# Python and Matplotlib Essentials for Scientists and Engineers

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

*Department of Physics and Astronomy*
*Texas A&M University-Commerce*

Morgan & Claypool Publishers

ISBN 978-1-6270-5620-5 (ebook)
ISBN 978-1-6270-5619-9 (print)
ISBN 978-1-6270-5622-9 (mobi)

DOI 10.1088/978-1-6270-5620-5

Version: 20150601

IOP Concise Physics
ISSN 2053-2571 (online)
ISSN 2054-7307 (print)

A Morgan & Claypool publication as part of IOP Concise Physics
Published by Morgan & Claypool Publishers, 40 Oak Drive, San Rafael, CA, 94903, USA

IOP Publishing, Temple Circus, Temple Way, Bristol BS1 6HG, UK

*For Janie—with love.*

# Contents

# Preface

*Python and Matplotlib Essentials for Scientists and Engineers* is intended to provide a starting point for scientists or engineers (or students of either discipline) who want to explore using Python and Matplotlib to work with data and/or simulations, and to make publication-quality plots. The active user base of Python and Matplotlib has been growing rapidly in recent years as people realize these packages have a very high level of functionality, are freely available for any likely operating system and are relatively simple to learn and use compared to similar software solutions.

No previous programming experience is needed before beginning this book, as my aim is to make this a stand-alone introduction to Python and Matplotlib. Indeed, my hope is that you the reader can take this introduction and discover for yourself in just a few hours whether Python and Matplotlib provide most if not all of the tools you need to get your work done and your publication-quality plots rendered.

The examples given in this book are available for download at the companion website pythonessentials.com.

# Acknowledgements

I would like to thank first of all my wife Janie, for her encouragement and support. I am grateful to Pim Schellart of the Astrophysics Department of Radboud University, Nijmegen, The Netherlands, for first introducing me to the Python language, and to Martin D Still of the Science Mission Directorate at NASA for introducing me to more advanced Matplotlib capabilities and helping me to get up to speed. Finally, thank you to the students and SARA colleagues who have commented on earlier versions of this manuscript.

# About the author

## Matt A Wood

Matt A Wood graduated with a BS degree in physics from Iowa State University, and Master's and PhD degrees in astronomy from the University of Texas at Austin. He spent a year as a NATO postdoctoral fellow at the Université de Montreal in Quebec before accepting a position as assistant professor at The Florida Institute of Technology. He spent the 2008–2009 academic year on sabbatical at Radboud University in Nijmegen, The Netherlands, where he was first introduced to the Python programming language. In 2012 he joined the Department of Physics and Astronomy at Texas A&M University-Commerce as department head. His current research focuses on mass-transfer binary star systems known as cataclysmic variables. He has been an author on more than 80 peer-reviewed publications and a similar number of non-refereed publications. He lives in Greenville, Texas, and when not doing astronomy or administrative tasks he enjoys playing guitar and bass, walking his doberman Dexter and exploring the world with his wife Janie.

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 1

# Introduction: why Python and Matplotlib?

## 1.1 Numerical analysis and publication-quality plots

As a scientist or engineer, you need a software package that will allow you to quickly and accurately analyze (including perhaps generating) your data and plot the results in a form that you can use in peer-reviewed publications or formal presentations—what I will be calling *publication-quality plots*. The software needs to be easy to learn and use, versatile, run on whatever computer operating system (OS) you are using and preferably be free of charge. Many people—at least initially—default to using Microsoft Excel (or some other spreadsheet application) because they have some familiarity with it. But while Excel can be useful for simple analyses and visualization of data, it is clunky and fairly limited in what it can achieve. And while it is possible—with effort—to produce a publication-quality plot with Excel, the default plots are not going to win any style awards and are not suitable for publication in professional journals or for presentations at professional conferences.

For many years, MATLAB[1] and IDL[2] have been industry standards for this kind of work, providing sophisticated development and visualization environments that allow rapid design and implementation of algorithms for use in computationally intensive data analysis, numeric computation and visualization projects. Both are high-level languages that are substantially easier to code for a typical problem than for example C/C++ or Fortran and the visualization and data-mining capabilities of both are excellent. However, all of this comes at a substantial cost. Node-locked commercial licenses for MATLAB start at over $2000 and academic pricing starts at

[1] www.mathworks.com/products/matlab
[2] www.exelisvis.com/IDL

doi:10.1088/978-1-6270-5620-5ch1

$500 per license, with additional charges for the various 'toolboxes' that may be required.

In the fields of physics and astronomy, the package SuperMongo[3] (SM) is fairly widely used for making publication-quality plots. It is a powerful interactive plotting package that allows the user to generate beautiful plots with a minimum number of simple commands or user-defined macros, to easily include LaTeX symbols in strings, and to save the plots as postscript files. If you know what this all means then you know this is very useful for technical publishing, and if not, do not worry—Python/Matplotlib can do all this and more. I still use SM for some of my plots and, while it also is not free, it is reasonably priced at $300 for a departmental site license with unlimited free upgrades and personal technical support from the authors. SM also only really runs on Unix-like systems (e.g. Linux and MacOS), so it is not cross-platform.

There are several good plotting packages available that are open source (i.e. free), including Gnuplot[4] and GNU Octave[5]. Gnuplot is a command-line driven plotting utility that is cross-platform and widely used, but which typically requires more time and effort on the part of the user to prepare publication-quality plots. The Octave language is very similar to MATLAB, such that code developed for MATLAB will typically run under Octave with little modification required. However, Octave uses Gnuplot to render output, with results that may not have the polish that MATLAB yields by default.

There are many, many other choices as well, and it is probably safe to say that any plotting package that has been around for more than a few years is *capable* of producing publication-quality plots, given enough time and effort on the part of the user. And this is really the key—you are busy, and if you are using a plotting package that requires an hour or more of your time to produce a publication-quality plot, and then another hour to produce a similar plot, and so on, then that package is keeping you from getting other important tasks done.

## 1.2 Enter Python

You are reading this book because you have heard or read that Python might be the solution you are looking for and I am here to tell you it very probably is! The Python language can probably do *far* more than you or I will ever need or want to do—visit python.org to explore in depth. My purpose here is not to give a comprehensive overview of the language, but to introduce you to the essential core features that will get you up to speed on simple data manipulation and analysis, and making the kinds of plots you need for your work. If you decide you want to mine websites for most commonly used words and phrases, or set up a beautiful graphical user interface (GUI), or write a fully functioning game, you can do that with Python, but you will need to go elsewhere to find out how.

[3] www.astro.princeton.edu/~rhl/sm
[4] www.gnuplot.info
[5] www.gnu.org/software/octave

Python was developed in the early 1990s by Guido van Rossum while at Stichting Mathematisch Centrum in the Netherlands. He is fondly known in the Python community as the Benevolent Dictator for Life (BDFL), but countless others have contributed to the development of the language and community packages. He chose the name Python because he was feeling irreverent and was a big fan of *Monty Python's Flying Circus*.

Python is a well-executed, object-oriented programming language comparable with Perl, Ruby, or Java. It uses a syntax that renders programs easy to read as well as to write. It comes with a large (and growing) standard library with extensive capabilities and can call modules that were written in a compiled language such as C/C++ or Fortran. Python can be used in interactive mode to test short pieces of code and includes a bundled development environment `IDLE` (Integrated DeveLopment Environment). Python is free to download and use, and runs on all major OSs. The language is copyrighted, but is freely re-distributable after modification as all releases of the language are open source[6].

In addition to basic data types such as numbers (integer, floating point, complex), strings and lists, Python also supports object-oriented programming with classes, allowing you to define your own object types. Exception handling of errors is cleanly implemented and the language is well suited to grouping code into modules and packages. Data types can be dynamically assigned and Python implements automatic memory management so you do not have to worry about allocating and freeing memory in your code. Some say entering `import antigravity` at the Python prompt allows one to fly[7], but that may be stretching reality just a bit.

To sum up this introduction: if you are a scientist or engineer looking for a numerical analysis and plotting system that is easy to learn and use, cross-platform and free, the combination of Python and Matplotlib is the grail you seek.

## 1.3 Resources

There are now countless books and websites devoted to Python and associated packages. The webpage wiki.python.org/moin/PythonBooks includes a long list of book titles sorted by category, as well as links to reviews. Some of the specific books that I have found useful in preparing this monograph include:

- Langtangen H P 2012 *A Primer on Scientific Programming with Python* 3rd edn (Berlin: Springer)
- Downey A 2012 *Think Python: How to Think Like a Computer Scientist* (Needham, MA: Green Tea)
- Fangohr H 2014 *Introduction to Python for Computational Science and Engineering* (free download at www.southampton.ac.uk/~fangohr/software).

[6] See www.opensource.org for the open source definition.
[7] Source: xkcd.com/353.

Many online resources exist as well. To list just a few that I have found useful:

- python.org is of course the definitive Python resource on the web. A good starting point is the Beginner's Guide at wiki.python.org/moin/BeginnersGuide.
- stackoverflow.com is a great question and answer site for programmers. Users vote up the best answers, so they show up first and are easiest to find.
- The Python course available at www.python-course.eu/index.php is very comprehensive and includes many tutorials and examples.
- Google has a Python class available at developers.google.com/edu/python/. The class includes text, lecture videos and many coding examples.

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 2

## Downloading and installation

There are a large number of Python distributions to choose from, depending on your OS. The page docs.python.org/2.7/using/index.html contains information on installation, set-up and usage for all major OSs. My recommendation at the time of writing is that you consider the Anaconda Python distribution available from Continuum Analytics[1]. Anaconda Python includes Matplotlib, NumPy, SciPy, IPython, etc—over 195 of the most popular Python packages for use in science, math, engineering and data analysis. The Anaconda distribution has the feature that it installs into a single directory and does not affect other Python installations on the system. It also does not require root or local administrator privileges to install and updates are very easily completed using their online repository.

Python currently has two major versions. Version 2.7 is now considered legacy and no further major releases are expected, but remains popular. Python 3.0, introduced in 2008, is under active development and is the future of the language. The current production version is 3.4. There are some differences between the two versions and some software packages have not yet been updated to work with version 3.x. Most Python programmers are still using Python 2.x and Mac and Linux systems both default to Python 2.x, at the time of writing. Guido van Rossum explains the changes from 2.x to 3.0 in *What's New in Python* 3.0 (docs.python.org/3/whatsnew/3.0.html). Relevant to his discussion, one final advantage of the Anaconda Python distribution is that it makes it trivial to switch between the Python 2.x and 3.x environments.

[1] Visit store.continuum.io/cshop/anaconda. Another popular choice is the Enthought Canopy Express Python distribution available from the site store.enthought.com/downloads.

doi:10.1088/978-1-6270-5620-5ch2

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 3

## First steps

### 3.1 Working with strings

#### 3.1.1 Hello, World!

Let us keep with tradition for our first example—start up your Python session by (a) entering `python` (or `ipython`) in a terminal window (MacOS or Linux) or (b) double clicking to start the Python interface (or Anaconda launcher) and enter the following[1]:

```
>>> print 'Hello, World!'
Hello, World
```

If you are using Python 3.x, you would enter `print("Hello, World!")`. You might have noticed when you invoked the Python shell in interactive mode that it prints the version number and copyright notice before the primary prompt, which defaults to the chevron (>>>). Continuation lines begin with the secondary prompt, which is three dots (. . .) by default.

Note that you can use either single or double quotes, allowing you to have an unmatched quote in your string:

```
>>> print "Hello, World's Best Dad!"
Hello, World's Best Dad!
```

[1] In this book, text that is in `monospaced font` represents text that you enter or that is returned by the computer. Text that is entered is colored black, and text that is returned is colored dark blue. Code snippets are in ivory-colored boxes with light gray borders and complete standalone programs are in ivory-colored boxes with gold borders.

doi:10.1088/978-1-6270-5620-5ch3

If you need to, you can always escape quote marks with a backslash (\), and if you need to include comments, just lead with a hash sign # at the prompt or in your programs:

```
>>> print 'Hello, World\'s Best Mom!'
# Seriously, she's awesome!
Hello, World's Best Mom!
>>>
```

Strings can include (\n) in them to give a line break:

```
>>> print 'Visualize\n Whirled\n  Peas'
Visualize
 Whirled
  Peas
>>>
```

### 3.1.2 Introduction to string methods

It is sometimes useful to `split()` and `join()` strings[2]. For example, if we wished to take the string `Visualize\n Whirled\n Peas` and convert this to a three-element list with the newlines and extraneous white space removed, we can accomplish this with

```
>>> s = 'Visualize\n Whirled\n Peas'
>>> a = s.strip().split()
>>> print a
['Visualize', 'Whirled', 'Peas']
```

We can also `join()` the list elements back into a single string with spaces as a separator:

```
>>> str = ' '           # define the separator character as a space
>>> s2 = str.join(a)    # or more simply: s2 = ' '.join(a)
>>> print s2
Visualize Whirled Peas
>>> type(s2)
<type 'str'>
```

[2] The functions `split()` and `join()` are *methods* of the `string` class. Methods are called using *dot notation*, for example `str.join()`. Methods, including how to define them, are discussed more fully in chapter 9.

### 3.1.3 String concatenation

In your programs for data analysis and plot making, you will probably mostly use strings to access files and to hold text data for plot labels. The following example shows how you might store your directory location in one variable and an input file name in another. Concatenation is as simple as putting a + symbol between the string variable names:

```
>>> directory = '/home/myhome/data/'
>>> infile = 'mydata.dat'
>>> file = directory + infile
>>> file
'/home/myhome/data/mydata.dat'
```

### 3.1.4 Slicing strings

Strings can be *sliced*, where we pull out just some particular character or subset of characters (we will do the same thing soon with lists). As in the C/C++ languages, indexing starts with subscript 0 (Fortran starts with index 1):

```
>>> a = 'Monty Python'
>>> a[0:5]
'Monty'
>>> a[:3]  # first 3 chars
'Mon'
>>> a[3:]  # all but first 3 chars
'ty Python'
>>> a[-4:] # last 4 chars
'thon'
>>> a[:-2] # all but last 4 chars
'Monty Pyth'
```

The length of a string is returned by `len()`:

```
>>> a = 'Well, she turned me into a newt!'
>>> len(a)
32
```

### 3.1.5 Example: a sequence of file names

Let us pause to explore a slightly more complex example, which also gives a sneak peak at the Python `for` loop syntax and the `range()` function. If we want to generate a sequence of file names (e.g., so that we might output intermediate

simulation results) we first need to know how to convert an integer to a string type so we can concatenate it into the file name. This is as simple as `str(i)` where `i` is a variable referring to an integer:

```
>>> i = 42
>>> type(i)
<type 'int'>

>>> s = str(i)
>>> print s
42
>>> type(s)
<type 'str'>
```

And we can create a file name using string concatenation:

```
>>> i = 42
>>> outfile = 'output'+str(i)+'.txt'
>>> print outfile
output42.txt
```

A potential problem with this direct approach is that if we create files `output0.txt` through `output100.txt`, a directory listing will return all files beginning with the string `output1` (`output1.txt`, `output10.txt`, `output11`,...`output100.txt`) before any files that start with `output2`, and so on. We can solve this problem by left padding the string created from integer `i` with zeros, so that instead of `1` to `100`, we have `001`, `002`, `...` `100`. We accomplish this with the `.zfill()` method:

```
>>> i = 42
>>> s = str(i).zfill(3)
>>> print s
042
```

Next, we need our list of integers, which we obtain with the `range()` function[3] The form of `range()` is `range([start,] stop[, step])` where `start` if

[3] Python 2.x includes the function `xrange()` which is an iterator object that returns the integers one at a time and so conserves memory when the calling argument is a very large integer. In Python 3.x, `range()` is an iterator object that behaves like `xrange()` in Python 2.x and the original `range()` is depreciated.

omitted defaults to 0 and `step` if omitted defaults to 1. If all three arguments are present, the function returns a list of integers `[start, start + step, start + 2*step, ...]`. If `step` is positive, the last element is the largest `start + i*step` less than `stop`. If `step` is negative, then the last element is the smallest `start + i*step` greater than `stop`. For example

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(1,11,2)
[1, 3, 5, 7, 9]
>>> range(0,-11,-2)
[0, -2, -4, -6, -8, -10]
```

Finally, we put it all together with a `for` loop. Note that when entering these commands, you will need to indent the lines beginning with `outfile` and `print` to indicate they are within the `for` loop. Python simply uses the indentation level to indicate which lines of code belong in a given block—curly brackets or `endif` statements are therefore not required. The standard indentation is four spaces, but you can use whatever you like, as long as you are consistent within a given block:

```
>>> basename = 'output'
>>> for i in range(1,101):
...   outfile = basename+str(i).zfill(3)+'.txt'
...   print outfile

output001.txt
output002.txt
output003.txt
   .
   .
   .
output100.txt
```

### 3.1.6 Error messages

Finally, note that if an error occurs, the interpreter prints an error message (which is usually helpful) and a *stack trace*. If in interactive mode using the interpreter, it returns to the primary prompt (`>>>`), and if the input came from a file (e.g., `% python myprog.py`) it prints the stack trace and then exits with a non-zero status. More on *exception handling* in section 3.2 below.

```
>>> a = 'spam'
>>> b = a + 1
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

## 3.2 Accessing user input

If you need to access user input when running your code, use the command `raw_input()`, which returns a string object, even if the response is a number:

```
>>> quest = raw_input("What is your quest? ")
What is your quest? To seek the Holy Grail.
>>> quest
To seek the Holy Grail.
>>> print 'Your quest:', quest
Your quest: To seek the Holy Grail.
>>> type(quest)
<type 'str'>

>>> num = raw_input("What is the answer? ")
What is the answer? 42
>>> print num
42
>>> type(num)
<type 'str'>
```

We use the `type()` function to confirm that `quest` and `num` are both string variables. Note that Python version 2.7 and earlier contains the function `input()`, which expects a valid Python expression, and evaluates it. This could be a (potentially complex) number (`2` or `12.3` or `1 + 2j`), or an expression (`3 + 2`), or a quote-enclosed string (`"I'm not dead"`). Thus, the statement `x = input("Enter something: ")` could result in `x` being of type `int`, `float`, `complex`, or `string`! If you are writing code just for your own use and want results as quickly as possible, you might opt to use `input()` rather than `raw_input()`, but the extra overhead for using `raw_input()` is small and offers protection against what should be invalid inputs to your programs. In addition, in Python 3.x the function `input()` behaves as `raw_input()` does in Python 2.7, so you might as well using the current standard from the outset.

If we need our input to be assigned to a variable of type `int` or `float` we can use the `int()` or `float()` functions to perform the conversion for us:

```
>>> myint = int(raw_input('Enter an integer: '))
Enter an integer: 42
>>> myint
42
>>> type(myint)
<type 'int'>
```

If you *do not* pass `int(raw_input("Enter an integer: "))` an integer—for example you enter `123.45` or `spam`—you will receive an error message and be dumped back to the Python prompt:

```
>>> myint = int(raw_input('Enter an integer: '))
Enter an integer: 42.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '42.0'
```

If instead of an integer you wanted a floating point (decimal) number, you could use instead `myfloat = float(raw_input("Enter number: "))`. If the user enters an integer when prompted, the integer will be promoted to a floating point number (e.g., entering either `2` or `2.0` will set `myfloat` to 2.0).

If you want to write user-tolerant code, you might include a test and let the user try again if he/she fumbles when typing the input:

```
>>> while True:
...     try:
...         xint = int(raw_input('Enter an integer: '))
...         print 'Your integer: ',xint
...         break
...     except ValueError:
...         print 'Not an integer. Try again ...'
...
Enter an integer: 2.2
Not an integer. Try again ...
Enter an integer: Spam, glorious spam
Not an integer. Try again ...
Enter an integer: 123
Your integer: 123
```

This code snippet is an example of handling an error exception[4]. When an integer is entered, the `break` statement breaks out of the `while` loop. If you are writing programs that will be widely used, then you will need to spend a fair amount of time worrying about error handling, but as our focus here is on getting you up to speed in Python and Matplotlib, let me simply direct you to section 8 of *The Python Tutorial* at docs.python.org[5] if you need or want a more in-depth treatment of this subject.

## 3.3 Your first Python program file

Using Python interactively is useful for very small tasks, but for anything substantial you will probably want to save your program to a file so you can tweak it until it does what you want and reuse it in the future. Remember that when you exit the interactive shell everything disappears.

While developing your code you have three primary methods to choose from. First, you can use an IDLE for your code development. Anaconda Python includes the Spyder[6] (Scientific PYthon Development EnviRonment) IDLE, which provides an interactive development environment with advanced editing features, interactive testing and debugging features. Second, you can simply have two windows open, one of which has your code file open in your favorite text editor (e.g., `vim`, `Sublime Text`, `Notepad`, `Notepad++` and `emacs`, to name just a few of the available options) and the other of which is running IPython. This is the method I typically use, as shown in figure 3.1, since IPython is far more capable than the standard Python interpreter. Third, you can use an *IPython Notebook*[7], which provides a web-based interactive environment where you can combine code (and results), explanatory text, mathematics, plots and animations into a single document that you can share with collaborators, students and/or instructors. Some users find it has all the features they need for their research notebooks (see figure 3.2). IPython Notebook is included with the Anaconda Python distribution. My recommendation is to try all three methods and see which works best for your coding style.

Python is often referred to as a *scripting* language, but the homepage of `python.org` states 'Python is a programming language that lets you work quickly and integrate systems more effectively'. You may be wondering what the difference is between a *script* and a *program*. The answer is: not much, really. Traditionally a program was a body of code written to be compiled (Fortran, C/C++, etc) and run independently, whereas scripts are typically run from within another program (Python, Perl, IDL, MATLAB). Scripts are usually smaller in scale than traditional programs as well, but scripts are certainly programs in that they are sets of instructions to be executed by the computer. So, feel free to call your codes either scripts or programs, as you prefer.

[4] This code snippet also gives a sneak peak at the `while` statement and *flow control*, discussed further in chapter 7.
[5] The link defaults to Python 3.x documentation, but the 2.x tutorial is just a click away.
[6] code.google.com/p/spyderlib/
[7] See ipython.org/notebook.html.

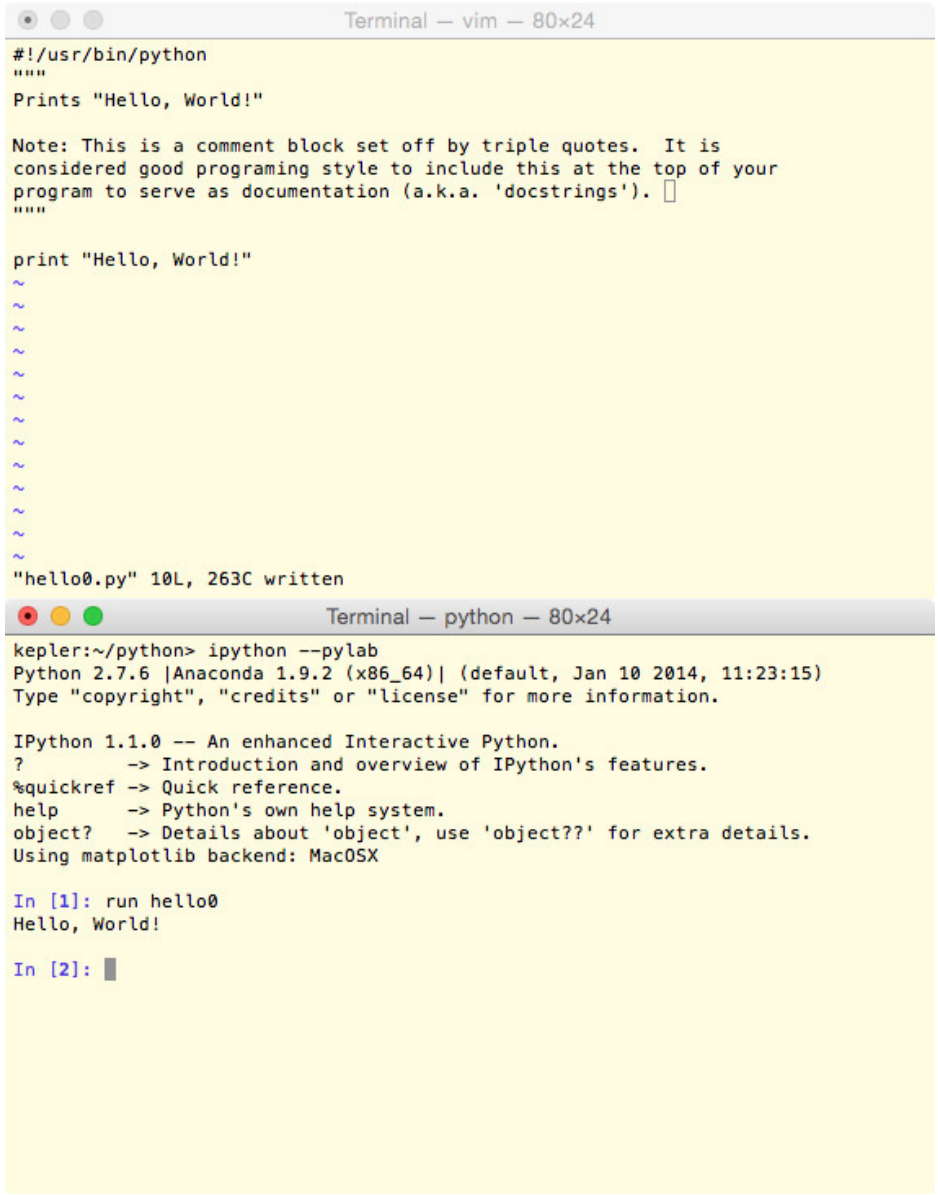


**Figure 3.1.** A simple terminal-based IPython development environment.

Our first program file (figure 3.1, top panel) simply prints `Hello, World!` and exits when it is run. Our program file could simply contain the single line `print "Hello, World!"`, and the result would be the same. The triple quotes set off a multi-line comment. This is used at the top of programs and functions to describe the workings of the routine, as we will discuss further below. The very first line `#!/usr/bin/env python` is a 'shebang' line that tells the program to

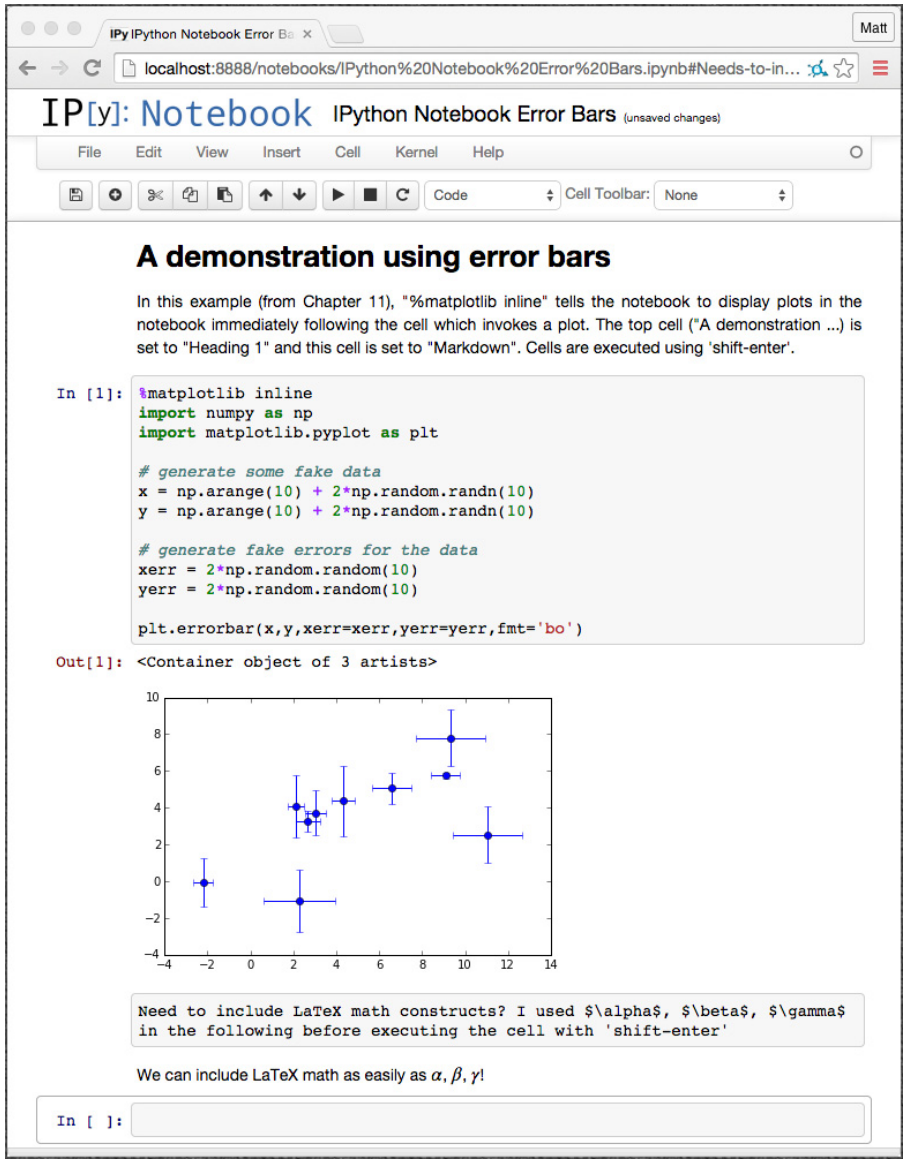


**Figure 3.2.** A sample IPython Notebook showing error bars.

activate the Python program given by your environment variable and should make the program portable between different machines and different OSs.

