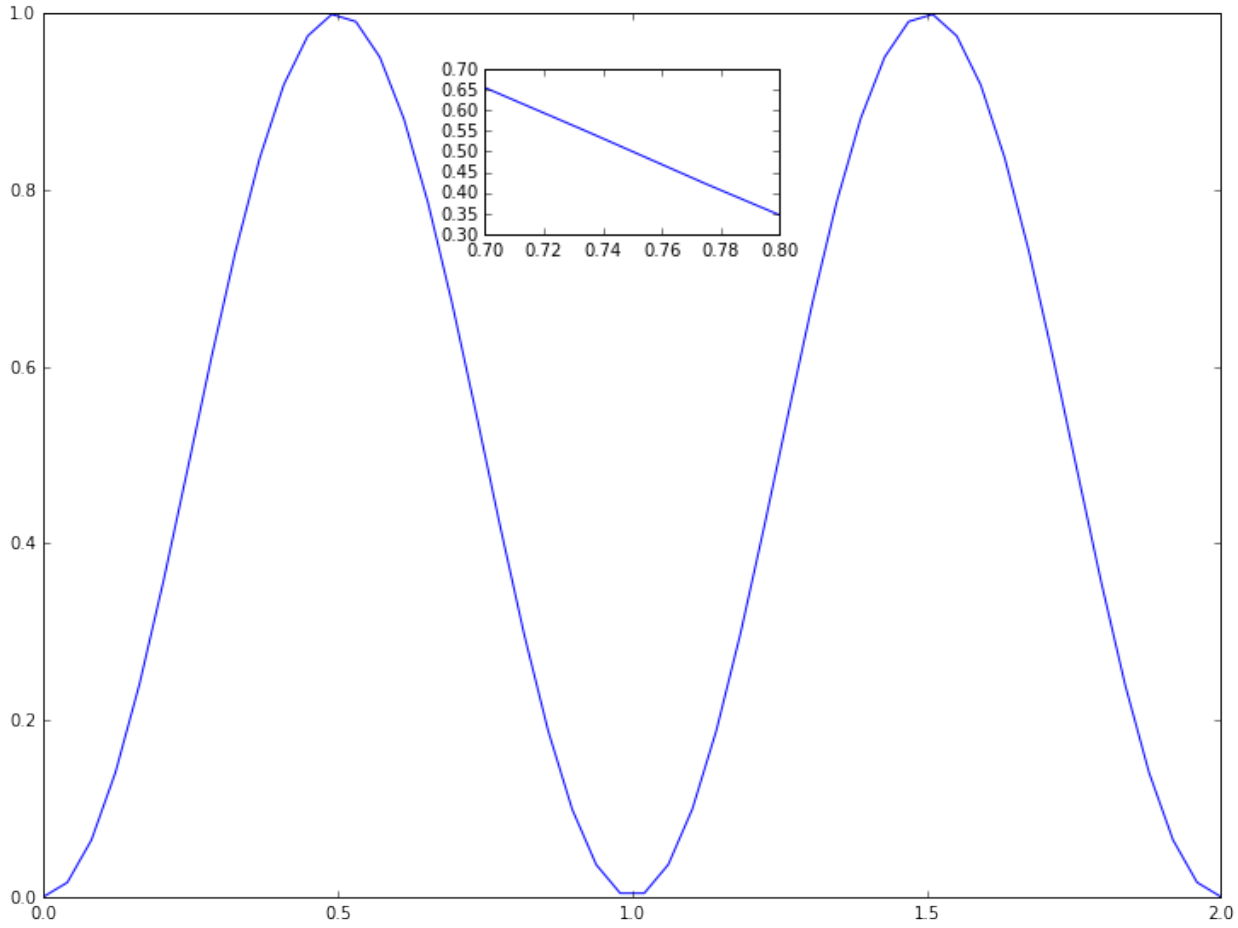
```
Out[21]: [<matplotlib.lines.Line2D at 0x10b0416a0>]
```

We will then set the range of the second axis:

```
In [22]: axis2.set_xbound(0.7, 0.8)
         axis2.set_ybound(0.3, 0.7)
```

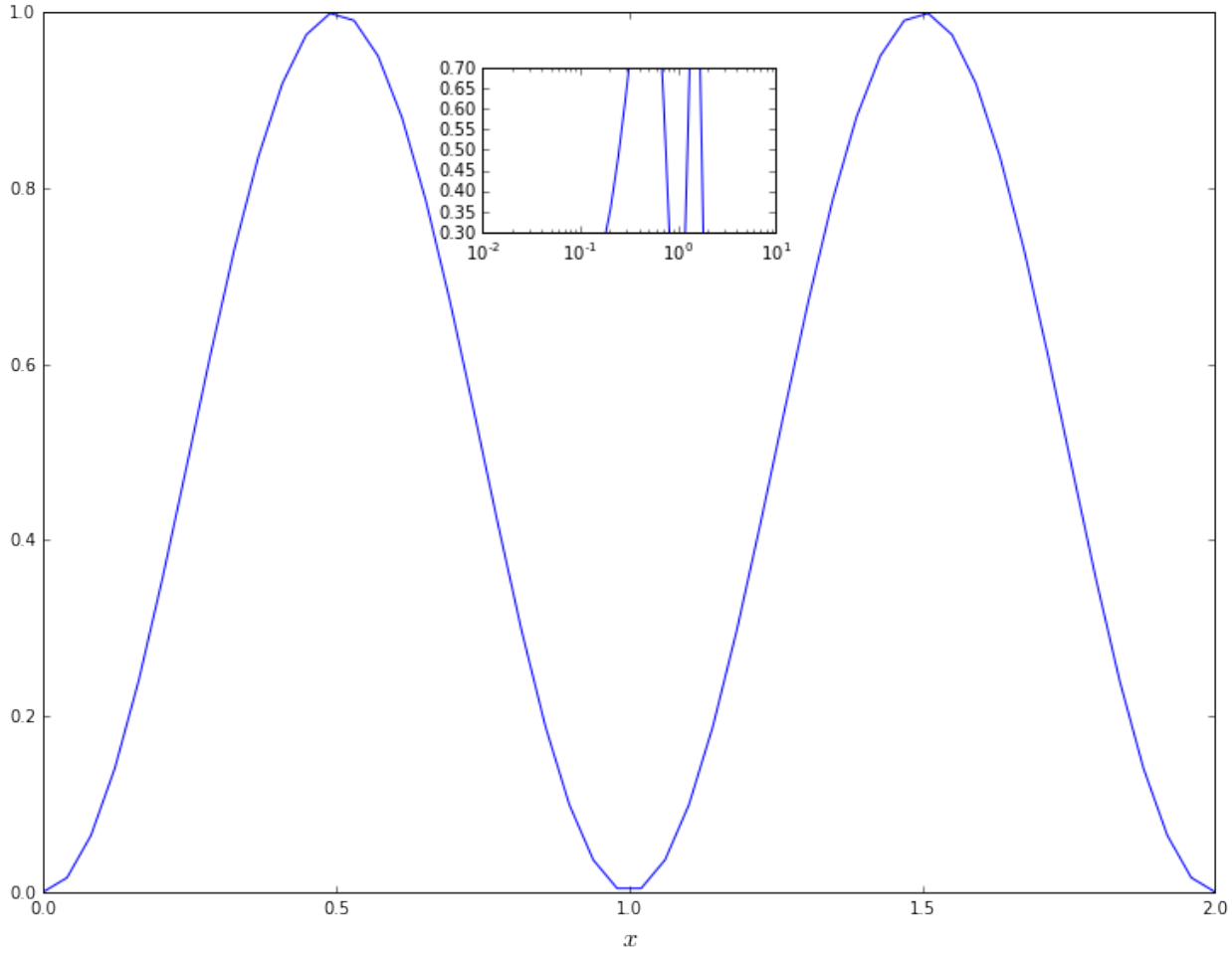
Finally, we'll see what it looks like:

```
In [23]: fig
```

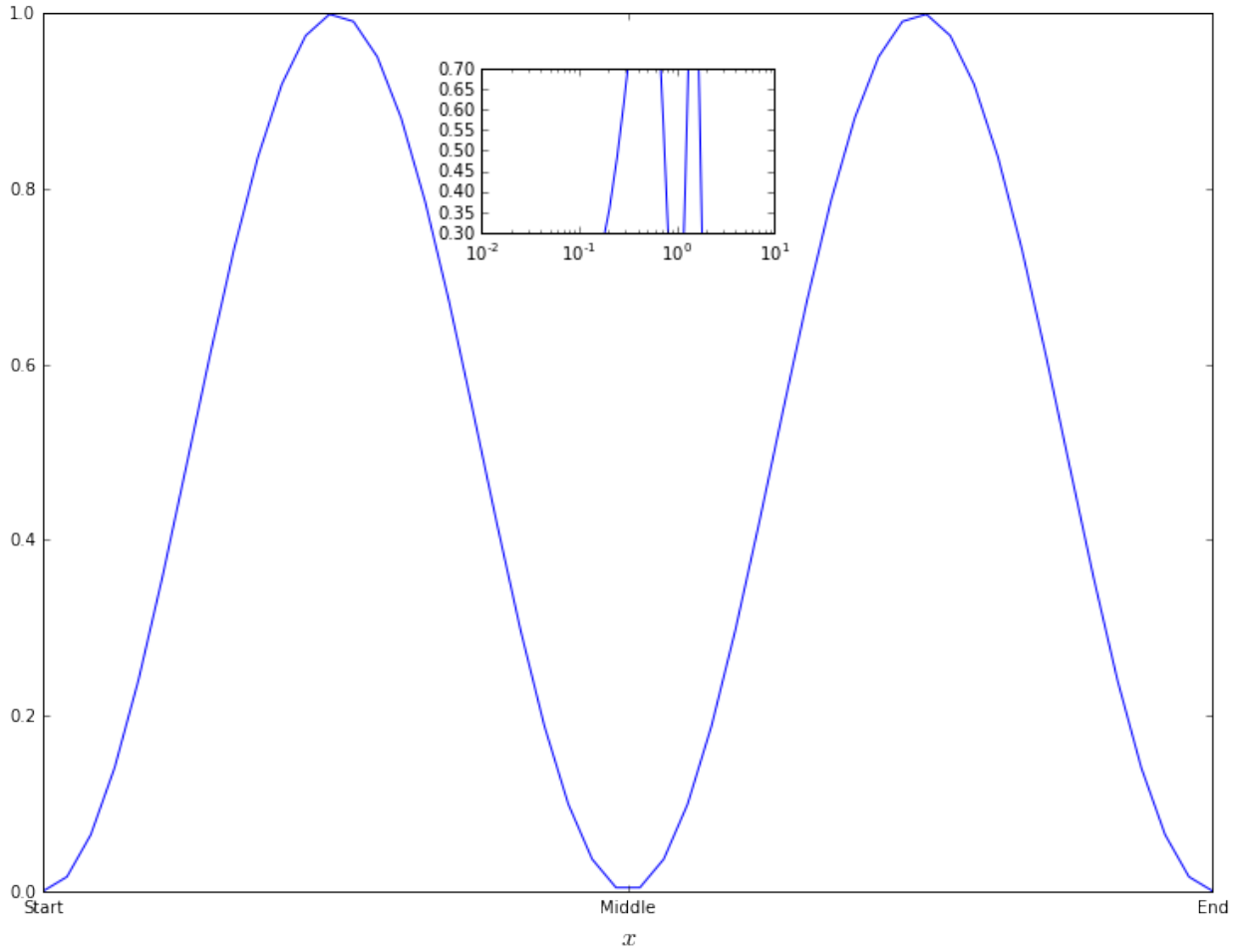


Each axis contains additional objects that can be modified.

```
In [24]: axis2.set_xscale('log')
         axis1.set_xlabel(r'$x$', fontsize=16)
         fig
```

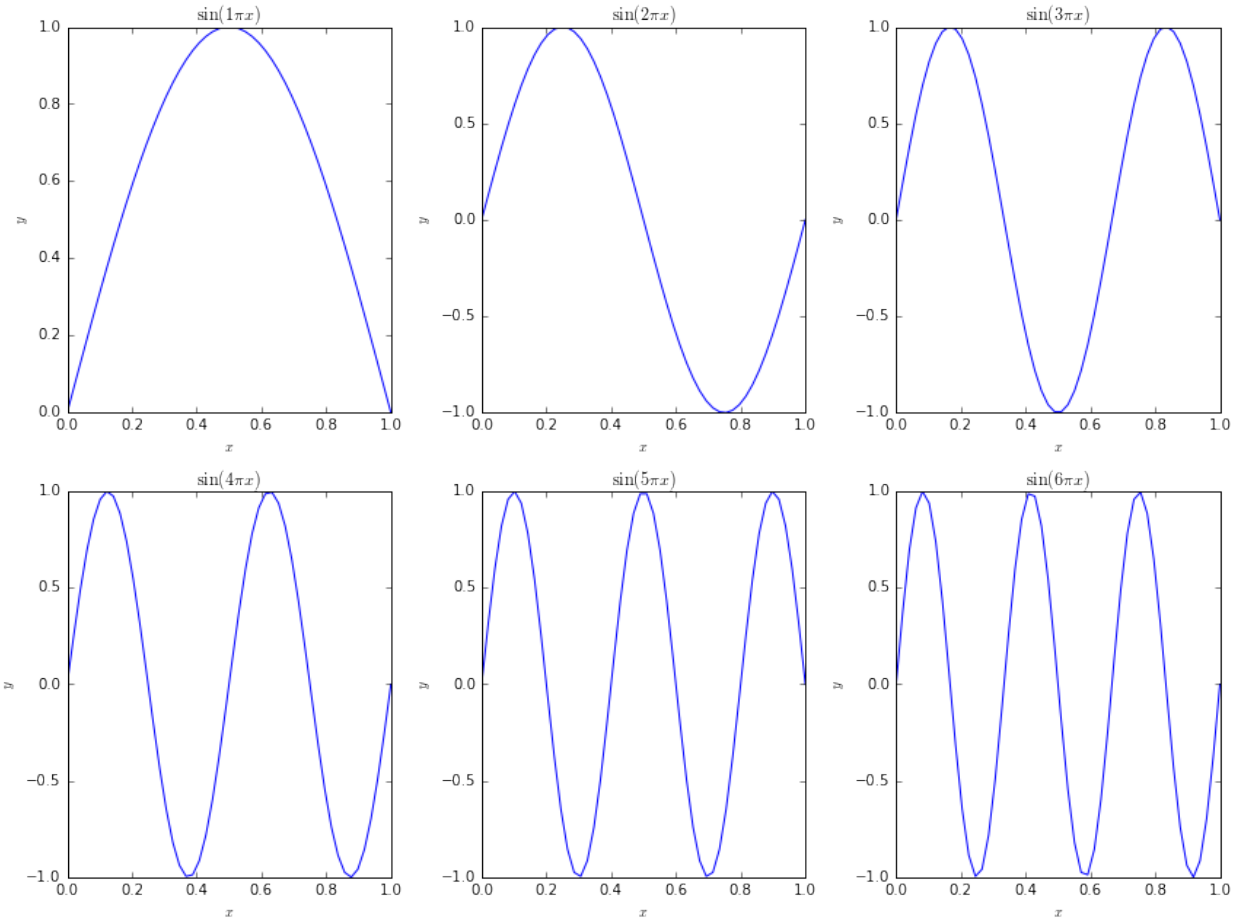



```
In [25]: axis1.set_xticks([0, 1, 2])
         axis1.set_xticklabels(['Start', 'Middle', 'End'])
         fig
```



Adding multiple axes by hand is often annoying (although sometimes necessary). There are a number of tools that can be used to simplify this in standard cases: `add_subplot` is the standard one. When you want a figure containing multiple subplots all the same size, with `r` rows and `c` columns, the command is `add_subplot(r,c,<subplot_number>)`. For example:

```
In [26]: fig = pyplot.figure(figsize=(12, 9))
         x = numpy.linspace(0.0, 1.0)
         for subplot in range(1, 7):
             axis = fig.add_subplot(2, 3, subplot)
             axis.plot(x, numpy.sin(numpy.pi*x*subplot))
             axis.set_xlabel(r'$x$')
             axis.set_ylabel(r'$y$')
             axis.set_title(r'$\sin({} \pi x)$'.format(subplot))
         fig.tight_layout();
```

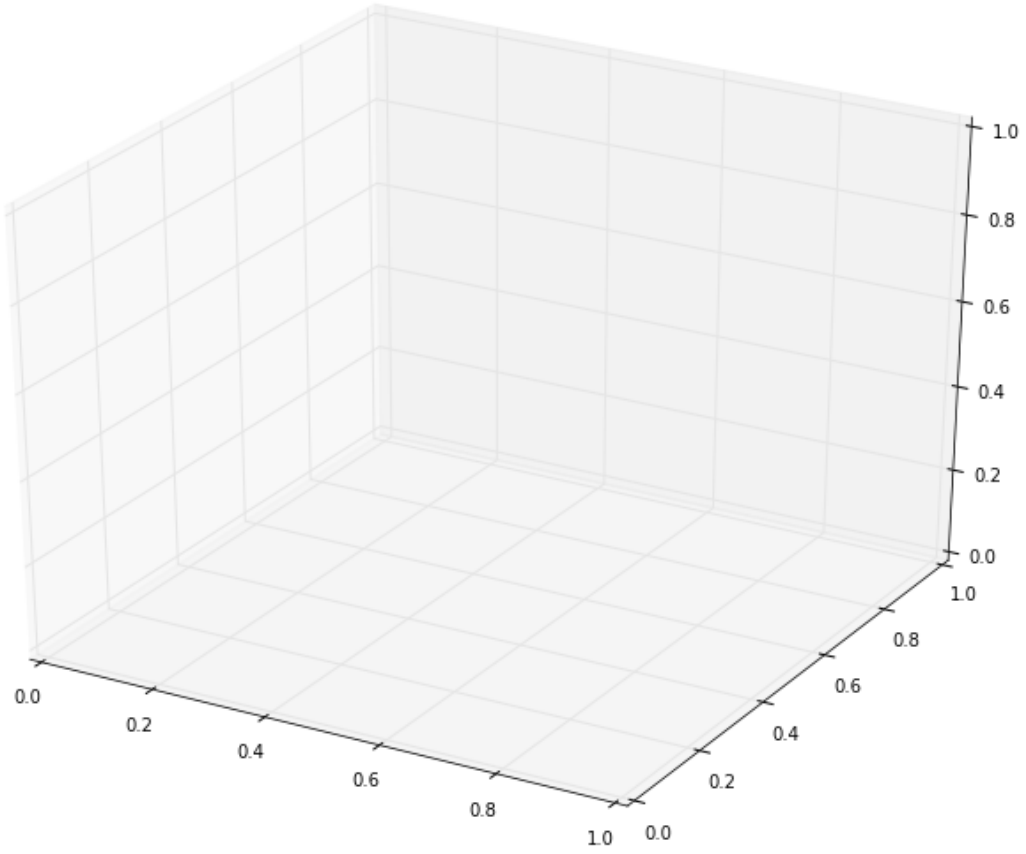


The `tight_layout` function call at the end ensures that the axis labels and titles do not overlap with other subplots.

7.3.3 Higher dimensions

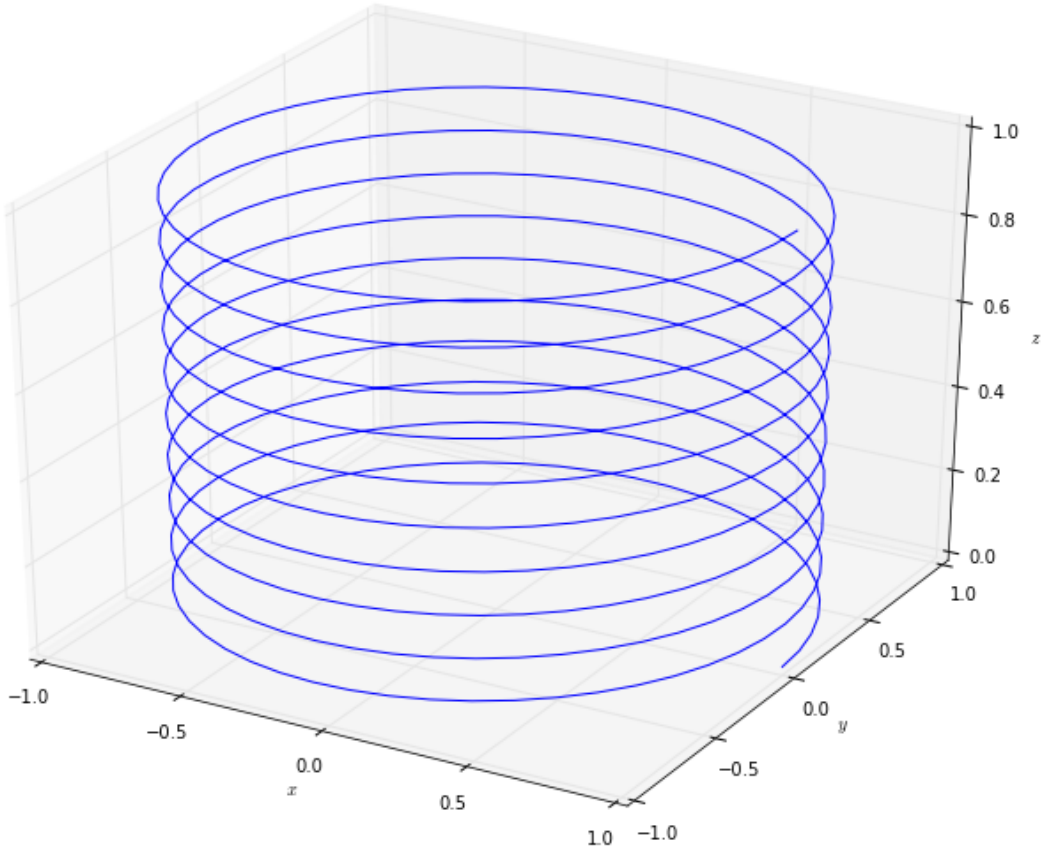
To plot three-dimensional objects, we need to modify the *axis* so that it knows a third dimension is required. To do this, we import another module and modify the command that sets up the axis object.

```
In [27]: from mpl_toolkits.mplot3d.axes3d import Axes3D

         fig = pyplot.figure(figsize=(12, 9))
         axis = fig.add_axes([0.1, 0.1, 0.8, 0.8], projection='3d')
```



We can then construct, for example, a parametric spiral:

```
In [28]: t = numpy.linspace(0.0, 10.0, 500)
         x = numpy.cos(2.0*numpy.pi*t)
         y = numpy.sin(2.0*numpy.pi*t)
         z = 0.1*t

         axis.plot(x, y, z)
         axis.set_xlabel(r'$x$')
         axis.set_ylabel(r'$y$')
         axis.set_zlabel(r'$z$')
         fig
```

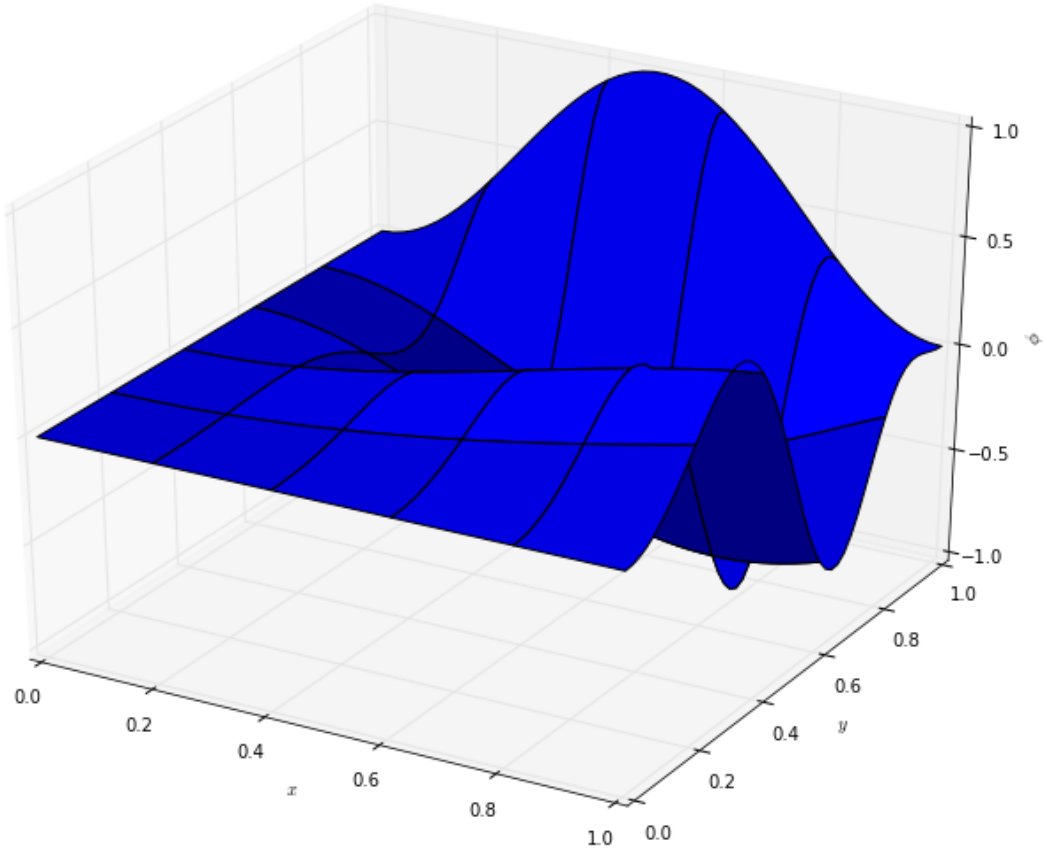


If we want to plot a surface, then we need to construct *2d arrays* containing the locations of the x and y coordinates, and a 2d array containing the "height" of the surface. For structured data (ie, where the x and y coordinates lie on a regular grid) the `meshgrid` function helps. For example, the function

$$\phi(x, y) = \sin^2(\pi x y)\cos(2\pi y^2), \qquad x \in [0, 1], \quad y \in [0, 1],$$

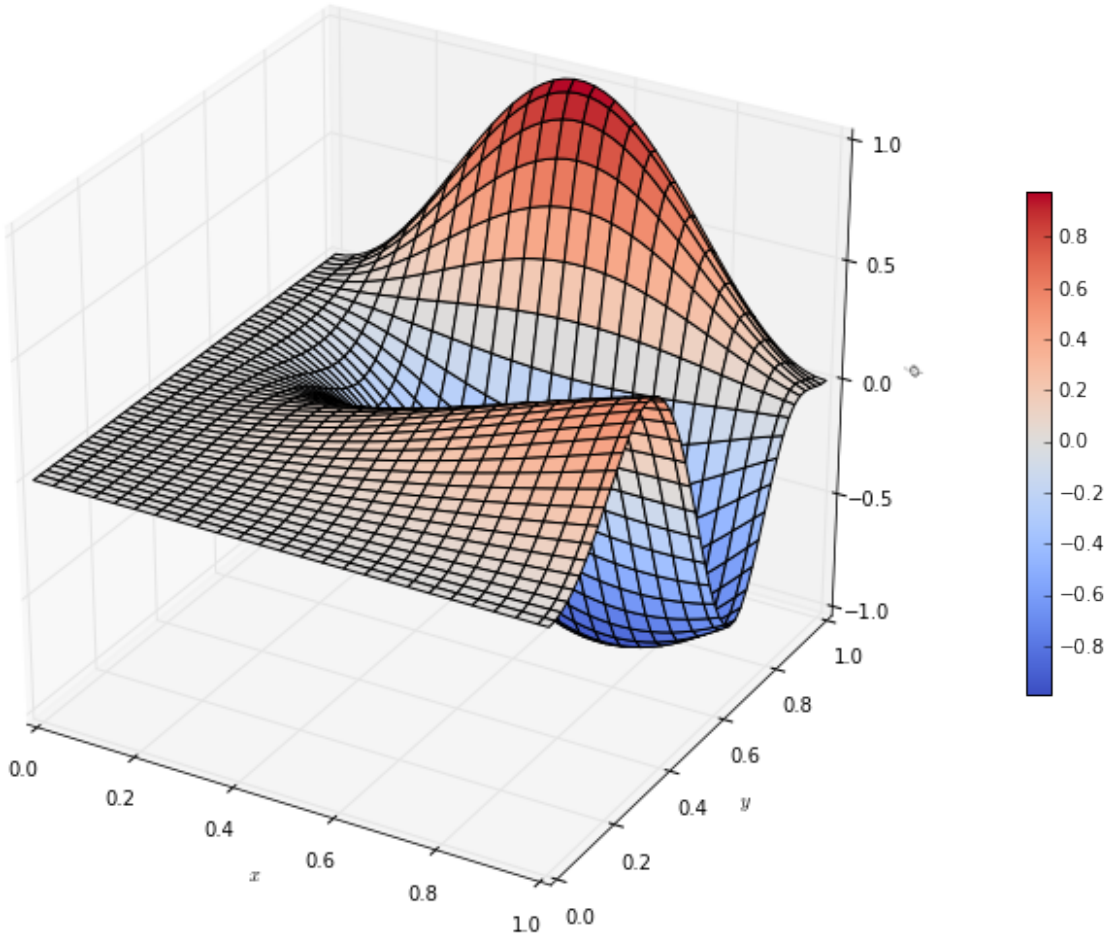
would be plotted using

```
In [29]: fig = pyplot.figure(figsize=(12, 9))
         axis = fig.add_axes([0.1, 0.1, 0.8, 0.8], projection='3d')

         x = numpy.linspace(0.0, 1.0)
         y = numpy.linspace(0.0, 1.0)

         X, Y = numpy.meshgrid(x, y)
         # x, y are vectors
         # X, Y are 2d arrays

         phi = numpy.sin(numpy.pi*X*Y)**2 * numpy.cos(2.0*numpy.pi*Y**2)

         axis.plot_surface(X, Y, phi)
         axis.set_xlabel(r'$x$')
         axis.set_ylabel(r'$y$')
         axis.set_zlabel(r'$\phi$');
```



There are a lot of options to modify the appearance of this plot. Important ones include the colormap (note the US spelling), which requires importing the `cm` module from `matplotlib`, and the `stride` parameters changing the appearance of the grid. For example

```
In [30]: from matplotlib import cm

         fig = pyplot.figure(figsize=(12, 9))
         axis = fig.add_axes([0.1, 0.1, 0.8, 0.8], projection='3d')

         p = axis.plot_surface(X, Y, phi, rstride=1, cstride=2, cmap = cm.coolwarm)
         axis.set_xlabel(r'$x$')
         axis.set_ylabel(r'$y$')
         axis.set_zlabel(r'$\phi$')
         fig.colorbar(p, shrink=0.5);
```



7.3.4 Further reading

As noted earlier, the matplotlib documentation contains a lot of details, and the gallery contains a lot of examples that can be adapted to fit. There is also an extremely useful document as part of Johansson's lectures on scientific Python, and an introduction by Nicolas Rougier.

7.4 scipy

`scipy` is a package for scientific Python, and contains many functions that are essential for mathematics. It works particularly well with `numpy`. We briefly introduced it above for tackling Linear Algebra problems, but it also includes

- Scientific constants
- Integration and ODE solvers
- Interpolation
- Optimization and root finding
- Statistical functions

and much more.

7.4.1 Integration

The *numerical quadrature* problem involves solving the *definite* integral

$$\int_a^b f(x)\,\mathrm{d}x,$$

or a suitable generalization. `scipy` has a module, `scipy.integrate`, that includes a number of functions to solve these types of problems. For example, to solve

$$I = \int_0^\pi \sin^2(x)\,\mathrm{d}x,$$

the `quad` function can be used as:

```
In [31]: from numpy import sin
         from scipy.integrate import quad

         def integrand(x):
             """
             The integrand \sin^2(x).

             Parameters
             ----------

             x : real (list)
                 The point(s) at which the integrand is evaluated

             Returns
             -------

             integrand : real (list)
                 The integrand evaluated at x
             """

             return sin(x)**2

         result = quad(integrand, 0.0, numpy.pi)
         print("The result is {}.".format(result))
```

```
The result is (1.5707963267948966, 1.743934249004316e-14).
```

The steps we have taken are:

1. Define the integrand by defining a function. This function takes the points at which the integrand is evaluated. By using `numpy` we can do this with a single command.
2. Import the `quad` function.
3. Call the `quad` function, passing the function defining the integrand, and the lower and upper limits.

The result we get back, as seen from the screen output, is *not just* I. It is a tuple containing both I, and also the *accuracy* with which `quad` believes it has computed the result. The quadrature is a *numerical approximation*, so can never be perfect. You should check this error estimate to ensure the result is "good enough" for your purposes.

We can also pass additional parameters if needed. Consider the problem

$$I_a = \int_0^\pi \sin^2(ax)\,\mathrm{d}x.$$

If we wanted to solve this for many values of a, say $a = 1, 2, \ldots, 5$, we could create a function taking a parameter, and then pass that parameter through:

```
In [32]: from numpy import sin
         from scipy.integrate import quad

         def integrand_param(x, a):
             """
             The integrand \sin^2(a x).

             Parameters
             ----------

             x : real (list)
                 The point(s) at which the integrand is evaluated
             a : real
                 The parameter for the integrand

             Returns
             -------

             integrand : real (list)
                 The integrand evaluated at x
             """

             return sin(a*x)**2

         for a in range(1, 6):
             result, accuracy = quad(integrand_param, 0.0, numpy.pi, args=(a,))
             print("For a={}, the result is {}.".format(a, result))
```

```
For a=1, the result is 1.5707963267948966.
For a=2, the result is 1.5707963267948966.
For a=3, the result is 1.5707963267948966.
For a=4, the result is 1.5707963267948966.
For a=5, the result is 1.5707963267948968.
```

Note that when passing the parameters using the `args` keyword argument, we put the parameters in a tuple. This shows how to pass more than one parameter: keep adding parameters to the argument list, and add them to the tuple. For example, to solve

$$I_{a,b} = \int_0^\pi \sin^2(ax + b)\,\mathrm{d}x$$

we write

```
In [33]: from numpy import sin
         from scipy.integrate import quad

         def integrand_param2(x, a, b):
             """
             The integrand \sin^2(a x + b).
```

```
        Parameters
        ----------

        x : real (list)
            The point(s) at which the integrand is evaluated
        a : real
            The parameter for the integrand
        b : real
            The second parameter for the integrand

        Returns
        -------

        integrand : real (list)
            The integrand evaluated at x
        """

        return sin(a*x+b)**2

for a in range(1, 3):
    for b in range(3):
        result, accuracy = quad(integrand_param2, 0.0, numpy.pi, args=(a, b))
        print("For a={}, b={}, the result is {}.".format(a, b, result))
```

```
For a=1, b=0, the result is 1.5707963267948966.
For a=1, b=1, the result is 1.570796326794897.
For a=1, b=2, the result is 1.570796326794897.
For a=2, b=0, the result is 1.5707963267948966.
For a=2, b=1, the result is 1.5707963267948961.
For a=2, b=2, the result is 1.5707963267948961.
```

7.4.2 Solving ODEs

There is a link between the solution of integrals and the solution of differential equations. Unfortunately, the numerical solution of an ODE is more complex than the solution of an integral. Fortunately, `scipy` contains a number of methods for these as well.

The methods in `scipy` solve ODEs of the form

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{f}\left(\vec{y}, t\right), \qquad \vec{y}(0) = \vec{y}_0.$$

For example, the ODE

$$\frac{\mathrm{d}y}{\mathrm{d}t} = e^{-t} - y, \qquad y(0) = 1$$

has $f(y, t) = e^{-t} - y$.

The method for using `scipy` is similar to the integration case.

1. Define a function that specifies the system, by defining the RHS.
2. Import the function that solves ODEs (`odeint`)
3. Call the function, passing the RHS function, the initial data $\vec{y}_0$, the times at which the solution is needed, and any parameters.

To solve our example, we use:

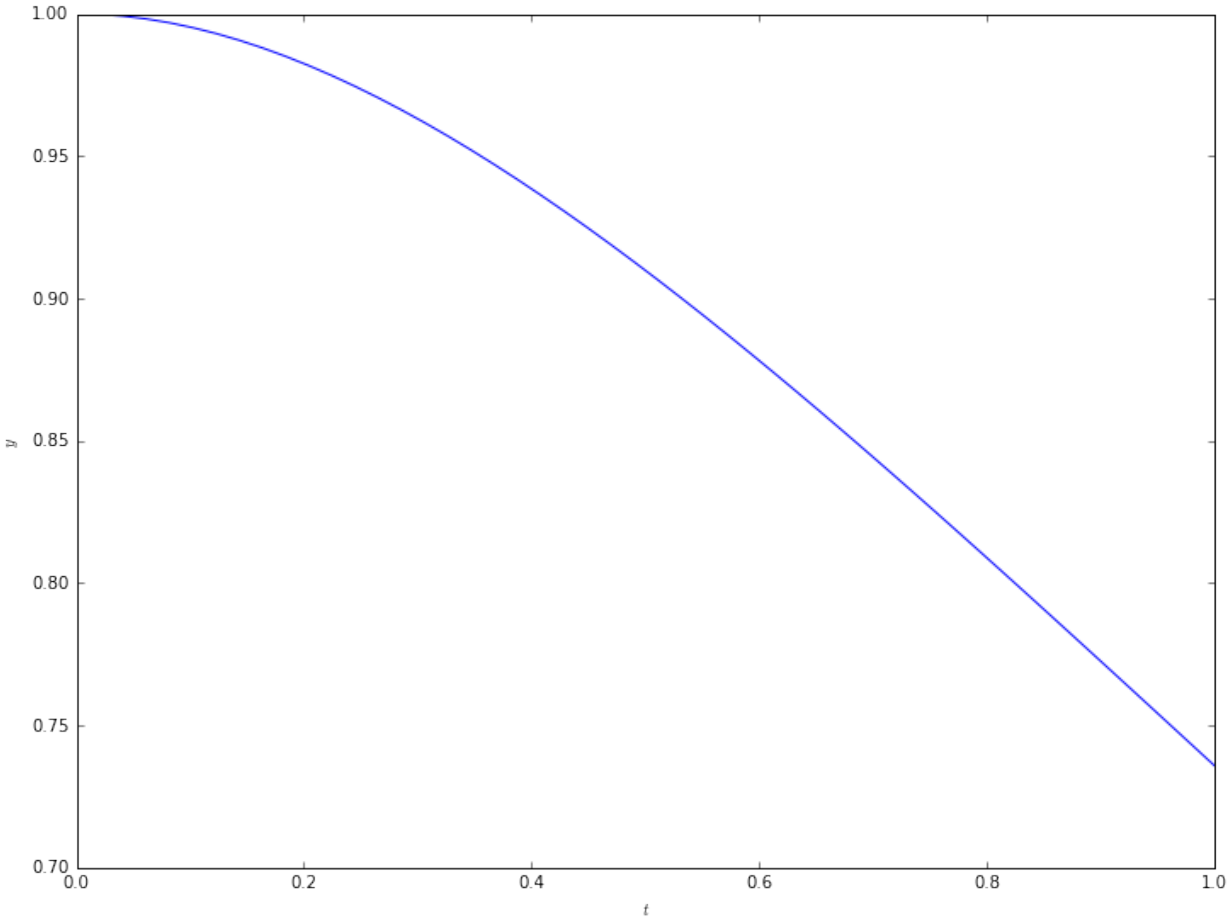
```
In [34]: from numpy import exp
         from scipy.integrate import odeint

         def dydt(y, t):
             """
             Defining the ODE dy/dt = e^{-t} - y.

             Parameters
             ----------

             y : real
                 The value of y at time t (the current numerical approximation)
             t : real
                 The current time t

             Returns
             -------

             dydt : real
                 The RHS function defining the ODE.
             """

             return exp(-t) - y

         t = numpy.linspace(0.0, 1.0)
         y0 = [1.0]

         y = odeint(dydt, y0, t)
         print("The shape of the result is {}.".format(y.shape))
         print("The value of y at t=1 is {}.".format(y[-1,0]))
```

```
The shape of the result is (50, 1).
The value of y at t=1 is 0.7357588629292717.
```

Note that the result for y is not a vector, but a two dimensional array. This is because `scipy` will solve a general *system* of ODEs. This scalar case is a system of size 1, but it still returns an array. To solve a system, the RHS function must take a vector for `y`, return a vector for `dydt`, and the initial data `y0` must be a vector. All these vectors must be the same size.

The output is the numerical approximation to y at the input times t, and can be immediately plotted:

```
In [35]: pyplot.plot(t, y[:,0])
         pyplot.xlabel(r'$t$')
         pyplot.ylabel(r'$y$')
         pyplot.show()
```



Passing parameters is also similar to the integration case. For example, consider the problem

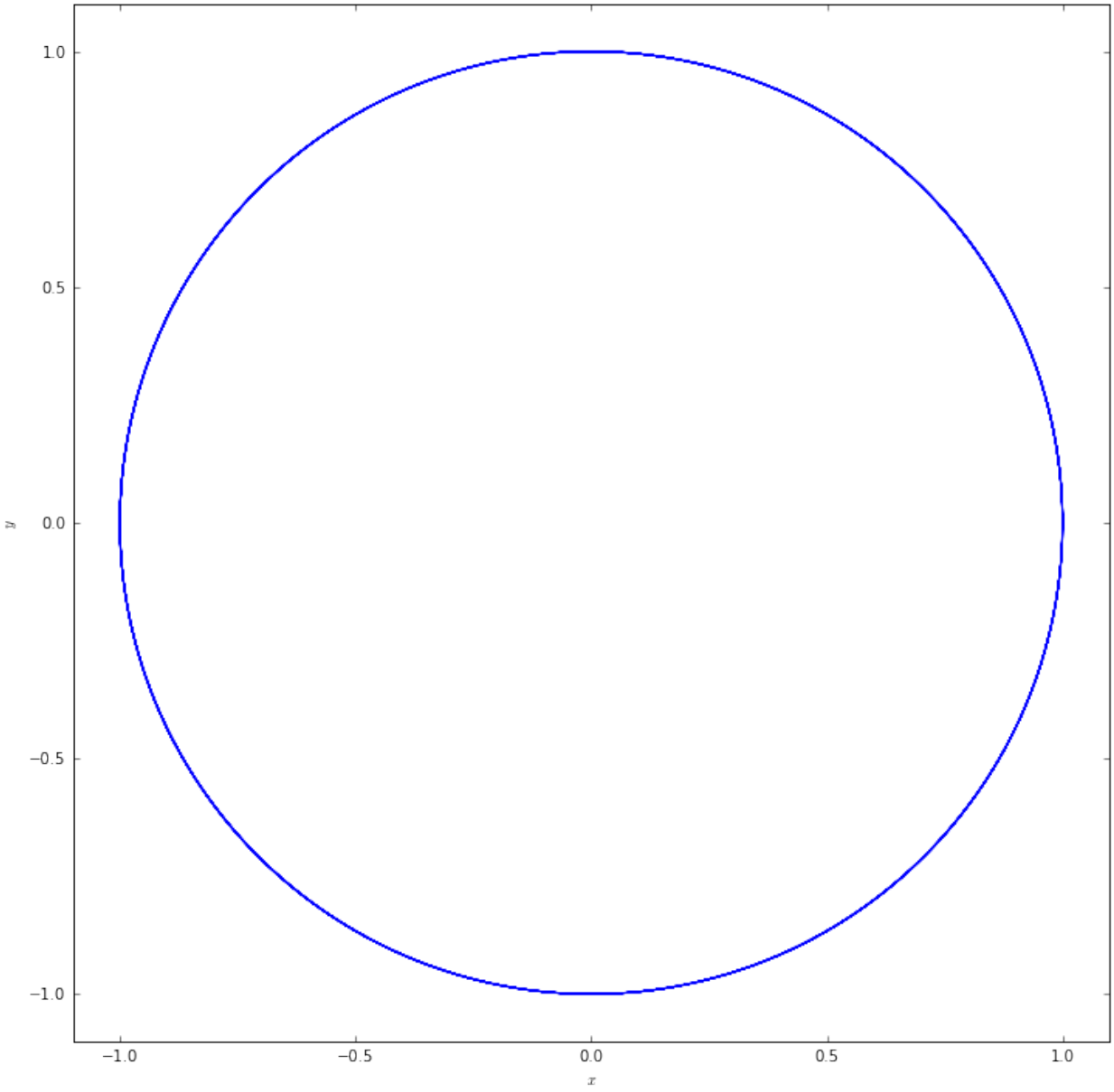
$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -y + \alpha \\ x \end{pmatrix}, \qquad \begin{pmatrix} x \\ y \end{pmatrix}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

If α is zero, the solution is a circle in the x, y plane. We solve this using `odeint`, denoting the state vector $\vec{z} = (x, y)^T$:

```
In [36]: import numpy
         from scipy.integrate import odeint

         def dzdt(z, t, alpha):
             """
             Defining the ODE dz/dt.

             Parameters
             ----------

             z : real, list
                 The value of z at time t (the current numerical approximation)
             t : real
                 The current time t
             alpha : real
                 Parameter

             Returns
```

```
        -------

        dzdt : real
            The RHS function defining the ODE.
        """

        dzdt = numpy.zeros_like(z)
        x, y = z
        dzdt[0] = -y + alpha
        dzdt[1] = x

        return dzdt

    t = numpy.linspace(0.0, 50.0, 1000)
    z0 = [1.0, 0.0]
    alpha = 1e-5

    z = odeint(dzdt, z0, t, args=(alpha,))
```

In [37]:
```
fig = pyplot.figure(figsize=(12,12))
ax = fig.add_subplot(1,1,1)
ax.plot(z[:,0], z[:,1])
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_xlim(-1.1, 1.1)
ax.set_ylim(-1.1, 1.1);
```



7.4.3 Further reading

Earlier we introduced `scipy` for Linear Algebra, and gave links there. Most of those links cover the full `scipy` package. The scipy documentation is comprehensive. Johansson also has a tutorial on scipy.