Our program as it stands is a bit inflexible. We can make it a little more versatile if we add a *command line argument*. In general, command line arguments allow us to input file names, variables, etc, into our program (`hello.py`) without having to prompt for them within the program itself. For example:

```
#!/usr/bin/python
"""
Prints "Hello, <username>" where <username> is an optional command
line argument.

Args:
  sys.argv[1]: string (optional)

Examples:
>>> python hello.py
Hello, Stranger!
>>> python hello.py Bob
Hello, Bob!
"""
import sys

if (len(sys.argv) > 1):
  print "Hello, "+sys.argv[1]+"!"
else:
  print "Hello, Stranger!"
```

Note that here we are importing the `sys` module to pass the command line arguments. `sys.argv[0]` is the program name and `sys.argv[1]` is our command line argument. The total number of command line arguments is given by `len(sys.argv)`.

Now would be a good time to save this code to a file called `hello.py`[8], after which you can run it using the command

```
% python hello.py Guinevere
Hello, Guinevere!
```

but for a program you will use frequently, it will be more convenient to make it an executable. In Unix-like systems, this is accomplished with the `chmod` (change mode) function, where `chmod +x` makes a script executable directly from the command line:

```
% chmod +x hello.py
% hello.py
Hello, Stranger!
% hello.py Merlin
Hello, Merlin!
```

[8] Example codes named in the text are available for download at pythonessentials.com.

As noted above, when I am developing code that uses Python/Matplotlib to analyze data and generate plots, I often have the code open in an editor (`vim`) in one window and an IPython shell running in another window in the same directory. Edit the code, save the file (without exiting), run the code by typing (note that we are using the IPython interpreter this time)

```
% ipython
[version etc. information clipped]
In [1]: run hello.py Arthur  # .py extension is optional within IPython
Hello, Arthur!
In [2]: run hello.py
Hello, Stranger!
```

The IPython shell numbers your commands for later use. In much of the rest of this book, I will continue to show the standard Python prompt `>>>` for examples, but when you are actually doing your own work, my strong recommendation is that you use IPython for everything (or an IDLE or IPython Notebook) whenever you are developing code.

You can also run an external program from the Python prompt `(>>>)` if IPython is not available. If there are no command line arguments, then you can use the `execfile()` function

```
>>> execfile("hello.py")
Hello, Stranger!
```

If you need to pass command line arguments, then things are not so straightforward:

```
>>> import sys
>>> import subprocess
>>> subprocess.call([sys.executable, 'hello.py','Brian'])
Hello, Brian!
```

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 4

# Working with numbers

## 4.1 A powerful calculator

We will do more with strings as we go, but now let us begin working with numbers. When you have some simple arithmetic to do and your calculator is not handy, just start up the Python interpreter. We start with integer arithmetic, since this is the only arithmetic that gives unexpected answers.

```
>>> 2 + 2
4
>>> 4 / 2
2
>>> 1/2
0
>>> -1/2
-1
```

Note the first two lines give the expected results, but that `1 / 2` yields `0` when in the Python 2.x interpreter, because the result was rounded down to the nearest integer (floor). This results because Guido 'BDFL' van Rossum adopted the typical rule from C (also true in Fortran) that the result of an equation is always of the same type as the operands. So, divide a float by a float, you obtain a float; divide an integer by an integer, you obtain an integer. The former case is fine, but the BDFL now considers the latter to be a design bug[1]. Although there are many codes out there that rely on this behavior, for the rest of us it is an annoyance we have to know about and avoid. The good news is that Python 3.x implements true division for both

[1] See python-history.blogspot.com/2009/03/problem-with-integer-division.html.

doi:10.1088/978-1-6270-5620-5ch4

integers and floats[2], so `1/2` will return `0.5`. Python 2.x allows you to execute the command `from __future__ import division` to yield this same behavior. If using Python 2.x and you have not run this command, you can simply put a decimal point after at least one of the integers in a division operation, which will raise the result to a floating number:

```
>>> 1/2.
0.5
>>> 10. * 1/2  # 10.*1 -> 10., then 10./2 -> 5.
5.0
>>> 10. * (1/2) # (1/2) evaluated first -> 0, then 10. * 0 -> 0.0
0.0
>>> from __future__ import division
>>> 10. * (1/2)
5.0
```

The order in which operations are completed in a statement with multiple mathematical operators is similar to other programing languages and can be remembered with the acronym PEMDAS (Parentheses, Exponentiation, Multiplication/Division, Addition/Subtraction).

Variables are easy to assign and work with. The statement

```
>>> a = 4.0
```

is an *assignment statement*. What happens when this is entered into the interpreter is that Python creates the float object `4.0` and binds the name `a` to that object.

In computer programs it is very common to iterate and so lines of code of the form

```
>>> a = a + 1
```

are common. As an algebraic statement, this makes no sense, but what this line of code is telling the interpreter (or compiler in a compiled language), is to evaluate the expression on the right-hand side of the equals sign and then assign the resulting value to the variable name on the left-hand side of the equation. Because this is such a common operation, Python has available the compact notation `a += 1` so, for example,

```
>>> a = 3
>>> a += 1
>>> a
4
```

[2] Integers and floating point numbers (floats) are stored differently in the computer memory.

```
>>> a *= 2
>>> a
8
```

Similarly, Python makes available subtraction of a constant with `-=` and division by a constant with `/=`. Note that the order is important: the expression `a += 1` increments the variable `a` by 1, while the expression `a =+ 1` assigns the value +1 to `a`!

Here are a few trivial examples of assigning variables and using them to calculate results:

```
>>> a = 4
>>> a
4
>>> b = 3
>>> a + b
7
>>> a / b  # Integer division!
1
>>> a / float(b)
1.3333333333333333
>>> c = 3.
>>> a / c
1.3333333333333333
```

When in interactive mode, the previous result is available through the variable `-`. This can be very useful when using Python as a calculator.

```
>>> secperday = 24 * 60 * 60
>>> secperday
86400
>>> secperyear = - * 365
>>> secperyear
31536000
```

A value can be assigned to more than one variable at a time:

```
>>> a = b = c = 5
>>> a
5
>>> b
5
>>> c
5
```

The modulus function `%` can be very useful in certain situations (for example, printing diagnostics every 100 time steps of a simulation):

```
>>> 8 % 2
0
>>> 8 % 3
2
>>> 8 % 3.1
1.7999999999999998
>>>
```

Adapting our code from section 3.1.5,

```
>>> basename = 'output'
>>> for i in range(1,101):
...     if i % 10 == 0:
...         outfile = basename+str(i).zfill(3)+'.txt'
...         print outfile

output010.txt
output020.txt
   .
   .
   .
output100.txt
```

It is probable that you will often want to use the value of $\pi$ in your programs. You can enter it yourself, but if you execute `from math import pi` at the command line or at the top of your program, you will have it available to machine precision when you need it. `math` is an example of a module and modules are extensions to Python that can be imported to extend the capabilities of the base language:

```
>>> from math import pi
>>> pi
3.1415926535897931
```

The math module provides access to the standard mathematical functions defined by the C standard and you have the option to simply include everything in the module with `from math import *` at the top of your programs, although as

discussed below it is generally safer to just import what you need to avoid conflicts in the namespace:

```
>>> from math import cos, log, log10
>>> cos(pi)              # trig functions expect angles in radians
-1.0
>>> log(e)
1.0
>>> log10(100.)
2.0
```

Here is a selected list of a few useful functions from the `math` module:

```
Function   Operation
fabs(x)    absolute value of x
log(x)     base-e logarithm of x
log10(x)   base-10 logarithm of x
pow(x,y)   x^y
sqrt(x)    square root of x
acos(x)    arc cosine of x, in radians
atan2(y,x) arctan(y/x), in range -pi,pi
degrees(x) converts x from radians to degrees
radians(x) converts x from degrees to radians
```

Complex numbers are also straightforward to work with, where `j` indicates the complex part of the value:

```
>>> a = 2 + 3j
>>> b = 4 - 9j
>>> a
(2+3j)
>>> b
(4-9j)
>>> a + b
(6-6j)
>>> a * b
(35-6j)
```

Complex numbers can also be created with the `complex(real, imag)` function. It is simple to extract the real and imaginary parts of a complex number if needed separately:

```
>>> a = complex(3.2, 5.4)
>>> print a
```

```
(3.2 + 5.4j)
>>> a.real
3.2
>>> a.imag
5.4
```

## 4.2 Lists, tuples and arrays

### 4.2.1 Lists

Python includes several *compound* data types. The *list* is a very useful sequence construct. It can be written as a list of comma-separated values between square brackets:

```
>>> a = ['spam', 'elderberry', 123, 4.0]
>>> a[0]
'spam'
>>> a[1]
'elderberry'
>>> a[2]
123
>>> a[3]
4.0
```

Note that the elements of a list do not have to all be of the same type—in the example, we have strings, an integer and a float all in the same list.

You can add elements on to the end of your list, replace elements within your list, remove items from your list, insert some, reverse the list, or clear the entire list:

```
>>> a.append('more spam')
>>> a
['spam', 'elderberry', 123, 4.0, 'more spam']
>>> #replace items
>>> a[1] = 'a newt'
['spam', 'a newt', 123, 4.0, 'more spam']
>>> # remove item at index i
>>> a.pop(3)
4.0
>>> a
['spam', 'a newt', 123, 'more spam']
>>> a.pop() # remove last by default
'more spam'
```

```
>>> a
['spam', 'a newt', 123]
>>> # insert an item
>>> a.insert(1,'spamalot')
>>> a
['spam', 'spamalot', 'a newt', 123]
>>> # reverse the order
>>> a.reverse()
>>> a
[123, 'a newt', 'spamalot', 'spam']
>>> # clear the list
>>> a = []
>>> a
[]
```

As noted briefly in the previous chapter, strings can be concatenated with the + sign:

```
>>> a = ['and','now','for']
>>> b = ['something','completely','different']
>>> c = a+b
>>> c
['and', 'now', 'for', 'something', 'completely', 'different']
```

Lists can also contain other lists, which is useful for some applications:

```
>>> john = ['John Smith',39]
>>> mary = ['Mary Jones',33]
>>> database = [john,mary]
>>> database
[['John Smith', 39], ['Mary Jones', 33]]
```

### 4.2.2 Slicing lists

Lists, like strings, can be sliced and indexed as needed:

```
>>> a = ['and', 'now', 'for', 'something', 'completely', 'different']
>>> a[3]
'something'
>>> a[0:3]           # items 0 through 2
['and', 'now', 'for']
>>> a[:3]            # items 0 through 2
['and', 'now', 'for']
>>> a[3:]            # items 3 through end
```

```
['something', 'completely', 'different']
>>> a[0:6:2]          # items 0 through 5 with stride=2
['and', 'for', 'completely']
>>> a[:]              # a copy of the entire array
['and', 'now', 'for', 'something', 'completely', 'different']
```

### 4.2.3 List comprehension

Creating lists is a frequently encountered task in Python programs, so Python has a useful and powerful compact syntax for accomplishing this, called *list comprehension*. The general syntax is

```
>>> mylist = [expression for a in list if conditional]
```

which is equivalent to

```
>>> for item in list:
>>>     if conditional:
>>>         expression
```

Here is a simple example to demonstrate the concept:

```
>>> x = [i*i for i in range(5)]
>>> x
>>> [0, 1, 4, 9, 16]
```

and if we only wanted to keep the even values, we can add a conditional statement:

```
>>> y = [i*i for i in range(5) if i*i % 2 == 0]
>>> y
>>> [0, 4, 16]
```

We can also use list comprehensions for string objects:

```
>>> motto = 'Keep Calm and Python'.split()
>>> [i.upper() for i in motto]
['KEEP', 'CALM', 'AND', 'PYTHON']
>>> [len(i) for i in motto]
[4, 4, 3, 6]
```

List comprehensions can be nested, with syntax:

```
[ expr for outer-loop-vbl in outer-seq for inner-loop-var in inner-seq ]
```

For example, to assign coordinate pairs for a rectangular grid:

```
>>> coords = [(x,y) for x in range(4) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0),
(2, 1), (2, 2), (3, 0), (3, 1), (3, 2)]
```

We will see additional uses of list comprehension below.

### 4.2.4 Tuples

Python also includes a data type called a *tuple* (in fact, our most recent example returned a list of tuples). Tuples, like lists, are sequences. What is special about tuples is that they cannot be changed—they are *immutable* (lists are *mutable*). You might think of a tuple as a 'constant list'. Tuples are indicated by simply separating some values with commas and are often enclosed in parentheses:

```
>>> 3, 2, 1
(3, 2, 1)
>>> (1, 2, 3)
(1, 2, 3)
```

The empty tuple is written as a `()`. You can slice tuples just as you can lists and you can convert a list to a tuple or vice versa if need be:

```
>>> a = [3, 2, 1]
>>> tuple(a)
(3, 2, 1)
>>> b = list(a)
>>> b
[3, 2, 1]
```

Tuples are used behind the scenes in Python and you may use them when calling functions. Again, they work very much like lists, with the exception that they cannot be changed.

The `zip()` function iterates over two or more sequences or iterables in parallel. Most commonly, it takes two or more lists and returns a list of tuples, where the *i*th tuple contains the *i*th element from each of the argument lists:

```
>>> a
[1, 2, 3]
```

```
>>> b
[4, 5, 6]
>>> zip(a,b)
[(1, 4), (2, 5), (3, 6)]    # list of tuples
```

It is possible to unzip a zipped tuple using the * operator:

```
>>> zipped = zip(a,b)
>>> a2, y2 = zip(*zipped)
>>> a2
(1, 2, 3)
>>> b2
(4, 5, 6)
>>> list(a2)
[1, 2, 3]
>>> list(b2)
[4, 5, 6]
```

The `zip()` function is commonly used in list comprehensions when two or more lists are involved in the expression:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> [i*j for i,j in zip(a,b)]
[4, 10, 18]
```

### 4.2.5 Lists caution #1: copying lists

An assignment statement such as `b = a` in Python does not make independent copies of lists. Instead, after the statement `b = a`, both `a` and `b` refer to the same list object, so if we change an element in `b` we have also changed `a`!

```
>>> a = [1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> b[2] = 10
>>> b
[1, 2, 10]
```

```
>>> a
[1, 2, 10]
>>> a is b  # tests if a and b refer to the same object
```

To make a copy of a list that does not refer to the same object, you can use `b = list(a)` or `b = copy.copy(a)` for simple lists[3]:

```
>>> a = [1, 2, 3]
>>> b = list(a)
>>> b[2] = 10
>>> b
[1, 2, 10]
>>> a
[1, 2, 3]
>>> import copy
>>> b = copy.copy(a)
>>> b[2] = 10
>>> b
[1, 2, 10]
>>> a
[1, 2, 3]
```

There may be situations where your list itself contains other objects like lists or class instances. If you have such a situation, you can use `new_list = copy.deepcopy(old_list)`. See section 9.3 below for more on when you would need to make a deep copy.

### 4.2.6 Lists caution #2: multiplying lists by a constant

When working with sequences (lists or tuples) of numbers we have to be careful. This is one instance where the default behavior of Python will give an unexpected result to those of us familiar with linear algebra. For example, you might have a list of numbers that you would like to have multiplied by a constant and so you might try the following:

```
>>> a = [1, 2, 3]
>>> b = 3 * a
>>> b
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

[3] For more information on the `copy` module, see docs.python.org/2/library/copy.html.

That is probably not what you expected! What you obtained was three copies of `a` concatenated together. You *can* achieve the behavior you want with the following list comprehension:

```
>>> a = [1, 2, 3]
>>> b = [3*i for i in a]
>>> b
[3, 6, 9]
```

This works, and you can use similar constructions for other arithmetic operations, but is a bit cumbersome. In science and engineering disciplines, we are mostly going to be dealing with *arrays* of numbers, which are not included as a core feature in Python. So, let us introduce the package NumPy, which you will probably import at the beginning of nearly all of your codes that work with numerical data.

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 5

# NumPy arrays

## 5.1 Creating and reshaping arrays

We just saw that the default Python behavior for dealing with lists of numbers was not what we really wanted. The NumPy module, however, will give us the tools we need. NumPy's main object is the *array*, which is a table of elements all of the same type, with an arbitrary number of dimensions (or axes) as needed. The number of dimensions or axes is called the *rank* of the array. For example, the coordinates of a single point `x1` in 3D space (for example, `[1.1, 0.2, 0.9]`) is represented by an array of rank 1 and that axis has a length of 3. An array `xpos` that holds the coordinates of four point masses might be given by

```
>>> xpos    # 2D array holding x,y,z positions
array([[ 0.1,  0.2,  1. ],
       [ 1.1, -0.3, -0.2],
       [ 3. , -3.1,  2.2],
       [-2.2,  1.1,  0.1]])
```

This is a 2D (rank 2) array. In this example, the first dimension (axis) has a length of 4 and the second has a length of 3.

We can access rows, columns and individual elements in the array just as we did for lists using indexing and slicing:

```
>>> xpos[0]
array([ 0.1,  0.2,  1. ])       # first row
>>> xpos[:,2]
```

doi:10.1088/978-1-6270-5620-5ch5  

```
array([ 1. , -0.2, 2.2, 0.1])  # last column
>>> xpos[2,0]
3.0
```

The recommended standard method for importing the `numpy` module is

```
>>> import numpy as np
```

The `numpy` array class is called `ndarray`. You can create an array from an existing (numerical-only) list `a` using

```
>>> x = np.array(a)
```

While it may be more convenient to import everything from the `numpy` module,

```
>>> from numpy import *
>>> x = array(a)
```

to accomplish the same task, experienced Python programmers usually advise against this, as it introduces many other definitions into your current namespace and this can potentially cause conflicts. For example, if you first import another module that also defined an `array` object and then used `from numpy import *` the new definition would override the other. The use of qualified names (e.g. `np.array()`) helps to avoid such collisions and also helps make clear where those definitions are coming from.

You can easily find and change the attributes of your arrays using methods of the class

```
>>> import numpy as np
>>> a = np.arange(6)      # NumPy arange returns an array object
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a = a.reshape(2,3)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.shape
(2, 3)                    # note: this returns a tuple
>>> a.ndim
2
>>> a.size
6
```

Arrays must be homogeneous, meaning all values have the same data type. So if you enter a mix of integers and floating point numbers the integers will be converted to floating point values and if there is a single complex number input all entries will be converted to complex numbers:

```
>>> a = np.array([1, 2, 3.])
>>> a
array([ 1., 2., 3.])
>>> a = np.array([1, 2, 3+3j])
>>> a
array([ 1.+0.j, 2.+0.j, 3.+3.j])
```

### 5.1.1 NumPy `arange`

Note that `arange()` works like the function `range()` we saw in section 3.1 above, but `arange()` returns an array instead of a list. As with `range` you specify at a minimum the `stop` value. You may not want your sequence to start at zero, or you may want your sequence to be floating point values and/or you may want to count down instead of up. As with `range()` these requirements are easy to accomplish:

```
>>> np.arange(1, 5)
array([1, 2, 3, 4])
>>> np.arange(5, 1, -1)
array([5, 4, 3, 2])
>>> np.arange(0, 1.0, 0.2)            # but see note below
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

### 5.1.2 NumPy `linspace`

As a general rule, it is not a good idea to use `arange()` with floating point values, because uncertainties in how the floating point precision will round the numbers means there may or may not be an extra value tagged on at the end (see section 5.6). It is much better practice to use the `numpy.linspace()` function that takes as an argument the number of elements to return, instead of the desired step size. Also, unlike `range()` or `arange()`, the function `linspace()` fills the array to include both end points. If this is not what you want you can pass `endpoint=False` as a keyword argument:

```
>>> np.linspace(0,1,5)
array([ 0.  ,  0.25,  0.5 ,  0.75,  1. ])
>>> np.linspace(0,1,5,endpoint=False)
array([ 0. ,  0.2,  0.4,  0.6,  0.8])
```

### 5.1.3 Other array creation methods

The array type can be declared explicitly when the array is created:

```
>>> a = np.array([1,2,3],dtype=complex)
>>> a
array([ 1.+0.j,  2.+0.j,  3.+0.j])
```

It may be useful in some situations to initialize the array to be filled with 1s or 0s (ones or zeros). These will fill with floating point values by default, but you can specify them to be other data types if needed. Note that the parameter specifying the shape is a tuple. There also exists the NumPy `empty()` function which initializes an array without initializing the values of the array elements:

```
>>> np.zeros(6)
array([ 0., 0., 0., 0., 0., 0.])
>>> np.zeros((2,3))
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> np.zeros((2,3),dtype=int)
array([[0, 0, 0],
       [0, 0, 0]])
>>> a = np.ones((3,2))
>>> a
array([[ 1., 1.],
       [ 1., 1.],
       [ 1., 1.]])
>>> a.shape
(3, 2)
```

## 5.2 Basic operations with arrays

Let us have a look at how to work with NumPy arrays. Arithmetic operations are performed *elementwise* and a new array is created and populated with the result:

```
>>> import numpy as np
>>> a = np.arange(10,50,10)
>>> a
array([10, 20, 30, 40])
>>> b = np.arange(4)
>>> b
```

```
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([10, 19, 28, 37])
>>> b**2
array([0, 1, 4, 9])
>>> 5*np.sqrt(a)
array([ 15.8113883, 22.36067977, 27.38612788, 31.6227766 ])
>>> a < 25
array([True, True, False, False], dtype=bool)
```

Note you could use the final operation to create a *mask* for another operation.

The * product operator operates elementwise in NumPy arrays—it is not standard matrix multiplication[1]. To obtain a matrix product use the `dot()` function. When using NumPy arrays, for example,

```
>>> a = arange(4.).reshape((2,2))
>>> a
array([[ 0., 1.],
       [ 2., 3.]])
>>> b = a + 2
>>> b
array([[ 2., 3.],
       [ 4., 5.]])
>>> a * b
array([[ 0., 3.],       # elementwise multiplication
       [ 8., 15.]])
>>> np.dot(a,b)
array([[  4., 5.],      # matrix multiplication
       [ 16., 21.]])
```

where the last statement `dot(a,b)` gives standard matrix multiplication if `a` and `b` are 2D arrays. If instead `a` and `b` are 1D arrays (i.e., *vectors*) then `dot(a,b)` returns the standard inner product of the vectors (without complex conjugation). The function `cross(a,b)` returns the cross product of vectors `a` and `b`:

```
>>> xhat = np.array([1., 0., 0.])
>>> yhat = np.array([0., 1., 0.])
>>> np.cross(xhat,yhat)
```

[1] MATLAB, for example calculates the matrix product when the * operator is used.

```
array([ 0.,  0.,  1.])

>>> a = np.array([1., 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.dot(a,b)
32.0
>>> np.cross(a,b)
array([-3.,  6., -3.])
```

You may sometimes want to perform operations and overwrite the original array:

```
>>> a = np.ones((3,2))
>>> b = np.arange(6).reshape(3,2)
>>> a
array([[ 1., 1.],
       [ 1., 1.],
       [ 1., 1.]])
>>>b
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> b += 1
>>> b
array([[1, 2],
       [3, 4],
       [5, 6]])
>>>b += 3*a         # Note: array a is converted to integer type here!
>>>b
array([[4, 5],
       [6, 7],
       [8, 9]])
```

It will often be useful to find the minimum or maximum value of an array. The array class provides methods `max()` and `min()` that return these. Having already imported `numpy`, we will use the `random.random()` function to return a list of pseudo-random numbers in the half-open interval [0.0, 1.0) with a uniform distribution[2]:

```
>>> a = np.random.random((3,2))
>>> a
array([[ 0.93143352, 0.23216296],
       [ 0.4620674 , 0.68159093],
       [ 0.6383356 , 0.58551648]])
```

[2] If you run this example, your numbers will differ from what is shown.