7.5 Exercise: Lorenz attractor

The Lorenz system is a set of ordinary differential equations which can be written

$$\frac{\mathrm{d}\vec{v}}{\mathrm{d}\vec{t}} = \vec{f}(\vec{v})$$

where the variables in the state vector $\vec{v}$ are

$$\vec{v} = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

and the function defining the ODE is

$$\vec{f} = \begin{pmatrix} \sigma\left(y(t) - x(t)\right) \\ x(t)\left(\rho - z(t)\right) - y(t) \\ x(t)y(t) - \beta z(t) \end{pmatrix}.$$

The parameters σ, ρ, β are all real numbers.

7.5.1 Exercise 1

Write a function `dvdt(v,t,params)` that returns $\vec{f}$ given $\vec{v}, t$ and the parameters σ, ρ, β.

7.5.2 Exercise 2

Fix $\sigma = 10, \beta = 8/3$. Set initial data to be $\vec{v}(0) = \vec{1}$. Using `scipy`, specifically the `odeint` function of `scipy.integrate`, solve the Lorenz system up to $t = 100$ for $\rho = 13, 14, 15$ and 28.

Plot your results in 3d, plotting x, y, z.

7.5.3 Exercise 3

Fix $\rho = 28$. Solve the Lorenz system twice, up to $t = 40$, using the two different initial conditions $\vec{v}(0) = \vec{1}$ and $\vec{v}(0) = \vec{1} + 10^{-5}$.

Show four plots. Each plot should show the two solutions on the same axes, plotting x, y and z. Each plot should show 10 units of time, ie the first shows $t \in [0, 10]$, the second shows $t \in [10, 20]$, and so on.

This shows the *sensitive dependence on initial conditions* that is characteristic of chaotic behaviour.

7.6 Exercise: Mandelbrot

The Mandelbrot set is also generated from a sequence, $\{z_n\}$, using the relation

$$z_{n+1} = z_n^2 + c, \qquad z_0 = 0.$$

The members of the sequence, and the constant c, are all complex. The point in the complex plane at c is in the Mandelbrot set only if the $|z_n| < 2$ for all members of the sequence. In reality, checking the first 100 iterations is sufficient.

Note: the Python notation for a complex number $x + \mathrm{i}y$ is `x + yj`: that is, `j` is used to indicate $\sqrt{-1}$. If you know the values of `x` and `y` then `x + yj` constructs a complex number; if they are stored in variables you can use `complex(x,y)`.

7.6.1 Exercise 1

Write a function that checks if the point c is in the Mandelbrot set.

7.6.2 Exercise 2

Check the points $c = 0$ and $c = \pm 2 \pm 2\mathrm{i}$ and ensure they do what you expect. (What *should* you expect?)

7.6.3 Exercise 3

Write a function that, given N

1. generates an $N \times N$ grid spanning $c = x + \mathrm{i}y$, for $-2 \leq x \leq 2$ and $-2 \leq y \leq 2$;
2. returns an $N \times N$ array containing one if the associated grid point is in the Mandelbrot set, and zero otherwise.

7.6.4 Exercise 4

Using the function `imshow` from `matplotlib`, plot the resulting array for a 100×100 array to make sure you see the expected shape.

7.6.5 Exercise 5

Modify your functions so that, instead of returning whether a point is inside the set or not, it returns the logarithm of the number of iterations it takes. Plot the result using `imshow` again.

7.6.6 Exercise 6

Try some higher resolution plots, and try plotting only a section to see the structure. **Note** this is not a good way to get high accuracy close up images!

7.7 Exercise: The shortest published Mathematical paper

A candidate for the shortest mathematical paper ever shows the following result:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5.$$

This is interesting as

> This is a counterexample to a conjecture by Euler ... that at least n nth powers are required to sum to an nth power, $n > 2$.

7.7.1 Exercise 1

Using Python, check the equation above is true.

7.7.2 Exercise 2

The more interesting statement in the paper is that

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5.$$

is

the smallest instance in which four fifth powers sum to a fifth power.

Interpreting "the smallest instance" to mean the solution where the right hand side term (the largest integer) is the smallest, we want to use Python to check this statement.

We are going to need to generate all possible combinations of four integers a, b, c, d and test if $a^5 + b^5 + c^5 + d^5$ matches e^5 where e is another integer.

The problem is the number of combinations grows very fast - the standard formula says that for a list of length ℓ there are

$$\binom{\ell}{k} = \frac{\ell!}{k!(\ell - k)!}$$

combinations of length k. For $k = 4$ as needed here we will have $\ell(\ell - 1)(\ell - 2)(\ell - 3)/24$ combinations.

Show, by getting Python to compute the number of combinations $N = \binom{\ell}{4}$ that N grows roughly as ℓ^4. To do this, plot the number of combinations and ℓ^4 on a log-log scale. Restrict to $\ell \leq 50$.

You may find the `combinations` function from the `itertools` package useful.

7.7.3 Exercise 3

With 17 million combinations to work with, we'll need to be a little careful how we compute. To check the interesting statement in the paper,

1. Construct a `numpy` array containing all integers in $1, \ldots, 144$ to the fifth power.
2. Construct a list of all combinations of four elements from this array.
3. Construct a list of sums of all these combinations.
4. Loop over one list and check if the entry appears in the other list (ie, use the `in` keyword).

By printing out any entries that pass this check, you should see only the solution given in the paper.

CHAPTER 8

Symbolic Python

8.1 Symbolic Python

In standard mathematics we routinely write down abstract variables or concepts and manipulate them without ever assigning specific values to them. An example would be the quadratic equation

$$ax^2 + bx + c = 0$$

and its roots $x_\pm$: we can write down the solutions of the equation and discuss the existence, within the real numbers, of the roots, without specifying the particular values of the parameters a, b and c.

In a standard computer programming language, we can write *functions* that encapsulate the solutions of the equation, but calling those functions requires us to specify values of the parameters. In general, the value of a variable must be given before the variable can be used.

However, there *do* exist *Computer Algebra Systems* that can perform manipulations in the "standard" mathematical form. Through the university you will have access to Wolfram Mathematica and Maple, which are commercial packages providing a huge range of mathematical tools. There are also freely available packages, such as SageMath and `sympy`. These are not always easy to use, as all CAS have their own formal languages that rarely perfectly match your expectations.

Here we will briefly look at `sympy`, which is a pure Python CAS. `sympy` is not suitable for complex calculations, as it's far slower than the alternatives. However, it does interface very cleanly with Python, so can be used inside Python code, especially to avoid entering lengthy expressions.

8.2 sympy

8.2.1 Setting up

Setting up `sympy` is straightforward:

```
In [1]: import sympy
        sympy.init_printing()
```

The standard `import` command is used. The `init_printing` command looks at your system to find the clearest way of displaying the output; this isn't necessary, but is helpful for understanding the results.

To do *anything* in `sympy` we have to explicitly tell it if something is a variable, and what name it has. There are two commands that do this. To declare a single variable, use

```
In [2]: x = sympy.Symbol('x')
```

To declare multiple variables at once, use

```
In [3]: y, z0 = sympy.symbols(('y', 'z_0'))
```

Note that the "name" of the variable does not need to match the symbol with which it is displayed. We have used this with `z0` above:

```
In [4]: z0
```

$$z_0$$

Once we have variables, we can define new variables by operating on old ones:

```
In [5]: a = x + y
        b = y * z0
        print("a={}. b={}.".format(a, b))
```

```
a=x + y. b=y*z_0.
```

```
In [6]: a
```

$$x + y$$

In addition to variables, we can also define general functions. There is only one option for this:

```
In [7]: f = sympy.Function('f')
```

8.2.2 In-built functions

We have seen already that mathematical functions can be found in different packages. For example, the sin function appears in `math` as `math.sin`, acting on a single number. It also appears in `numpy` as `numpy.sin`, where it can act on vectors and arrays in one go. `sympy` re-implements many mathematical functions, for example as `sympy.sin`, which can act on abstract (`sympy`) variables.

Whenever using `sympy` we should use `sympy` functions, as these can be manipulated and simplified. For example:

```
In [8]: c = sympy.sin(x)**2 + sympy.cos(x)**2
```

```
In [9]: c
```

$$\sin^2(x) + \cos^2(x)$$

```
In [10]: c.simplify()
```

$$1$$

Note the steps taken here. `c` is an object, something that `sympy` has created. Once created it can be manipulated and simplified, using the methods on the object. It is useful to use tab completion to look at the available commands. For example,

```
In [11]: d = sympy.cosh(x)**2 - sympy.sinh(x)**2
```

Now type `d.` and then tab, to inspect all the available methods. As before, we could do

```
In [12]: d.simplify()
```

$$1$$

but there are many other options.

8.2.3 Solving equations

Let us go back to our quadratic equation and check the solution. To define an *equation* we use the `sympy.Eq` function:

```
In [13]: a, b, c, x = sympy.symbols(('a', 'b', 'c', 'x'))
         quadratic_equation = sympy.Eq(a*x**2+b*x+c, 0)
         sympy.solve(quadratic_equation)
```

$$\left[\left\{a : -\frac{1}{x^2}\left(bx + c\right)\right\}\right]$$

What happened here? `sympy` is not smart enough to know that we wanted to solve for `x`! Instead, it solved for the first variable it encountered. Let us try again:

```
In [14]: sympy.solve(quadratic_equation, x)
```

$$\left[\frac{1}{2a}\left(-b + \sqrt{-4ac + b^2}\right), \quad -\frac{1}{2a}\left(b + \sqrt{-4ac + b^2}\right)\right]$$

This is our expectation: multiple solutions, returned as a list. We can access and manipulate these results:

```
In [15]: roots = sympy.solve(quadratic_equation, x)
         xplus, xminus = sympy.symbols(('x_{+}', 'x_{-}'))
         xplus = roots[0]
         xminus = roots[1]
```

We can substitute in specific values for the parameters to find solutions:

```
In [16]: xplus_solution = xplus.subs([(a,1), (b,2), (c,3)])
         xplus_solution
```

$$-1 + \sqrt{2}i$$

We have a list of substitutions. Each substitution is given by a tuple, containing the variable to be replaced, and the expression replacing it. We do not have to substitute in numbers, as here, but could use other variables:

```
In [17]: xminus_solution = xminus.subs([(b,a), (c,a+z0)])
         xminus_solution
```

$$-\frac{1}{2a}\left(a + \sqrt{a^2 - 4a\left(a + z_0\right)}\right)$$

```
In [18]: xminus_solution.simplify()
```

$$-\frac{1}{2a}\left(a + \sqrt{-a\left(3a + 4z_0\right)}\right)$$

We can use similar syntax to solve *systems* of equations, such as

$$\begin{aligned} x + 2y &= 0, \\ xy &= z_0. \end{aligned}$$

```
In [19]: eq1 = sympy.Eq(x+2*y, 0)
         eq2 = sympy.Eq(x*y, z0)
         sympy.solve([eq1, eq2], [x, y])
```

$$\left[\left(-\sqrt{2}\sqrt{-z_0}, \quad \frac{\sqrt{2}}{2}\sqrt{-z_0}\right), \quad \left(\sqrt{2}\sqrt{-z_0}, \quad -\frac{\sqrt{2}}{2}\sqrt{-z_0}\right)\right]$$

8.2.4 Differentiation and integration

Differentiation

There is a standard function for differentiation, `diff`:

```
In [20]: expression = x**2*sympy.sin(sympy.log(x))
         sympy.diff(expression, x)
```

$$2x\sin\left(\log\left(x\right)\right) + x\cos\left(\log\left(x\right)\right)$$

A parameter can control how many times to differentiate:

```
In [21]: sympy.diff(expression, x, 3)
```

$$\frac{1}{x}\left(-3\sin\left(\log\left(x\right)\right) + \cos\left(\log\left(x\right)\right)\right)$$

Partial differentiation with respect to multiple variables can also be performed by increasing the number of arguments:

```
In [22]: expression2 = x*sympy.cos(y**2 + x)
         sympy.diff(expression2, x, 2, y, 3)
```

$$4y\left(-2xy^2\sin\left(x+y^2\right) + 3x\cos\left(x+y^2\right) + 4y^2\cos\left(x+y^2\right) + 6\sin\left(x+y^2\right)\right)$$

There is also a function representing an *unevaluated* derivative:

```
In [23]: sympy.Derivative(expression2, x, 2, y, 3)
```

$$\frac{\partial^5}{\partial x^2 \partial y^3}\left(x\cos\left(x+y^2\right)\right)$$

These can be useful for display, building up a calculation in stages, simplification, or when the derivative cannot be evaluated. It can be explicitly evaluated using the `doit` function:

```
In [24]: sympy.Derivative(expression2, x, 2, y, 3).doit()
```

$$4y\left(-2xy^2\sin\left(x+y^2\right) + 3x\cos\left(x+y^2\right) + 4y^2\cos\left(x+y^2\right) + 6\sin\left(x+y^2\right)\right)$$

Integration

Integration uses the `integrate` function. This can calculate either definite or indefinite integrals, but will *not* include the integration constant.

```
In [25]: integrand=sympy.log(x)**2
         sympy.integrate(integrand, x)
```

$$x\log^2\left(x\right) - 2x\log\left(x\right) + 2x$$

```
In [26]: sympy.integrate(integrand, (x, 1, 10))
```

$$-20\log\left(10\right) + 18 + 10\log^2\left(10\right)$$

The definite integral is specified by passing a tuple, with the variable to be integrated (here `x`) and the lower and upper limits (which can be expressions).

Note that `sympy` includes an "infinity" object `oo` (two `o`'s), which can be used in the limits of integration:

```
In [27]: sympy.integrate(sympy.exp(-x), (x, 0, sympy.oo))
```

$$1$$

Multiple integration for higher dimensional integrals can be performed:

```
In [28]: sympy.integrate(sympy.exp(-(x+y))*sympy.cos(x)*sympy.sin(y), x, y)
```

$$-\frac{e^{-x}}{4}e^{-y}\sin(x)\sin(y) - \frac{e^{-x}}{4}e^{-y}\sin(x)\cos(y) + \frac{e^{-x}}{4}e^{-y}\sin(y)\cos(x) + \frac{e^{-x}}{4}e^{-y}\cos(x)\cos(y)$$

```
In [29]: sympy.integrate(sympy.exp(-(x+y))*sympy.cos(x)*sympy.sin(y),
                         (x, 0, sympy.pi), (y, 0, sympy.pi))
```

$$\frac{1}{4e^{2\pi}} + \frac{1}{2e^{\pi}} + \frac{1}{4}$$

Again, there is an unevaluated integral:

```
In [30]: sympy.Integral(integrand, x)
```

$$\int \log^2(x)\,dx$$

```
In [31]: sympy.Integral(integrand, (x, 1, 10))
```

$$\int_1^{10} \log^2(x)\,dx$$

Again, the `doit` method will explicitly evaluate the result where possible.

8.2.5 Differential equations

Defining and solving differential equations uses the pattern from the previous sections. We'll use the same example problem as in the `scipy` case,

$$\frac{\mathrm{d}y}{\mathrm{d}t} = e^{-t} - y, \qquad y(0) = 1.$$

First we define that y is a function, currently unknown, and t is a variable.

```
In [32]: y = sympy.Function('y')
         t = sympy.Symbol('t')
```

`y` is a general function, and can be a function of anything at this point (any number of variables with any name). To use it consistently, we *must* refer to it explicitly as a function of t everywhere. For example,

```
In [33]: y(t)
```

$$y(t)$$

We then define the differential equation. `sympy.Eq` defines the equation, and `diff` differentiates:

```
In [34]: ode = sympy.Eq(y(t).diff(t), sympy.exp(-t) - y(t))
         ode
```

$$\frac{d}{dt}y(t) = -y(t) + e^{-t}$$

Here we have used `diff` as a method applied to the function. As `sympy` can't differentiate $y(t)$ (as it doesn't have an explicit value), it leaves it unevaluated.

We can now use the `dsolve` function to get the solution to the ODE. The syntax is very similar to the `solve` function used above:

```
In [35]: sympy.dsolve(ode, y(t))
```

$$y(t) = (C_1 + t)\,e^{-t}$$

This is simple enough to solve, but we'll use symbolic methods to find the constant, by setting $t = 0$ and $y(t) = y(0) = 1$.

```
In [36]: general_solution = sympy.dsolve(ode, y(t))
         value = general_solution.subs([(t,0), (y(0), 1)])
         value
```

$$1 = C_1$$

We then find the specific solution of the ODE.

```
In [37]: ode_solution = general_solution.subs([(value.rhs,value.lhs)])
         ode_solution
```

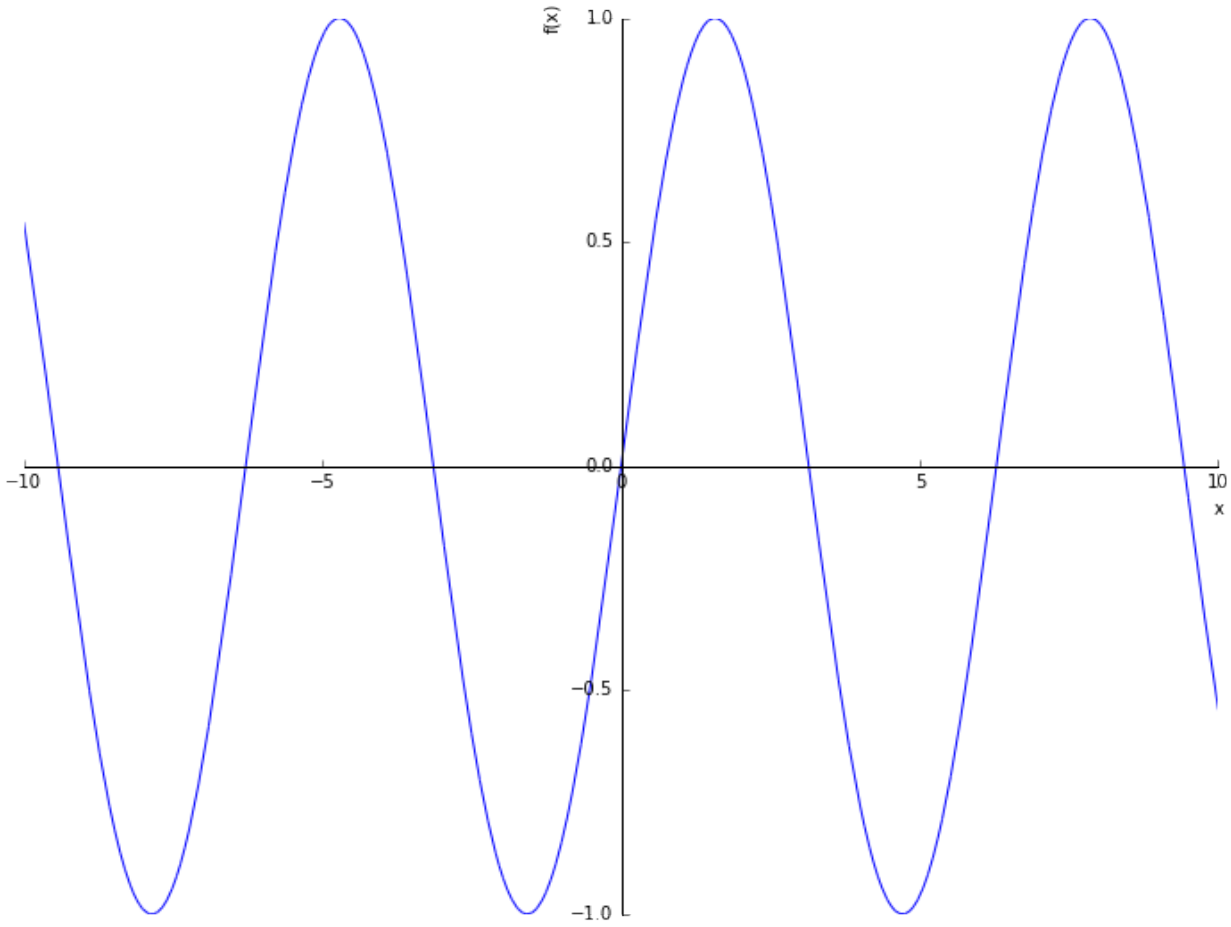
$$y(t) = (t+1)\,e^{-t}$$

8.2.6 Plotting

`sympy` provides an interface to `matplotlib` so that expressions can be directly plotted. For example,