```
>>> a.min()
0.23216295741014037
>>> a.max()
0.93143351615165648
>>> a.sum()
3.5311068835151014
```

It might be that you need the maximum or minimum of a given row or column, in which case you could specify which axis to search:

```
>>> a = np.array([0, 1, 3, 2, 4, 5]).reshape(3,2)
>>> a
array([[0, 1],
       [3, 2],
       [4, 5]])
>>> a.min(axis=0)
array([0, 1])
>>> a.max(axis=1)
array([1, 3, 5])
```

### 5.2.1 Copying arrays

As when copying lists as discussed above, the statement `y = x` does not make an independent copy of `x` in the variable `y`—they both point to the same object. If we are working with `numpy` arrays and need to make a copy of an array, we use `.copy()` or `y = np.array(x)` as in

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])
>>> y = x.copy()
>>> y
array([0, 1, 2])
>>> y[2] = 10
>>> y
array([ 0, 1, 10])
>>> x
array([0, 1, 2])

>>> # or
>>> y = np.array(x)
>>> y is x            # tests if x and y refer to the same object
False
```

If you simply use `y = x`, then changing any element of `y` also changes the corresponding element in `x` since they both refer to the same object—be careful out there.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])
>>> y = x
>>> y is x
True
>>> y[2] = 10       # this changes x[2] as well
>>> y
array([ 0, 1, 10])
>>> x
array([ 0, 1, 10])
```

## 5.3 Dictionaries

The Python *dictionary* object provides a very flexible means of storing information. Perhaps you have a list that has the mass densities in units of g $cm^{-3}$ for selected substances:

```
rho = [1.000, 0.93, 2.7]
```

For this to be useful we need to know what substance each of these list items represents. This is where a dictionary can be useful. A dictionary object can be created as follows using curly brackets {} and *key-value* pairs (or simply *items*) each separated by a colon `:` where, in our example, the substance name is the key:

```
>>> rho = {'water':1.000, 'ice':0.93, 'aluminum':2.7}
>>> print rho
{'water': 1.0, 'aluminum': 2.7, 'ice': 0.93}
```

Note that the printed order of the key-value pairs is not the same as what we input, because the information is stored as a *hashtable*. If you enter the same statements on your computer, the order may be different from that above. This behavior is not a problem because the dictionary is not accessed using an index as a sequence is, but rather by the key.

With the above definition for `rho`, we can retrieve the density of ice using a statement

```
>>> rho['ice']
0.93
```

We can add to the dictionary and print it, and can return the length of the dictionary using `len()`:

```
>>> rho['gold'] = 19.3
>>> print rho
'water': 1.0, 'aluminum': 2.7, 'gold': 19.3, 'ice': 0.93
>>> len(rho)
4
```

We can check whether a dictionary contains an item of interest (or not):

```
>>> 'ice' in rho
True
>>> 'platinum' in rho
False
```

The keys and values can each be extracted into new lists:

```
>>> rho.keys()
['water', 'aluminium', 'gold', 'ice']
>>> rho.values()
[1.0, 2.7, 19.3, 0.93]
```

We can sort a dictionary by the keys in alphabetical order using the `sorted()` function:

```
>>> for substance in sorted(rho):
...     print substance, rho[substance]
...
aluminium 2.7
gold 19.3
ice 0.93
water 1.0
```

Key-value pairs can be removed from a dictionary using `del`:

```
>>> del rho['gold']
>>> print rho
{'water': 1.0, 'aluminium': 2.7, 'ice': 0.93}
```

As with lists and arrays, to make an independent copy of a dictionary use `.copy()`:

```
>>> rho_new = rho.copy()
```

For more information on dictionaries in Python see, for, example chapter 6 of Langtangen H P 2012 *A Primer on Scientific Programming with Python* 3rd edn (Berlin: Springer).

## 5.4 Basic statistics

Statistics are easy to compute using NumPy. The function `np.random.random()` returns an array of pseudo-random numbers in the half-interval [0,1):

```
>>> x = np.random.random(10)
>>> x
array([ 0.55996936, 0.38046019, 0.62875143,
        0.12421172, 0.93876425, 0.80777689,
        0.53750113, 0.73750162, 0.61521331,
        0.65951292])
>>> np.median(x)
0.62198237032837067
>>> np.average(x)       # can specify weights for this
0.59896628192996249
>>> np.mean(x)
0.59896628192996249
>>> x.std()             # standard deviation
0.2148276320926009
>>> x.var()             # normalized with N (not N – 1)
0.046150911510513891
```

## 5.5 Universal functions

The `numpy` module includes *vectorized* versions of the trigonometric and exponential families of functions, for example, `sin()`, `cos()` and `exp()`. These vectorized versions can take arrays as arguments and will return arrays corresponding elementwise to the values of the passed arrays:

```
>>> tau = np.linspace(0,-4,5)  # exponential example
>>> tau
array([ 0., -1., -2., -3., -4.])
>>> efold = np.exp(tau)
>>> efold
array([ 1.        , 0.36787944, 0.13533528, 0.04978707, 0.01831564])
>>>
```

```
>>> a = np.linspace(0,2*np.pi,5)       # sine example, a in radians
>>> a
array([ 0.        , 1.57079633, 3.14159265, 4.71238898, 6.28318531])
>>> b = np.sin(a)
>>> b
array([  0.00000000e+00,   1.00000000e+00,  1.22464680e-16,
        -1.00000000e+00,  -2.44929360e-16])
```

We will use vectorized functions in several examples below.

## 5.6 Precision and round-off error

Before moving on, note that we sampled our `a` array in the example above at the five values [0, $\pi$/2, $\pi$, 3$\pi$/2, 2$\pi$], which should give for the `b` array the values [0, 1, 0−1, 0]. While three of the results were as expected, two of the 'zero' values are instead returned as numbers of order $10^{-16}$. This is the result of *round-off error*, also called representation error. The cause is that on most machines running Python, the mantissas (or 'significands') of floating point numbers are represented as binary numbers with 53 bits of precision, which translates to approximately 16 decimal digit precision. Pi itself is an irrational number, so only the value of zero is exactly zero in binary in our example. The others will be represented internally with binary values that are very close to, but not exactly equal to the desired values. The effects of round-off error can be clearly seen in the following example:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 - 0.3
5.551115123125783e-17
```

Round-off error is a fact of life when programming and is the reason why it is best to avoid comparing floats as equal in conditional statements. The following example code would seem to print the numbers from 0.0 to 0.9 in increments of 0.1 and then stop when t = 1.0. The actual behavior is that the conditional expression `t != 1.0` never tests as `True` and so the loop is infinite. The built-in function `repr()` returns a string containing the full ('official') string representation of an object, whereas the `str()` function returns an 'informal'—potentially less accurate—string representation of the object:

```
>>> t = 0.
>>> while t != 1.0:
...     print repr(t)
...     t += 0.1
...
0.0
0.1
```

```
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
1.0999999999999999
1.2
1.3
.
.
.                              # Infinite loop!
```

Rather than testing for equality, it is much safer to check that you have reached the target value within some *tolerance*. For example, the following code terminates at `0.8999...`, as intended:

```
>>> t = 0.
>>> epsilon = 1.e-6
>>> while abs(t - 1.0) > epsilon:
...     print repr(t)
...     t += 0.1
```

## 5.7 NumPy matrix objects

The NumPy module includes the *matrix* object, which is a subset of the array class. Matrix objects inherit the attributes and methods of array objects, but matrix objects are strictly 2D, unlike arrays which can have any dimension. Matrix objects will be most useful for those performing linear algebra. The primary difference between a `matrix` object and `array` objects is that the multiplication operator `*` yields matrix–matrix, vector–matrix, or matrix–vector multiplication. If you are coming from a MATLAB background, you may find the behavior of the `matrix` objects comfortably familiar.

```
>>> import numpy as np
>>> a = np.linspace(1.,3.,3)       # a is a list object
>>> am_row = np.matrix(aa)         # row matrix
>>> am_row
matrix([[ 1.,  2.,  3.]])
>>> am_row.shape
(1, 3)
```

```
>>> type(am_row)
<class 'numpy.matrixlib.defmatrix.matrix'>

>>> am_col = am_row.transpose()     # or amrow.T
>>> am_col                          # column matrix
matrix([[ 1.],
        [ 2.],
        [ 3.]])
>>> am_col.shape
(3, 1)
```

Matrices can also be created using a MATLAB-like syntax:

```
>>> a = np.matrix(' 2. 3. ; 4. 5.')
>>> a
matrix([[ 2., 3.],
        [ 4., 5.]])
>>> a.T                 # matrix transpose
matrix([[ 2., 4.],
        [ 3., 5.]])
>>> a.I                 # matrix inverse
matrix([[-2.5,  1.5],
        [ 2. , -1. ]])
>>> a * a.I             # matrix multiplication =>the identity matrix
matrix([[ 1., 0.],
        [ 0., 1.]])
```

The `numpy` module contains the `linalg` routines, which are optimized for linear algebra. For example, it is trivial to solve a matrix equation of the form $ax = b$ for vector $x$.

```
>>> a = np.matrix(np.linspace(2.,5.,4).reshape((2,2)))
>>> a                           # another route to matrix a above
matrix([[2.,  3.],
        [4., 5.]])
>>> b = np.matrix([[2,3]]).T     # transpose
>>> b
matrix([[2],
        [3]])
>>> x = np.linalg.solve(a,b)
>>> x
matrix([[-0.5],
        [1. ]])
>>> np.allclose(np.dot(a,x),b)   # check that solution is correct
True
```

If you are working with very large matrices, you should consider instead using the `scipy.linalglinalg` module, because typically SciPy is built using the optimized ATLAS[3] LAPACK and BLAS libraries, which results in very fast linear algebra performance. However, in this case you will need to use the `array` class instead of the `matrix` class.

We have not discussed SciPy up to this point, but it is worth mentioning that essentially everything available in NumPy is also available in SciPy. Often the routines are identical, but when they differ the SciPy routines are usually faster. To quote the SciPy FAQ[4]:

> In an ideal world, NumPy would contain nothing but the array data type and the most basic operations: indexing, sorting, reshaping, basic elementwise functions, et cetera. All numerical code would reside in SciPy. However, one of NumPy's important goals is compatibility, so NumPy tries to retain all features supported by either of its predecessors. Thus NumPy contains some linear algebra functions, even though these more properly belong in SciPy. In any case, SciPy contains more fully featured versions of the linear algebra modules, as well as many other numerical algorithms. If you are doing scientific computing with Python, you should probably install both NumPy and SciPy. Most new features belong in SciPy rather than NumPy.

[3] math-atlas.sourceforge.net
[4] www.scipy.org/scipylib/faq.html

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 6

# File input and output

## 6.1 Reading from a file

### 6.1.1 General form: numbers and text

Up to this point we have been focusing on Python language basics and have primarily been working with the interpreter directly. For actual practical use, however, you will be likely to need a program that will read something from a file, manipulate those data and output the results to a new file (which may be a new data file, or a plot output file, or both).

The function `open()` returns a file object and most often takes two arguments, the first being the filename and the second the *mode* to be used. The mode will be 'r' if reading from the file. If writing to the file, use 'w' if any existing contents are to be replaced or 'a' if appending to the existing contents. The mode 'r+' is available if you will be both reading from and writing to the file. If the mode argument is missing, 'r' is assumed. If a binary file is to be written or read, append a 'b' to the mode (e.g., 'wb', 'rb', or 'r+b'). The 'b' is not required for Unix-based systems (e.g., Mac and Linux), but is for Windows, so include it for platform independence. After reading is completed, close the file with `close()`.

Let us assume we have a file called `data.dat` that contains the lines

```
1.0 black hole
2.0 red   dwarfs
3.0 blue  planets
```

The `read()` command will return the entire file as a single string (including the newline character `\n`) if no argument is passed, or just the number of bytes passed as

doi:10.1088/978-1-6270-5620-5ch6  

an argument, which for example can be useful for reading binary files. The `seek(0)` command resets the current position back to the beginning of the file and `close()` closes the file:

```
>>> f = open('data.dat','r')
>>> f.name                    # returns the filename
'data.dat'
>>> f.read()
'1.0 black hole\n2.0 red  dwarfs\n3.0 blue planets\n'
>>> f.seek(0)                 # reset to beginning of file
>>> f.read(10)                # read and return 10 bytes
'1.0 black '
>>> f.close()
```

The file can alternatively be read using `readlines()`, which returns a list containing the lines in the file:

```
>>> f = open('data.dat','r')
>>> f.readlines()
['1.4 blue planets\n', '3.2 red dwarfs\n',
'2.1 green giants\n', '1.0 'black holes\n']
```

Using `readlines()`, each line is returned as a newline-terminated string and so really needs some additional processing to be useful. If we wanted each element on each line to be stored as an element in a 2D list `mydata[][]`, we might use something like the following code, made executable with `chmod +x read_data.py`.

```
#!/usr/bin/env python


f=open('data.dat','r')                  # open file

mydat = []                              # initialize empty nested list
for line in f.readlines():              # outer for loop
  row=[]                                # init empty list for row items
  for i in line.strip().split():        # inner for loop
    row.append(i)                       # append each item to list 'row'
  row[0] = float(row[0])                # first column is a number
  mydat.append(row)                     # append to list 'mydat'

print mydat
```

When we run the code at the terminal prompt, we obtain

```
% read_data.py
[[1.0, 'black', 'hole'], [2.0, 'red', 'dwarfs'], [3.0, 'blue', 'planets']]
```

This method is both simple and general, and can be used for essentially any kind of file. What this code does is to open the file and then step through line by line using the statement `for line in f.readlines():`. For each line, the newline character is stripped with the string method `strip()`, the elements are split into a list assuming a space for the separation character using method `split()` and the first element of each line is converted to a floating point value using `float()`. Finally, the new rank 1 list is appended onto the end of the nested list `mydat`.

Now, it turns out we can improve on the above code. First, we did not have an explicit `close()` statement in our example. The file will be closed when we exit the program, but in a more complicated code that retrieves data from hundreds of files it is possible that, even with the close statements, the files might not be closed 'quickly enough' and lead to a 'too many files open' error from the OS. If instead we use a `with` block as shown in the following example, then the file is closed immediately after the contents are retrieved. Using a `with` block is now considered the preferred method of accessing files. Next, it turns out that `readlines()` is not even needed and can slow down the execution of your code significantly because it results in the entire file being stored in the memory, which can be a problem if your data files are huge. Instead, you can just iterate on the file object itself, as it is already an iterable object! This is memory efficient, fast and yields simpler code.

So, if we wanted to read in a data file that we knew contained (any number of) columns of numbers, we could use the following program (`read_numdata.py`) which makes use of a list comprehension:

```
#!/usr/bin/env python
import sys

mydat = []
with open(sys.argv[1]) as f:
    for line in f:
        mydat.append([float(x) for x in line.split()])

print mydat
```

This program takes the file name from the command line, iterates on the file itself and iterates within the `append` line. Indeed, this can be made even more compact with the use of a nested list comprehension:

```
mydat = []
with open(sys.argv[1]) as f:
    mydat = [[float(x) for x in line.split()] for line in f]
```

and for example if we know there are two columns of data in our file and we want to put these into `numpy` array objects, we could add the lines

```
a = np.array(mydat)
x = a[:,0]              # first column
y = a[:,1]              # second column
```

Now that we have discussed the 'hard' way to accomplish this, let us discuss the easier path that `numpy` affords for reading files containing numerical data.

### 6.1.2 NumPy `loadtxt` and `genfromtxt`

Perhaps the most common situation you will encounter is needing to read a multi-column file of numerical data. NumPy provides the function `loadtxt()` which does exactly this. Let us say we have a two-column file `co2data_1980-2014.txt` containing the mean monthly carbon dioxide $CO_2$ in parts per million (ppm) since 1980 as measured at Mauna Loa, Observatory, Hawaii[1],

```
# decimal   C02
#  date     ppm
1980.042   337.80
1980.125   338.28
1980.208   340.04
   .         .
   .         .
   .         .
2014.792   395.93
2014.875   397.13
2014.958   398.78
```

where you will notice that the first two lines are comments serving as column headers.

[1] www.esrl.noaa.gov/gmd/ccgg/trends/

To read this file, all we need do is

```
>>> import numpy as np
>>> a = np.loadtxt('co2data_1980-2014.txt')
>>> a
array([[ 1980.042,  337.8  ],          # Jan 1980
       [ 1980.125,  338.28 ],          # Feb 1980
       [ 1980.208,  340.04 ],
                  .
                  .
                  .
       [ 2014.792,  395.93 ],
       [ 2014.875,  397.13 ],
       [ 2014.958,  398.78 ]])         # Dec 2014
```

Note that because the first two rows of the file start with the comment symbol # they are ignored by `loadtxt()`. If you have some number of rows at the top of your file that you want to skip but that do not start with #, you can simply include the `skiprows` keyword when calling `loadtxt()`. When using `skiprows` comment lines are included in the count of skipped lines, so it will behave as you want it to, no matter if the rows you want to skip begin with # or not.

For example, given our $CO_2$ data file of monthly averages, if we wanted to not read in the first ten years of data, we would need to skip the two header lines and 120 data lines, for a total of 122 lines:

```
>>> a = np.loadtxt('co2data.txt',skiprows=122)
>>> a
array([[1990.125,  354.88 ],
       [1990.208,  355.65 ],
       [1990.292,  356.27 ],
                  .
                  .
                  .
       [2014.958,  398.78 ]])
```

The actual data file from NOAA we are referring to (`co2_mm_mlo.txt`) contains 682 rows of measurements (at the time of writing) dating back to 1958, where a typical line looks like

```
1980   1    1980.042    337.80    337.80    337.95    30
```

Here column 1 is the year, column 2 is the month, column 3 is the decimal date, columns 4, 5 and 6 are different estimations of the $CO_2$ concentration averages and the last column is the number of days going into the monthly average.

If we want to read this file directly, we can simply read everything with

```
>>> a = np.loadtxt('co2_mm_mlo.txt')
```

and this would give us an array with

```
>>> np.shape(a)
(682, 7)
```

We could then copy the data of interest (the 3rd and 4th columns) to two 1D arrays as follows:

```
x = a[:,2]
y = a[:,3]
```

but `loadtxt()` provides a more direct solution. If we just want the decimal date and the direct average concentration, we can use the keyword `usecols` to specify which of these columns we want to read, where the index starts at zero for the first column. Even more useful, we can `unpack` the data and load them directly into 1D arrays that we can later pass to (for example) the Matplotlib `plot()` functions:

```
>>> x, y = np.loadtxt('co2_mm_mlo.txt',usecols=(2,3),unpack=True)
>>> x
array([1958.208,  1958.292, ... 2014.792, 2014.875, 2014.958])
>>> y
 array([315.71, 317.45, 317.5, ... 395.93, 397.13, 398.78])
```

The `unpack` parameter, if set `True`, transposes the returned array allowing a statement of the form `x, y, z = loadtxt(...)` to be used. The function `loadtxt()` has additional parameters that may be useful in special circumstances, but the above will probably work for most files you will need to read in practice.

If your data file contains missing data and/or if you just want more control over how your data file is read, the NumPy function `genfromtxt()` is a good choice[2].

[2] See docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html.

It can take missing data into account because it loops twice over the data. On the first pass, it converts each line into a sequence of strings and on the second it converts to the appropriate data type.

### 6.1.3 Reading and working with dates and times

Python includes the `time` and `datetime` modules for working with date and time values. If you have a file that includes dates in the ISO 8601 international standard format[3], (e.g. `2015-01-26T16:14:49Z`), you can read this into a string using the general method of section 6.1.1 and then parse the string to a `datetime` object using the following:

```
>>> import datetime
>>> s = "2015-01-27T16:14:49Z"
>>> d = datetime.datetime.strptime(s,"%Y-%m-%dT%H:%M:%SZ")
>>> d
datetime.datetime(2015, 1, 27, 12, 14, 49)
>>> print d
2015-01-27 16:14:49
```

Then you can use methods of the `datetime` module to return useful information or to reformat the date and time using `.strftime()`[4]:

```
>>> print d.strftime("%A, %B %d, %Y")
2015-01-27 16:14:49
```

The `datetime` module can return the current date and time (`datetime.now()`), can calculate the difference between two dates, etc, but this is beyond the scope of this book. For more information, see docs.python.org/2/library/datetime.html and also section 6.1.4 below.

### 6.1.4 Reading files with Astropy

Members of the astronomical research community have contributed the Astropy[5] package, which has the useful general function `read()` which reads in tabular

[3] See, e.g., en.wikipedia.org/wiki/ISO_8601 and perhaps also xkcd.com/1179.
[4] For a list of all the `.strftime()` format codes, see strftime.org.
[5] Visit www.astropy.org. If using Anaconda Python, installation is as simple as typing `conda install astropy` at a terminal command prompt.

data and attempts to guess the format by trying the known supported formats (which include basic ASCII, HTML, LaTeX, CSV, FITS, HDF5 and many others). For example, if you have a file named `co2data.dat` that contains the last three lines of the $CO_2$ data with month columns added as both text and integer:

```
  date      ppm     month mon_i
2014.792  395.93    Oct    10
2014.875  397.13    Nov    11
2014.958  398.78    Dec    12
```

you can read this file and examine the results using the following:

```
>>> from astropy.io import ascii
>>> data = ascii.read('co2data.dat')
>>> data
<Table rows=3 names=('date','ppm','month','mon_i')>
array([(2014.792, 395.93, 'Oct', 10), (2014.875, 397.13, 'Nov', 11),
       (2014.958, 398.78, 'Dec', 12)],
      dtype=[('date', '<f8'), ('ppm', '<f8'), ('month', 'S3'),
             ('mon_i', '<i8')])


>>> print data
  date    ppm   month  mon_i
-------- ------ -----  -----
2014.792 395.93   Oct     10
2014.875 397.13   Nov     11
2014.958 398.78   Dec     12
>>>
```

Note that the `ascii.read()` function was able to determine that the first line contains `names` and that the data types of the four columns were `float`, `float`, `string` and `int`, respectively. The full utility of the `astropy` data table object is beyond the scope of this text, but note that the column names can be accessed via

```
>>> data.colnames
['date', 'ppm', 'month', 'mon_i']
```

and data can be assigned to NumPy array objects using

```
>>> ppm = np.array(data['ppm'])
>>> ppm
array([395.93,  397.13,  398.78])
>>>
```

Tables, like lists, are *mutable* so data in them can be changed in place, and rows and columns can be deleted or added as needed. As noted above, `ascii.read()` from the Astropy library can also read LaTeX tables directly, so if we had found our data in the LaTeX source code of a paper on `arxiv.org` and saved it to a file on our local disk as `co2data.tex`

```
\begin{table}
\begin{tabular}{cccc}
date & ppm & month & mon_i \\
2014.792 & 395.93 & Oct & 10 \\
2014.875 & 397.13 & Nov & 11 \\
2014.958 & 398.78 & Dec & 12 \\
\end tabular
\end table
```

we could read that file using the following and obtain exactly the same table object as we obtained above. A related function discussed below that may be useful to you is `ascii.write()`, which can write your data as a LaTeX table[6].

```
data = ascii.read('co2data.tex')
```

*Reading dates and times with Astropy*

As you might imagine, astronomers are particularly interested in time—it is required for someone who, for example, wants to measure a change in the spin period of a pulsar over decades or the exact arrival time of a gamma-ray burst. So, it is not surprising that the Astropy module contains features that surpass those available in the `datetime` and `time` modules discussed in section 6.1.3 above[7],

[6] In the current example, the table above was created with the command `ascii.write(data, 'co2data.tex',format='latex')`.

[7] See astropy.readthedocs.org/en/latest/time. From the documentation, 'All time manipulations and arithmetic operations are done internally using two 64-bit floats to represent time… [T]he Time object maintains sub-nanosecond precision over times spanning the age of the universe'.

with a specific focus on time formats (e.g., ISO 8601 and Julian date) and time standards (e.g., UTC, TAI, etc) used in astronomy.

Let us assume that we have read the time string in ISO 8601 format with nanosecond accuracy. We can then convert it to an Astropy `Time` object:

```
>>> from astropy.time import Time

>>> s = '2015-01-27T16:14:49.123456789Z'
>>> t = Time(t)
>>> print t
2015-01-27T16:14:49.123456789Z
>>> type(t)
<Time object: scale='utc' format='isot' value=2015-01-27T16:14:49.123>
```

and then can easily print out the equivalent Julian date, modified Julian date, or convert to another time scale. If you do not need this functionality, then jump to the next section, but if you do need this functionality, then something like this module may have been on your wish list for years. See the documentation for more information and send your thanks to the developers.

```
>>> t.byear                 # returns Besselian year
2015.073952510752
>>> t.datetime              # returns datetime object
datetime.datetime(2015, 1, 27, 16, 14, 49, 123)
>>> t.jd                    # returns Julian date (JD)
2457050.1769574475
>>> t.mjd                   # returns modified Julian date (MJD)
57049.67695744742
>>> t.tai                   # returns International Atomic Time (TAI)
<Time object: scale='tai' format='isot' value=2015-01-27T16:15:24.123>
>>> t.tt                    # returns Terrestrial Time (TT)
<Time object: scale='tt' format='isot' value=2015-01-27T16:15:56.307>
>>> t.tdb                   # returns Barycentric Coordinate time (TCB)
<Time object: scale='tdb' format='isot' value=2015-01-27T16:15:56.308>
```

You have control over the precision of the printed output with the `precision` attribute, which gives the number of digits after the decimal point when outputting a value that includes seconds. The default is three and the maximum precision is nine:

```
>>> t = Time('2015-01-27T16:14:49.123456789Z')
>>> print t
2015-01-27T16:14:49.123
```

```
>>> t.precision = 6
>>> print t
2015-01-27T16:14:49.123457
>>> t.precision = 9
>>> print t
2015-01-27T16:14:49.123456789
```

Finally, note that the Astropy module can also read and write files using the binary HDF5 and FITS formats, as we discuss in section 6.2.4 below.

## 6.2 Writing to a file

### 6.2.1 Formatted output

When writing output from your Python codes, the default formatting will often not be formatted attractively. For example, `print` will often output many more digits than are needed:

```
>>> from numpy import pi
>>> print 'pi is approximately',pi
pi is approximately 3.14159265359
```

If you are saving measurements that are only significant to four digits (e.g., 101.2, 3.002, $1.734 \times 10^5$) then the default behavior will not only take up unnecessary disk space, but will be misleading since someone opening that file at a later time would not know the true precision of the measurements from the file contents alone.

*The old way: the string modulo operator*

Before Python 2.6, string formatting was accomplished using the *string modulo operator* '`%`'. This method is still in wide use (and is available in Python 3.x), so you should be familiar with it, but there may come a time when this old style of formatting will be removed from the language.

For example, the format string is on the left side of the string modulo operator `%` and on the right side is a tuple containing the values to be used, in order, in the format string. Note that if only a single item is to the right of the modulo operator no parentheses are required.

```
>>> print "%d %s at $%4.2f each" % (20, 'liters', 1.17234)
20 liters at $1.17 each

>>> print "pi is approximately %6.4f" % pi
pi is approximately 3.1416
```

If you are using Python 3.x, your statement would be

```
>>> print("%d %s at $%4.2f each" % (20, 'liters', 1.17234))
20 liters at $1.17 each
```

If you have programmed in any other languages, the format codes for this method should be easily understandable. The general form is `%[flag]width[.precision]specifier`, where

```
Flags
    - : left justify
    + : a sign character (+ or -) will precede the number
    0 : left pad the number with zeros instead of space

Width
    Minimum number of characters to be printed.

Precision
    For integer specifier (d, i, o, x) the minimum number of digits.
    For e, E and f, the number of digits after the decimal point.
    For g and G, the maximum number of significant digits.
    For s, the maximum number of characters.

Specifiers:
    c       : character
    d or i  : signed decimal integer
    e or E  : scientific notation with e or E
    f       : decimal floating point
    g or G  : use the shorter of e, E, or f
    s       : string of characters
    u       : unsigned decimal integer
```

So, the code `%4.2f` means print a floating point number with a width of four characters in the format `N.NN` (the decimal point counts as one character). The code `%3i` means print an integer with three character spaces, including a sign if negative. There are additional specifiers not listed here for binary, octal and hexadecimal numbers.

Although it might appear that the formatting is part of the `print` function, this is not the case. Instead the string object is acted upon by the modulo operator, which returns another string, and it is this returned string that is passed to the print function:

```
>>> s = "%d %s at $%4.2f each" % (20, 'liters', 1.17234)
>>> print s
20 liters at $1.17 each
```

*The new (Pythonic) way:* `format`
A new method was added in Python 2.6 and is intended to be the default method going forward. If you are new to Python, this is the method you should learn and use in your codes. The general syntax is

```
format_string.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

which perhaps is not terribly clear, so let us convert our examples from above to the new method:

```
>>> print "{0:d} {1} at ${2:4.2f} each".format(20, 'liters', 1.17234)
20 liters at $1.17 each
>>> print "pi is approximately {0:6.4f}".format(pi)
pi is approximately 3.1416
```

As before with the string modulo operator, we again have a format string on the left which has fields that will be replaced, however, here we indicate these fields with curly brackets {}. The curly brackets and any format codes within will be replaced by the formatted value of one of the arguments to the `.format()` object. In the examples above, the positional arguments `{0}`, `{1}` and `{2}` were explicitly stated, along with format codes. If the arguments are in the same order as you want things printed, then you can leave them out. Similarly, if you do not care about the exact formatting of the arguments, you can also leave out those specifiers:

```
>>> print "{} {} at ${} each".format(20, 'liters', 1.17234)
20 liters at $1.17234 each
>>> print "pi is approximately {:6.4f}".format(pi)
pi is approximately 3.1416
```

However, if you want to use the arguments in a different order, or if you want to use an argument more than once, then you do need to specify the positional parameters

```
>>> print "First arg: {0} ... Second arg: {1}".format('1st','2nd')
First arg: 1st ... Second arg: 2nd
>>> print "Second arg: {1} ... First arg: {0}".format('1st','2nd')
Second arg: 2nd ... First arg: 1st
>>> print "{0}, {0}, {0}, {0}".format('Spam!')
Spam!, Spam!, Spam!, Spam!
```

You may have noticed that the general syntax for `.format()` allowed keyword arguments. This feature could be quite useful for complicated print statements, as it makes it easier to map from the arguments back to the format string:

```
>>> print "{s} {v} a {obj}!".format(obj='shrubbery', s='We', v='want')
We want a shrubbery!
```

Fields can be left-, right-, or center-aligned if needed:

```
Option  Effect
'<'    Field is left-aligned within specified space (default for
           strings).
'>'    Field is right-alighted within specified space (default for
           numbers).
'='    Valid for numeric types. Force zero padding after sign but
           before digits.
'^'    Field is centered within the available space.
```

Here is a slightly more complex example where we have a list we would like to print centered one item per line between vertical bars. We use the `join()` method from section 3.1 above, using the newline character `\n` as the separator and iterating over the elements in the list:

```
>>> motto = ['Keep', 'Calm', 'and', 'Python']
>>> print '\n'.join('|{:^8s}|'.format(k) for k in motto)
|  Keep  |
|  Calm  |
|  and   |
| Python |
```

*Printing unicode characters*
In science and engineering, you often want to output the results of a fit to data and the fit parameters will have formal errors associate with them, something of the form $a \pm \sigma_a$. You can of course accomplish this with

```
a = 5.34 +- 0.02
```

but if you employ *unicode characters* you can output the '±' to the terminal (or file). The unicode character for the '±' symbol is `\u00B1`. If you include this in your

string with a 'u' in front of the string to tell Python to interpret the string as unicode, you can obtain the result in this form using

```
>>> print u'a = 5.34 \u00B1 0.02'
```

which outputs `a = 5.34 ± 0.02` to the terminal or your file. Note, it may still be preferable to use the '+-' combination depending on your application. A full list of unicode characters is available at unicode-table.com.

*Printing integers with commas*
Finally and somewhat randomly, if you are printing large integers, you might prefer to make them more easily readable for humans. Python lets you use a thousands separator:

```
>>> print ":,"format(123456789)
123,456,789
```

### 6.2.2 Writing text and numbers to a file

As discussed above, we can open a file for writing using `f = open('output.txt','w')`. We can then write to the file using the `f.write()` command which takes as an argument a string optionally modified using a `.format()` statement:

```
>>> f = open('output.txt','w')
>>> f.write('This is the first line of the file.\n')
>>> f.write("{} {} at \${:4.2f} each\n"format(20, 'liters', 1.17234)
>>> f.close()
```

As noted above, it is preferable to use a `with` statement, which automatically closes the file after the enclosed block of code finishes executing:

```
>>> with open('output.txt','w') as f:
...     f.write('This is the first line of the file.\n')
...     f.write("{} {} at \${:4.2f} each\n".format(20, 'liters', 1.17234)
...
```

After executing either of these statement blocks from the interpreter (or a file), your working directory will contain the file `output.txt` with the following lines:

```
This is the first line of the file.
20 liters at $1.17 each
```

If you have numerical data that you want to write in columns to a file, there are several ways you can do this, four of which are shown in the example below (`fwrite_demo.py`). All four give identical output. The first example is the most straightforward and perhaps the first thing you would think of if coming to Python with previous experience in C/C++ or Fortran. Example 2 brings the `for` loop inside a list comprehension, saving one line ('Flat is better than nested'). Examples 3 and 4 both use the `with` statement, saving another line. These use `f.write()` and `f.writelines()`, respectively, where `f.write()` writes a single line to a file and `f.writelines()` writes a sequence of strings to the file. The `writelines()` method requires the entire sequence to be created in memory before writing to the file and so example 4 is less memory efficient than example 3. Therefore, of the examples shown, I recommend example 3 as the best, however, NumPy provides the `savetxt()` function, which is in practice what you will probably use to write columns of numbers to a file.

```
import numpy as np

x = np.arange(5.)
y = x**2

# Example 1
f = open('mydat1.txt','w')
for i in range(len(x)):
    f.write("{} {}\n".format(x[i],y[i]))
f.close()

# Example 2
f = open('mydat2.txt','w')
[f.write("{} {}\n".format(x[i],y[i])) for i in range(len(x))]
f.close()

# Example 3
with open('mydat3.txt','w') as f:
    [f.write("{} {}\n".format(i,j)) for i,j in zip(x,y)]

# Example 4
with open('mydat4.txt','w') as f:
    f.writelines(["{} {}\n".format(i,j) for i,j in zip(x,y)])
```

### 6.2.3 NumPy `savetxt`

Saving an array to a text file is straightforward using the `numpy savetxt()` function. To save your two-column data to a file you could simply enter

```
>>> import numpy as np
>>> x = np.linspace(1,3,3)          # array([1., 2., 3.])
>>> y = x*x                         # array([1., 4., 9.])
>>> np.savetxt('test.out',(x,y))
```

which will output a file containing the `x` values on the first line of the file and the `y` values on the second line of the file, with all values by default printed in exponential format to machine precision, which is not convenient:

```
1.0000000000000000e+00 2.0000000000000000e+00
    3.0000000000000000e+00 4.0000000000000000e+00
1.0000000000000000e+00 4.0000000000000000e+00
    9.0000000000000000e+00 1.6000000000000000e+01
```

We can write our arrays in columns by including `np.transpose((x,y))` in the call to `savetxt()` and we can format our data to the appropriate number of significant figures by including the `fmt` keyword argument.

If using the same `x` and `y` arrays we call `savetxt()` using

```
>>> np.savetxt('mydat1.txt',np.transpose((x,y)),fmt='%5.2f %5.1f')
```

then our output file `mydat1.txt` contains

```
1.00  1.0
2.00  4.0
3.00  9.0
4.00 16.0
```

If you would like to include header or footer lines, you can pass strings to the `header` and `footer` keyword arguments. This example also demonstrates that if you only include a single format specifier, it will be used for all values:

```
>>> hdr = "This is my data file"
>>> np.savetxt('mydat2.txt',np.transpose((x,y)),fmt='%5.1f',header=hdr)
```

Then `mydat2.txt` contains

```
# This is my data file
1.0  1.0
2.0  4.0
3.0  9.0
4.0  16.0
```

If you would like to include multiple comment lines at the top of your file, you can do something similar to the following:

```
>>> hdr = "This is my data file\n"
>>> hdr += "Columns: x y"
>>> np.savetxt('mydat3.txt',np.transpose((x,y)),fmt='%5.1f',header=hdr)
```

Then `mydat3.txt` contains

```
# This is my data file
# Columns: x y
 1.0  1.0
 2.0  4.0
 3.0  9.0
 4.0  16.0
```

### 6.2.4 Astropy `write`

*Writing ASCII tabular data with Astropy*
If you are using the `astropy` module, you may wish to use the included `write()` function instead of `savetxt()`:

```
>>> from astropy.table import Table
>>> from astropy.io import ascii
>>> x = np.array([1,2,3])
>>> y = x**2
>>> data = Table([x,y],names=['x','y'])
>>> ascii.write(data, 'values.dat')
```

The new file `values.dat` will contain

```
x y
1 1
2 4
3 9
```

As noted above, the Astropy `write()` function is very flexible and can also write your data in several useful formats. If you want to write your file with the column headings written as a comment line (so the file could be read directly with `np.loadtxt()`):

```
>>> ascii.write(data, 'values.dat',format='commented_header')
```

then the resulting file `values.dat` will contain

```
# x y
1 1
2 4
3 9
```

*Writing LaTeX files*
If you want to write your data as a LaTeX table,

```
>>> ascii.write(data, 'values.tex',format='commented_header')
```

then `values.tex` will contain

```
\begin{table}
\begin{tabular}{cc}
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9
\end{tabular}
\end{table}
```

*Other ASCII formats*
You can write ASCII files with the formats `no_header`, `html`, `csv`, or one of several other supplied format options. If you want to format your output to keep only a certain precision, use the the `formats` keyword:

```
>>> x = np.linspace(1,2,4)
>>> x
array([1.         ,  1.33333333,  1.66666667,  2.       ])
>>> y = x**2
>>> y
array([1.         ,  1.77777778,  2.77777778,  4.       ])
>>> data = Table([x,y],names=['x','y'])
>>> ascii.write(data, 'values.dat', formats='x':'%4.2f', 'y':'%4.2f')
```

Now the file `values.dat` contains

```
xa ya
1.00 1.00
1.33 1.78
1.67 2.78
2.00 4.00
```

*Writing binary files with Astropy*
The Astropy `table` module can also read and write files in the widely used HDF5 (Hierarchical Data Format)[8] as well as the FITS format[9]. The HDF was designed to store and organize large volumes of numerical data efficiently and flexibly. A significant advantage of using HDF5 files is that because the files are stored in a binary format, there is no loss in precision and the read/write times are significantly improved over ASCII files. A disadvantage is that the file itself is not directly human-readable.

Another significant advantage of HDF5 is that HDF5 files can contain *multiple* tables. This feature could be useful in the very common situation where you need to write snapshots of the state of a computational simulation as it evolves some system through time. Each snapshot can be written as a new table in the file. In the example here, we write two tables to our output file. Note that each table needs a `path`

[8] See www.hdfgroup.org/HDF5.
[9] FITS stands for *Flexible Image Transport System*. The FITS format is widely used by astronomers and although a binary file format, has the advantage that the metadata are included in a human-readable (ASCII) header. See fits.gsfc.nasa.gov/fits_home.html.

defined on the write statement to label the table within the file. The example is contained in the file `apy_write.py`.

```
import numpy as np
from astropy.table import Table
x = np.array([1, 2, 3])
y = x**2

data = Table([x,y], names=['x', 'y'])

data.write('values.hdf5',path='step1',overwrite=True)
x *= 2
y = x**2

data = Table([x,y], names=['x', 'y'])

data.write('values.hdf5',path='step2',append=True)
```

When we execute this code with `% python apy_writehdf5.py`, it creates the new file `values.hdf5`. The default behavior of the `write` function is that if the specified file already exists, the program will exit with an error message. Include `overwrite=True` if you simply want to overwrite the data in an existing file, but use this with caution if your computations take hours or days to complete. The `append=True` allows adding a new table to an existing file.

To read files in HDF5, the `path` keyword must again be specified to retrieve the desired table. If we execute the following code in the file `apy_readhdf5.py`

```
import numpy as np
from astropy.table import Table

data = Table.read('values.hdf5', path='step1')
print 'Step1:\n',data

data = Table.read('values.hdf5', path='step2')
print '\nStep 2:\n',data
```

we obtain the output

```
Step1:
 x   y
--- ---
  1   1
  2   4
  3   9


Step2:
x    y
--- ---
  2   4
  4  16
  6  36
```

There are other available packages for reading and writing HDF5 files in Python, including the h5py package available at www.h5py.org. Quoting from the website, '[t]he h5py package provides a Pythonic interface to the HDF5 binary data format' and there exists the book *Python and HDF5* written Andrew Collette, the lead author of the h5py package. The h5py package provides a lower-level interface to HDF5 files, but may include features you need that the Astropy package does not.

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 7

## Simple programing: flow control

To write more advanced programs, you will need to use basic *flow control* commands, some of which we have used in previous examples. In this chapter, we will first discuss *conditionals*, then `if-elif-else` blocks, `for` blocks, `while` loops and related items. In the following chapter, we will introduce functions. As noted above, Python uses the indentation level to indicate which lines of code belong in a block, obviating, for example, the need for brackets as used in, for example, the C/C++ programming languages or the `end <whatever>` statement used in Fortran and other languages. The convention among Python coders is to indent four white spaces per new block of code.

### 7.1 Conditionals

Python uses the boolean `True` and `False` objects in conditional statements:

```
>>> a = True
>>> b = False
>>> a
True
>>> b
False
>>> type(a)
<type 'bool'>
```

doi:10.1088/978-1-6270-5620-5ch7

The `True` and `False` objects behave as expected with `and`, `or` and `not` boolean logic statements:

```
>>> True and True
True
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
```

In our programs, we often need to test whether some condition is `True` or `False` before executing some block of code:

```
>>> a = 10      # the value 10 is assigned to variable a
>>> a < 20      # test if a < 20
True
>>> a > 20      # test if a > 20
False
>>> a == 10     # test if a is equal to 10
True
>>> a == 20     # test if a is equal to 20
False
>>> a != 10     # test if a is not equal to 10
False
>>> a != 20     # test if a is not equal to 20
True
>>> a <= 20     # test if a is less than or equal to 20
True
>>> a <= 10     # test if a is less than or equal to 10
True
>>> a % 10 == 0  # test if a is divisible by 10 w/o remainder
True
```

## 7.2 `if-elif-else` statements

One of the most basic statement types is the `if` statement to choose between different code blocks depending on the result of a conditional test. For example,

```
>>> a = int(raw_input('Enter an integer: '))
Please enter an integer: 27
>>> if x < 0:
...     print 'Stop being negative!'
... elif x == 0:
...     print 'I got nothin!'
... elif x == 1:
...     print 'One is the loneliest number.'
... else:
...     print "I'm positive!"
...
I'm positive!
```

The statement `elif` is short for *else if* and these, as well as the `else` statement, are optional.

## 7.3 `for` loops

In Python, the `for` statement loops over the elements in a sequence. While that sequence *can* be a sequence of numbers,

```
>>> for i in range(3):
...     print i
...
0
1
2
```

it may be that you want to loop over the elements in a list of strings:

```
>>> a = ['star','galaxy','universe']
>>> for s in a:
...     print s
...
star
galaxy
universe
```

If you need to iterate over the indices of a sequence, you can do so by using the `range()` and `len()` functions:

```
>>> a = ['star','galaxy','universe']
>>> for i in range(len(a)):
...    print i, a[i]
...
0 star
1 galaxy
2 universe
```

You can also accomplish this behavior with the `enumerate()` function:

```
>>> for i, obj in enumerate(a)
...   print i, obj
...
0 star
1 galaxy
2 universe
```

The `enumerate()` function can greatly simplify the situation when you need to have a collection of items and want to know all the unique pairs. For example, to calculate the gravitational potential energy of $N$ point masses, we use the formula

$$U_{\text{grav}} = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} U_{ij} = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{GM_i M_j}{r_{ij}} \tag{7.1}$$

where $G$ is the gravitational constant and $r_{ij}$ is the distance between masses $M_i$ and $M_j$. So given some object `particles` that contains the mass and position vectors for all particles in the system (see section 9.1 below), we could implement this double sum in Python and hence identify all the unique pairs using the `enumerate()` function as in the following example (where we assume $N = 3$ particles):

```
>>> for i, particle1 in enumerate(particles):
...     for particle2 in particles[i+1:]:
...         print particle1, particle2
...
0 1
0 2
```

```
0 3
1 2
1 3
2 3
```

Should you actually want to implement this, you may find it useful to know that you can find the distance between two vector positions using `np.linalg.norm()`:

```
>>> a = np.array([0, 0, 0])
>>> b = np.array([1, 0, 0])
>>> np.linalg.norm(a-b)
1.0
>>> b = np.array([1,1,0])
>>> np.linalg.norm(a-b)
1.4142135623730951
>>> b = np.array([1,1,1])
>>> np.linalg.norm(a-b)
1.7320508075688772
```

## 7.4 `while` statements

An alternative method which can be useful is to loop over a sequence while some condition is `True` and to stop when that condition becomes `False`:

```
>>> i = 0
>>> while (i <=3):
...     print i
...     i = i + 1
...
0
1
2
3
```

The potential problem is that, if we are not careful, the condition might not be met and we then have an infinite loop. Generally, it is safer to use `for` loops.

## 7.5 `break`, `continue` and `pass` statements

Just as in other languages, a `break` statement breaks out of the current loop. In section 3.2 we saw how to use a `while` loop to handle an error exception on user

input, using the statement `break` to break out of the loop. Here is a trivial example using the `break` statement:

```
>>> for i in range(100):
...     print i
...     if ( i == 3):
...         break
...
0
1
2
3
```

The `continue` statement skips the rest of the statements in the current loop block and continues to the next iteration of the loop. The following example program `testwhile.py` demonstrates the use of the `while`, `break` and `continue` statements:

```
#! /usr/bin/env python

while True:
    s = raw_input("What is your favorite color?: ")
    if s == 'blue':
        break
    else:
        print 'Try again.'
        continue
print 'Right. Off you go!'
```

Make the code executable with `% chmod +x testwhile.py` then run

```
% testwhile.py
What is your favorite color?: red
Try again.
What is your favorite color?: green
Try again.
What is your favorite color?: blue
Right. Off you go!
%
```

The `pass` statement is a null operation. It is useful as a placeholder when a statement is required by the syntax of the language, but no code needs to be executed. An example of its use is given in the following chapter.

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 8

## Functions and modules

### 8.1 Introduction: coding best practices

Functions and modules allow us to gather code statements into a single block that can be *called* from anywhere else in the program. Functions are useful for avoiding code duplication and also make the overall program easier to modify and easier to understand. For example, you might write a function that searches through a list of alphabetized names to return a person's phone number. Your first working code might simply perform a linear search, starting at the beginning of the list each time the function is called and comparing item-by-item until the desired name is found—fine for prototyping with only a few hundred names and easy to write. But if this function needs to be called thousands to millions of times and the list holds a million names, you will need to replace it with a function that implements a more efficient sort strategy with the same interface to the calling program. Indeed, it is good programming style to write your program so that, as much as is practical, all tasks are put into functions where each function does one thing and is simple to understand. When writing a larger code than the simple examples we have discussed so far, your overall development time will generally be reduced if you follow some 'best practices' for coding:

1. Know exactly what the overall code, and each function within the code, is supposed to do.
2. Include a comment block at the top that explains the purpose of the function, and the inputs and outputs, and that includes example results from calling the function.
3. Debug as you develop your code. Start with something very simple that works and build in one new feature at a time, always maintaining a working code.
4. Keep your code simple so others can understand it.
5. Use variable names that are descriptive and maintain consistent naming conventions.

doi:10.1088/978-1-6270-5620-5ch8

Some blocks of code will be useful in multiple projects. An example would be the standard math routines (e.g., `log10()`, `sqrt()`, `sin()`, etc.). These of course are so useful that they are part of the distribution of any language. But perhaps your simulation program writes output files in a specific format and you have several different programs that you use to analyze and visualize the results. You *could* cut and paste the lines of code that read your file format from program 1 to program 2, to program 3, etc., but then if you change the output format of your simulation program you have to update all of your other programs. Instead, it is more efficient to put your `read_file()` code into a *module* that you can `import` into any program. Then if you change the file format, you only need to change the code in the module—not in each program separately.

Tim Peters posted the following to the `python-list` on June 4, 1999, with the title *The Python Way*[1]. It has since come to be known as *The Zen of Python* and is an 'easter egg' available at any time using `import this` at the interpreter prompt:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one---and preferably only one---obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea---let's do more of those!
```

## 8.2 Simple Python functions and modules

You can define a function interactively,

[1] Source: www.wefearchange.org/2010/06/import-this-and-zen-of-python.html.

```
>>> def hello(name):
...  print 'Hello, '+name+'!'
...
>>> hello('Sir Robin')
Hello, Sir Robin!
```

but of course you will generally want to save your code in files for later use or simply for a more efficient code development process. A *module* is simply a file that contains one or more function definitions and associated statements.

The simple form of a function definition is

```
def myfunc(arg1,arg2, ..., argN):
  """
  Docstring to describe purpose, arguments, and
  return value(s), if any
  """
  # statements that do something with the arguments

  return result1[, result2, ...]  # return statement optional
```

The function *takes* an argument(s) `arg1` (etc), does something with it (them) and *returns* a result. The returned result is called the *return value*. Note that it is possible for a function to take no arguments and return no explicit result, but when defining a function you must include the empty parentheses after the function name, even if you do not pass any arguments to the function.

The following example shows a common use of the `pass` function introduced in the previous chapter. A valid function definition must have at least one statement following the definition line, so when developing a program or module, you may use the `pass` statement as a placeholder statement for a function you have not yet written (sometimes called a program *stub*), or instead you might opt to include a comment that describes what the function will eventually do:

```
def donothing():
  pass

def donothing2():
  """ Compute the slacker coefficient """
```

Neither of the functions `donothing1()` or `donothing2()` saved in the file `donothing.py` (module name `donothing`) explicitly return anything, but even

functions without an explicit `return` statement still return the object `None`. The object `None` is an object that has its own type—it is not the string `'None'`.

```
>>> import donothing
>>> donothing.donothing()      # does nothing
>>> donothing.donothing2()     # also does nothing
>>> x = donothing.donothing()
>>> print x
None
>>> type(None)
<type 'NoneType'>
```

As another example, here is a function that converts miles per hour (`mph`) to meters per second (`mps`), saved in a file named `conv.py`:

```
def mph2mps(mph):
    """
    Converts miles per hour to meters per second
    """
    mps = mph * 0.44704
    return mps
```

If we import and call this code interactively, the interpreter prints the result to the terminal, but no variables are actually set. In order to set a variable to the result, we have to call with something of the form `x = func_name(arg)`

```
>>> import conv
>>> conv.mph2mps(10.)
4.4704
>>> speedmks = conv.mph2mps(10.)
>>> print speedmks
4.4704
```

The special object `__name__` is set to the function name if imported and to `__main__` if run as a program. For example, consider the single-line module file `namedemo.py`:

```
print "I am",__name__
```

Notice how the output changes between importing the code and running the code from the IPython interpreter with `run` or the shell prompt:

```
% ipython
In [1]: import namedemo.py
I am namedemo
In [2]: run namedemo.py
I am __main__
In [3]: quit
% python namedemo.py
I am __main__
```

This behavior is very useful, because it allows us to include a block of code in a module that only executes if the module is `run` directly, but does not run if the module is imported by another module or the *main* module (the top-level program or when in the interpreter). It is considered good programming practice to include statements that demonstrate the functionality of the module if the module is run directly, using an `if __name__ = '__main__':` statement. For example, let us revisit our conversion module, where we now also include the inverse to our original function and a `main()` function that gives examples of running the functions in the module `conv`:

```
def mph2mps(mph):
    """
    Converts miles per hour to meters per second
    """
    mps = mph * 0.44704
    return mps

def mps2mph(mps):
    """
    Converts meters per second to miles per hour
    """
    mph = mps / 0.44704
    return mph

def main():
    """
    Code to demonstrate the functions in this module
    mph2mps(1) -> 0.44704
    mps2mph(10) -> 22.369
    """
    print "Module conv: convert mph <-> mps\n"
    print "Examples"
```

```
    print '1 mile per hour    = ',mph2mps(1),'meters per second'
    print '10 meters per second = ',mps2mph(10),'miles per hour'

if __name__ == "__main__":
    main()
```

Accessing the module `conv` via `import` and `run` results in the following, where we also demonstrate how to access the *docstrings* using `print module.function.__doc__`:

```
% ipython
In [1]: run conv.py
Module conv: convert between mph <-> mps

Examples
1 mile per hour      = 0.44704 meters per second
10 meters per second = 22.3693629205 miles per hour

In [2]: import conv
In [3]: print conv.mph2mps.__doc__

    Converts miles per hour to meters per second

In [3]: print conv.mps2mph.__doc__

    Converts meters per second to miles per hour

In [4]: print conv.main.__doc__

    Code to demonstrate the functions in this module
    mph2mps(1) -> 0.44704
    mps2mph(10) -> 22.369
```

## 8.3 Functions with keyword arguments

Python also has the ability to call functions with *keyword arguments*. These are useful for two related reasons. First, the value of some variable can be assigned a default value, but that value can be overridden if needed. For example, you may have a function that calculates the free fall time $t$ from a given height and the final speed $v$ at the end of that fall, assuming gravity is constant. The 1D equation for the speed is $v = at$, where we solve the familiar equation

$$y = \frac{1}{2}at^2$$

for time

$$t = \sqrt{\frac{2y}{a}}.$$

So our function definition might be as follows, in a file named `pos_vel_vs_time.py`:

```
def pos_vel_vs_time(y):
    from math import sqrt
    a = 9.8          # m/s^2
    t = sqrt(2*y/a)
    v = a * t
    return t, v
```

which we could run with

```
>>> from pos_vel_vs_time import pos_vel_vs_time
>>> height = 1.0
>>> t, v = pos_vel_vs_time(1)
>>> print "time to fall", height," m = ", t, "s"
time to fall 5 m = 0.4517539514526256 s
>>> print "final velocity = ", v, "m/s"
final velocity = 4.427188724235731m/s
```

We could use a keyword argument to make this slightly more interesting. We would like the default height to be 1 m, and the default acceleration to be the gravitational acceleration at the surface of the Earth, but would also like the ability to use different values for the height and acceleration if desired. We then have for our function

```
def pos_vel_vs_time(y=1.0, a=9.8):
    from math import sqrt
    t = sqrt(2*y/a)
    v = a * t
    return t, v
```

We can now call this function without any arguments and the defaults will be used. We can call it using the keyword arguments and, if using the keyword arguments, the order does not matter:

```
>>> pos_vel_vs_time()
(0.4517539514526256, 4.427188724235731)
>>> pos_vel_vs_time(1, 9.8)
(0.4517539514526256, 4.427188724235731)
>>> pos_vel_vs_time(y=5, a=0.1)
(10.0, 1.0)
>>> pos_vel_vs_time(a=0.1, y=5)
(10.0, 1.0)
```

Keyword arguments are used a great deal in calls to the Matplotlib plotting functions discussed in chapter 10 below.