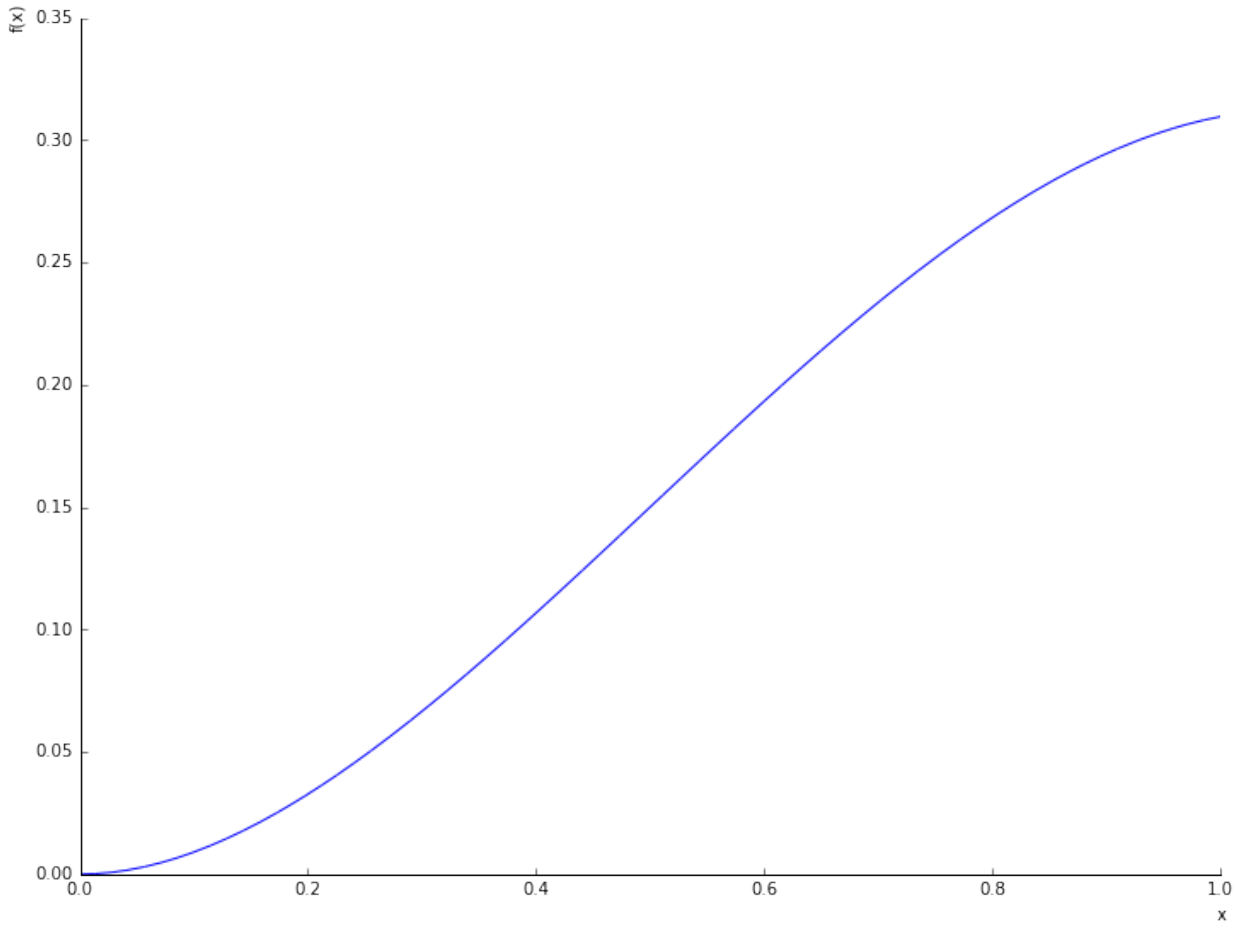
```
In [38]: %matplotlib inline
         from matplotlib import rcParams
         rcParams['figure.figsize']=(12,9)
```

```
In [39]: sympy.plot(sympy.sin(x));
```



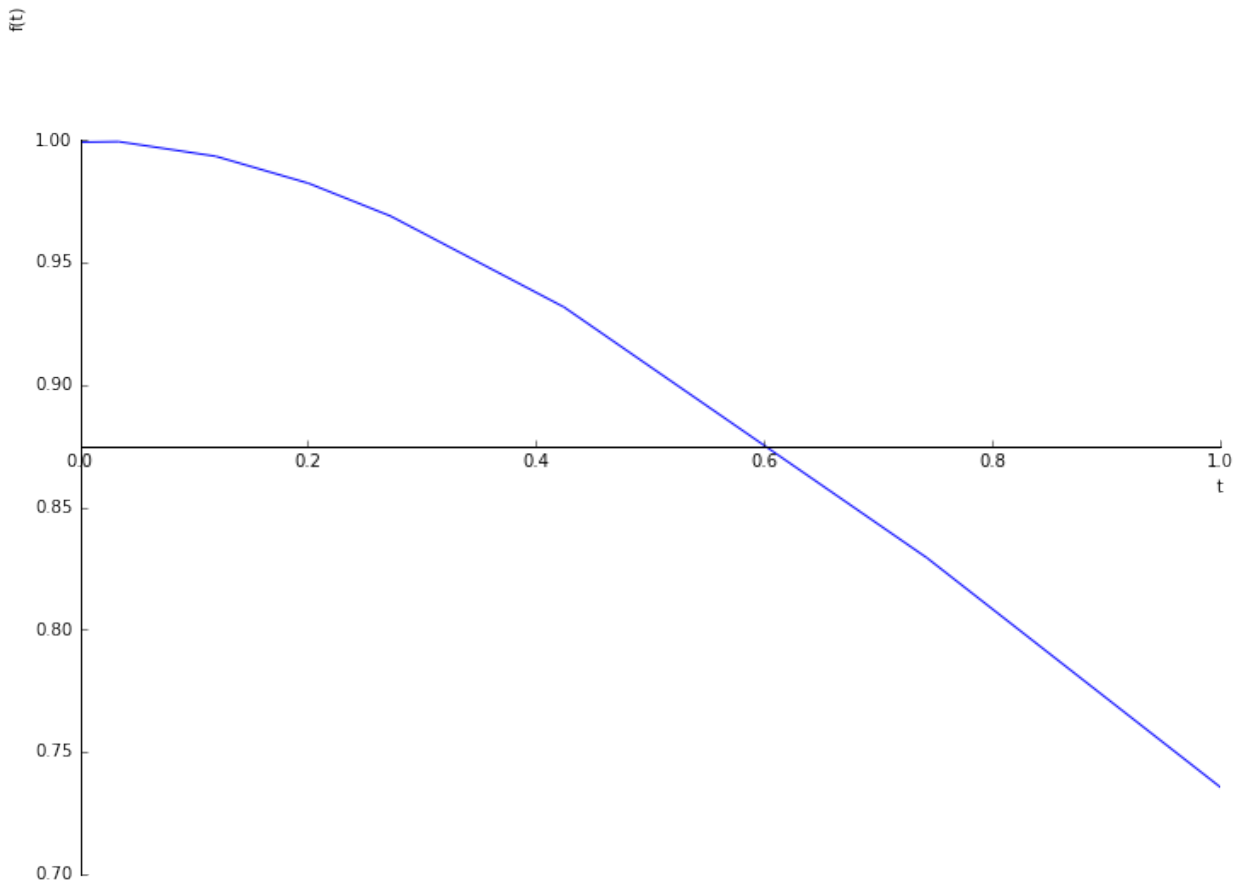
We can explicitly set limits, for example

```
In [40]: sympy.plot(sympy.exp(-x)*sympy.sin(x**2), (x, 0, 1));
```



We can plot the solution to the differential equation computed above:

```
In [41]: sympy.plot(ode_solution.rhs, xlim=(0, 1), ylim=(0.7, 1.05));
```



This can be *visually* compared to the previous result. However, we would often like a more precise comparison, which requires numerically evaluating the solution to the ODE at specific points.

8.2.7 lambdify

At the end of a symbolic calculation using `sympy` we will have a result that is often long and complex, and that is needed in another part of another code. We could type the appropriate expression in by hand, but this is tedious and error prone. A better way is to make the computer do it.

The example we use here is the solution to the ODE above. We have solved it symbolically, and the result is straightforward. We can also solve it numerically using `scipy`. We want to compare the two.

First, let us compute the `scipy` numerical result:

```
In [42]: from numpy import exp
         from scipy.integrate import odeint
         import numpy

         def dydt(y, t):
             """
             Defining the ODE dy/dt = e^{-t} - y.

             Parameters
             ----------

             y : real
```

```
            The value of y at time t (the current numerical approximation)
        t : real
            The current time t

        Returns
        -------

        dydt : real
            The RHS function defining the ODE.
        """

        return exp(-t) - y

    t_scipy = numpy.linspace(0.0, 1.0)
    y0 = [1.0]

    y_scipy = odeint(dydt, y0, t_scipy)
```

We want to evaluate our `sympy` solution at the same points as our `scipy` solution, in order to do a direct comparison. In order to do that, we want to construct a function that computes our `sympy` solution, without typing it in. That is what `lambdify` is for: it creates a function from a sympy expression.

First let us get the expression explicitly:

```
In [43]: ode_expression = ode_solution.rhs
         ode_expression
```

$$(t+1)\,e^{-t}$$

Then we construct the function using `lambdify`:

```
In [44]: from sympy.utilities.lambdify import lambdify

         ode_function = lambdify((t,), ode_expression, modules='numpy')
```

The first argument to `lambdify` is a tuple containing the arguments of the function to be created. In this case that's just `t`, the time(s) at which we want to evaluate the expression. The second argument to `lambdify` is the expression that we want converted into a function. The third argument, which is optional, tells `lambdify` that where possible it should use `numpy` functions. This means that we call the function using `numpy` arrays, it will calculate using `numpy` array expressions, doing the whole calculation in a single call.

We now have a function that we can directly call:

```
In [45]: print("sympy solution at t=0: {}".format(ode_function(0.0)))
         print("sympy solution at t=0.5: {}".format(ode_function(0.5)))
```

```
sympy solution at t=0: 1.0
sympy solution at t=0.5: 0.9097959895689501
```

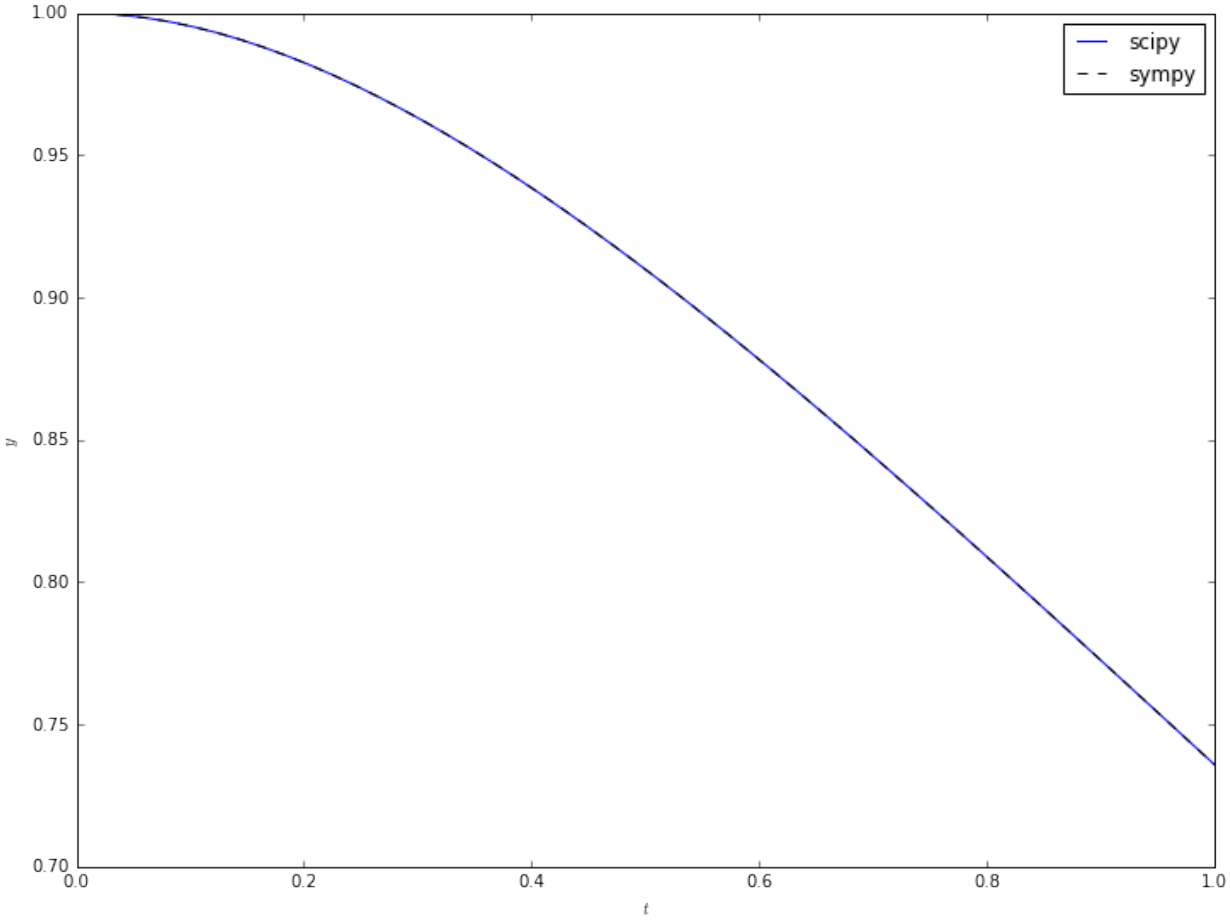
And we can directly apply this function to the times at which the `scipy` solution is constructed, for comparison:

```
In [46]: y_sympy = ode_function(t_scipy)
```

Now we can use `matplotlib` to plot both on the same figure:

```
In [47]: from matplotlib import pyplot
         pyplot.plot(t_scipy, y_scipy[:,0], 'b-', label='scipy')
         pyplot.plot(t_scipy, y_sympy, 'k--', label='sympy')
         pyplot.xlabel(r'$t$')
         pyplot.ylabel(r'$y$')
```

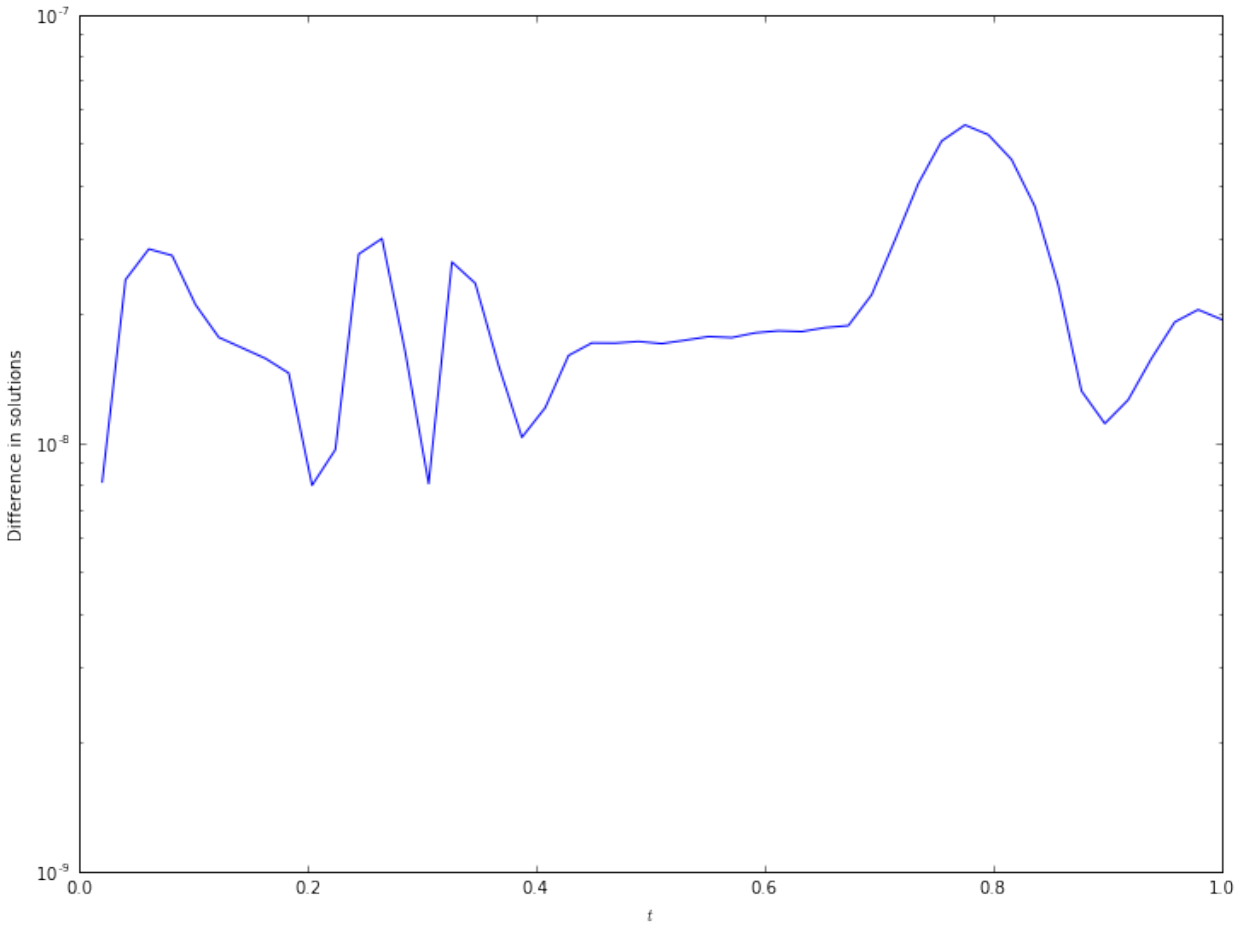
```
        pyplot.legend(loc='upper right')
        pyplot.show()
```



We see good visual agreement everywhere. But how accurate is it?

Now that we have `numpy` arrays explicitly containing the solutions, we can manipulate these to see the differences between solutions:

```
In [48]: pyplot.semilogy(t_scipy, numpy.abs(y_scipy[:,0]-y_sympy))
         pyplot.xlabel(r'$t$')
         pyplot.ylabel('Difference in solutions');
```



The accuracy is around 10^{-8} everywhere - by modifying the accuracy of the `scipy` solver this can be made more accurate (if needed) or less (if the calculation takes too long and high accuracy is not required).

8.3 Further reading

`sympy` has detailed documentation and a useful tutorial.

8.4 Exercise : systematic ODE solving

We are interested in the solution of

$$\frac{\mathrm{d}y}{\mathrm{d}t} = e^{-t} - y^n, \qquad y(0) = 1,$$

where $n > 1$ is an integer. The "minor" change from the above examples mean that `sympy` can only give the solution as a power series.

8.4.1 Exercise 1

Compute the general solution as a power series for $n = 2$.

8.4.2 Exercise 2

Investigate the help for the `dsolve` function to straightforwardly impose the initial condition $y(0) = 1$ using the `ics` argument. Using this, compute the specific solutions that satisfy the ODE for $n = 2, \ldots, 10$.

8.4.3 Exercise 3

Using the `removeO` command, plot each of these solutions for $t \in [0, 1]$.

CHAPTER 9

Statistics

9.1 Statistics

There are many specialized packages for dealing with data analysis and statistical programming. One very important code that you will see in MATH1024, Introduction to Probability and Statistics, is R. A Python package for performing similar analysis of large data sets is pandas. However, simple statistical tasks on simple data sets can be tackled using `numpy` and `scipy`.

9.2 Getting data in

A data file containing the monthly rainfall for Southampton, taken from the Met Office data can be downloaded from this link. We will save that file locally, and then look at the data.

The first few lines of the file are:

```
In [1]: !head southampton_precip.txt
```

```
#Year Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
1855 85.6 54.3 61.3 10.1 60.0 43.9 101.0 47.9 88.4 187.5 28.2 55.4
1856 93.5 50.6 36.3 127.3 55.7 40.3 16.5 64.7 67.6 74.5 38.7 87.1
1857 72.3 10.6 54.4 60.7 19.0 38.2 43.7 66.3 93.6 191.4 57.1 25.0
1858 27.0 33.1 22.9 94.1 65.7 14.1 69.6 55.5 75.2 66.2 50.1 116.6
1859 59.6 78.3 49.7 92.4 36.8 45.7 66.6 58.3 135.3 119.8 125.1 127.1
1860 129.2 29.3 59.3 47.6 88.7 205.0 84.7 115.0 99.2 53.2 80.2 127.7
1861 20.7 60.2 76.4 10.2 41.3 100.8 103.5 22.2 78.0 27.7 164.3 53.2
1862 104.0 20.1 124.2 57.5 123.9 53.8 52.8 36.3 29.7 171.8 22.4 72.7
1863 129.4 32.4 38.7 20.5 55.2 94.6 26.4 63.9 98.7 115.3 60.7 64.4
```

We can use `numpy` to load this data into a variable, where we can manipulate it. This is not ideal: it will lose the information in the header, and that the first column corresponds to years. However, it is simple to use.

```
In [2]: import numpy
```

```
In [3]: data = numpy.loadtxt('southampton_precip.txt')
```

```
In [4]: data
```

```
Out[4]: array([[ 1855. ,    85.6,    54.3, ...,   187.5,    28.2,    55.4],
               [ 1856. ,    93.5,    50.6, ...,    74.5,    38.7,    87.1],
               [ 1857. ,    72.3,    10.6, ...,   191.4,    57.1,    25. ],
               ...,
               [ 1997. ,    16.4,   112.2, ...,    64.5,   151.4,   100.5],
```

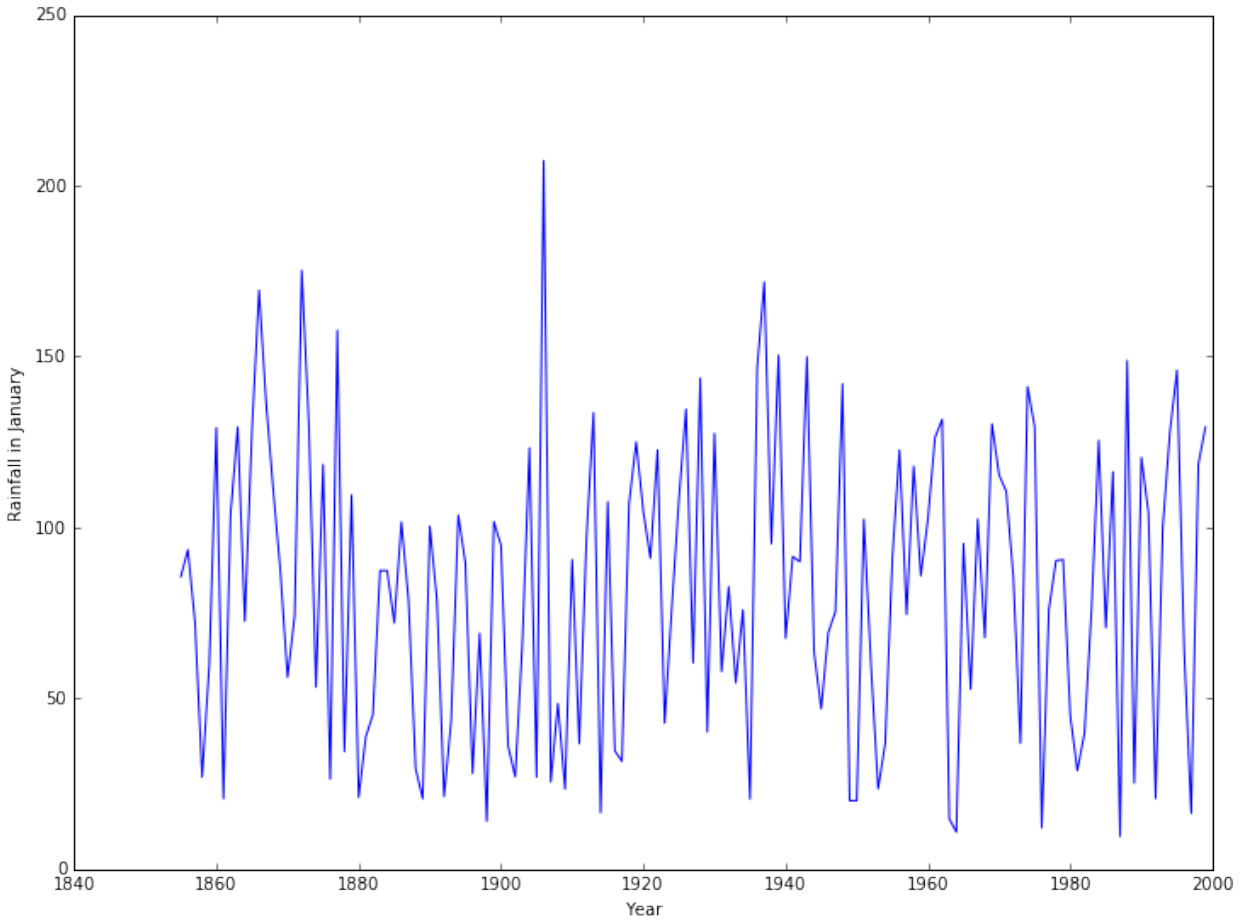
```
       [ 1998. ,   118.5,     9.5, ...,   135.1,    59. ,    87.3],
       [ 1999. ,   129.4,    28.8, ...,    66.8,    49.6,   138.8]])
```

We see that the first column - the year - has been converted to a floating point number, which is not helpful. However, we can now split the data using standard `numpy` operations:

```
In [5]: years = data[:, 0]
        rainfall = data[:, 1:]
```

We can now plot, for example, the rainfall in January for all years:

```
In [6]: %matplotlib inline
        from matplotlib import rcParams
        rcParams['figure.figsize']=(12,9)

In [7]: from matplotlib import pyplot

In [8]: pyplot.plot(years, rainfall[:,0])
        pyplot.xlabel('Year')
        pyplot.ylabel('Rainfall in January');
```



9.3 Basic statistical functions

`numpy` contains a number of basic statistical functions, such as `min`, `max` and `mean`. These will act on entire arrays to give the "all time" minimum, maximum, and average rainfall:

```
In [9]: print("Minimum rainfall: {}".format(rainfall.min()))
        print("Maximum rainfall: {}".format(rainfall.max()))
        print("Mean rainfall: {}".format(rainfall.mean()))
```

```
Minimum rainfall: 0.0
Maximum rainfall: 280.7
Mean rainfall: 67.03591954022988
```

Of more interest would be either

1. the mean (min/max) rainfall in a given month for all years, or
2. the mean (min/max) rainfall in a given year for all months.

So the mean rainfall in the first year, 1855, would be

```
In [10]: print ("Mean rainfall in 1855: {}".format(rainfall[0, :].mean()))
```

```
Mean rainfall in 1855: 68.63333333333334
```

Whilst the mean rainfall in January, averaging over all years, would be

```
In [11]: print ("Mean rainfall in January: {}".format(rainfall[:, 0].mean()))
```

```
Mean rainfall in January: 81.86482758620689
```

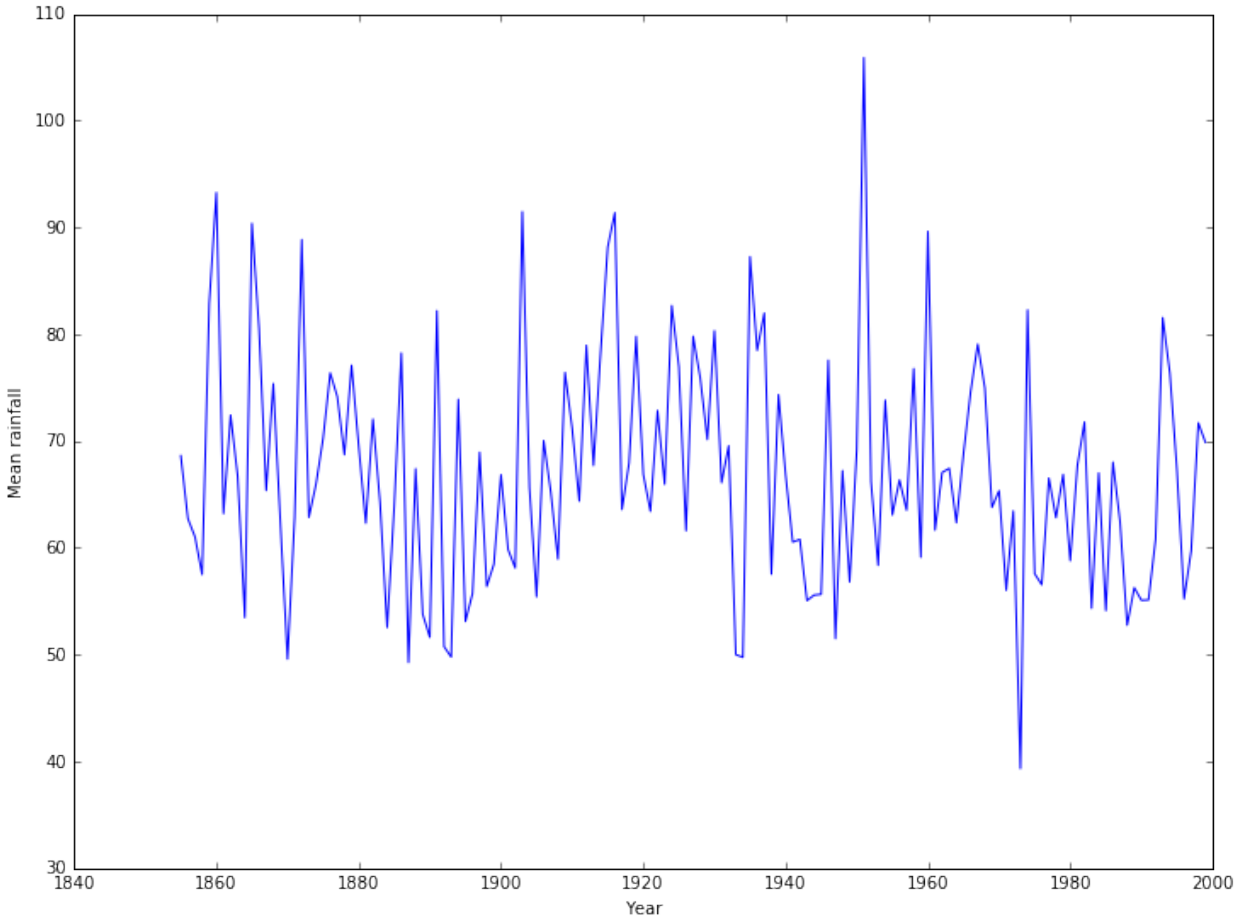
If we wanted to plot the mean rainfall per year, across all years, this would be tedious - there are 145 years of data in the file. Even computing the mean rainfall in each month, across all years, would be bad with 12 months. We could write a loop. However, `numpy` allows us to apply a function along an axis of the array, which does this is one operation:

```
In [12]: mean_rainfall_in_month = rainfall.mean(axis=0)
         mean_rainfall_per_year = rainfall.mean(axis=1)
```

The `axis` argument gives the direction we want to *keep* - that we do not apply the operation to. For this data set, each row contains a year and each column a month. To find the mean in a given month we want to keep the row information (`axis` 0) and take the mean over the column. To find the mean in a given year we want to keep the column information (`axis` 1) and take the mean over the row.

We can now plot how the mean varies with each year.

```
In [13]: pyplot.plot(years, mean_rainfall_per_year)
         pyplot.xlabel('Year')
         pyplot.ylabel('Mean rainfall');
```

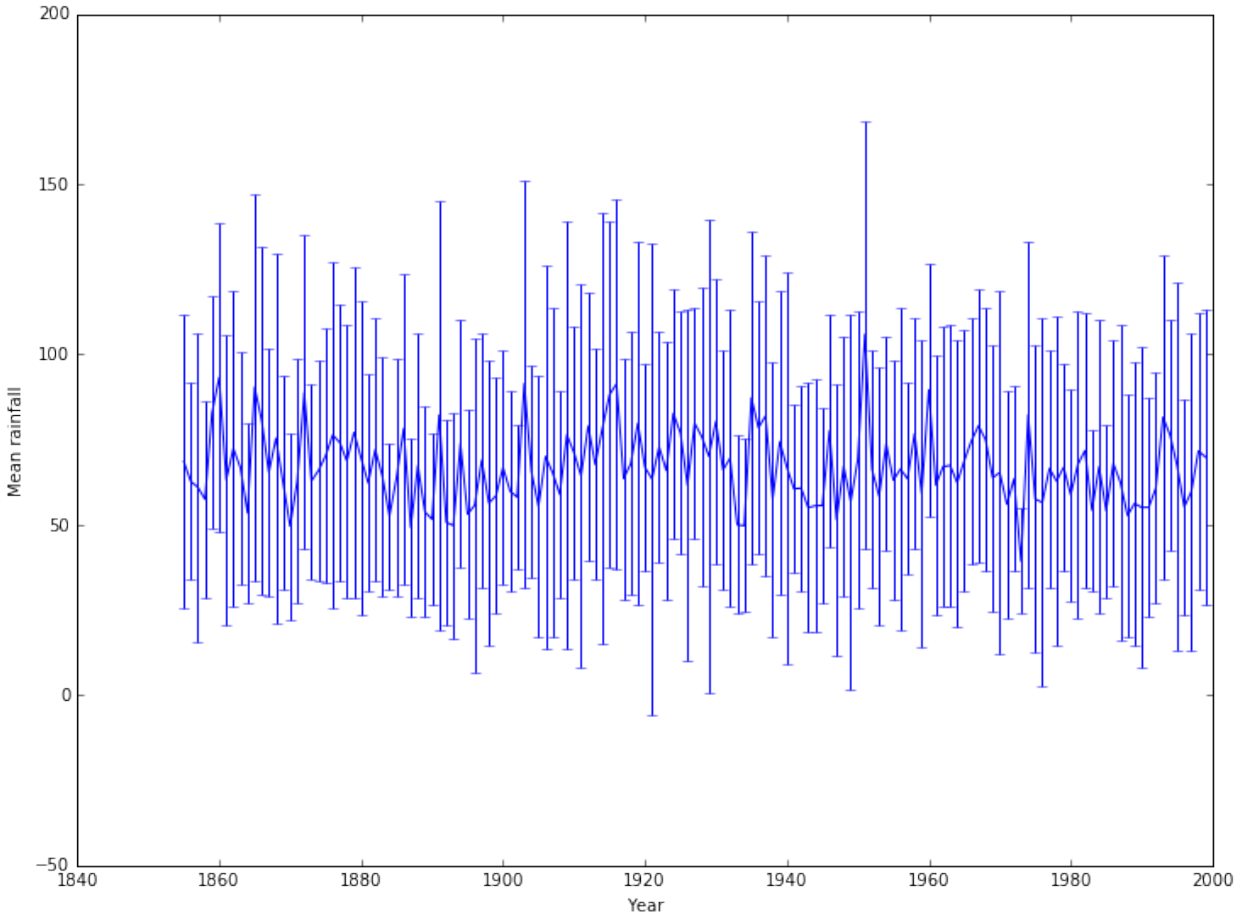


We can also compute the standard deviation:

```
In [14]: std_rainfall_per_year = rainfall.std(axis=1)
```

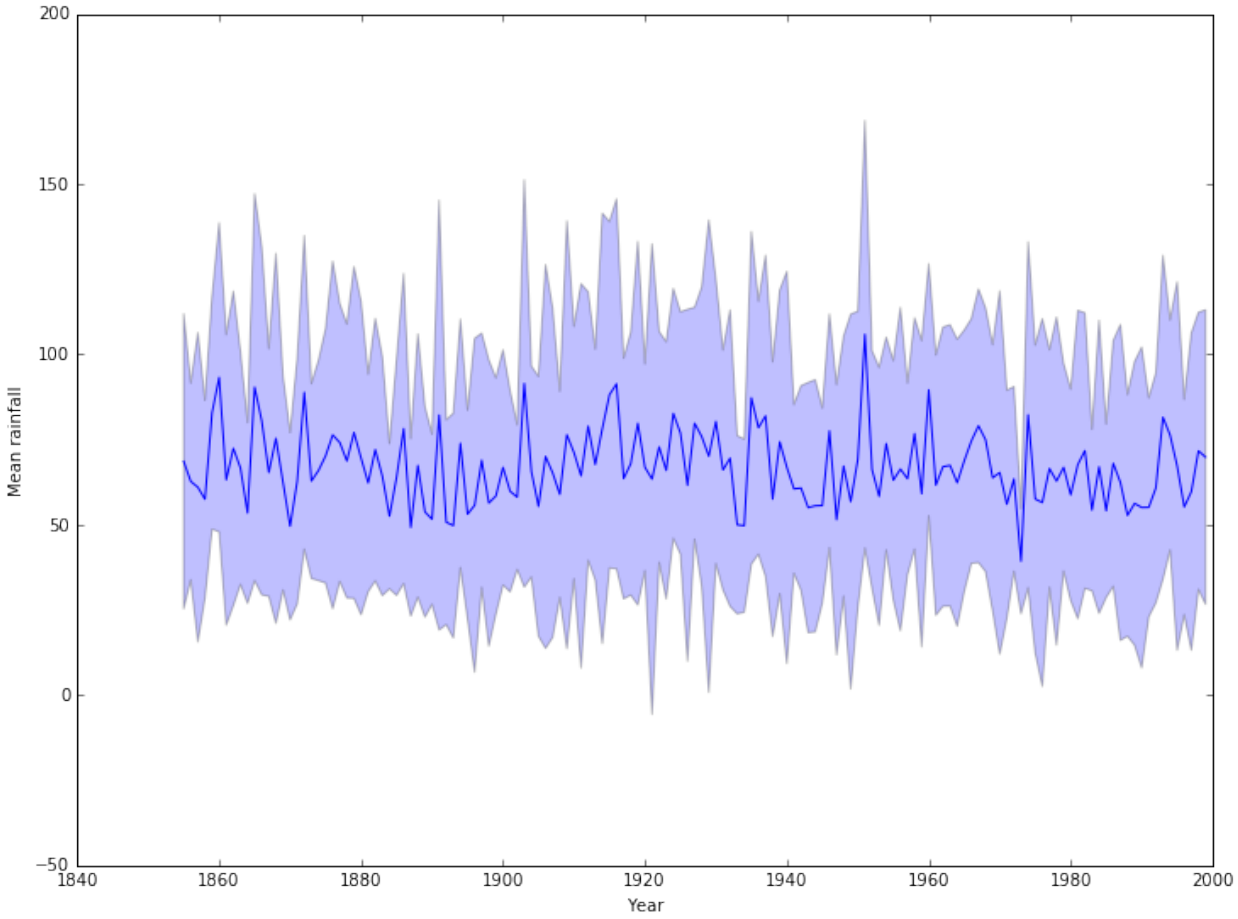
We can then add confidence intervals to the plot:

```
In [15]: pyplot.errorbar(years, mean_rainfall_per_year, yerr = std_rainfall_per_year)
         pyplot.xlabel('Year')
         pyplot.ylabel('Mean rainfall');
```



This isn't particularly pretty or clear: a nicer example would use better packages, but a quick fix uses an alternative `matplotlib` approach:

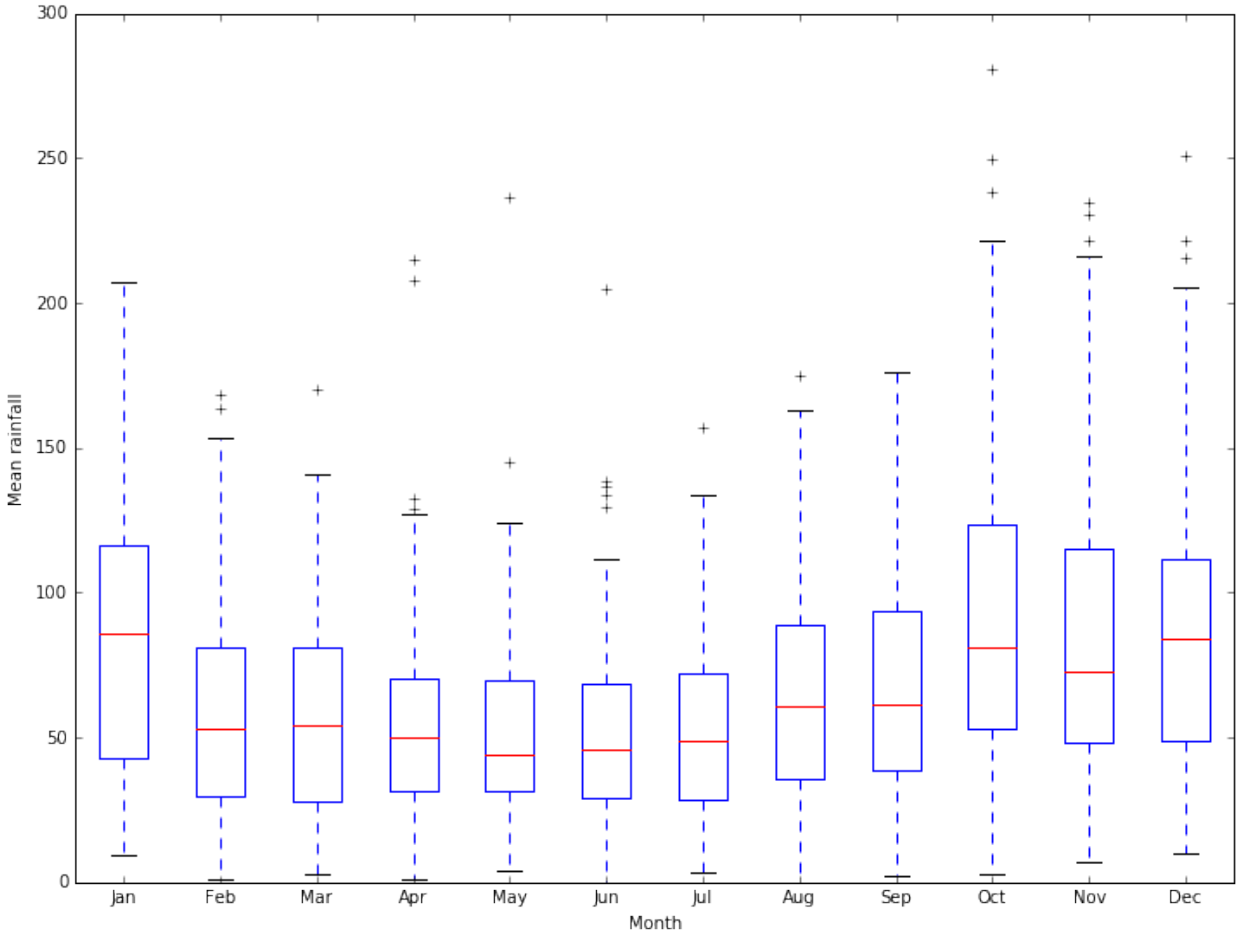
```
In [16]: pyplot.plot(years, mean_rainfall_per_year)
         pyplot.fill_between(years, mean_rainfall_per_year - std_rainfall_per_year,
                             mean_rainfall_per_year + std_rainfall_per_year,
                             alpha=0.25, color=None)
         pyplot.xlabel('Year')
         pyplot.ylabel('Mean rainfall');
```



9.4 Categorical data

Looking at the means by month, it would be better to give them names rather than numbers. We will also summarize the available information using a boxplot:

```
In [17]: months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'N
         pyplot.boxplot(rainfall, labels=months)
         pyplot.xlabel('Month')
         pyplot.ylabel('Mean rainfall');
```