## 8.4 Functional programming: list comprehension, `lambda`, `map` and `filter`

### 8.4.1 Introduction

Python includes *functional programming* capabilities in addition to *procedural programming* capabilities. Many common programming languages (e.g., C, Fortran, Pascal, BASIC) are procedural, meaning that the program contains a series of computational steps to be completed. Procedures (also variously called routines, subroutines, or functions) can call other procedures, but the net result is a series of programming statements that are completed one after the other. Functional programming is also modular in approach, but functional programming languages tend to de-emphasize or even remove the imperative elements of procedural programming[2]. Python is one of several languages (e.g., C++, Java, Perl, MATLAB, Visual Basic .NET) that are *multi-paradigm*. Here we will briefly introduce the most useful functional programming features of Python.

### 8.4.2 List comprehension and generator comprehension

We introduced *list comprehension* in section 4.2.3 above, but repeat the essentials here for completeness. The general form of a list comprehension is

```
>>> mylist = [expression for a in list if conditional]
```

[2] For a more in-depth treatment of the topics in this subsection, see docs.python.org/2/howto/functional.html.

where the brackets '[' and ']' help remind us that the result will be a list object. A simple example using a list comprehension is

```
>>> motto = 'Keep Calm and Python'.split()
>>> print motto
['Keep', 'Calm', 'and', 'Python']
>>> shout = [i.upper() for i in motto]
>>> for i in shout:
...     print i
...
KEEP
CALM
AND
PYTHON
```

Here is a more complex example using a nested list comprehension that returns a list of prime numbers from 0 to 49. The list `noprimes` contains all the numbers from 4 to 49 that are divisible by 2, 3, 4, … 7 (many of them more than once, e.g., 12 occurs four times). The `primes` list is created by finding the integers between 2 and 49 that are not contained in the `noprimes` list:

```
>>> noprimes = [j for i in range(2, 8) for j in range(i*2, 50, i)]
>>> primes = [k for k in range(2, 50) if k not in noprimes]
>>> print primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

This method is reasonably efficient for finding small primes, but for finding very large primes the above code would quickly fill all available system memory with the `no primes` list. In such a case, a *generator* comprehension would be more appropriate, since generator objects simplify the task of writing iterators and maintain their state between calls. That is, instead of storing the entire list, generators only return one item of the list for each time they are called, so are more efficient than list comprehension when the list is just an intermediate step and does not actually need to be stored.

Here is a trivial example of a generator comprehension statement. Note that the surrounding parentheses indicate the statement returns a generator object:

```
>>> x = (x**2 for x in range(3))
>>> type(x)
<generator object <genexpr> at ...>
>>> x.next()
0
```

```
>>> x.next()
1
>>> x.next()
4
>>> x.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The same behavior is available if we write a generator function saved to `genfunc.py`:

```
def gen_isquared(N):
    for i in range(N):
        yield i*i
```

This can be used as

```
>>> from genfunc import gen_isquared
>>> isquared = gen_isquared(3)
>>> isquared
<generator object gen_isquared at ...>
>>> isquared.next()
0
>>> isquared.next()
1
>>> isquared.next()
4
>>> isquared.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

There is a lot more to say about generators, but they are not really *essential* so are arguably beyond the scope of this book. For more on generators, see www.python.org/dev/peps/pep-0255 and this link from the jeffknupp.com blog.

### 8.4.3 The `lambda` function

The `lambda` function (or `lambda` operator) provides a convenient method to create small unnamed functions that are used in-place, when needed and where created. They are often used in conjunction with the `map()`, `reduce()` and `filter()`

functions, and can improve both the conciseness and readability of code if used well. Lambda functions can take any number of arguments but return just one value as the result of evaluating a single expression. Lambda functions also have their own local namespace and cannot access variables other than those in their parameter list and in the global namespace.

The syntax of a lambda function is

```
lambda [arg 1[, arg2, ..., argN]] : expression
```

For example,

```
>>> mult = lambda x, y: x*y
>>> mult(2,3)
6
>>> mult(3,5)
15
```

The `lambda` function can be useful in situations where a simple function needs to be passed as an argument to an equation solver or minimizer. For example, the `scipy.optimize` module contains many useful optimization algorithms. Here is a simple example using Brent's method as implemented in `minimize_scalar` to find the minimum of the univariate function $f(x) = (x + 4)(x - 2)^2$, which is shown in figure 8.1:

```
>>> from scipy.optimize import minimize_scalar
>>> f = lambda x: (x + 4) * (x - 2)**2
>>> res = minimize_scalar(f, method='brent')
>>> print res.x
2.0
```

Another use of the `lambda` function is for those who use the `Tkinter`[3] or `wxPython`[4] to make GUIs. This example (`lambda_tkinter.py`), written by Michael Driscoll[5], demonstrates how useful the `lambda` function can be in this context. A full discussion of `Tkinter` is beyond the scope of this book, but the key lines in this example are the first `btn22 = ...` and `btn44 = ...` lines. Note that these statements create a `tk.button()` instance and bind to the `printNum()`

[3] Tkinter is the most commonly used GUI programming toolkit for Python, but there are others. See wiki.python.org/moin/TkInter for links to more information and tutorials.
[4] www.wxpython.org
[5] www.blog.pythonlibrary.org/2010/07/19/the-python-lambda

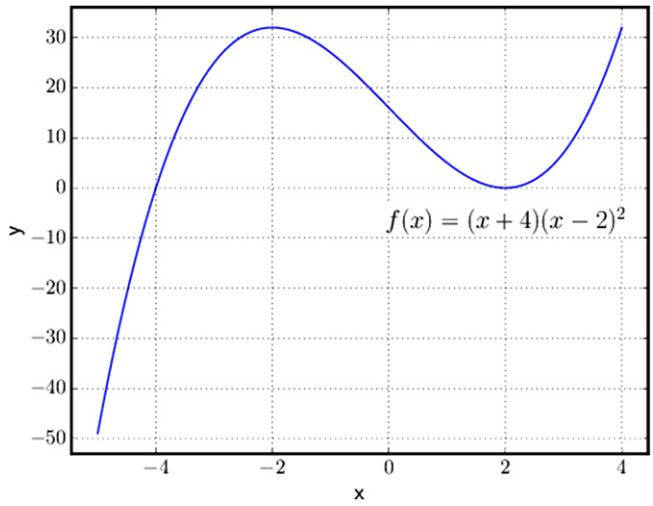


**Figure 8.1.** The function $(x + 4)(x - 2)^2$ has a minimum at $x = 2.0$.

function in a single line. The `lambda` function is assigned to the button's `command` parameter, calling `printNum()`. In this case, the button object is said to 'call back' to the function object specified as its command. It turns out that using the `lambda` function to implement so-called *callbacks* to the `Tkinter` (or `wxPython`) GUI frameworks is one of the most frequent uses of the `lambda` function:

```
import Tkinter as tk

class App:
   """"""

   #----------------------------------------------------
   def __init__(self, parent):
       """Constructor"""
       frame = tk.Frame(parent)
       frame.pack()

       btn22 = tk.Button(frame, text="22",
                         command=lambda: self.printNum(22))
       btn22.pack(side=tk.LEFT)
       btn44 = tk.Button(frame, text="44",
                         command=lambda: self.printNum(44))
       btn44.pack(side=tk.LEFT)
       quitBtn = tk.Button(frame, text="QUIT", fg="red",
                           command=frame.quit)
       quitBtn.pack(side=tk.LEFT)
```

```
    #--------------------------------------------------
    def printNum(self, num):
        """"""
        print "You pressed the %s button" % num

if __name__ == "__main__":
    root = tk.Tk()
    app = App(root)
    root.mainloop()
```

### 8.4.4 The `map` function

The `map()` function has similar functionality to list comprehension. It allows us to map one list onto another using some function. In other words,

```
result = map(function, sequence)
```

calls `function(element)` for each of the sequences elements and assigns the resulting list to the variable `result`.

So recalling our conversion functions, we have the following example that uses `map()` with `lambda` and avoids the function definition for `mph2mps()` altogether:

```
>>> mph = [20, 40, 60, 80, 100]
>>> map(lambda x: x * 0.44704, mph)
[8.9408, 17.8816, 26.8224, 35.7632, 44.704]
```

And this can be reduced to a single line using a list comprehension as follows:

```
>>> map(lambda x: x * 0.44704, [i*20 for i in range(1,6)])
```

The `map()` function can also be applied to more than one list at a time with the proviso that both lists have to have the same length:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> map(lambda a, b: a*b, a, b)
[4, 10, 18]
```

This can also be accomplished with list comprehension, as noted above,

```
>>> [i*j for i,j in zip(a,b)]
[4, 10, 18]
```

### 8.4.5 The `filter` function

The `filter()` function extracts elements from a sequence object for which the function returns `True`. For example, to keep only the positive values of a list,

```
>>> a = range(-4,4)
>>> a
[-4, -3, -2, -1, 0, 1, 2, 3]
>>> filter((lambda x: x > 0), a)
[1, 2, 3]
```

To extend this, we might have a data set for which we would like to remove data points that are more than some number of standard deviations away from the mean, a process called *sigma clipping*. For a true normal (i.e., Gaussian) distribution, we expect 98.7% of the samples to fall between $\pm 3\sigma$. For 1000 numbers drawn from a normal distribution, we expect roughly three samples to fall outside this range. In the following example, we draw 1000 samples from a normal distribution with mean 0.0 and $\sigma = 1.0$, then filter those to return the values that are more than $3\sigma$ from the mean:

```
>>> a = np.random.normal(0,1,1000)
>>> outliers = filter((lambda x: x<-3 or x>3),a)
>>> for i in outliers:
...     print ":5.2f".format(i),
...
 3.03 -3.02 3.12 -3.44
```

The comma at the end of the `print` statement suppresses the default newline. Note that the same result can be obtained with a list comprehension which is arguably easier to read:

```
>>> outliers2 = [i for i in a if i<-3 or i>3]
>>> outliers2 == outliers
True
```

Should you actually need to sigma clip your data, you can use the function `sigmaclip()` from `scipy.stats`. Using the same array `a` from the previous example, `sigmaclip()` will return an array with the four outliers removed. Note that this routine is iterative, so after having removed outliers, the mean and standard deviation of the culled sample are again computed and any outliers removed. This continues until no outliers remain—use with caution:

```
>>> from scipy.stats import sigmaclip
>>> c, low, pup = sigmaclip(a,3,3)
>>> len(c)
96
```

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 9

## Classes and class methods

### 9.1 Introduction

Up to this point, we have used many of Python's built in object types—strings, lists, tuples, etc. The *class* object lets us define our own object type. This is a very powerful and flexible feature, but it is also a bit more involved than what we have discussed so far[1]. As we have discussed, Python supports *object-oriented programming*. Everything in Python is an object, and the program executes by operating on the objects.

For our example, let us consider what object type would be useful for a code that computed the time evolution of a system of point particles subject to physical forces (typically pairwise between the particles). Such a code is called an *N-body* code and these are used widely in physics and astrophysics. For our specific example, let us assume that we have $N$ point masses interacting via the gravitational force[2].

Conceptually, the method is straightforward. A single step in time $\delta t$ in the simulation consists of:

1. Calculate the net gravitational force $\vec{F}_i$ for each particle $i$ resulting from the other $N-1$ particles.
2. Update each particle's velocity using $\vec{v}_i^{\,\text{new}} = \vec{v}_i^{\,\text{old}} + \frac{\vec{F}_i}{M_i}\delta t$.
3. Update each particle's position using $\vec{r}_i^{\,\text{new}} = \vec{r}_i^{\,\text{old}} + \vec{v}_i^{\,\text{new}}\delta t$.

Once the particle positions are updated, the code again calculates the forces and so on.

[1] For an excellent discussion of Python classes and their implementation, see Downey A 2012 *Think Python: How to Think Like a Computer Scientist* (Needham, MA: Green Tea) www.greenteapress.com/thinkpython.

[2] In many $N$-body codes, the masses of all particles are identical, which reduces computation time. For more on $N$-body methods (and more), see Bodenheimer P, Laughlin G P, Różyczka M and Yorke H W 2007 *Numerical Methods in Astrophysics: An Introduction* (Boca Raton, FL: Taylor and Francis).

doi:10.1088/978-1-6270-5620-5ch9

Thus, for each particle, we must keep track of the position, velocity and mass. The way this is typically accomplished if using a sequential programming language is to use seven arrays, each of length $N$, which hold the positions ($x_i$, $y_i$, $z_i$), velocities ($vx_i$, $vy_i$, $vz_i$) and masses ($m_i$) for each of the $i$ particles. However, making use of the object-oriented programming features of Python, we can define a new data type `Particle`. A user-defined type is also called a *class*. We can define this new class and assign an object to it as follows:

```
>>> class Particle(object):
...     """ Represents a particle: position, velocity, mass"""
...
>>> print Particle
<class '__main__.Particle'>
>>> p = Particle()
>>> print p
<__main__.Particle object at 0x...>
```

The new object is called an *instance* of the class and creating the new object is called *instantiation.*

## 9.2 Class attributes

We can assign positions, velocities and masses using dot notation. These elements are called *attributes* of the class.

```
>>> p.x = 3.
>>> p.y = 4.
>>> p.z = 0.
>>> p.vx = 1.
>>> p.vy = 1.
>>> p.vz = 0.
>>> p.mass = 1.0
```

The values of the attributes can be retrieved again with dot notation and we can use them in any valid expression:

```
>>> print p.x
3.0
>>> r = np.sqrt(p.x**2 + p.y**2 + p.z**2)
>>> print r
5.0
```

```
>>> vr = np.sqrt(p.vx**2 + p.vy**2 + p.vz**2)
>>> print vr
1.41421356237
>>> position = np.array([p.x,p.y,p.z])
>>> position
array([3., 4., 0.])
```

An instance can be passed as an argument to a function. For example, we probably want a function that prints the attributes of a particle:

```
def print_p(p):
    print "Position:{} {} {}\nVelocity:{} {} {}\nMass: {}"\
          .format(p.x,p.y,p.z,p.vx,p.vy,p.vz,p.m)
```

Then we would have

```
>>> print_p(p1)
Position: 3.0 4.0 0.0
Velocity: 1.0 1.0 0.0
Mass: 1.0
```

Note that even without having implemented a function to print the values of the attributes of our instance, we can always print all the attributes of our object using the `pprint` module, which 'pretty prints' any Python data structure in a form which could be used as input to the interpreter:

```
>>> from pprint import pprint
>>> pprint(vars(p))
'mass': 1.0, 'vx': 1.0, 'vy': 1.0, 'vz': 0.0, 'x': 3.0, 'y': 4.0, 'z': 0.0
```

Functions can return instances. Here is an example function `collision_inelastic()` that returns the center of mass position and velocity of two particles, with the mass equal to the sum of the two particle masses:

```
def collision_inelastic(p1,p2):
    """ returns center of mass position, velocity, sum(mass) """
    pcm = Particle()
    pcm.m = p1.m + p2.m
```

```
    pcm.x = (p1.m*p1.x + p2.m*p2.x) / pcm.m
    pcm.y = (p1.m*p1.y + p2.m*p2.y) / pcm.m
    pcm.z = (p1.m*p1.z + p2.m*p2.z) / pcm.m

    pcm.vx = (p1.m*p1.vx + p2.m*p2.vx) / pcm.m
    pcm.vy = (p1.m*p1.vy + p2.m*p2.vy) / pcm.m
    pcm.vz = (p1.m*p1.vz + p2.m*p2.vz) / pcm.m
    return pcm
```

When we run this we obtain the expected result and it is an instance of the `Particle()` class:

```
>>> from pprint import pprint
>>> p1 = Particle()
>>> p2 = Particle()
>>>
>>> p1.x = 0. ; p1.y = 0. ; p1.z = 0.
>>> p1.vx = 0. ; p1.vy = 0. ; p1.vz = 0.
>>> p1.m = 1.
>>>
>>> p2.x = 1. ; p2.y = 0. ; p2.z = 0.
>>> p2.vx = 1. ; p2.vy = 0. ; p2.vz = 0.
>>> p2.m = 3.
>>>
>>> pcm = collision_inelastic(p1,p2)
>>>
>>> pprint(vars(pcm))
'm': 4.0, 'vx': 0.75, 'vy': 0.0, 'vz': 0.0, 'x': 0.75, 'y': 0.0, 'z': 0.0
>>> type(pcm)
<type 'instance'>
```

Like lists, class attributes are mutable, so for example we could have a function `accrete()` that adds to the mass of one of our particles:

```
def accrete(p,dm):
p.m += dm
```

When run, assuming `p1` is already an instance of the `Particle()` class,

```
>>> p1.m = 1.
>>> accrete(p1,0.5)
>>> p1.m
1.5
```

## 9.3 Copying and deep copying

To obtain an independent copy of an instance, again use the `copy` module. As with lists, a statement `p2 = p1` has the result that `p2` and `p1` refer to the same object, so changing the value of an attribute of `p2` will change the value of the same attribute in `p1`:

```
>>> p1 = Particle()
>>> p1.x  = 3. ; p1.y = 4.  ; p1.z = 0.
>>> p1.vx = 1. ; p1.vy = 1. ; p1.vz = 0.
>>> p1.m = 1.
>>> p2 = p1     # Both p1 and p2 refer to the same object!
>>> p1 is p2
True
>>> import copy
>>> p2 = copy.copy(p1)

>>> print_p(p1)
Position: 3.0 4.0 0.0
Velocity: 1.0 1.0 0.0
Mass: 1.0
>>> print_p(p2)
Position: 3.0 4.0 0.0
Velocity: 1.0 1.0 0.0
Mass: 1.0
```

Instead of having the position and velocity attributes defined as we did above, we might have opted to create the individual classes `Position` and `Velocity` for each of these vector quantities and then used them within the `Particle` container object:

```
class Position(object):
    """ Represents particle position """

class Velocity(object):
    """ Represents particle velocity """

class Particle(object):
    """ Represents a particle

    attributes: mass, position, velocity """
```

Our initialization would be

```
>>> p1 = Particle()

>>> p1.m = 1.

>>> p1.pos = Position()
>>> p1.pos.x = 3.
>>> p1.pos.y = 4.
>>> p1.pos.z = 0.

>>> p1.vel = Velocity()
>>> p1.vel.x = 1.
>>> p1.vel.y = 1.
>>> p1.vel.z = 0.
```

In this case, if we use `copy` in an attempt to make an independent copy of the object `p1`, we run into a problem. The `pos` and `vel` instances are not copied—they still refer to the original objects, so changing `p2.vel` also changes `p1.vel`:

```
>>> import copy
>>> p2 = copy.copy(p1)
>>> p2 is p1
False
>>> p2.pos is p1.pos    # These refer to the same object.
True
>>> p2.vel is p1.vel    # same here
True
```

In such a case, we must make use of the `deepcopy` function available in the `copy` module. This function copies all levels of objects and so does return a completely independent copy of the object. It is slower than `copy`, but there are times when it is unavoidable:

```
>>>p2 = copy.deepcopy(p1)
>>>p2 is p1
False
>>>p2.pos is p1.pos
False
>>>p2.vel is p1.vel
False
```

## 9.4 Methods

Methods are functions associated with a particular class. For example `upper()` is a method associate with the string class:

```
>>> s = "ni".upper()
>>> print s
'NI'
```

Returning to our original definition of the `Particle` class[3], we can bring our print function into the class definition and make it a print method:

```
class Particle(object):

    def print_p(self):
        print "Position:{} {} {}\nVelocity:{} {} {}\nMass: {}"
             .format(self.x,self.y,self.z,self.vx,self.vy,self.vz,self.m)
```

The convention is to use `self` as the first parameter of a method. Note that a method is invoked using dot notation, as in the `upper()` example above:

```
>>> from particle import *
>>> p = Particle()
>>> p.x = 0. ; p.y = 0. ; p.z = 0.
>>> p.vx = 0. ; p.vy = 0. ; p.vz = 0.
>>> p.m = 1.0

>>> p.print_p()
Position: 0.0 0.0 0.0
Velocity: 0.0 0.0 0.0
Mass: 1.0
```

If we include our `accrete` function as a method, we can invoke as follows, since `p` is assigned to `self` by default:

```
def accrete(self,dm):
    self.m += dm
```

[3] Note: the complete definition of the Particle class as discussed in this chapter is available at the companion website pythonessentials.com in the file `particle_class.py`.

```
>>> p.accrete(0.5)
>>> print p.m
1.5
```

When a class is instantiated, the `__init__` method is invoked, if present. This is where you should set default values for the attributes of your class. For our particle example, we might have inside class `Particle`:

```
class Particle(object):

    def __init__(self, x = 0., y = 0., z = 0.,
                       vx = 0., vy = 0., vz = 0., m = 1.):

        self.x = x
        self.y = y
        self.z = z

        self.vx = vx
        self.vy = vy
        self.vz = vz

        self.m = m
```

Now when we create `p`, the values of the attributes are set, even if we call the method with no arguments:

```
>>> from particle import *
>>> p = Particle()
>>> p.print_p()
Position: 0.0 0.0 0.0
Velocity: 0.0 0.0 0.0
Mass: 1.0
```

We can create `p` using some or all of the arguments. We can call using positional arguments or keyword arguments:

```
>>> p = Particle(1.,2.,3.,4.,5.,6.,7.)
>>> p.print_p()
Position: 1.0 2.0 3.0
```

```
Velocity: 4.0 5.0 6.0
Mass: 7.0

>>> p = Particle(1.,2.,3.)
>>> p.print_p()
Position: 1.0 2.0 3.0
Velocity: 0.0 0.0 0.0
Mass: 1.0

>>> p = Particle(vx=3.,vy=4.,vz=5.)
>>> p.print_p()
Position: 0.0 0.0 0.0
Velocity: 3.0 4.0 5.0
Mass: 1.0
```

The `__str__` method is another special method that is designed to return a string representation of an object. We can modify our `print_p()` function to fill this role. The `__str__` method is invoked when you print an object. Within class `Particle`, we have

```
def __str__(self):
    return   "Position:{} {} {}\nVelocity:{} {} {}\nMass: {}"\
          .format(self.x,self.y,self.z,self.vx,self.vy,self.vz,self.m)
```

```
>>> p = Particle(1.,2.,3.,4.,5.,6.,7.)
>>> print p
Position: 1.0 2.0 3.0
Velocity: 4.0 5.0 6.0
Mass: 7.0
```

You can *overload operators* if needed, such as +, −, ×, ÷. For example if we wanted the + symbol to represent an inelastic collision as discussed above, we could define the method `__add__` to our class:

```
def __add__(self,other):
    """ returns center of mass position, velocity, sum(mass) """

    pcm = Particle()
    pcm.m = self.m + other.m
```

```
    pcm.x = (self.m*self.x + other.m*other.x) / pcm.m
    pcm.y = (self.m*self.y + other.m*other.y) / pcm.m
    pcm.z = (self.m*self.z + other.m*other.z) / pcm.m

    pcm.vx = (self.m*self.vx + other.m*other.vx) / pcm.m
    pcm.vy = (self.m*self.vy + other.m*other.vy) / pcm.m
    pcm.vz = (self.m*self.vz + other.m*other.vz) / pcm.m
    return pcm
```

Then invoking this method is as simple as

```
>>> p1 = Particle(vy=10.)
>>> p2 = Particle(x=1.,m=3.)
>>> print p1
Position: 0.0 0.0 0.0
Velocity: 0.0 10.0 0.0
Mass: 1.0
>>> print p2
Position: 1.0 0.0 0.0
Velocity: 0.0 0.0 0.0
Mass: 3.0
>>> pcm = p1 + p2
>>> print pcm
Position: 0.75 0.0 0.0
Velocity: 0.0 2.5 0.0
Mass: 4.0
```

This ends our discussion of Python class objects but, as you can imagine, we have barely scratched the surface of the discussion of classes or object-oriented programming. For more information, see chapter 9 of *The Python Tutorial* (docs.python.org/2/tutorial/classes.html).