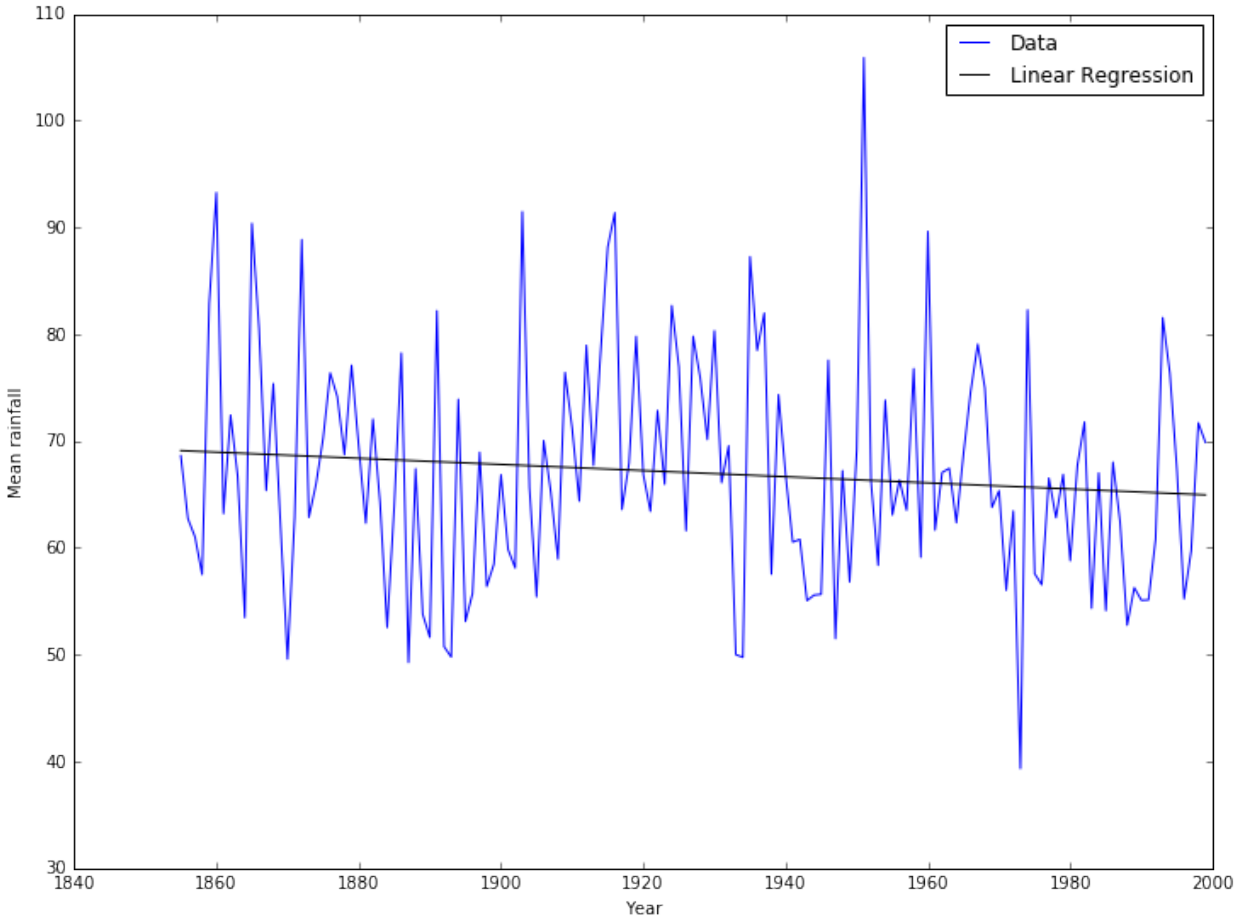


Much better ways of working with categorical data are available through more specialized packages.

9.5 Regression

We can go beyond the basic statistical functions in `numpy` and look at other standard tasks. For example, we can look for simple trends in our data with a *linear regression*. There is a function to compute the linear regression in `scipy` we can use. We will use this to see if there is a trend in the mean yearly rainfall:

```
In [18]: from scipy import stats

         slope, intercept, r_value, p_value, std_err = stats.linregress(years, mean_rainfal

         pyplot.plot(years, mean_rainfall_per_year, 'b-', label='Data')
         pyplot.plot(years, intercept + slope*years, 'k-', label='Linear Regression')
         pyplot.xlabel('Year')
         pyplot.ylabel('Mean rainfall')
         pyplot.legend();
```




```
In [19]: print("The change in rainfall (the slope) is {}.".format(slope))
         print("However, the error estimate is {}.".format(std_err))
         print("The correlation coefficient between rainfall and year"
               " is {}.".format(r_value))
         print("The probability that the slope is zero is {}.".format(p_value))
```

```
The change in rainfall (the slope) is -0.028739338949246847.
However, the error estimate is 0.021587122926201515.
The correlation coefficient between rainfall and year is -0.11064686384415015.
The probability that the slope is zero is 0.18520267346715713.
```

It looks like there's a good chance that the slight decrease in mean rainfall with time is a real effect.

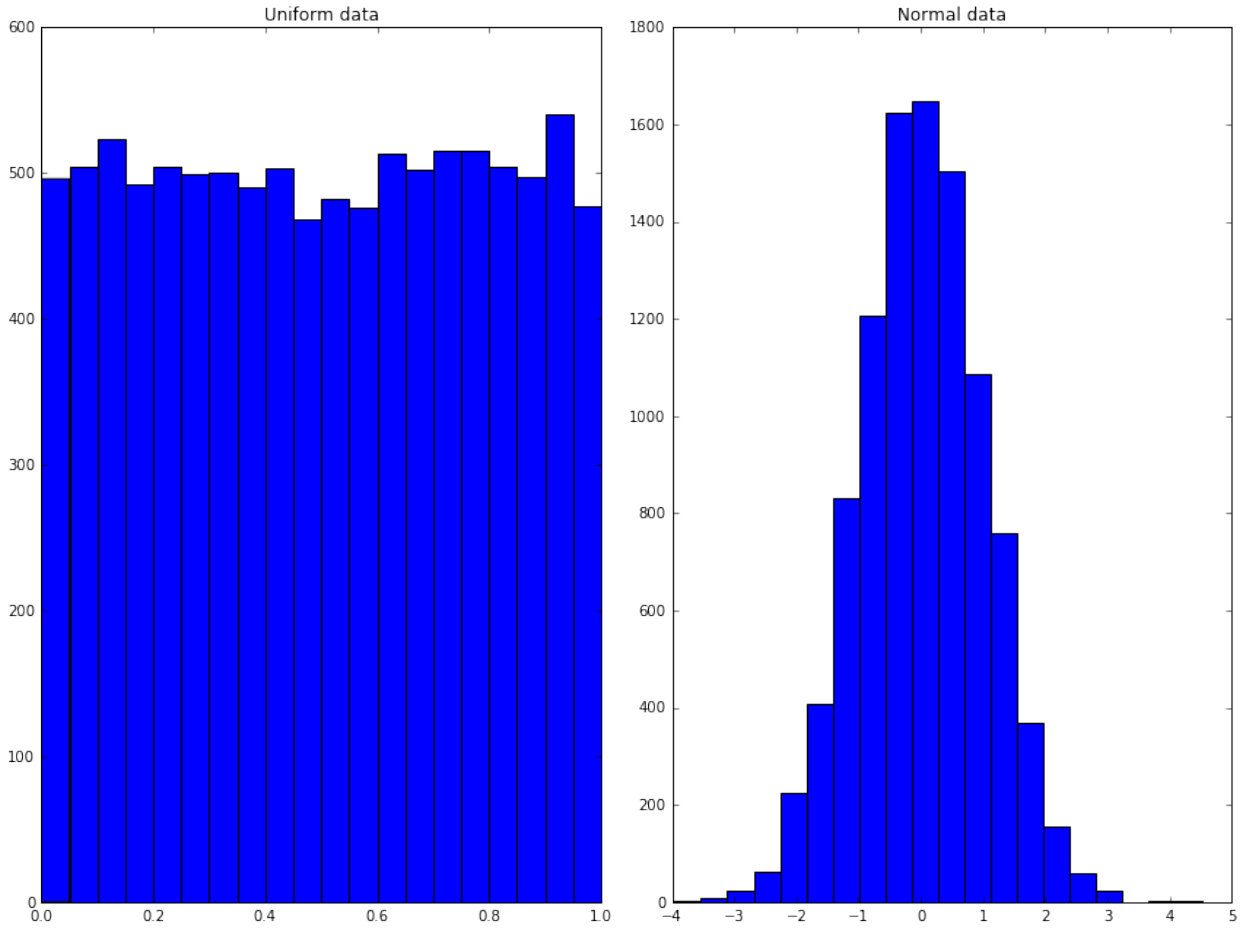
9.6 Random numbers

Random processes and random variables may be at the heart of probability and statistics, but computers cannot generate anything "truly" random. Instead they can generate *pseudo-random* numbers using random number generators (RNGs). Constructing a random number generator is a *hard problem* and wherever possible you should use a well-tested RNG rather than attempting to write your own.

Python has many ways of generating random numbers. Perhaps the most useful are given by the `numpy.random <http://docs.scipy.org/doc/numpy/reference/routines.random.html>`__ module, which can generate a `numpy` array filled with random numbers from various distributions. For example:

```
In [20]: from numpy import random

         uniform = random.rand(10000)
         normal = random.randn(10000)

         fig = pyplot.figure()
         ax1 = fig.add_subplot(1,2,1)
         ax2 = fig.add_subplot(1,2,2)
         ax1.hist(uniform, 20)
         ax1.set_title('Uniform data')
         ax2.hist(normal, 20)
         ax2.set_title('Normal data')
         fig.tight_layout()
         fig.show();
```

```
/Users/ih3/anaconda/lib/python3.4/site-packages/matplotlib/figure.py:397: UserWarning: matp
  "matplotlib is currently using a non-GUI backend, "
```

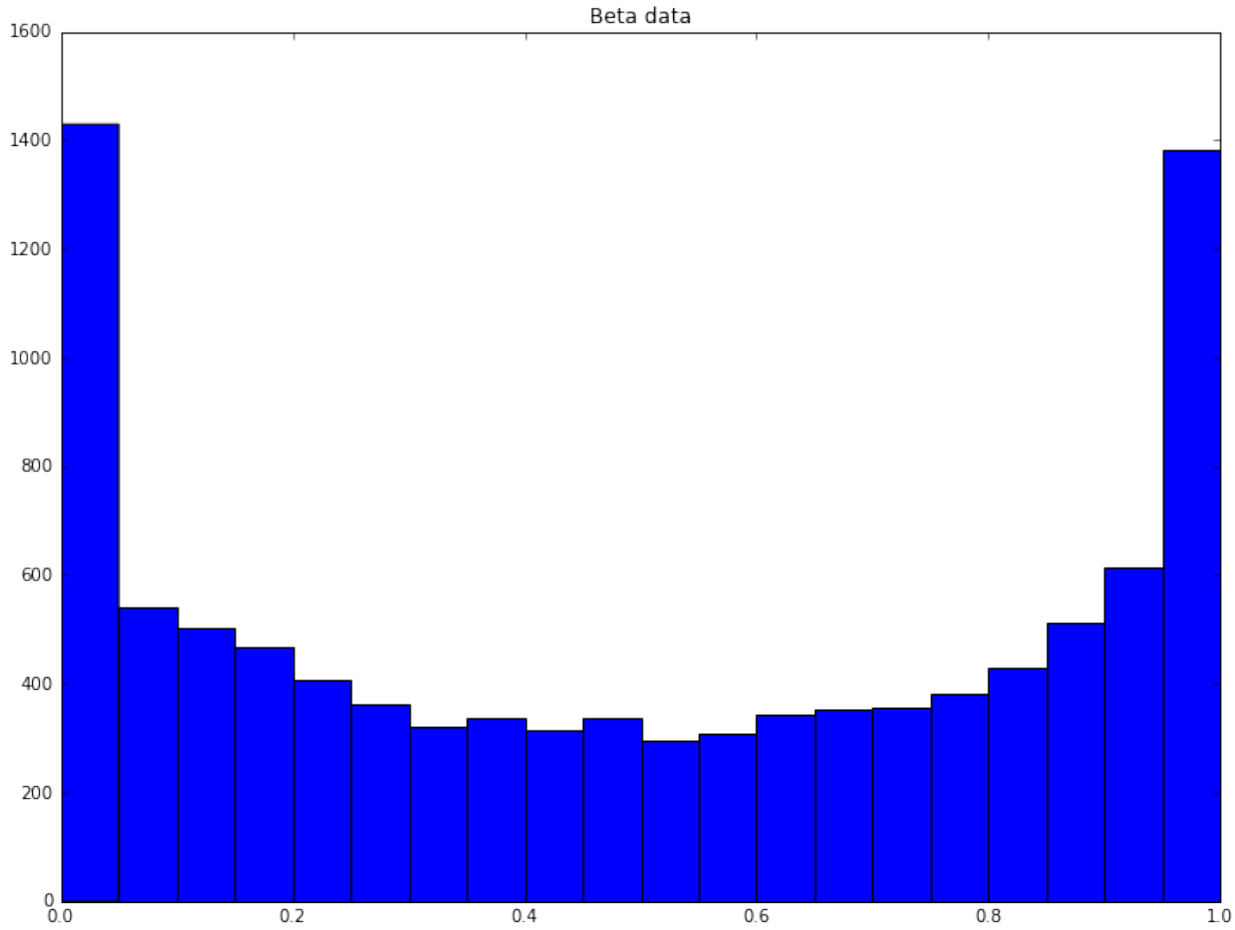


9.6.1 More distributions

Whilst the standard distributions are given by the convenience functions above, the *full documentation of ‘‘numpy.random‘* <http://docs.scipy.org/doc/numpy/reference/routines.random.html>‘__ shows many other distributions available. For example, we can draw $10,000$ samples from the Beta distribution using the parameters

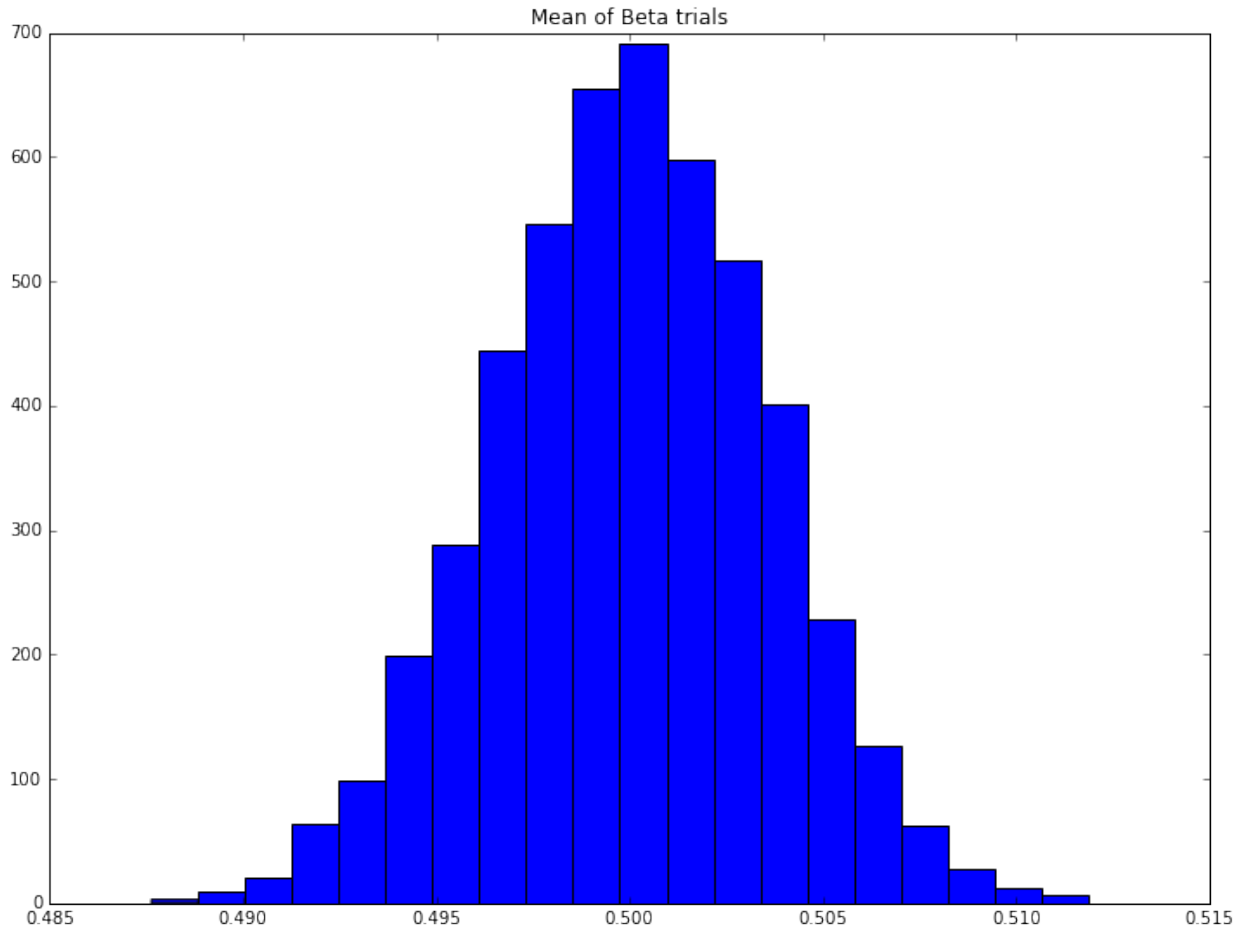
$\alpha = 1/2 = \beta$ as

```
In [21]: beta_samples = random.beta(0.5, 0.5, 10000)

         pyplot.hist(beta_samples, 20)
         pyplot.title('Beta data')
         pyplot.show();
```



We can do this $5,000$ times and compute the mean of each set of samples:

```
In [22]: n_trials = 5000
         beta_means = numpy.zeros((n_trials,))

         for trial in range(n_trials):
             beta_samples = random.beta(0.5, 0.5, 10000)
             beta_means[trial] = numpy.mean(beta_samples)

         pyplot.hist(beta_means, 20)
         pyplot.title('Mean of Beta trials')
         pyplot.show();
```



Here we see the *Central Limit Theorem* in action: the distribution of the means appears to be normal, despite the distribution of any individual trial coming from the Beta distribution, which looks very different.

9.7 Exercise: Anscombe's quartet

Four separate datasets are given:

x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

9.7.1 Exercise 1

Using standard `numpy` operations, show that each dataset has the same mean and standard deviation, to two decimal places.

9.7.2 Exercise 2

Using the standard `scipy` function, compute the linear regression of each data set and show that the slope and correlation coefficient match to two decimal places.

9.7.3 Exercise 3

Plot each dataset. Add the best fit line. Then look at the description of Anscombe's quartet, and consider in what order the operations in this exercise *should* have been done.

CHAPTER 10

Exceptions and Testing

10.1 Exceptions and Testing

Things go wrong when programming all the time. Some of these "problems" are errors that stop the program from making sense. Others are problems that stop the program from working in specific, special cases. These "problems" may be real, or we may want to treat them as special cases that don't stop the program from running.

These special cases can be dealt with using *exceptions*.

10.2 Exceptions

Let's define a function that divides two numbers.

```
In [1]: from __future__ import division
```

```
In [2]: def divide(numerator, denominator):
            """
            Divide two numbers.

            Parameters
            ----------

            numerator: float
                numerator
            denominator: float
                denominator

            Returns
            -------

            fraction: float
                numerator / denominator
            """
            return numerator / denominator
```

```
In [3]: print(divide(4.0, 5.0))
```

```
0.8
```

But what happens if we try something *really stupid*?

```
In [4]: print(divide(4.0, 0.0))
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-4-316ba9717160> in <module>()
----> 1 print(divide(4.0, 0.0))

<ipython-input-2-f5b027ed84cf> in divide(numerator, denominator)
     17         numerator / denominator
     18     """
---> 19     return numerator / denominator

ZeroDivisionError: float division by zero
```

So, the code works fine until we pass in input that we shouldn't. When we do, this causes the code to stop. To show how this can be a problem, consider the loop:

```
In [5]: denominators = [1.0, 0.0, 3.0, 5.0]
        for denominator in denominators:
            print(divide(4.0, denominator))

4.0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-5-05c6ef547bde> in <module>()
      1 denominators = [1.0, 0.0, 3.0, 5.0]
      2 for denominator in denominators:
----> 3     print(divide(4.0, denominator))

<ipython-input-2-f5b027ed84cf> in divide(numerator, denominator)
     17         numerator / denominator
     18     """
---> 19     return numerator / denominator

ZeroDivisionError: float division by zero
```

There are three sensible results, but we only get the first.

There are many more complex, real cases where it's not obvious that we're doing something wrong ahead of time. In this case, we want to be able to *try* running the code and *catch* errors without stopping the code. This can be done in Python:

```
In [6]: try:
            print(divide(4.0, 0.0))
        except ZeroDivisionError:
            print("Dividing by zero is a silly thing to do!")

Dividing by zero is a silly thing to do!

In [7]: denominators = [1.0, 0.0, 3.0, 5.0]
        for denominator in denominators:
            try:
                print(divide(4.0, denominator))
            except ZeroDivisionError:
                print("Dividing by zero is a silly thing to do!")

4.0
Dividing by zero is a silly thing to do!
```

```
1.3333333333333333
0.8
```

The idea here is given by the names. Python will *try* to execute the code inside the `try` block. This is just like an `if` or a `for` block: each command that is indented in that block will be executed in order.

If, and only if, an error arises then the `except` block will be checked. If the error that is produced matches the one listed then instead of stopping, the code inside the `except` block will be run instead.

To show how this works with different errors, consider a different silly error:

```
In [8]: try:
            print(divide(4.0, "zero"))
        except ZeroDivisionError:
            print("Dividing by zero is a silly thing to do!")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-220d79bf294a> in <module>()
      1 try:
----> 2     print(divide(4.0, "zero"))
      3 except ZeroDivisionError:
      4     print("Dividing by zero is a silly thing to do!")

<ipython-input-2-f5b027ed84cf> in divide(numerator, denominator)
     17         numerator / denominator
     18     """
---> 19     return numerator / denominator

TypeError: unsupported operand type(s) for /: 'float' and 'str'
```

We see that, as it makes no sense to divide by a string, we get a `TypeError` instead of a `ZeroDivisionError`. We could catch both errors:

```
In [9]: try:
            print(divide(4.0, "zero"))
        except ZeroDivisionError:
            print("Dividing by zero is a silly thing to do!")
        except TypeError:
            print("Dividing by a string is a silly thing to do!")
```

```
Dividing by a string is a silly thing to do!
```

We could catch *any* error:

```
In [10]: try:
             print(divide(4.0, "zero"))
         except:
             print("Some error occured")
```

```
Some error occured
```

This doesn't give us much information, and may lose information that we need in order to handle the error. We can capture the exception to a variable, and then use that variable:

```
In [11]: try:
             print(divide(4.0, "zero"))
         except (ZeroDivisionError, TypeError) as exception:
             print("Some error occured: {}".format(exception))
```

```
Some error occured: unsupported operand type(s) for /: 'float' and 'str'
```

Here we have caught two possible types of error within the tuple (which *must*, in this case, have parantheses) and captured the specific error in the variable `exception`. This variable can then be used: here we just print it out.

Normally best practise is to be as specific as possible on the error you are trying to catch.

10.2.1 Extending the logic

Sometimes you may want to perform an action *only* if an error did not occur. For example, let's suppose we wanted to store the result of dividing 4 by a divisor, and also store the divisor, but *only* if the divisor is valid.

One way of doing this would be the following:

```
In [12]: denominators = [1.0, 0.0, 3.0, "zero", 5.0]
         results = []
         divisors = []
         for denominator in denominators:
             try:
                 result = divide(4.0, denominator)
             except (ZeroDivisionError, TypeError) as exception:
                 print("Error of type {} for denominator {}".format(exception, denominator)
             else:
                 results.append(result)
                 divisors.append(denominator)
         print(results)
         print(divisors)
```

```
Error of type float division by zero for denominator 0.0
Error of type unsupported operand type(s) for /: 'float' and 'str' for denominator zero
[4.0, 1.3333333333333333, 0.8]
[1.0, 3.0, 5.0]
```

The statements in the `else` block are only run if the `try` block succeeds. If it doesn't - if the statements in the `try` block raise an exception - then the statements in the `else` block are not run.

10.2.2 Exceptions in your own code

Sometimes you don't want to wait for the code to break at a low level, but instead stop when you know things are going to go wrong. This is usually because you can be more informative about what's going wrong. Here's a slightly artificial example:

```
In [13]: def divide_sum(numerator, denominator1, denominator2):
             """
             Divide a number by a sum.

             Parameters
             ----------

             numerator: float
                 numerator
             denominator1: float
                 Part of the denominator
             denominator2: float
                 Part of the denominator

             Returns
```

```
            -------

            fraction: float
                numerator / (denominator1 + denominator2)
            """

            return numerator / (denominator1 + denominator2)
```

```
In [14]: divide_sum(1, 1, -1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-14-a77bb3466659> in <module>()
----> 1 divide_sum(1, 1, -1)

<ipython-input-13-0b7e8c50456d> in divide_sum(numerator, denominator1, denominator2)
     20     """
     21
---> 22     return numerator / (denominator1 + denominator2)

ZeroDivisionError: division by zero
```

It should be obvious to the code that this is going to go wrong. Rather than letting the code hit the `ZeroDivisionError` exception automatically, we can *raise* it ourselves, with a more meaningful error message:

```
In [15]: def divide_sum(numerator, denominator1, denominator2):
            """
            Divide a number by a sum.

            Parameters
            ----------

            numerator: float
                numerator
            denominator1: float
                Part of the denominator
            denominator2: float
                Part of the denominator

            Returns
            -------

            fraction: float
                numerator / (denominator1 + denominator2)
            """

            if (denominator1 + denominator2) == 0:
                raise ZeroDivisionError("The sum of denominator1 and denominator2 is zero

            return numerator / (denominator1 + denominator2)
```

```
In [16]: divide_sum(1, 1, -1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-16-a77bb3466659> in <module>()
----> 1 divide_sum(1, 1, -1)
```

```
<ipython-input-15-49244a562615> in divide_sum(numerator, denominator1, denominator2)
     21
     22     if (denominator1 + denominator2) == 0:
---> 23         raise ZeroDivisionError("The sum of denominator1 and denominator2 is zero!"
     24
     25     return numerator / (denominator1 + denominator2)

ZeroDivisionError: The sum of denominator1 and denominator2 is zero!
```

There are a large number of standard exceptions in Python, and most of the time you should use one of those, combined with a meaningful error message. One is particularly useful: `NotImplementedError`.

This exception is used when the behaviour the code is about to attempt makes no sense, is not defined, or similar. For example, consider computing the roots of the quadratic equation, but restricting to only real solutions. Using the standard formula

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

we know that this only makes sense if $b^2 \geq 4ac$. We put this in code as:

```
In [17]: from math import sqrt

         def real_quadratic_roots(a, b, c):
             """
             Find the real roots of the quadratic equation a x^2 + b x + c = 0, if they ex

             Parameters
             ----------

             a : float
                 Coefficient of x^2
             b : float
                 Coefficient of x^1
             c : float
                 Coefficient of x^0

             Returns
             -------

             roots : tuple
                 The roots

             Raises
             ------

             NotImplementedError
                 If the roots are not real.
             """

             discriminant = b**2 - 4.0*a*c
             if discriminant < 0.0:
                 raise NotImplementedError("The discriminant is {} < 0. "
                                           "No real roots exist.".format(discriminant))
```

```
            x_plus = (-b + sqrt(discriminant)) / (2.0*a)
            x_minus = (-b - sqrt(discriminant)) / (2.0*a)

            return x_plus, x_minus
```

```
In [18]: print(real_quadratic_roots(1.0, 5.0, 6.0))
```

```
(-2.0, -3.0)
```

```
In [19]: real_quadratic_roots(1.0, 1.0, 5.0)
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-19-0fda03c09b58> in <module>()
----> 1 real_quadratic_roots(1.0, 1.0, 5.0)

<ipython-input-17-f4ffff0c1b94> in real_quadratic_roots(a, b, c)
     31     if discriminant < 0.0:
     32         raise NotImplementedError("The discriminant is  < 0. "
---> 33                                   "No real roots exist.".format(discriminant))
     35     x_plus = (-b + sqrt(discriminant)) / (2.0*a)

NotImplementedError: The discriminant is -19.0 < 0. No real roots exist.
```

10.3 Testing

How do we know if our code is working correctly? It is not when the code runs and returns some value: as seen above, there may be times where it makes sense to stop the code even when it is correct, as it is being used incorrectly. We need to test the code to check that it works.

Unit testing is the idea of writing many small tests that check if simple cases are behaving correctly. Rather than trying to *prove* that the code is correct in all cases (which could be very hard), we check that it is correct in a number of tightly controlled cases (which should be more straightforward). If we later find a problem with the code, we add a test to cover that case.

Consider a function solving for the real roots of the quadratic equation again. This time, if there are no real roots we shall return `None` (to say there are no roots) instead of raising an exception.

```
In [20]: from math import sqrt

         def real_quadratic_roots(a, b, c):
             """
             Find the real roots of the quadratic equation a x^2 + b x + c = 0, if they ex

             Parameters
             ----------

             a : float
                 Coefficient of x^2
             b : float
                 Coefficient of x^1
             c : float
                 Coefficient of x^0

             Returns
             -------
```

```
        roots : tuple or None
            The roots
        """

        discriminant = b**2 - 4.0*a*c
        if discriminant < 0.0:
            return None

        x_plus = (-b + sqrt(discriminant)) / (2.0*a)
        x_minus = (-b + sqrt(discriminant)) / (2.0*a)

        return x_plus, x_minus
```

First we check what happens if there are imaginary roots, using $x^2 + 1 = 0$:

```
In [21]: print(real_quadratic_roots(1, 0, 1))
```

```
None
```

As we wanted, it has returned `None`. We also check what happens if the roots are zero, using $x^2 = 0$:

```
In [22]: print(real_quadratic_roots(1, 0, 0))
```

```
(0.0, 0.0)
```

We get the expected behaviour. We also check what happens if the roots are real, using $x^2 - 1 = 0$ which has roots ± 1:

```
In [23]: print(real_quadratic_roots(1, 0, -1))
```

```
(1.0, 1.0)
```

Something has gone wrong. Looking at the code, we see that the `x_minus` line has been copied and pasted from the `x_plus` line, without changing the sign correctly. So we fix that error:

```
In [24]: from math import sqrt

         def real_quadratic_roots(a, b, c):
             """
             Find the real roots of the quadratic equation a x^2 + b x + c = 0, if they ex

             Parameters
             ----------

             a : float
                 Coefficient of x^2
             b : float
                 Coefficient of x^1
             c : float
                 Coefficient of x^0

             Returns
             -------

             roots : tuple or None
                 The roots
             """
```

```
            discriminant = b**2 - 4.0*a*c
            if discriminant < 0.0:
                return None

            x_plus = (-b + sqrt(discriminant)) / (2.0*a)
            x_minus = (-b - sqrt(discriminant)) / (2.0*a)

            return x_plus, x_minus
```

We have changed the code, so now have to re-run *all* our tests, in case our change broke something else:

```
In [25]: print(real_quadratic_roots(1, 0, 1))
         print(real_quadratic_roots(1, 0, 0))
         print(real_quadratic_roots(1, 0, -1))
```

```
None
(0.0, 0.0)
(1.0, -1.0)
```

As a final test, we check what happens if the equation degenerates to a linear equation where $a = 0$, using $x + 1 = 0$ with solution -1:

```
In [26]: print(real_quadratic_roots(0, 1, 1))
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-26-e790de2bb87e> in <module>()
----> 1 print(real_quadratic_roots(0, 1, 1))

<ipython-input-24-a2282acd2dc3> in real_quadratic_roots(a, b, c)
     26         return None
     27
---> 28     x_plus = (-b + sqrt(discriminant)) / (2.0*a)
     29     x_minus = (-b - sqrt(discriminant)) / (2.0*a)
     30

ZeroDivisionError: float division by zero
```

In this case we get an exception, which we don't want. We fix this problem:

```
In [27]: from math import sqrt

         def real_quadratic_roots(a, b, c):
             """
             Find the real roots of the quadratic equation a x^2 + b x + c = 0, if they ex

             Parameters
             ----------

             a : float
                 Coefficient of x^2
             b : float
                 Coefficient of x^1
             c : float
                 Coefficient of x^0

             Returns
             -------
```

```
        roots : tuple or float or None
            The root(s) (two if a genuine quadratic, one if linear, None otherwise)

        Raises
        ------

        NotImplementedError
            If the equation has trivial a and b coefficients, so isn't solvable.
        """

        discriminant = b**2 - 4.0*a*c
        if discriminant < 0.0:
            return None

        if a == 0:
            if b == 0:
                raise NotImplementedError("Cannot solve quadratic with both a"
                                          " and b coefficients equal to 0.")
            else:
                return -c / b

        x_plus = (-b + sqrt(discriminant)) / (2.0*a)
        x_minus = (-b - sqrt(discriminant)) / (2.0*a)

        return x_plus, x_minus
```

And we now must re-run all our tests again, as the code has changed once more:

```
In [28]: print(real_quadratic_roots(1, 0, 1))
         print(real_quadratic_roots(1, 0, 0))
         print(real_quadratic_roots(1, 0, -1))
         print(real_quadratic_roots(0, 1, 1))
```

```
None
(0.0, 0.0)
(1.0, -1.0)
-1.0
```

10.3.1 Formalizing tests

This small set of tests covers most of the cases we are concerned with. However, by this point it's getting hard to remember

1. what each line is actually testing, and
2. what the correct value is meant to be.

To formalize this, we write each test as a small function that contains this information for us. Let's start with the $x^2 - 1 = 0$ case where the roots are ± 1:

```
In [29]: from numpy.testing import assert_equal, assert_allclose

         def test_real_distinct():
             """
             Test that the roots of x^2 - 1 = 0 are \pm 1.
             """
```

```
            roots = (1.0, -1.0)
            assert_equal(real_quadratic_roots(1, 0, -1), roots,
                         err_msg="Testing x^2-1=0; roots should be 1 and -1.")

In [30]: test_real_distinct()
```

What this function does is checks that the results of the function call match the expected value, here stored in `roots`. If it didn't match the expected value, it would raise an exception:

```
In [31]: def test_should_fail():
             """
             Comparing the roots of x^2 - 1 = 0 to (1, 1), which should fail.
             """

             roots = (1.0, 1.0)
             assert_equal(real_quadratic_roots(1, 0, -1), roots,
                          err_msg="Testing x^2-1=0; roots should be 1 and 1."
                          " So this test should fail")

         test_should_fail()

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-31-ccb1cf91e65e> in <module>()
      9                   " So this test should fail")
     10
---> 11 test_should_fail()

<ipython-input-31-ccb1cf91e65e> in test_should_fail()
      6     roots = (1.0, 1.0)
      7     assert_equal(real_quadratic_roots(1, 0, -1), roots,
----> 8                  err_msg="Testing x^2-1=0; roots should be 1 and 1."
      9                  " So this test should fail")
     10

/Users/ih3/anaconda/lib/python3.4/site-packages/numpy/testing/utils.py in assert_equal(actu
    288         assert_equal(len(actual), len(desired), err_msg, verbose)
    289         for k in range(len(desired)):
--> 290             assert_equal(actual[k], desired[k], 'item=%r%s' % (k, err_msg), verbose
    291         return
    292     from numpy.core import ndarray, isscalar, signbit

/Users/ih3/anaconda/lib/python3.4/site-packages/numpy/testing/utils.py in assert_equal(actu
    352     # Explicitly use __eq__ for comparison, ticket #2552
    353     if not (desired == actual):
--> 354         raise AssertionError(msg)
    355
    356 def print_assert_equal(test_string, actual, desired):

AssertionError:
Items are not equal:
item=1
Testing x^2-1=0; roots should be 1 and 1. So this test should fail
 ACTUAL: -1.0
 DESIRED: 1.0
```

Testing that one floating point number equals another can be dangerous. Consider $x^2 - 2x + (1 - 10^{-10}) = 0$ with roots 1.1 ± 10^{-5}):

```
In [32]: from math import sqrt

         def test_real_distinct_irrational():
             """
             Test that the roots of x^2 - 2 x + (1 - 10**(-10)) = 0 are 1 \pm 1e-5.
             """

             roots = (1 + 1e-5, 1 - 1e-5)
             assert_equal(real_quadratic_roots(1, -2.0, 1.0 - 1e-10), roots,
                          err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")

         test_real_distinct_irrational()
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-32-e01bced6ccc9> in <module>()
     10                  err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")
     11
---> 12 test_real_distinct_irrational()

<ipython-input-32-e01bced6ccc9> in test_real_distinct_irrational()
      8     roots = (1 + 1e-5, 1 - 1e-5)
      9     assert_equal(real_quadratic_roots(1, -2.0, 1.0 - 1e-10), roots,
---> 10                  err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")
     12 test_real_distinct_irrational()

/Users/ih3/anaconda/lib/python3.4/site-packages/numpy/testing/utils.py in assert_equal(actu
    288         assert_equal(len(actual), len(desired), err_msg, verbose)
    289         for k in range(len(desired)):
--> 290             assert_equal(actual[k], desired[k], 'item=%r%s' % (k, err_msg), verbose
    291         return
    292     from numpy.core import ndarray, isscalar, signbit

/Users/ih3/anaconda/lib/python3.4/site-packages/numpy/testing/utils.py in assert_equal(actu
    352     # Explicitly use __eq__ for comparison, ticket #2552
    353     if not (desired == actual):
--> 354         raise AssertionError(msg)
    355
    356 def print_assert_equal(test_string, actual, desired):

AssertionError:
Items are not equal:
item=0
Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.
 ACTUAL: 1.0000100000004137
 DESIRED: 1.00001
```

We see that the solutions match to the first 14 or so digits, but this isn't enough for them to be *exactly* the same. In this case, and in most cases using floating point numbers, we want the result to be "close enough": to match the expected precision. There is an assertion for this as well:

```
In [33]: from math import sqrt
```

```
        def test_real_distinct_irrational():
            """
            Test that the roots of x^2 - 2 x + (1 - 10**(-10)) = 0 are 1 \pm 1e-5.
            """

            roots = (1 + 1e-5, 1 - 1e-5)
            assert_allclose(real_quadratic_roots(1, -2.0, 1.0 - 1e-10), roots,
                            err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")

        test_real_distinct_irrational()
```

The `assert_allclose` statement takes options controlling the precision of our test.

We can now write out all our tests:

```
In [34]: from math import sqrt
         from numpy.testing import assert_equal, assert_allclose

         def test_no_roots():
             """
             Test that the roots of x^2 + 1 = 0 are not real.
             """

             roots = None
             assert_equal(real_quadratic_roots(1, 0, 1), roots,
                          err_msg="Testing x^2+1=0; no real roots.")

         def test_zero_roots():
             """
             Test that the roots of x^2 = 0 are both zero.
             """

             roots = (0, 0)
             assert_equal(real_quadratic_roots(1, 0, 0), roots,
                          err_msg="Testing x^2=0; should both be zero.")

         def test_real_distinct():
             """
             Test that the roots of x^2 - 1 = 0 are \pm 1.
             """

             roots = (1.0, -1.0)
             assert_equal(real_quadratic_roots(1, 0, -1), roots,
                          err_msg="Testing x^2-1=0; roots should be 1 and -1.")

         def test_real_distinct_irrational():
             """
             Test that the roots of x^2 - 2 x + (1 - 10**(-10)) = 0 are 1 \pm 1e-5.
             """

             roots = (1 + 1e-5, 1 - 1e-5)
             assert_allclose(real_quadratic_roots(1, -2.0, 1.0 - 1e-10), roots,
                             err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")

         def test_real_linear_degeneracy():
```

```
    """
    Test that the root of x + 1 = 0 is -1.
    """

    root = -1.0
    assert_equal(real_quadratic_roots(0, 1, 1), root,
                 err_msg="Testing x+1=0; root should be -1.")
```

```
In [35]: test_no_roots()
         test_zero_roots()
         test_real_distinct()
         test_real_distinct_irrational()
         test_real_linear_degeneracy()
```

10.3.2 Nose

We now have a set of tests - a *testsuite*, as it is sometimes called - encoded in functions, with meaningful names, which give useful error messages if the test fails. Every time the code is changed, we want to re-run all the tests to ensure that our change has not broken the code. This can be tedious. A better way would be to run a single command that runs all tests. `nosetests` is that command.

The easiest way to use it is to put all tests in the same file as the function being tested. So, create a file `quadratic.py` containing

```
from math import sqrt
from numpy.testing import assert_equal, assert_allclose

def real_quadratic_roots(a, b, c):
    """
    Find the real roots of the quadratic equation a x^2 + b x + c = 0, if they exist.

    Parameters
    ----------

    a : float
        Coefficient of x^2
    b : float
        Coefficient of x^1
    c : float
        Coefficient of x^0

    Returns
    -------

    roots : tuple or float or None
        The root(s) (two if a genuine quadratic, one if linear, None otherwise)

    Raises
    ------

    NotImplementedError
        If the equation has trivial a and b coefficients, so isn't solvable.
    """

    discriminant = b**2 - 4.0*a*c
    if discriminant < 0.0:
        return None
```

```
    if a == 0:
        if b == 0:
            raise NotImplementedError("Cannot solve quadratic with both a"
                                      " and b coefficients equal to 0.")
        else:
            return -c / b

    x_plus = (-b + sqrt(discriminant)) / (2.0*a)
    x_minus = (-b - sqrt(discriminant)) / (2.0*a)

    return x_plus, x_minus

def test_no_roots():
    """
    Test that the roots of x^2 + 1 = 0 are not real.
    """

    roots = None
    assert_equal(real_quadratic_roots(1, 0, 1), roots,
                 err_msg="Testing x^2+1=0; no real roots.")

def test_zero_roots():
    """
    Test that the roots of x^2 = 0 are both zero.
    """

    roots = (0, 0)
    assert_equal(real_quadratic_roots(1, 0, 0), roots,
                 err_msg="Testing x^2=0; should both be zero.")

def test_real_distinct():
    """
    Test that the roots of x^2 - 1 = 0 are \pm 1.
    """

    roots = (1.0, -1.0)
    assert_equal(real_quadratic_roots(1, 0, -1), roots,
                 err_msg="Testing x^2-1=0; roots should be 1 and -1.")

def test_real_distinct_irrational():
    """
    Test that the roots of x^2 - 2 x + (1 - 10**(-10)) = 0 are 1 \pm 1e-5.
    """

    roots = (1 + 1e-5, 1 - 1e-5)
    assert_allclose(real_quadratic_roots(1, -2.0, 1.0 - 1e-10), roots,
                 err_msg="Testing x^2-2x+(1-1e-10)=0; roots should be 1 +- 1e-5.")

def test_real_linear_degeneracy():
    """
    Test that the root of x + 1 = 0 is -1.
    """

    root = -1.0
    assert_equal(real_quadratic_roots(0, 1, 1), root,
                 err_msg="Testing x+1=0; root should be -1.")
```

Then, in a terminal or command window, switch to the directory containing this file. Then run

```
nosetests quadratic.py
```

You should see output similar to

```
nosetests quadratic.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.006s

OK
```

Each dot corresponds to a test. If a test fails, `nose` will report the error and move on to the next test. `nose` automatically runs every function that starts with `test`, or every file in a module starting with `test`, or more. The documentation gives more details about using `nose` in more complex cases.

To summarize: when trying to get code working, tests are essential. Tests should be simple and cover as many of the easy cases and as much of the code as possible. By writing tests as functions that raise exceptions, and using a testing framework such as `nose`, all tests can be run rapidly, saving time.

10.3.3 Test Driven Development

There are many ways of writing code to solve problems. Most involve planning in advance how the code should be written. An alternative is to say in advance what tests the code should pass. This *Test Driven Development* (TDD) has advantages (the code always has a detailed set of tests, features in the code are always relevant to some test, it's easy to start writing code) and some disadvantages (it can be overkill for small projects, it can lead down blind alleys). A detailed discussion is given by Beck's book, and a more recent discussion in this series of conversations.