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 10

## Making plots with Matplotlib

It is time to make some plots. The module `matplotlib.pyplot` is a large and growing collection of functions that make Matplotlib work similarly to MATLAB. Each `pyplot` function changes some aspect of the plot area, such as creating the figure and a plotting area within the figure, adding lines, points, text, changing the fonts and axis labels, etc. We will explore several demonstrative examples below, but when it comes time for you to make your own publication-quality plots, I recommend you visit the Matplotlib thumbnail gallery at matplotlib.org/gallery.html for examples that may be more closely related to the vision that you have for your own figures.

### 10.1 Simple line and point plots

Basic plots are almost trivial to create with the `plot()` function. If you pass `plot()` a 1D list of numbers, it assumes the values are $y$ values and the $x$ values are assumed to be integers starting with 0. The default is to connect points with lines. For example, start an IPython shell and enter the commands

```
In [1]: import matplotlib.pyplot as plt
In [2]: plt.plot([1,2,4,7])
In [3]: plt.show()
```

and you should obtain a plot that looks something like figure 10.1.

For a simple point plot (see figure 10.2), the code file (`point_plot_demo.py`) might be

doi:10.1088/978-1-6270-5620-5ch10  

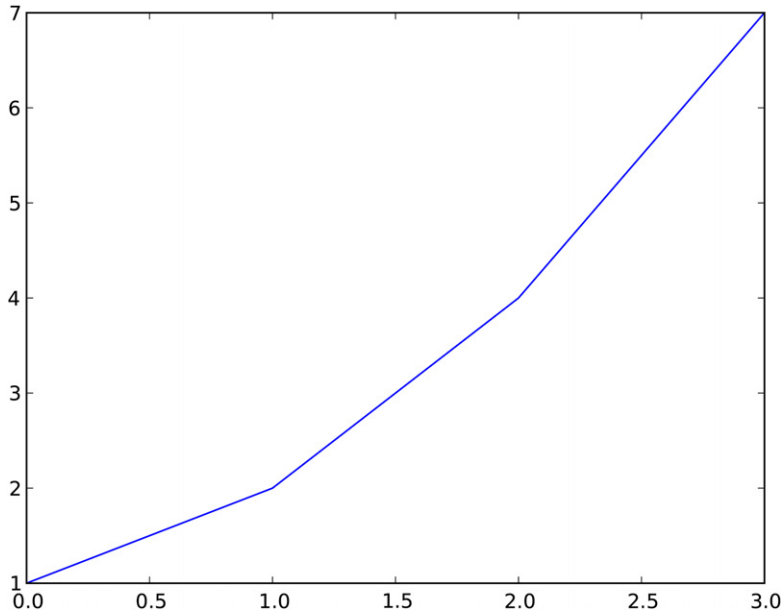


**Figure 10.1.** A simple line plot.

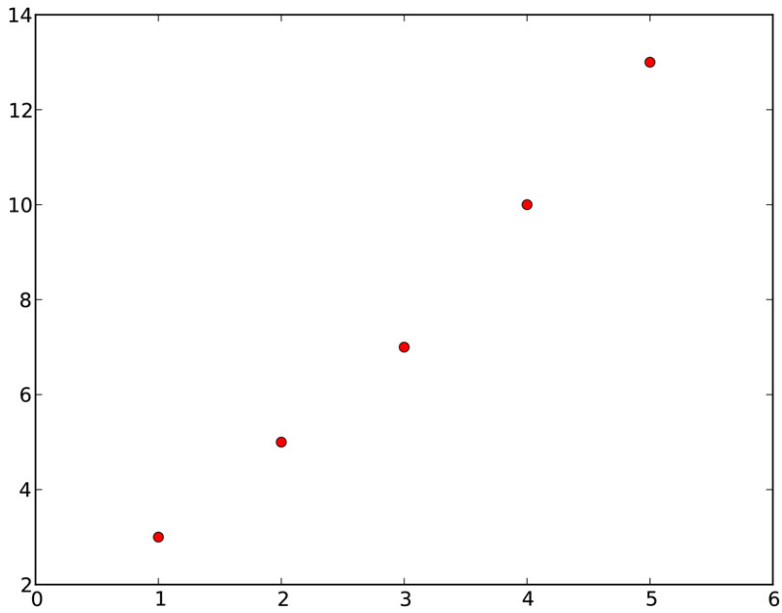


**Figure 10.2.** A simple point plot.

```
# Point plot example
import matplotlib.pyplot as plt
x = [1,2,3,4,5]
y = [3,5,7,10,13]
plt.axis([0,6,2,14])
plt.plot(x,y,'ro')
plt.show()
```

Note that to leave a little white space between the plotted points and the plot frame we have specified the plot axes with the `plt.axis([xmin,xmax,ymin,ymax])` command, which expects a list as an argument (you do not pass the values directly).

The complete list of arguments to `plt.plot()` for setting the line/point type and color is too long to include here, but a useful subset includes

```
Character Description
'-'       solid line
'–'       dashed line
'-.'      dash-dot line
':'       dotted line
'.'       point marker
','       pixel marker
'o'       circle marker
's'       square marker
'^'       triangle marker
'v'       upside down triangle
'*'       star marker
'+'       plus marker
'D'       diamond marker
'd'       thin diamond marker
```

```
Character Color
'b'       blue
'g'       green
'r'       red
'k'       black
'w'       white
'c'       cyan
'm'       magenta
'y'       yellow
```

You can also specify grayscale intensities as a string ('0.6'), or specify the color as a hex string ('#5D0603'). You can also specify the `markersize`.

To plot multiple data sets on the same axis, just call `plt.plot()` multiple times. The axes will autoscale to fit all of the data as shown in figure 10.3, but because `plt.plot()` does not automatically insert white space between the data sets and the axes, you will usually want to tweak your final plot ranges manually, as in the previous example. Note this code (`multiple_data_demo.py`) also demonstrates the use of the `legend()` function:

```
# Multiple datasets on one plot
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(1,19,.4)
y1 = np.log10(x)
y2 = 0.01 *x**2
y3 = 0.9*np.sin(x)
plt.plot(x,y1,'r-',label='y1')
plt.plot(x,y2,'b^',label='y2')
plt.plot(x,y3,'go',label='y3')
plt.plot(x,y1+y2+y3,'+',label='y1+y2+y3')
plt.legend(loc=2)
plt.show()
```

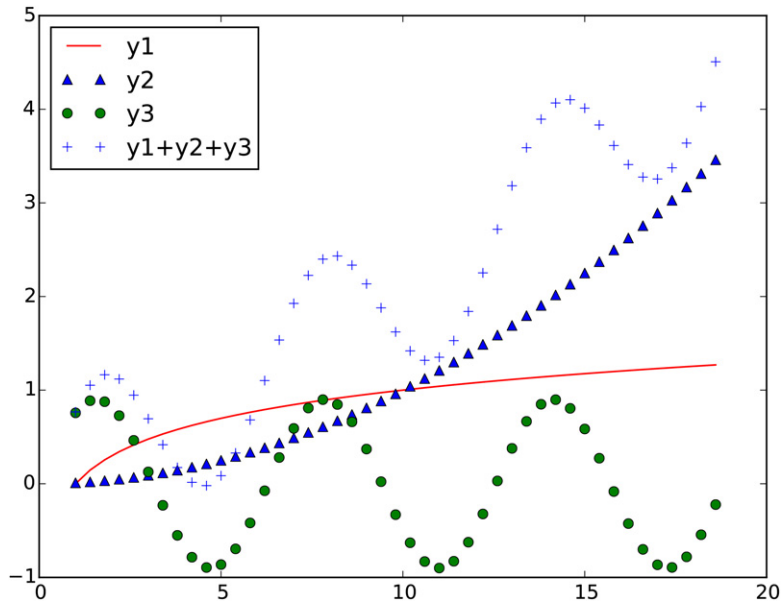


**Figure 10.3.** A figure showing multiple data sets and a legend.

## 10.2 Including error bars

If you are plotting data values, there is a good chance you will also want to include error bars on your plots. Here is an example (`errorbar.py`) using `np.random.randn()` to generate pseudo-random numbers from the *standard normal* distribution—i.e., a Gaussian distribution with mean 0 and variance 1—as well as the `np.random.random()` function that returns values from a uniform distribution in the half-open interval [0.0, 1.0) (see figure 10.4):

```
# errorbar example

import numpy as np
import matplotlib.pyplot as plt

# generate some fake data
x = np.arange(10) + 2*np.random.randn(10)
y = np.arange(10) + 2*np.random.randn(10)

# generate fake errors for the data
xerr = 2*np.random.random(10)
yerr = 2*np.random.random(10)

plt.errorbar(x,y,xerr=xerr,yerr=yerr,fmt='bo')

plt.show()
```

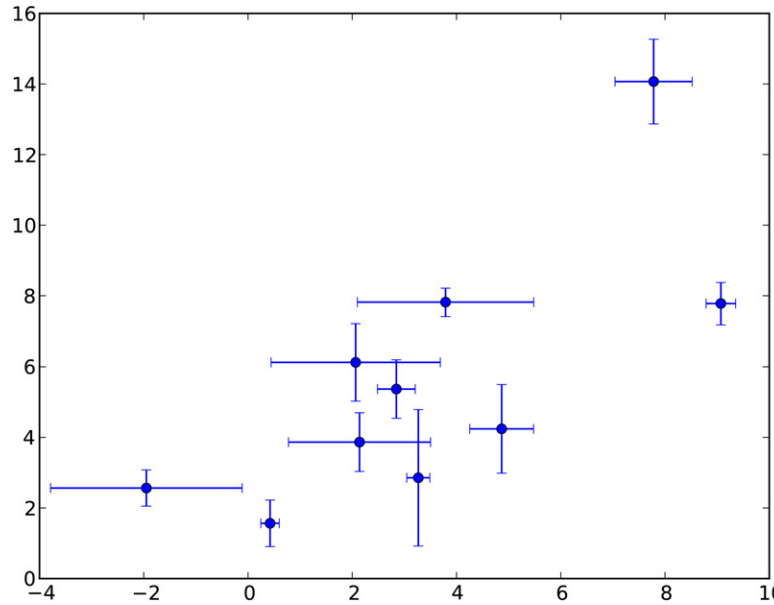


**Figure 10.4.** An example demonstrating the use of error bars.

If you only had error bars in your $y$ values and it was the same value (e.g., $\sigma = 0.2$) for all data points, you could simply use

```
plt.errorbar(x,y,yerr=0.2,fmt='bo')
```

## 10.3 Multiple plots on a page

It will often be useful to have multiple subplots together in a single figure. You might show raw data in a top panel and processed data (with the same $x$ values) below, or a long time series that spans several panels, etc. To show multiple panels within a given figure, you will use the `subplot()` function as in the following example (`subplot_demo.py`; see figure 10.5):

```
# subplot example

import numpy as np
import matplotlib.pyplot as plt

def func(x):
    return np.sin(2*np.pi*x)

x1 = np.arange(0.0,4.0,0.1)
x2 = np.arange(0.0,4.0,0.01)

y1 = func(x1)
y2 = func(x2)
y1n = y1 + 0.1*np.random.randn(len(x1))

plt.figure()

plt.subplot(211)

plt.plot(x1,y1,'bo',x2,y2,'r:')

plt.subplot(212)

plt.plot(x1,y1n,'bo',x2,y2,'r:')

plt.show()
```

Note that here we first call `plt.figure()` to initialize our figure space. This is optional but good practice. We next specify `plt.subplot(211)` where the

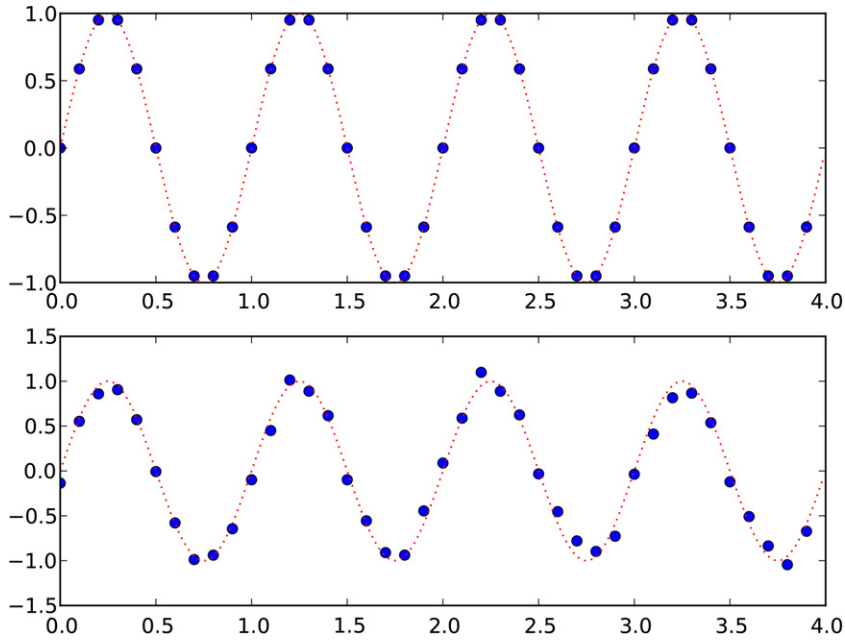


**Figure 10.5.** Demonstrating multiple plots on a page.

argument `211` says make two plots vertically and one horizontally, and we are plotting in the first (top left) plot until we give a new `subplot()` command. If you had, for example, a 2×2 grid of subplots, then the order `221`, `222`, `223` and `224` would be top left, top right, bottom left and bottom right, respectively.

## 10.4 Histogram plots

There are times when you will want to present your data as a histogram. In general you will read in data, and you will know what kind of spacing you need. The routine `hist()` plots histograms in Matplotlib. If you only pass the array of data, the routine will pick the minimum and maximum data values, the spacing and the number of bins to use. Most often, you will want to specify the bin widths and boundaries. The following example (`histogram.py`) shows how you might do this using `randn()` to load an array with random numbers drawn from a normal distribution with a width of 1.0 (see figure 10.6):

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(500)
width = 0.5
histmin = np.floor(min(x))
histmax = np.ceil(max(x))+width
```

```
bins = np.arange(histmin,histmax,width)
plt.hist(x,bins=bins)
plt.show()
```

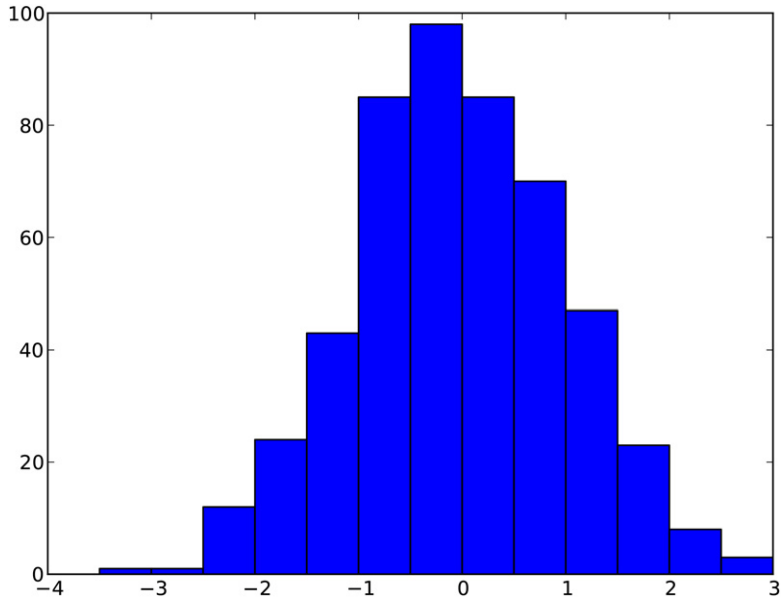


**Figure 10.6.** A simple histogram.

## 10.5 Quick and easy plotting routines for two-column data

Often when working on a project, you will just need a quick way to display two-column data as a line plot. The example below (`pltxy.py`) is a code I keep as an executable in my `~/bin` directory for just such a purpose[1]. This example demonstrates passing the input file name on the command line and also outputs a syntax usage message if the program is called with no arguments. The program adds a bit of white space between the displayed data and the plot frame for aesthetic appeal. If a second command line argument is passed, this is used as the title of the plot:

```
#!/usr/bin/env python
"""
Plot 2 column data file using a line to connect points
"""

import sys
```

[1] I also keep a similar code `poixy.py` that plots points instead of lines.

```
from numpy import loadtxt
import matplotlib.pyplot as plt

if (len(sys.argv) >1):
    infile = sys.argv[1]
    x, y = loadtxt(infile,unpack=True,usecols=(0,1))
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(x,y)
    xrange = max(x) - min(x)
    yrange = max(y) - min(y)
    minx = min(x)-0.05*xrange
    maxx = max(x)+0.05*xrange
    miny = min(y)-0.05*yrange
    maxy = max(y)+0.05*yrange
    ax.axis([minx,maxx,miny,maxy])
    if (len(sys.argv) >2):
        title(sys.argv[2])

    plt.show()
else:
    print "syntax: pltxy <infile>[title]"
```

## 10.6 Customization: text on plots, `rc params` and inset figures

The default fonts are sufficient for many plots, but for making publication-quality plots, you will often want to increase the font size, you may want to use a different font and you may find it useful to include LaTeX commands for special symbols. These changes can be made using `rcParams` as shown in the example below (`inset+label_demo.py`). Inset plots can also be useful on occasion and these are straightforward to add using the `plt.axes()` function. Finally, figures can be saved using the `plt.savefig()` function, with a user-defined value of dots per inch (`dpi`). The example reads in some NASA *Kepler* mission data for the binary star system V344 Lyrae, plots these data as points and then uses inset plots to 'zoom in' and show the character of the light curve at higher resolution (see figure 10.7):

```
import sys
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np

status = 0  # plot label sizes, colors, etc.
xsize=14
ysize=10
```

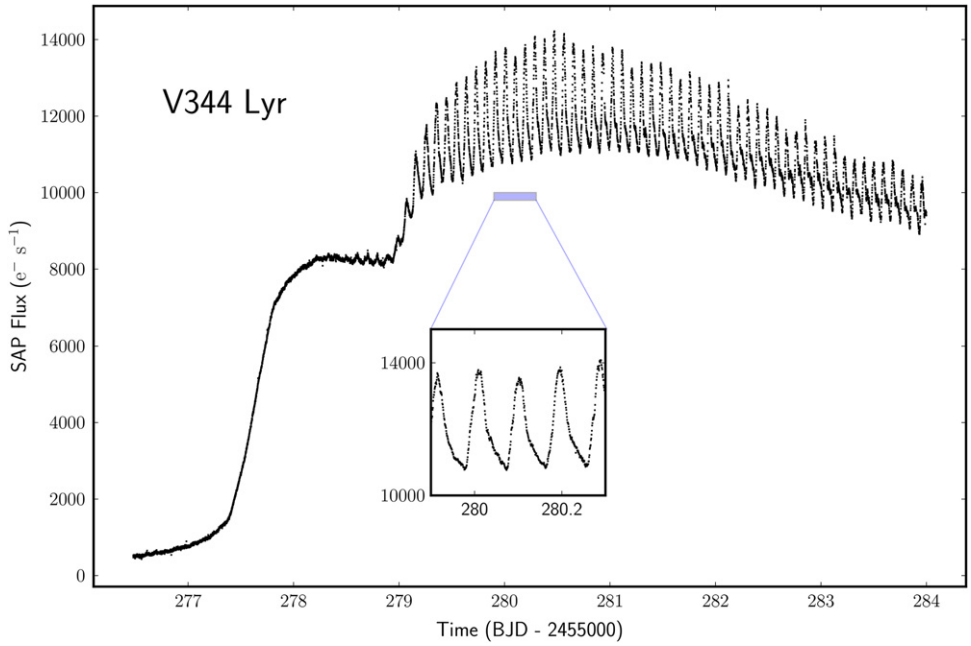


**Figure 10.7.** A plot with inset figures and text.

```
dpi = 300
labelsize = 18
ticksize = 16
lcolor = '#0000ff'
lwidth = 1.0
fcolor = '#ffff00'

plt.rc('text', usetex=True)  # use LaTeX codes for labels
plt.rc('font',**{'family':'sans-serif','sans-serif':['sans-serif']})
plt.params = {#'backend': 'png',
    'axes.linewidth': 2.0,
    'axes.labelsize': labelsize,
    'axes.font': 'sans-serif',
    'axes.fontweight' : 'bold',
    'text.fontsize': 12,
    'legend.fontsize': 12,
    'xtick.labelsize': ticksize,
    'ytick.labelsize': ticksize}
plt.rcParams.update(plt.params)

infile1 = 'V344Lyr_demo.dat'
plotfile = 'inset+label.pdf'
x1, y1 = np.loadtxt(infile1,unpack=True,usecols=(0,1))

fig = plt.figure(figsize=[xsize,ysize])
ax = plt.axes([0.16,0.2,0.75,0.7])
```

```
# first panel
plt.plot(x1,y1,'k,')
xrange = max(x1) - min(x1)
yrange = max(y1) - min(y1)
minx = min(x1)-0.05*xrange
maxx = max(x1)+0.05*xrange
miny = min(y1)-0.05*yrange
maxy = max(y1)+0.05*yrange
plt.axis([minx,maxx,miny,maxy])
plt.xlabel('Time (BJD - 2455000)',labelpad=10)
plt.ylabel(r'SAP Flux ($\rm e^-\ s^{-1}$)')

# Plot Label 'V344 Lyr' at coordinates 0.15, 0.83
plt.text(0.15, 0.83,'V344 Lyr', horizontalalignment='center',
     verticalalignment='center',fontsize=30,
     transform = ax.transAxes)

# Inset
ax.add_patch(Rectangle((279.9,9800),.4,200,alpha=0.3))
plt.plot([279.3,279.9],[6430,9800],'b',alpha=0.3)
plt.plot([280.97,280.3],[6430,9800],'b',alpha=0.3)

ax1 = plt.axes([0.45,0.31,0.15,0.2])
plt.axis([279.9,280.3,10000,15000])
plt.plot(x1,y1,'k,')
plt.xticks([280,280.2],('280','280.2'))
plt.yticks([10000,14000])

plt.savefig(plotfile,dpi=dpi)

plt.show()
```

## 10.7 Image plots with `imshow`

Color maps can be a very effective means of conveying information. The Matplotlib function `imshow` provides the means to work with images. The following example (`imshow_demo.py`) defines an ($x$,$y$) grid using the NumPy `meshgrid` function and then evaluates $z = -\sin(x)\sin(y)$ over that grid. The resulting data are rendered with `imshow` and a color bar is added and labeled. When run, the code produces figure 10.8.

```
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
```

```
fig = plt.figure()
ax = fig.add_subplot(111)

x = np.linspace(0, 1 * np.pi, 200)
y = np.linspace(0, 2 * np.pi, 200)
x, y = np.meshgrid(x,y)

z = -np.sin(x)*np.sin(y)

ax = plt.imshow(z,cmap=cm.jet,origin='lower')
cbar = fig.colorbar(ax)
cbar.ax.get_yaxis().labelpad=15
cbar.ax.set_ylabel('Z Value',rotation=270)

plt.xlabel('X Label')
plt.ylabel('Y Label')

plt.show()
```

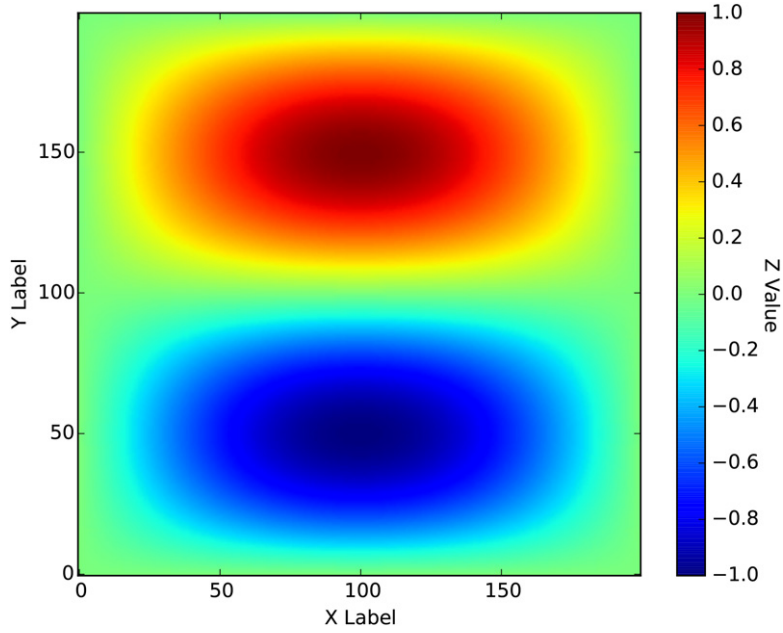


**Figure 10.8.** An `imshow` demonstration plot.

## 10.8 3D plots

### 10.8.1 3D scatter plots

If you have data of three variables you would like to plot in 3D, use functions within the `mplot3d` module[2]. Here is a simple example (`3DPoints.py`) where 200 random points are colored by their distance from the origin:

```
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import matplotlib.cm as cm

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
n = 200

xs = np.random.rand(n)
ys = np.random.rand(n)
zs = np.random.rand(n)
rs = np.sqrt(xs*xs + ys*ys + zs*zs)

ax.scatter(xs, ys, zs, c=rs, marker='o')

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

When this file is run, the result will be something that looks like figure 10.9. The plot shown on the screen is interactive—you can use the mouse (click–drag) to change the viewing angle.

### 10.8.2 3D wireframe and surface plots

Wireframe and surface plots are similarly straightforward to generate. Here is a simple code (`wire3D.py`) demonstrating the generatation of a wireframe plot (figure 10.10):

[2] For more examples, see the *mplot3d tutorial* at matplotlib.org/mpl_toolkits/mplot3d/tutorial.html.