Even if TDD does not work for you, testing itself is extremely important.

CHAPTER 11

Iterators and Generators

11.1 Iterators and Generators

In the section on loops we introduced the `range` function, and said that you should think about it as creating a list of numbers. In Python `2.X` this is exactly what it does. In Python `3.X` this is *not* what it does. Instead it creates the numbers one at a time. The difference in speed and memory usage is enormous for very large lists - examples are given here and here.

We can recreate one of the examples from Meuer's slides in detail:

```
In [1]: def naivesum_list(N):
            """
            Naively sum the first N integers
            """
            A = 0
            for i in list(range(N + 1)):
                A += i
            return A
```

We will now see how much memory this uses:

```
In [2]: %load_ext memory_profiler

In [3]: %memit naivesum_list(10**4)

peak memory: 32.38 MiB, increment: 0.50 MiB

In [4]: %memit naivesum_list(10**5)

peak memory: 35.95 MiB, increment: 3.57 MiB

In [5]: %memit naivesum_list(10**6)

peak memory: 70.75 MiB, increment: 34.79 MiB

In [6]: %memit naivesum_list(10**7)

peak memory: 426.25 MiB, increment: 382.75 MiB

In [7]: %memit naivesum_list(10**8)

peak memory: 3856.96 MiB, increment: 3744.59 MiB
```

We see that the memory usage is growing very rapidly - as the list gets large it's growing as N.

Instead we can use the `range` function that yields one integer at a time:

```
In [8]: def naivesum(N):
            """
            Naively sum the first N integers
            """
            A = 0
            for i in range(N + 1):
                A += i
            return A

In [9]: %memit naivesum(10**4)

peak memory: 34.35 MiB, increment: 0.13 MiB

In [10]: %memit naivesum(10**5)

peak memory: 34.38 MiB, increment: 0.01 MiB

In [11]: %memit naivesum(10**6)

peak memory: 34.40 MiB, increment: 0.01 MiB

In [12]: %memit naivesum(10**7)

peak memory: 34.41 MiB, increment: 0.00 MiB

In [13]: %memit naivesum(10**8)

peak memory: 34.41 MiB, increment: 0.00 MiB
```

We see that the *memory* usage is unchanged with N, making it practical to run much larger calculations.

11.1.1 Iterators

The `range` function is returning an *iterator* here. This is an object - a general thing - that represents a stream, or a sequence, of data. The iterator knows how to create the first element of the stream, and it knows how to get the next element. It does not, in general, need to know all of the elements at once.

As we've seen above this can save a lot of memory. It can also save time: the code does not need to construct all of the members of the sequence before starting, and it's quite possible you don't need all of them (think about the "Shortest published mathematical paper" exercise).

An iterator such as `range` is very useful, and there's a lot more useful ways to work with iterators in the `itertools` module. These functions that return iterators, such as `range`, are called *generators*, and it's useful to be able to make your own.

11.1.2 Making your own generators

Let's look at an example: finding all primes less than N that can be written in the form $4k - 1$, where k is an integer.

We're going to need to calculate all prime numbers less than or equal to N. We could write a function that returns all these numbers as a list. However, if N gets large then this will be expensive, both in time and memory. As we only need one number at a time, we can use a generator.

```
In [14]: def all_primes(N):
             """
             Return all primes less than or equal to N.

             Parameters
             ----------
```

```
    N : int
        Maximum number

    Returns
    -------

    prime : generator
        Prime numbers
    """

    primes = []
    for n in range(2, N+1):
        is_n_prime = True
        for p in primes:
            if n%p == 0:
                is_n_prime = False
                break
        if is_n_prime:
            primes.append(n)
            yield n
```

This code needs careful examination. First it defines the list of all prime numbers that it currently knows, `primes` (which is initially empty). Then it loops through all integers n from 2 to N (ignoring 1 as we know it's not prime).

Inside this loop it initially assumes that n is prime. It then checks if any of the known primes exactly divides n (`n%p == 0` checks if $n \bmod p = 0$). As soon as it finds such a prime divisor it knows that n is not prime it resets the assumption with this new knowledge, then `break`s out of the loop. This statement stops the `for p in primes` loop early, as we don't need to look at later primes.

If no known prime ever divides n then at the end of the `for p in primes` loop we will still have `is_n_prime` being `True`. In this case we must have n being prime, so we add it to the list of known primes and return it.

It is precisely this point which makes the code above define a generator. We return the value of the prime number found

1. using the `yield` keyword, not the `return` keyword, and
2. we return the value as soon as it is known.

It is the use of the `yield` keyword that makes this function a generator.

This means that only the latest prime number is stored for return.

To use the iterator within a loop, we code it in the same way as with the `range` function:

```
In [15]: print("All prime numbers less than or equal to 20:")
         for p in all_primes(20):
             print(p)
```

```
All prime numbers less than or equal to 20:
2
3
5
7
11
13
17
19
```

To see what the generator is actually doing, we can step through it one call at a time using the built in `next` function:

```
In [16]: a = all_primes(10)

In [17]: next(a)

Out[17]: 2

In [18]: next(a)

Out[18]: 3

In [19]: next(a)

Out[19]: 5

In [20]: next(a)

Out[20]: 7

In [21]: next(a)

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-21-3f6e2eea332d> in <module>()
----> 1 next(a)

StopIteration:
```

So, when the generator gets to the end of its iteration it raises an exception. As seen in previous sections, we could surround the `next` call with a `try` block to capture the `StopIteration` so that we can continue after it finishes. This is effectively what the `for` loop is doing.

We can now find all primes (less than or equal to 100, for example) that have the form $4k - 1$ using

```
In [22]: for p in all_primes(100):
             if (1+p)%4 == 0:
                 print("The prime {} is 4 * {} - 1.".format(p, int((1+p)/4)))

The prime 3 is 4 * 1 - 1.
The prime 7 is 4 * 2 - 1.
The prime 11 is 4 * 3 - 1.
The prime 19 is 4 * 5 - 1.
The prime 23 is 4 * 6 - 1.
The prime 31 is 4 * 8 - 1.
The prime 43 is 4 * 11 - 1.
The prime 47 is 4 * 12 - 1.
The prime 59 is 4 * 15 - 1.
The prime 67 is 4 * 17 - 1.
The prime 71 is 4 * 18 - 1.
The prime 79 is 4 * 20 - 1.
The prime 83 is 4 * 21 - 1.
```

11.2 Exercise : twin primes

A *twin prime* is a pair (p_1, p_2) such that both p_1 and p_2 are prime and $p_2 = p_1 + 2$.

11.2.1 Exercise 1

Write a generator that returns twin primes. You can use the generators above, and may want to look at the itertools module together with its recipes, particularly the `pairwise` recipe.

11.2.2 Exercise 2

Find how many twin primes there are with $p_2 < 1000$.

11.2.3 Exercise 3

Let π_N be the number of twin primes such that $p_2 < N$. Plot how π_N/N varies with N for $N = 2^k$ and $k = 4, 5, \ldots 16$. (You should use a logarithmic scale where appropriate!)

11.3 Exercise : a basis for the polynomials

In the section on classes we defined a `Monomial` class to represent a polynomial with leading coefficient 1. As the $N + 1$ monomials $1, x, x^2, \ldots, x^N$ form a basis for the vector space of polynomials of order N, $\mathbb{P}^N$, we can use the `Monomial` class to return this basis.

11.3.1 Exercise 1

Define a generator that will iterate through this basis of $\mathbb{P}^N$ and test it on $\mathbb{P}^3$.

11.3.2 Exercise 2

An alternative basis is given by the monomials

$$\begin{aligned} p_0(x) &= 1, \\ p_1(x) &= 1 - x, \\ p_2(x) &= (1 - x)(2 - x), \\ \cdots &\quad \cdots, \\ p_N(x) &= \prod_{n=1}^{N} (n - x). \end{aligned}$$

Define a generator that will iterate through this basis of $\mathbb{P}^N$ and test it on $\mathbb{P}^4$.

11.3.3 Exercise 3

Use these generators to write another generator that produces a basis of $\mathbb{P}^3 \times \mathbb{P}^4$.

CHAPTER 12

Classes and OOP

12.1 Classes and Object Oriented Programming

In an earlier section we discussed *classes* as a way of representing an abstract object, such as a polynomial. The resulting code

```
In [1]: class Polynomial(object):
            """Representing a polynomial."""
            explanation = "I am a polynomial"

            def __init__(self, roots, leading_term):
                self.roots = roots
                self.leading_term = leading_term
                self.order = len(roots)

            def display(self):
                string = str(self.leading_term)
                for root in self.roots:
                    if root == 0:
                        string = string + "x"
                    elif root > 0:
                        string = string + "(x - {})".format(root)
                    else:
                        string = string + "(x + {})".format(-root)
                return string

            def multiply(self, other):
                roots = self.roots + other.roots
                leading_term = self.leading_term * other.leading_term
                return Polynomial(roots, leading_term)

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
                print("My roots are {}.".format(self.roots))
```

allowed polynomials to be created, displayed, and multiplied together. However, the language is a little cumbersome. We can take advantage of a number of useful features of Python, many of which carry over to other programming languages, to make it easier to use the results.

Remember that the `__init__` function is called when a variable is created. There are a number of special class functions, each of which has two underscores before and after the name. This is another Python *convention* that is

effectively a rule: functions surrounded by two underscores have special effects, and will be called by other Python functions internally. So now we can create a variable that represents a specific polynomial by storing its roots and the leading term:

```
In [2]: p_roots = (1, 2, -3)
        p_leading_term = 2
        p = Polynomial(p_roots, p_leading_term)
        p.explain_to("Alice")
        q = Polynomial((1,1,0,-2), -1)
        q.explain_to("Bob")
```

```
Hello, Alice. I am a polynomial.
My roots are (1, 2, -3).
Hello, Bob. I am a polynomial.
My roots are (1, 1, 0, -2).
```

Another special function that is very useful is `__repr__`. This gives a *representation* of the class. In essence, if you ask Python to `print` a variable, it will print the *string* returned by the `__repr__` function. This was the role played by our `display` method, so we can just change the name of the function, making the `Polynomial` class easier to use. We can use this to create a simple string representation of the polynomial:

```
In [3]: class Polynomial(object):
            """Representing a polynomial."""
            explanation = "I am a polynomial"

            def __init__(self, roots, leading_term):
                self.roots = roots
                self.leading_term = leading_term
                self.order = len(roots)

            def __repr__(self):
                string = str(self.leading_term)
                for root in self.roots:
                    if root == 0:
                        string = string + "x"
                    elif root > 0:
                        string = string + "(x - {})".format(root)
                    else:
                        string = string + "(x + {})".format(-root)
                return string

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
                print("My roots are {}.".format(self.roots))
```

```
In [4]: p = Polynomial(p_roots, p_leading_term)
        print(p)
        q = Polynomial((1,1,0,-2), -1)
        print(q)
```

```
2(x - 1)(x - 2)(x + 3)
-1(x - 1)(x - 1)x(x + 2)
```

The final special function we'll look at (although there are many more, many of which may be useful) is `__mul__`. This allows Python to *multiply* two variables together. We did this before using the `multiply` method, but by using the `__mul__` method we can multiply together two polynomials using the standard `*` operator. With this we can take the product of two polynomials:

```
In [5]: class Polynomial(object):
            """Representing a polynomial."""
            explanation = "I am a polynomial"

            def __init__(self, roots, leading_term):
                self.roots = roots
                self.leading_term = leading_term
                self.order = len(roots)

            def __repr__(self):
                string = str(self.leading_term)
                for root in self.roots:
                    if root == 0:
                        string = string + "x"
                    elif root > 0:
                        string = string + "(x - {})".format(root)
                    else:
                        string = string + "(x + {})".format(-root)
                return string

            def __mul__(self, other):
                roots = self.roots + other.roots
                leading_term = self.leading_term * other.leading_term
                return Polynomial(roots, leading_term)

            def explain_to(self, caller):
                print("Hello, {}. {}.".format(caller,self.explanation))
                print("My roots are {}.".format(self.roots))
```

```
In [6]: p = Polynomial(p_roots, p_leading_term)
        q = Polynomial((1,1,0,-2), -1)
        r = p*q
        print(r)
```

```
-2(x - 1)(x - 2)(x + 3)(x - 1)(x - 1)x(x + 2)
```

We now have a simple class that can represent polynomials and multiply them together, whilst printing out a simple string form representing itself. This can obviously be extended to be much more useful.

12.1.1 Inheritance

As we can see above, building a complete class from scratch can be lengthy and tedious. If there is another class that does much of what we want, we can build on top of that. This is the idea behind *inheritance*.

In the case of the `Polynomial` we declared that it started from the `object` class in the first line defining the class: `class Polynomial(object)`. But we can build on any class, by replacing `object` with something else. Here we will build on the `Polynomial` class that we've started with.

A *monomial* is a polynomial whose leading term is simply 1. A monomial *is* a polynomial, and could be represented as such. However, we could build a class that knows that the leading term is always 1: there may be cases where we can take advantage of this additional simplicity.

We build a new monomial class as follows:

```
In [7]: class Monomial(Polynomial):
            """Representing a monomial, which is a polynomial with leading term 1."""
```

```
            def __init__(self, roots):
                self.roots = roots
                self.leading_term = 1
                self.order = len(roots)
```

Variables of the `Monomial` class *are* also variables of the `Polynomial` class, so can use all the methods and functions from the `Polynomial` class automatically:

```
In [8]: m = Monomial((-1, 4, 9))
        m.explain_to("Caroline")
        print(m)
```

```
Hello, Caroline. I am a polynomial.
My roots are (-1, 4, 9).
1(x + 1)(x - 4)(x - 9)
```

We note that these functions, methods and variables may not be exactly right, as they are given for the general `Polynomial` class, not by the specific `Monomial` class. If we *redefine* these functions and variables inside the `Monomial` class, they will *override* those defined in the `Polynomial` class. We do not have to override all the functions and variables, just the parts we want to change:

```
In [9]: class Monomial(Polynomial):
            """Representing a monomial, which is a polynomial with leading term 1."""
            explanation = "I am a monomial"

            def __init__(self, roots):
                self.roots = roots
                self.leading_term = 1
                self.order = len(roots)

            def __repr__(self):
                string = ""
                for root in self.roots:
                    if root == 0:
                        string = string + "x"
                    elif root > 0:
                        string = string + "(x - {})".format(root)
                    else:
                        string = string + "(x + {})".format(-root)
                return string
```

```
In [10]: m = Monomial((-1, 4, 9))
         m.explain_to("Caroline")
         print(m)
```

```
Hello, Caroline. I am a monomial.
My roots are (-1, 4, 9).
(x + 1)(x - 4)(x - 9)
```

This has had no effect on the original `Polynomial` class and variables, which can be used as before:

```
In [11]: s = Polynomial((2, 3), 4)
         s.explain_to("David")
         print(s)
```

```
Hello, David. I am a polynomial.
My roots are (2, 3).
4(x - 2)(x - 3)
```

And, as `Monomial` variables are `Polynomials`, we can multiply them together to get a `Polynomial`:

```
In [12]: t = m*s
         t.explain_to("Erik")
         print(t)
```

```
Hello, Erik. I am a polynomial.
My roots are (-1, 4, 9, 2, 3).
4(x + 1)(x - 4)(x - 9)(x - 2)(x - 3)
```

In fact, we can be a bit smarter than this. Note that the `__init__` function of the `Monomial` class is identical to that of the `Polynomial` class, just with the `leading_term` set explicitly to `1`. Rather than duplicating the code and modifying a single value, we can *call* the `__init__` function of the `Polynomial` class directly. This is because the `Monomial` class is built on the `Polynomial` class, so knows about it. We regenerate the class, but only change the `__init__` function:

```
In [13]: class Monomial(Polynomial):
             """Representing a monomial, which is a polynomial with leading term 1."""
             explanation = "I am a monomial"

             def __init__(self, roots):
                 Polynomial.__init__(self, roots, 1)

             def __repr__(self):
                 string = ""
                 for root in self.roots:
                     if root == 0:
                         string = string + "x"
                     elif root > 0:
                         string = string + "(x - {})".format(root)
                     else:
                         string = string + "(x + {})".format(-root)
                 return string
```

```
In [14]: v = Monomial((2, -3))
         v.explain_to("Fred")
         print(v)
```

```
Hello, Fred. I am a monomial.
My roots are (2, -3).
(x - 2)(x + 3)
```

We are now being very explicit in saying that a `Monomial` *really is* a `Polynomial` with `leading_term` being `1`. Note, that in this case we are calling the `__init__` function directly, so have to explicitly include the `self` argument.

By building on top of classes in this fashion, we can build classes that transparently represent the objects that we are interested in.

Most modern programming languages include some object oriented features. Many (including Python) will have more complex features than are introduced above. However, the key points where

- a single variable representing an object can be defined,
- methods that are specific to those objects can be defined,
- new classes of object that inherit from and extend other classes can be defined,

are the essential steps that are common across nearly all.

12.2 Exercise: Equivalence classes

This exercise repeats that from the earlier chapter on classes, but explicitly includes new class methods to make equivalence classes easier to work with.

An *equivalence class* is a relation that groups objects in a set into related subsets. For example, if we think of the integers modulo 7, then 1 is in the same equivalence class as 8 (and 15, and 22, and so on), and 3 is in the same equivalence class as 10. We use the tilde $3 \sim 10$ to denote two objects within the same equivalence class.

Here, we are going to define the positive integers programmatically from equivalent sequences.

12.2.1 Exercise 1

Define a Python class `Eqint`. This should be

1. Initialized by a sequence;
2. Store the sequence;
3. Define its representation (via the `__repr__` function) to be the integer length of the sequence;
4. Redefine equality (via the `__eq__` function) so that two `Eqints` are equal if their sequences have the same length.

12.2.2 Exercise 2

Define a `zero` object from the empty list, and three `one` objects, from a single object list, tuple, and string. For example

```
one_list = Eqint([1])
one_tuple = Eqint((1,))
one_string = Eqint('1')
```

Check that none of the `one` objects equal the zero object, but all equal the other `one` objects. Print each object to check that the representation gives the integer length.

12.2.3 Exercise 3

Redefine the class by including an `__add__` method that combines the two sequences. That is, if `a` and `b` are `Eqints` then `a+b` should return an `Eqint` defined from combining `a` and `b`s sequences.

Note

Adding two different *types* of sequences (eg, a list to a tuple) does not work, so it is better to either iterate over the sequences, or to convert to a uniform type before adding.

12.2.4 Exercise 4

Check your addition function by adding together all your previous `Eqint` objects (which will need re-defining, as the class has been redefined). Print the resulting object to check you get `3`, and also print its internal sequence.

12.2.5 Exercise 5

We will sketch a construction of the positive integers from *nothing*.

1. Define an empty list `positive_integers`.
2. Define an `Eqint` called `zero` from the empty list. Append it to `positive_integers`.
3. Define an `Eqint` called `next_integer` from the `Eqint` defined by *a copy of* `positive_integers` (ie, use `Eqint(list(positive_integers))`. Append it to `positive_integers`.
4. Repeat step 3 as often as needed.

Use this procedure to define the `Eqint` equivalent to 10. Print it, and its internal sequence, to check.

12.3 Exercise: Rational numbers

Instead of working with floating point numbers, which are not "exact", we could work with the rational numbers $\mathbb{Q}$. A rational number $q \in \mathbb{Q}$ is defined by the *numerator* n and *denominator* d as $q = \frac{n}{d}$, where n and d are *coprime* (ie, have no common divisor other than 1).

12.3.1 Exercise 1

Find a Python function that finds the greatest common divisor (`gcd`) of two numbers. Use this to write a function `normal_form` that takes a numerator and divisor and returns the coprime n and d. Test this function on $q = \frac{3}{2}$, $q = \frac{15}{3}$, and $q = \frac{20}{42}$.

12.3.2 Exercise 2

Define a class `Rational` that uses the `normal_form` function to store the rational number in the appropriate form. Define a `__repr__` function that prints a string that *looks like* $\frac{n}{d}$ (**hint**: use `len(str(number))` to find the number of digits of an integer, and use `\n` to start a new line). Test it on the cases above.

12.3.3 Exercise 3

Overload the `__add__` function so that you can add two rational numbers. Test it on $\frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 1$.

12.3.4 Exercise 4

Overload the `__mul__` function so that you can multiply two rational numbers. Test it on $\frac{1}{3} \times \frac{15}{2} \times \frac{2}{5} = 1$.

12.3.5 Exercise 5

Overload the `__rmul__ <https://docs.python.org/2/reference/datamodel.html?highlight=rmul#object.__rmul__>`__ function so that you can multiply a rational by an *integer*. Check that $\frac{1}{2} \times 2 = 1$ and $\frac{1}{2} + (-1) \times \frac{1}{2} = 0$. Also overload the `__sub__` function (using previous functions!) so that you can subtract rational numbers and check that $\frac{1}{2} - \frac{1}{2} = 0$.

12.3.6 Exercise 6

Overload the `__float__` function so that `float(q)` returns the floating point approximation to the rational number `q`. Test this on $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{11}$.

12.3.7 Exercise 7

Overload the `__lt__` function to compare two rational numbers. Create a list of rational numbers where the denominator is $n = 2, \ldots, 11$ and the numerator is the floored integer $n/2$, ie `n//2`. Use the `sorted` function on that list (which relies on the `__lt__` function).

12.3.8 Exercise 8

The Wallis formula for π is

$$\pi = 2 \prod_{n=1}^{\infty} \frac{(2n)^2}{(2n-1)(2n+1)}.$$

We can define a partial product π_N as

$$\pi_N = 2 \prod_{n=1}^{N} \frac{(2n)^2}{(2n-1)(2n+1)},$$

each of which are rational numbers.

Construct a list of the first 20 rational number approximations to π and print them out. Print the sorted list to show that the approximations are always increasing. Then convert them to floating point numbers, construct a `numpy` array, and subtract this array from π to see how accurate they are.

CHAPTER 13

Indices and tables

- search