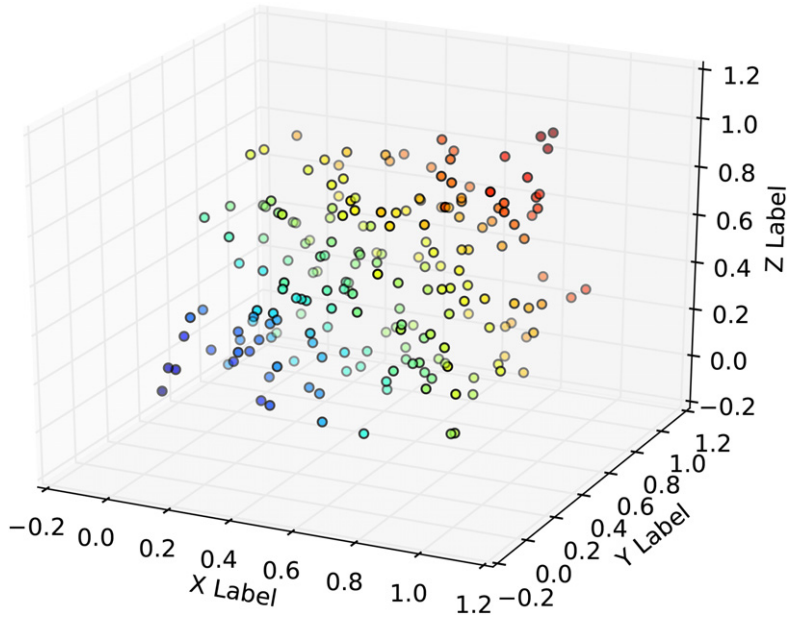


**Figure 10.9.** A 3D scatter plot.

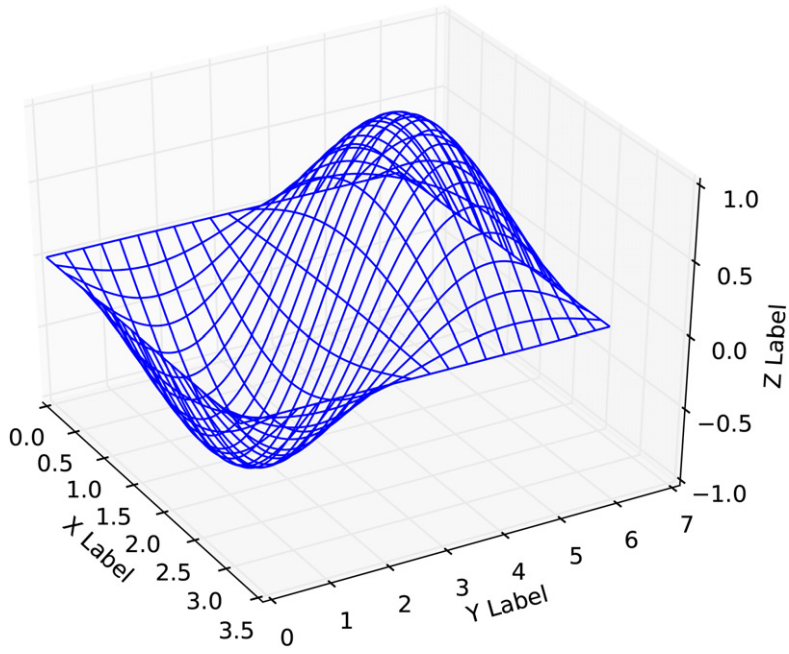


**Figure 10.10.** A 3D wireframe plot.

```
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.linspace(0, 1 *np.pi, 100)
y = np.linspace(0, 2 *np.pi, 100)
x, y = np.meshgrid(x,y)
z = -np.sin(x)*np.sin(y)

ax.plot_wireframe(x, y, z, rstride=5, cstride=5)

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

Note that we use the NumPy function `meshgrid()` in the previous example. This function makes it easy to generate a mesh of $x$–$y$ values from 1D arrays that can then be used in statements that assign values to a 2D $z$ array. For example:

```
>>> x = np.linspace(0., 1., 4)
>>> y = np.linspace(0., 2., 3)
>>> x
array([ 0.        , 0.33333333,  0.66666667, 1.        ])
>>> y
array([ 0., 1., 2.])
>>> x, y = np.meshgrid(x,y)
>>> x
array([[ 0.        , 0.33333333,  0.66666667, 1.        ],
       [ 0.        , 0.33333333,  0.66666667, 1.        ],
       [ 0.        , 0.33333333,  0.66666667, 1.        ]])
>>> y
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.]])
>>> z = x*x + y*y
>>> z
```

```
array([[ 0.        ,  0.11111111, 0.44444444, 1.        ],
       [ 1.        ,  1.11111111, 1.44444444, 2.        ],
       [ 4.        ,  4.11111111, 4.44444444, 5.        ]])
```

For the surface plot, let us step up the complexity just a bit by using a color map to shade the surface according to the $z$ values, making the surface slightly transparent with `alpha=0.7`, adding a contour plot on the base plane and adding a colorbar to the right of the plot (see figure 10.11). The following lines of code replace the line beginning `ax.plot_wireframe(...)` in the wire3D.py example:

```
surf = ax.plot_surface(x, y, z, rstride=4, cstride=4,
                       cmap=cm.coolwarm,alpha=0.7)
fig.colorbar(surf,shrink=0.5)
cset = ax.contourf(x, y, z, zdir='z',cmap=cm.coolwarm,offset=-1)
```

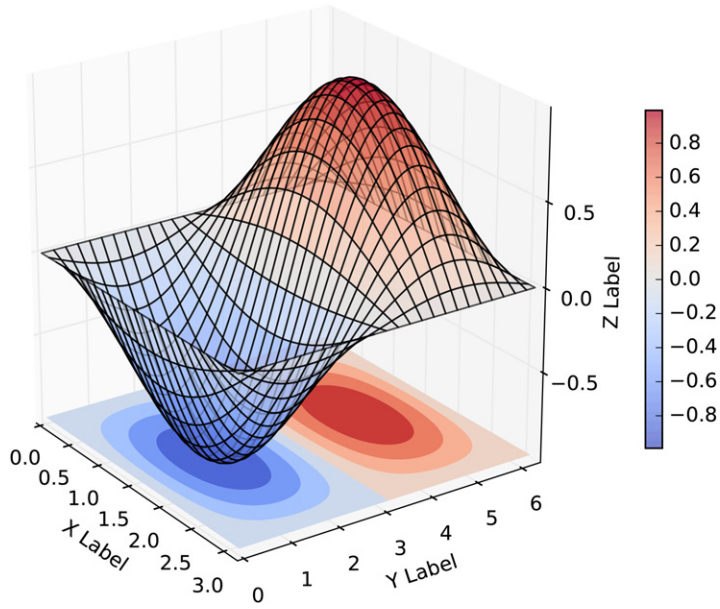


**Figure 10.11.** A 3D surface plot with a contour plot base and semi-transparent surface.

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 11

# Applications

Python has a substantial body of existing packages that can be imported to efficiently solve many classes of problems. In this chapter, I highlight just a few of these that I find particularly useful.

## 11.1 Fits to data

### 11.1.1 Linear least squares: fitting a polynomial

NumPy and SciPy includes several functions for fitting data. The NumPy function `polyfit()` will return the coefficients of the best fit polynomial of degree `n`. The coefficients are given in decreasing powers. In the example below, we create data using a polynomial of order 4 and then fit that with polynomials of order 1 through 4. The example (`polyfit.py`) also makes use of the NumPy `poly1d` function that takes a list of coefficients as argument and allows evaluation of the polynomial:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5., 5., 25)

# y = (0.5 * x**4) + (-4 * x**3) + (3 * x**2) + (-2 * x) + 1

f = np.poly1d([0.5,-4,3,-2,1]) # returns polynomial shown above
y = f(x)
```

doi:10.1088/978-1-6270-5620-5ch11

```
c1 = np.polyfit(x,y,1)  # fit a line (returns coefficients of fit)
c2 = np.polyfit(x,y,2)  # fit a parabola
c3 = np.polyfit(x,y,3)  # fit a 3rd order polynomial
c4 = np.polyfit(x,y,4)  # fit a 4th order polynomial

print "Fit coefficients, 1st through 4th order"
print "{}\n{}\n{}\n{}\n".format(c1,c2,c3,c4)

p1 = np.poly1d(c1)
p2 = np.poly1d(c2)
p3 = np.poly1d(c3)
p4 = np.poly1d(c4)

plt.plot(x,y,'ko',label='Data')
plt.plot(x,p1(x),'g-',label='1st order')
plt.plot(x, p2(x),'b-',label='2nd order')
plt.plot(x,p3(x),'c-',label='3rd order')
plt.plot(x,p4(x),'r-',label='4th order')

plt.legend()

plt.show()
```

When we execute the code, it prints out the fitted coefficients. As expected the final fit returns the original coefficients and a perfect fit (see figure 11.1):

```
In [1]: run polyfit.py
[ -66.86111111  101.27729552]
[ 14.54513889 -66.86111111 -30.03298611]
[ -4.          14.54513889  -2.         -30.03298611]
[ 0.5 -4.   3.  -2.   1. ]
```

### 11.1.2 Non-linear least squares

*SciPy* `optimize.curve_fit()`

Probably more useful than fitting a polynomial is generalized non-linear least squares fitting. Here we will initialize some noisy data around the function $y = a + bxe^{-cx}$. We use the `curve_fit()` function of the SciPy `optimize` module, which actually calls the function `leastsq()` but presents a slightly simpler interface for the programmer. In the example (`curve_fit_demo.py`), we use a `lambda` function, set our weights equal to unity for all points and initialize our guesses for the parameters to be 1.0. The function `curve_fit()` uses the Levenburg–Marquardt gradient (steepest descent) method. It returns the best-fit

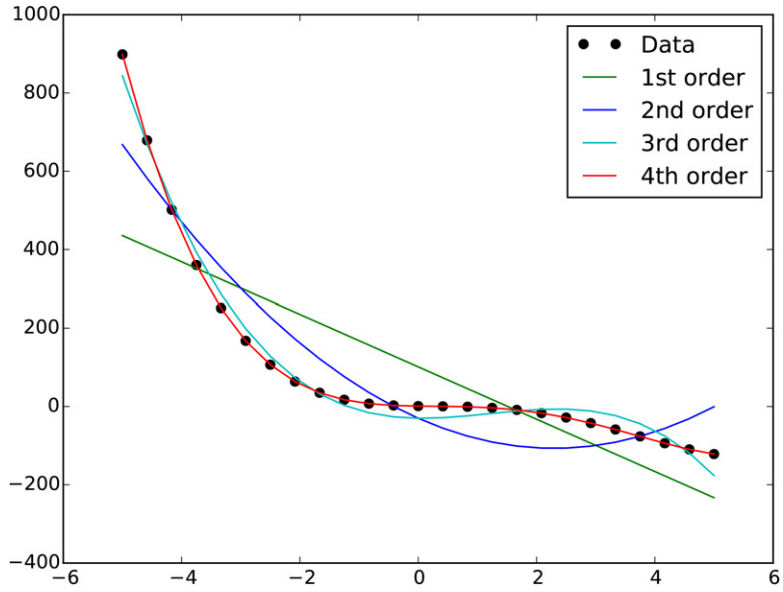


**Figure 11.1.** A polynomial fit example.

values for the parameters and the covariance matrix. The standard errors are given by the square-root of the diagonal elements of the covariance matrix:

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

func = lambda x, a, b, c: a + b*x * np.exp(-c*x)

x = np.linspace(0, 6, 20)
noise = 0.05 * np.random.normal(size=len(x))
y = func(x,1,2,1) + noise

sigma = np.ones(len(x))
p0 = np.ones(3)          # initial guesses for a, b, c

p, pcov = curve_fit(func,x,y,p0,sigma)

perr = np.sqrt(np.diag(pcov))

print 'Best-fit:'
print u'a = {:g} \u00B1 {:g}'.format(p[0],perr[0])
print u'b = {:g} \u00B1 {:g}'.format(p[1],perr[1])
print u'c = {:g} \u00B1 {:g}'.format(p[2],perr[2])
```

```
xfit = np.linspace(0,6,100)

plt.plot(x,y,'ko',label='Data')
plt.plot(xfit, func(xfit,p[0],p[1],p[2]),'b-',label='Fit')
plt.legend()
plt.text(4,1.5,"$y = a + bx e^{-cx}$",fontsize=20)
plt.axis([-0.5,6.5,0.92,1.85])
plt.show()
```

When we run the code, it prints the best fit solution and plots the fit over the generated data (see figure 11.2):

```
In [1]: run curve_fit
Best-fit:
a = 1.05713 ± 0.0149848
b = 1.90339 ± 0.082478
c = 1.04477 ± 0.0332015
```

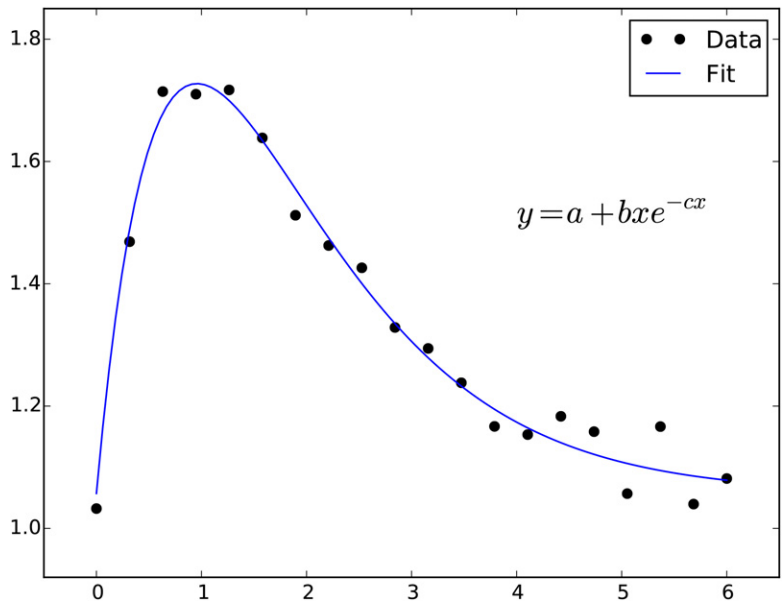


**Figure 11.2.** A linear least squares example with `curve_fit`.

*SciPy* `optimize.leastsq()`

Next we present an example of fitting a sine curve to a generated curve with added noise using the function `leastsq()` (see figure 11.3). In this example (`singen+fit.py`), all of the data have the same weight, but it should be obvious from the code how to included point-by-point weighting if appropriate for your data. The code for this is a bit long.

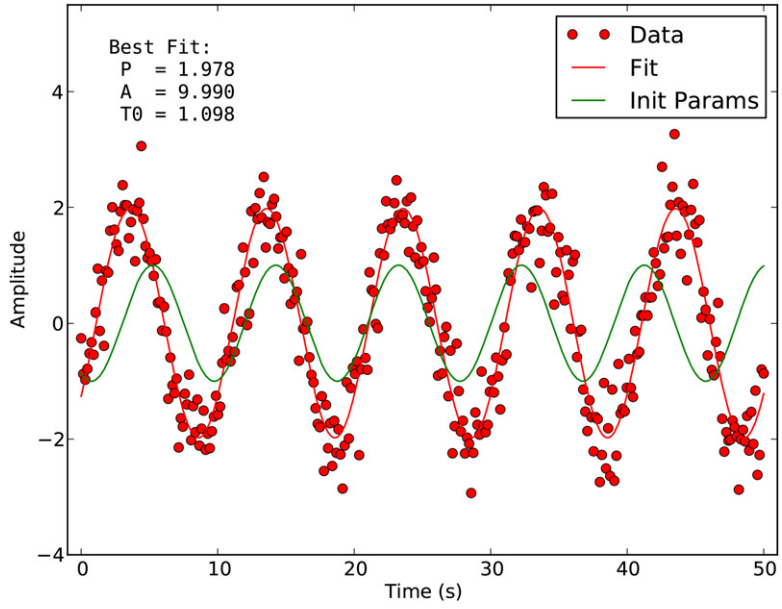


**Figure 11.3.** An example of a non-linear least squares fit to sinusoidal data with noise.

```
#!/usr/bin/env /python
''''''

SinGen+Fit
Generate sine curve with noise, then fit using Levenberg-Marquardt
method.
''''''

import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize
from scipy.optimize import leastsq

# Generate the data
npts = 330
amp = 2.
per = 10.
T0  = 1.0
noise_amp = 0.5
```

```
t = np.linspace(0.,50.,npts)
y = amp*np.sin(2*np.pi/per*(t-T0))+noise_amp*np.random.randn(npts)


fitfunc = lambda p, t: p[1]*np.sin(2*np.pi/p[0]*(t-p[2]))
errfunc = lambda p, t, y: fitfunc(p,t) - y

p0 = [9.,1.,3.] # Initial guesses for fit parameters
yguess = p0[1]*np.sin(2*np.pi/p0[0]*(t-p0[2]))
p1,cov,info,mesg, success = optimize.leastsq(errfunc, p0,args=(t,y),
                                             full_output=True)

chisq=sum(info["fvec"]*info["fvec"])
dof = len(t) - len(p1)
rmsred = np.sqrt(chisq/dof)
print "Converged with chi squared ",chisq
print "degrees of freedom, dof  ",dof
print "sqrt(chisq/dof)      ",rmsred

print "Best Fit, with 1-sigma errors:"
print "P = %.3f +- %.3f" % (p1[0],np.sqrt(cov[0,0])*np.sqrt(chisq/dof))
print "A = %.3f +- %.3f" % (p1[1],np.sqrt(cov[1,1])*np.sqrt(chisq/dof))
print "T0 = %.3f +- %.3f" % (p1[2],np.sqrt(cov[2,2])*np.sqrt(chisq/dof))
plt.plot(t,y,"ro",t,fitfunc(p1,t),"r-")
plt.plot(t,yguess,"g-")
plt.ylim(-4,5.5)
plt.xlim(-1,51)

plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend(('Data','Fit','Init Params'))

ax = plt.axes()
s = 'Best Fit:\n P = %.3f\n A = %.3f\n T0 = %.3f'%(p1[1],p1[0],p1[2])
plt.text(2.0, 3.5,s, family='monospace',fontsize=12)

plt.show()
```

### 11.1.3 Linear systems of equations

Both SciPy and NumPy have linear algebra solvers. The SciPy routines may be faster because they are always compiled with BLAS/LAPACK support, whereas this is optional for NumPy, and can be much faster if SciPy is built using the optimized ATLAS LAPACK and BLAS libraries. Solving a system of linear equations is quite straightforward. Say you have the system of equations

$$
\begin{aligned}
1x + 2y + 3z + 4w &= 9 \\
3x + 4y + 9z + 6w &= 8 \\
8x + 3y + 8z + 1w &= 7 \\
7x + 4y + 3z + 2w &= 7.
\end{aligned}
$$

We can solve this by initializing a nested NumPy array `A` to the coefficients on the left-hand side and `B` to the values on the right-hand side of the equals signs. The code (`lineq.py`) to solve this particular problem is

```
import numpy as np
from scipy import linalg

A = np.array([[1, 2, 3, 4], [3, 4, 9, 6], [8, 3, 8, 1], [7, 4, 3, 2]])
B = np.array([9, 8, 7, 7])

x = linalg.solve(A,B)

print x
```

which when run gives

```
In [1]: run lineq
[ 3.65   -6.5875 -0.975   5.3625]
```

The `linalg` package contains many more linear algebra functions, including functions for solving matrix inversions, banded matrices, triangular matrices, eigenvalue problems, decompositions, etc. Low-level BLAS functions are available using `scipy.linalg.blas` and low-level LAPACK functions are available using `scipy.linalg.lapack`.

## 11.2 Numerical integration

SciPy provides several choices for integration routines in the `scipy.integrate` module. The `quad()` function integrates a function from limits `a` to `b` (which can be $\pm\infty$) using a scheme from the Fortran library QUADPACK. It returns the result of the integral and an estimate of the absolute error in the result. Higher dimensional integrals are available using `dblquad()`, `tplquad()`, and `nquad()` for 2, 3 and $N$ dimensions, respectively. In this example (`quad_demo.py`) we integrate $\int_0^\infty \mathrm{e}^{-2x}\,\mathrm{d}x$ which of course has the analytical solution 1/2:

```
import numpy as np
from scipy import integrate

# integrate exp(-2x) from 0 to np.inf (np.inf = +infinity)
f = lambda x: np.exp(-2.*x)
y, yerr = integrate.quad(f, 0, np.inf)

print y, yerr
```

```
% python quad_demo.py
0.5 7.7350316838e-11
```

## 11.3 Integrating ordinary differential equations

Ordinary differential equations can be solved with `scipy.integrate.odeint()`, which uses LSODA from the Fortran library odepack. Implementation is straightforward, as all that is needed are initial conditions and a function that returns the derivatives. For example, the Lorenz system consists of three coupled ordinary differential equations that model the essentials of atmospheric convection and provide a now-classic demonstration of chaotic behavior in a deterministic system[1]:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x)$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z$$

where $x$, $y$ and $z$ indicate the state of the system, and $\sigma$, $\rho$ and $\beta$ are system parameters.

For our example (`lorenz_demo.py`), the function definition returns the derivatives of the Lorenz equations, the initial position is set in `r0`, the time array is assigned to `t` and the `odeint()` function integrates the trajectory. The result is shown in figure 11.4.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from mpl_toolkits.mplot3d import Axes3D

def func(r, t):
  x, y, z = r

  dxdt = sigma*(y - x)   # The Lorenz equations
  dydt = x*(rho - z) - y
  dzdt = x*y - beta*z
  return dxdt, dydt, dzdt
```

[1] See, e.g., mathworld.wolfram.com/LorenzAttractor.html.

```
r0 = (0.1, 0.0, 0.0)  # Initial position

sigma = 10.0          # Constants
rho   = 28.0
beta  = 8.0/3.0

t = np.linspace(0, 100, 10000)

pos = odeint(func, r0, t)

x = pos[:, 0]
y = pos[:, 1]
z = pos[:, 2]

fig = plt.figure(figsize=(12,10))
ax = fig.gca(projection='3d')

ax.plot(x, y, z)

plt.show()
```

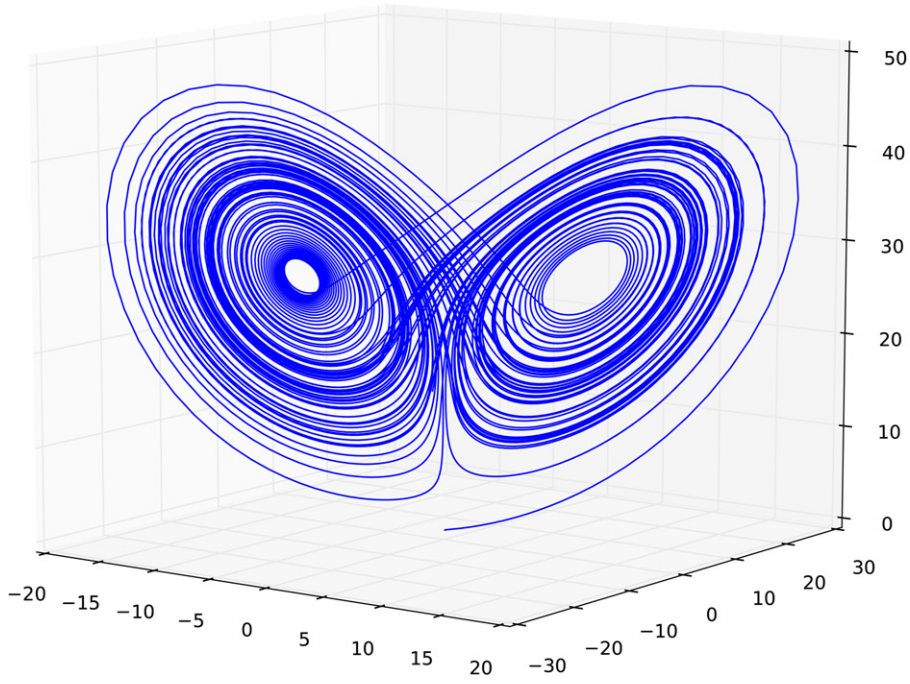


**Figure 11.4.** A 3D plot of the Lorenz attractor.

## 11.4 Fourier transforms

Fourier transform are useful for exploring the harmonic content of time series (or spatial) data[2]. The common convention is

$$F(k) = \int_{-\infty}^{\infty} f(t) \mathrm{e}^{-2\pi \mathrm{i} k t} \mathrm{d}t$$

where $F(k)$ is the Fourier transform of series $f(t)$. When this integral is expressed as a sum suitable for numerical evaluation[3], we have the discrete Fourier transform (DFT)

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left( -\mathrm{i} 2\pi f_k t_i \right)$$

where the coefficients $c_k$ provide an estimate of the power in the time series at the frequency. The direct coding of the above results in the following example, where in addition we have applied the normalization such that an input sine curve with an amplitude of $A$ will return a peak in the Fourier transform with an amplitude of 1.0. In the example, the DFT is implemented in the function `dft()`, and we use this to demonstrate the calculation of the amplitude spectrum on a time series consisting of two sinusoids where we have randomly deleted 80% of the points. For large data sets, the DFT can be very slow, as it requires $O(N^2)$ operations. If your data are equally spaced, then you will be able to use a fast Fourier transform (FFT) to compute your estimate of the amplitude spectrum. The FFT requires only $O(N \log N)$ operations.

In the example (`ft_demo.py`) we include a call to the NumPy function `rfft()` from the `fft` module (using the gapless data set), as well as the `rfftfreq()` function which returns the frequencies of the calculated amplitudes (see figure 11.5):

```
import numpy as np
from numpy.fft import rfft, rfftfreq
import matplotlib.pyplot as plt
from cmath import exp, pi
def dft(t,y,freq):
    nt = len(t)
    nf = len(freq)
    c = np.zeros(nf,complex)
```

[2] A classic and comprehensive text introducing Fourier transforms is Bracewell R 1999 *The Fourier Transform & Its Applications* (New York: McGraw-Hill).

[3] See chapter 7 of Newman M 2012 *Computational Physics* (Scotts Valley, CA: CreateSpace) for a thorough discussion of applications of Fourier transforms.

```
    for k in range(nf):
        f = freq[k]
        for i in range(nt):
            c[k] += y[i]*exp(-2j*pi*f*t[i])
    return 2.*abs(c)/nt

npts = 1000
twopi = 2.* pi

A1 = 1.5 ; P1 = 10.
A2 = 1.  ; P2 = 3.

t = np.linspace(0.,100.,npts)
y = A1*np.sin((twopi/P1)*t) + A2*np.sin((twopi/P2)*t)

frac_points = 0.2 # fraction of points to select

r = np.random.rand(y.shape[0])
t_subset = t[r >= (1.0 - frac_points)]
y_subset = y[r >= (1.0 - frac_points)]

nfreq = 1000
freq_dft = np.linspace(0.,1.,nfreq)

amp_dft = dft(t_subset, y_subset, freq_dft)

amp_fft = np.abs(rfft(y)) * 2 / npts
freq_fft = rfftfreq(npts,t[1]-t[0])

fig = plt.figure()
ax = plt.subplot(311)                # time series
plt.plot(t_subset,y_subset,'bo',ms=3)
plt.plot(t,y,'r',alpha=0.2)

ax = plt.subplot(312)                # Discrete Fourier Transform
plt.plot(freq_dft, amp_dft)

ax = plt.subplot(313)                # FFT
plt.plot(freq_fft, amp_fft)
plt.xlim([0,1])

plt.show()
```

The SciPy module contains the `signal` module which contains a long list of useful functions, including `lombscargle()` which returns an estimate of the power spectral density using the Lomb–Scargle periodogram. To call this function,

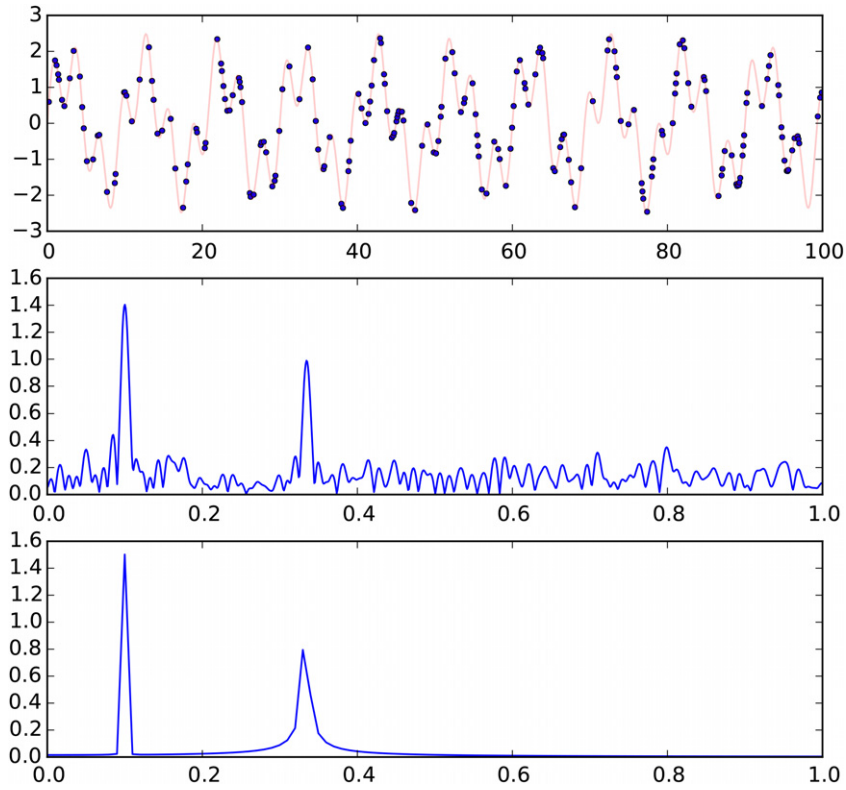


**Figure 11.5.** A demonstration of a Fourier transform. The upper panel shows the original time series (red) and the randomly selected points (blue points). The middle panel shows the DFT amplitude spectrum of the randomly selected points and the lower panel shows the FFT amplitude spectrum of the red curve.

we only need to calculate the angular frequencies and call as follows using our previous definitions:

```
ang_freqs = freqs * twopi
pgm     = lombscargle(t_subset, y_subset, ang_freqs)
amp_pgm = np.sqrt(4 * (pgm / len(t_subset)))          # normalize
```

## 11.5 Writing sound files

It can be fun and perhaps even useful to turn your data into a sound file. The following example[4] does just that (`dat2wav.py`). It reads in two-column data,

[4] Based on codingmess.blogspot.com/2010/02/how-to-make-wav-file-with-python.html.

normalizes to the range -16384 to +16384, then uses the `wave` module functions to create and write a WAV file:

```
#!/usr/bin/env python
"""
Program: dat2wav

Description
  Read in time series data (2-col) and normalize.
  Write sound (.wav) file

  Based on http://codingmess.blogspot.com/2010/02
                   how-to-make-wav-file-with-python.html

"""
import numpy as np
import wave
import sys

class SoundFile:

    def __init__(self, signal):
        self.file = wave.open(outfile, 'wb')
        self.signal = signal
        self.sr = samplerate

   def write(self):
        self.file.setparams((1, 2, self.sr, samplerate*4, 'NONE',
                             'noncompressed'))
        self.file.writeframes(self.signal)
        self.file.close()

def np2str(y):
    """Convert NumPy vector to string (h =>data formatted as ints)"""
    signal = ".join(wave.struct.pack('h', item) for item in y)
    return signal

# User can set sample rate on command line (optional)
samplerate = 44100
if (len(sys.argv) >3):
    samplerate = int(sys.argv[3])
if (len(sys.argv) >2):
   infile = sys.argv[1]
```

```
    outfile = sys.argv[2]
    x, y = np.loadtxt(infile,usecols=(0,1),unpack=True)

# Normalize and output
    ynorm = 16384 / max(abs(y))
    ydata = y * ynorm

    ssignal = np2str(ydata)
    f = SoundFile(ssignal)
    f.write()

else:
    print "syntax: dat2wav <infile> <outfile> [samplerate]"
```

**IOP** Concise Physics

# Python and Matplotlib Essentials for Scientists and Engineers

**Matt A Wood**

# Chapter 12

## Visualization and animations

### 12.1 VPython

VPython[1] is based on the `visual` 3D graphics module contributed by David Scherer in 2000. The `visual` package allows you to place 3D objects in a scene. The scene is updated many times per second, allowing animations and user-controlled scene rotations and scalings that are fluid. The `vpython.org` website contains links to tutorial videos that provide a good introduction. Here is a simple example[2] (`ball+box.py`) that simulates a ball bouncing between two walls. Figure 12.1 shows a screen capture from a slightly modified version of this code that creates a new sphere every 15 time steps. In the code, the `visual` module is imported on the first line. Next, the scene is initialized, where `center=(0,3,0)` indicates the coordinates to which the camera points. The next three lines initialize the wall and floor objects. The ball object is initialized next at position `(-4.5, 4, 0)`, radius = 0.5 and with a red color. The next line initializes the vector velocity. The `while` loop is infinite, so the program is terminated by closing the plot window. The command `rate(100)` indicates that no more than 100 time steps should be taken per second, which is needed for fast computers to prevent the entire simulation from zipping by too fast to register. The ball object's position is updated with the velocity as a vector statement. Normal velocity components are mirrored upon collision with the walls or floor and the $y$ velocity is updated using $v_y^{\text{new}} = v_y^{\text{old}} - g\,\mathrm{d}t$ on all time steps except when there is a floor collision. Note that

[1] Tagline: *3D Programming for Ordinary Mortals*. VPython is available from vpython.org. To install VPython in Anaconda, type `conda install -c mwcraig vpython` at a terminal prompt. To build VPython and dependencies from source, see the instructions at github.com/mwcraig/conda-vpython-recipes.
[2] Based on the *bounce example* from vpython.org/contents/bounce_example.html.

doi:10.1088/978-1-6270-5620-5ch12

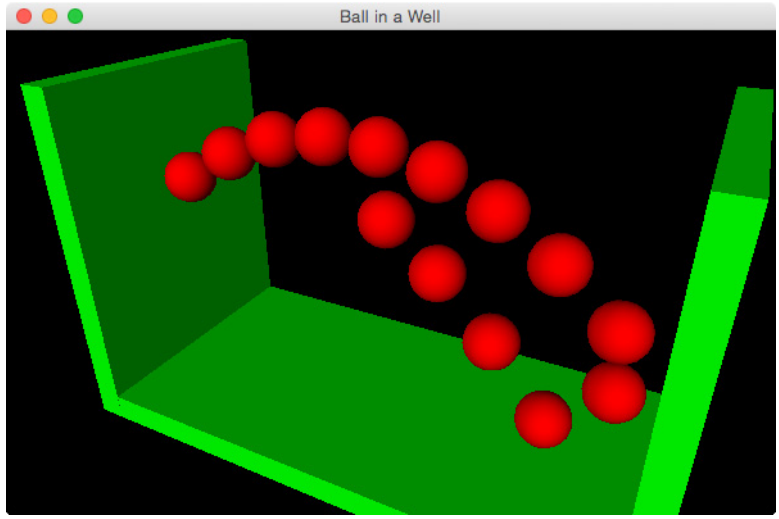


**Figure 12.1.** A VPython simulation of a ball bouncing between walls.

there are no graphics calls in the `while` loop—just calculations updating positions and velocities:

```
from visual import *

scene = display(title='Ball in a Well',width=600,height=400,
                center=(0,3,0))
floor = box (pos=(0,0,0), length=10, height=0.5, width=5,
color=color.green)
lwall = box (pos=(-5,3,0), length=.5, height=6.5, width=5,
color=color.green)
rwall = box (pos=(5,3,0), length=.5, height=6.5, width=5,
color=color.green)

ball = sphere (pos=(-4.5,4,0), radius=0.5, color=color.red)
ball.velocity = vector(8,4,0)
dt = 0.01

while 1:
    rate (100)
    ball.pos = ball.pos + ball.velocity*dt
    if ball.y < ball.radius:
        ball.velocity.y = abs(ball.velocity.y)
    else:
        ball.velocity.y = ball.velocity.y - 9.8*dt
    if ball.x < -5+ball.radius or ball.x > 5-ball.radius:
        ball.velocity.x = -ball.velocity.x
```

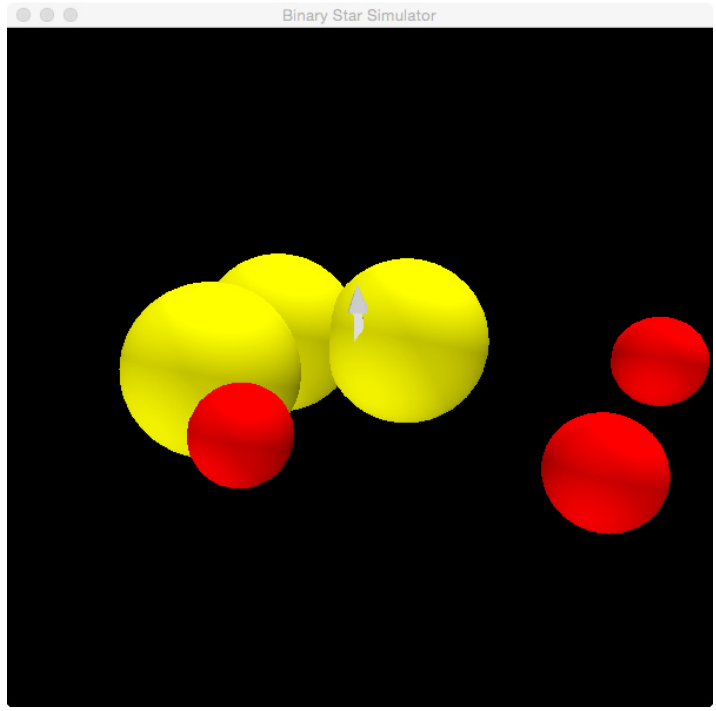


**Figure 12.2.** A VPython binary star simulation.

Here is an example of a binary star simulator (`binstar_vpython.py`). The calculations are performed in MKS units to simplify the programming. Here we have increased the ambient and directional lighting, and make use of the `mag` (magnitude) function when creating our $\hat{r}_{12}$ unit vector. We include an `arrow` object, the tail of which is at the center of mass, to indicate the direction of system angular momentum. In figure 12.2 we show three positions of the simulation.

```
# binary star simulator

from visual import *
import numpy as np

scene = display(title='Binary Star Simulator',width=600,height=600,
                center=(0,3,0))
scene.ambient=color.gray(0.4)
distant_light(direction=(0,-1,0),color=color.gray(0.4))

Rsun = 7.e8
Msun = 2.e30
G = 6.67e-11
```

```
M1 = 1.0*Msun ; R1 = 1.0*Rsun # solar
M2 = 0.5*Msun ; R2 = 0.5*Rsun # M dwarf

M = M1+M2
a = 5*Rsun
a1 = a*M2/M
a2 = a*M1/M
Porb = np.sqrt(4*np.pi**2 / (G*M) * a**3)
v1circ = 2 * np.pi * a1 / Porb
v2circ = 2 * np.pi * a2 / Porb
v1 = 0.8 * v1circ
v2 = v1 * M1/M2

print 'M1, M2, M = ', M1, M2, M
print 'a, a1, a2 = ', a, a1, a2
print 'porb (s) = ',Porb

s1 = sphere (pos=(-a1,0,0), radius=R1, color=color.yellow)
s2 = sphere (pos=(a2,0,0), radius=R2, color=color.red)
cm = arrow(pos=(0,0,0), axis=(0,0,Rsun),color=color.white)
s1.velocity = vector(0,v1,0)
s2.velocity = vector(0,-v2,0)
dt = 250   #seconds

while 1:
    rate (100)
    r12 = s2.pos - s1.pos
    r = mag(r12)
    r12hat = r12 / mag(r12)
    accel1 = G * M2 / r**2 * r12hat
    accel2 = G * M1 / r**2 * -r12hat
    s1.velocity = s1.velocity + accel1*dt
    s2.velocity = s2.velocity + accel2*dt
    s1.pos = s1.pos + s1.velocity*dt
    s2.pos = s2.pos + s2.velocity*dt
```

## 12.2 Making figures with Mayavi

Mayavi[3] is intended to provide an interface for making interactive visualizations of 3D data and is an alternative to Matplotlib. It is available as a stand-alone program, but relevant to us it is also callable from Python programs and interactively from IPython. Mayavi is a very powerful package and a full description of it is outside the scope of this book. My intent here is to simply bring it to your attention with a couple of small examples. Should Mayavi be useful to you, there are a large number of examples on the website to get you started. For our first example, the code below

[3] For download, installation, documentation and a gallery of examples, visit code.enthought.com/projects/mayavi.

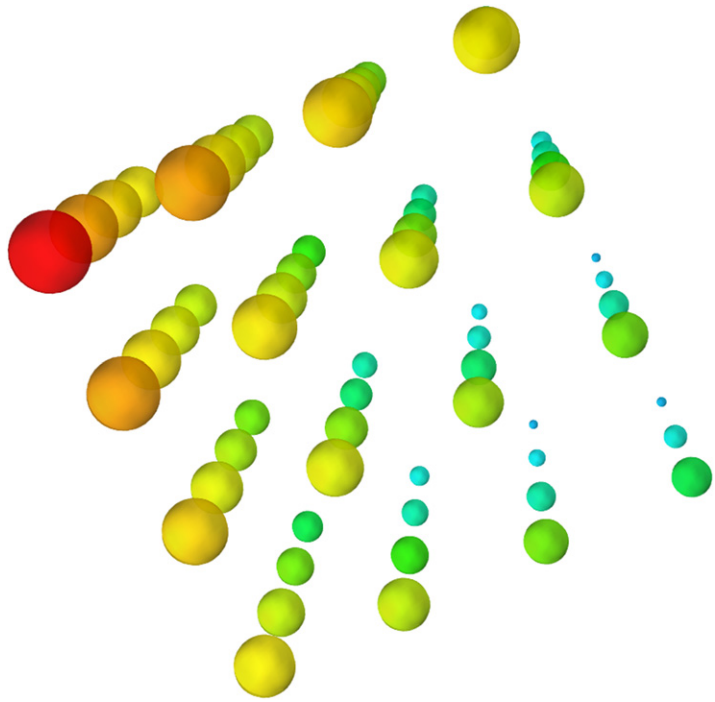

**Figure 12.3.** A Mayavi figure of points on a 3D mesh.

(`points3D_mayavi.py`) uses the NumPy `meshgrid()` function to generate a 3D mesh, then plots points on that mesh, scaling both the color and size of the spherical glyphs by the distance to the origin (see figure 12.3):

```
import numpy as np
from mayavi import mlab

x = np.linspace(1,4,4)

x, y, z = np.meshgrid(x, x, x)
r = np.sqrt(x*x + y*y + z*z)

mlab.figure(bgcolor=(1,1,1),size=(1200,1200))
mlab.points3d(x,y,z,r,resolution=16,opacity=0.7)
mlab.show()
```

As our second Mayavi example, the code below (`Y10_5_mayavi.py`) generates a plot of the $Y_{10}^{5}(\theta, \phi)$ spherical harmonic and saves the image to a file. The

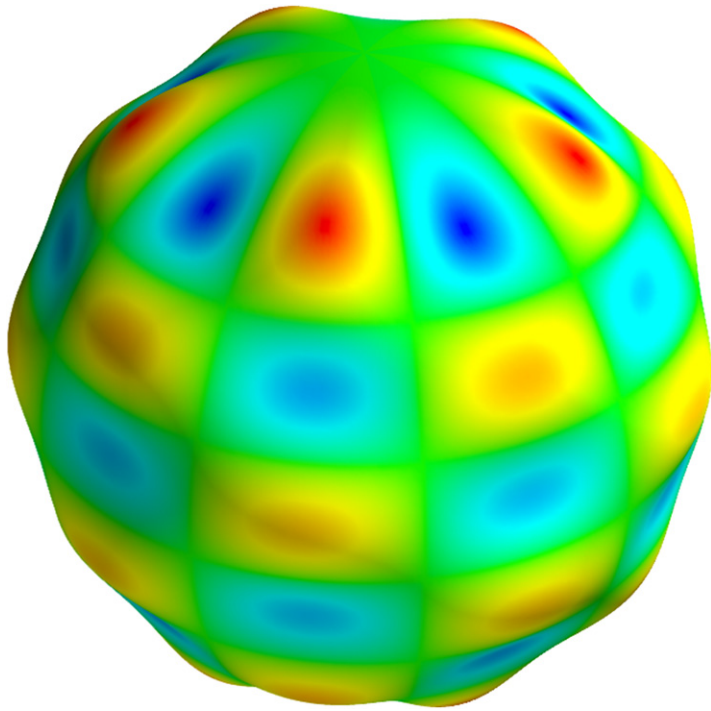

**Figure 12.4.** A Mayavi figure showing spherical harmonic $Y_{10}^{5}(\theta, \phi)$.

argument `scalars=r` instructs Mayavi to use the `r` values to scale the color map on the surface (see figure 12.4):

```
from numpy import pi, cos, sin, sqrt, mgrid
from mayavi import mlab

a = -3./256*sqrt(1001/pi)

dp = pi/250.0
[theta,phi] = mgrid[0:pi+dp:dp,0:2*pi+dp:dp]

r = 1 + 0.1 * a * cos(5*phi) * sin(theta)**5 \
    *(323*cos(theta)**5 - 170*cos(theta)**3 + 15*cos(theta))

x = r*sin(theta)*cos(phi)
y = r*sin(theta)*sin(phi)
z = r*cos(theta)

fig = mlab.figure(bgcolor=(1.,1.,1.),size=(1200,1200))
s = mlab.mesh(x, y, z,scalars=r,figure=fig)
#mlab.savefig('Y10.5_mayavi.png')
mlab.show()
```

## 12.3 Animations

Matplotlib contains the `animation` module which can be useful for several purposes. As discussed above, animations are excellent for visualizing physical phenomena. In addition, animations can be very effectively used during public talks or for web publications. Even a simple value versus time plot can be brought to life by animation. For example, the following code[4] reads in the $CO_2$ data discussed in section 6.1.2 (see figure 12.5). The line object is created with `l,= plt.plot([], [], 'r-')` and is of type `Line2D`. It is the only element which changes during the animation. The line object is updated with the function `update_line()`, which is what is called repeatedly to generate the animation. Here, the ellipsis '`...`' serve as a placeholder for a variable number of '`:`' slices and `:num` of course indicates to return only the first `num-1` values of the array. The result of this function definition is that each subsequent call sets the data for the line object to contain one additional data pair, up to the complete data set for the final call. The `interval` keyword argument specifies a delay of 20 ms between frames. The `blit` keyword if set `True` tells `FuncAnimation` to only redraw the pixels of the plot that have changed, which can speed up the animations considerably[5]. Finally, the animation is saved in the file `anim_co2.mp4`[6].

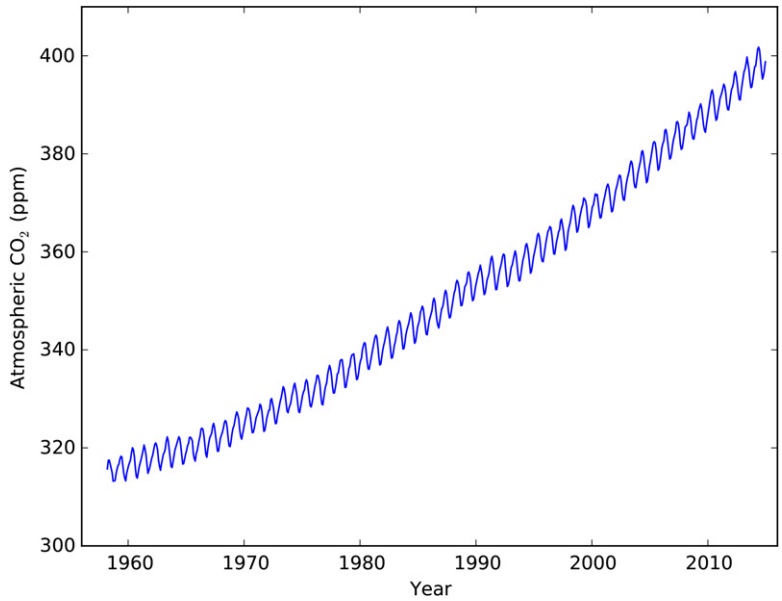


**Figure 12.5.** The final frame of the animated $CO_2$ movie.

[4] Based on `basic_example.py` (`anim_co2.py`) from the matplotlib.org animation examples page matplotlib.org/1.4.2/examples/animation/index.html. A nice tutorial is available at jakevdp.github.io/blog/2012/08/18/matplotlib-animation-tutorial.

[5] If using OS X, you will need to specify `blit=False` due to the way in which the OS works. This has been a known issue for several years now and apparently is not a simple fix. For more information, see github.com/matplotlib/matplotlib/issues/531.

[6] Available at pythonessentials.org/anim_co2.mp4.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def update_line(num, data, line):
    line.set_data(data[...,:num])
    return line,

fig = plt.figure()

data = np.loadtxt('co2_mm_mlo0.txt',usecols=(2,3),unpack=True)
nlines = data.shape[1]

l, = plt.plot([], [], 'r-')
plt.xlim(1956, 2016)
plt.ylim(300, 410)
plt.xlabel('Year')
plt.ylabel('Atmospheric CO$_2$ (ppm)')
line_ani = animation.FuncAnimation(fig, update_line, nlines,
            fargs=(data, l), interval=20, blit=False)
line_ani.save('anim_co2.mp4')

plt.show()
```

**IOP** Concise Physics

Python and Matplotlib Essentials
for Scientists and Engineers

**Matt A Wood**

# Chapter 13

## Interfacing with other languages

It is possible to call C/C++ and Fortran routines from within a Python program and vice versa. You might want to do this if you have a working and often-used computationally intensive routine in one of these languages, but you prefer the more user-friendly interface that Python can present. Perhaps surprisingly, calling a Fortran routine is less involved than interfacing with C/C++ as long as we use `f2py`[1], which is now part of the NumPy distribution and is what we will explore in this chapter. If you have the need to interface with C/C++, see the documentation at docs.python.org/2/extending/extending.html.

Here we will use the specific example of a Fortran subroutine (`dft` in file `dftsub.f`) that calculates a simple DFT, as discussed in section 11.4. The subroutine calculates the normalized amplitudes, such that when passed a noise-free sine curve with amplitude 1.0, the calculated amplitude will equal 1.0 at the appropriate frequency. The implementation here does not require that the input data be equally spaced, as is required when using an FFT.

```
      subroutine dft(t,y,freq,amp,numt,nfreq)

c input:  t(numt), y(numt)  - times and values
c         freq(nfreq)       - frequency values
c output: amp(nfreq)        - normalized amplitudes

      implicit double precision(a-h,o-z)
      dimension t(numt), y(numt), freq(nfreq), amp(nfreq)
```

[1] See, for example, docs.scipy.org/doc/numpy/user/c-info.python-as-glue.html.

doi:10.1088/978-1-6270-5620-5ch13  

```
      twopi = 2 * 3.141592653589793

      do ifreq = 1,nfreq
         f = freq(ifreq)
         fr = 0.
         fi = 0.
         do i = 1,numt
            a = twopi * f * t(i)
            c = cos(a)
            s = sin(a)
            fr = fr + y(i) * c
            fi = fi + y(i) * s
         end do
         fr = fr/numt
         fi = fi/numt
         ff = fr*fr + fi*fi
         amp(ifreq) = 2.*sqrt(ff)
      end do

      return
      end
```

The equivalent Python implementation of this is function `dft()` from our module `dft_py.py`,

```
def dft(t,y,freq):
    from numpy import zeros
    from cmath import exp, pi
    nt = len(t)
    nf = len(freq)
    c = zeros(nf,complex)
    for k in range(nf):
        f = freq[k]
        for i in range(nt):
            c[k] += y[i]*exp(-2j*pi*f*t[i])
    return 2.*abs(c)/nt
```

where both implementations return the same values to six decimal places.

Given our Fortran subroutine, we can use `f2py` to create a module that can be imported and used:

```
% f2py -c -m dftsub dftsub.f
```

This creates a file on your system with the basename `dftsub` and an extension that is the appropriate extension for a Python extension module on your platform (e.g., `.so`, `.pyd`, etc). The module `dftsub` is now importable, but all array dimensions must be declared in the calling function.

In this example (`dft-compare.py`) we generate a time series data set of 10 000 points consisting of two periods, of ten and three seconds, of different amplitudes. We calculate the DFT at 1000 frequency points using both the Python code and the code in the Fortran-derived module (`dftsub.so`). We use the `time()` function from the `time` module to determine how much time is spent in each of the two DFT routines and find that the Python DFT function takes some 150 times longer to complete than the Fortran subroutine:

```
#!/usr/bin/env python
"""
DFT Comparison (Python vs. Fortran)
Generate 2-sine curve, then call Python and Fortran DFT function.
"""
from numpy import *
import matplotlib.pyplot as plt
import time
import dftsub
import dft_py

npts = 10000
twopi = 2.* pi

A1 = 2. ; P1 = 10.
A2 = 1. ; P2 = 3.

t = linspace(0.,300.,npts)
y = A1*sin((twopi/P1)*t) + A2*sin((twopi/P2)*t)

nfreq = 1000
freq = linspace(0.,1.,nfreq)
amp_f = zeros(nfreq,dtype='float')

t0p = time.time()
amp_py = dft_py.dft(t,y,freq)
t1p = time.time()

t0f = time.time()
dftsub.dft(t,y,freq,amp_f,npts,nfreq)
t1f = time.time()

dtp = t1p - t0p
dtf = t1f - t0f

print "Time in Fortran subroutine", dtf
print "Time in Python subroutine ", dtp
```

```
In [1]: run dft-compare
Time in Fortran subroutine 0.25775885582
Time in Python subroutine  39.9592280388
```