Introduction to Python for Science

Release 0.9.30

David J. Pine

September 15, 2014

CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 Introduction to Python and its use in science

This manual is meant to serve as an introduction to the Python programming language and its use for scientific computing. It's ok if you have never programmed a computer before. This manual will teach you how to do it from the ground up.

The Python programming language is useful for all kinds of scientific and engineering tasks. You can use it to analyze and plot data. You can also use it to numerically solve science and engineering problems that are difficult or even impossible to solve analytically.

While we want to marshall Python's powers to address scientific problems, you should know that Python is a general purpose computer language that is widely used to address all kinds of computing tasks, from web applications to processing financial data on Wall Street and various scripting tasks for computer system management. Over the past decade it has been increasingly used by scientists and engineers for numerical computations, graphics, and as a "wrapper" for numerical software originally written in other languages, like Fortran and C.

Python is similar to Matlab and IDL, two other computer languages that are frequently used in science and engineering applications. Like Matlab and IDL, Python is an *interpreted* language, meaning you can run your code without having to go through an extra step of compiling, as required for the C and Fortran programming languages. It is also a *dynamically typed* language, meaning you don't have to declare variables and set aside memory before using them. Don't worry if you don't know exactly what these terms mean. Their primary significance for you is that you can write Python code, test, and use it quickly with a minimum of fuss.

One advantage of Python over similar languages like Matlab and IDL is that it is free. It can be downloaded from the web and is available on all the standard computer platforms, including Windows, MacOS, and Linux. This also means that you can use Python without being tethered to the internet, as required for commercial software that is tied to a remote license server.

Another advantage is Python's clean and simple syntax, including its implementation of *object oriented* programming (which we do not emphasize in this introduction).

An important disadvantage is that Python programs can be slower than compiled languages like C. For large scale simulations and other demanding applications, there can be a considerable speed penalty in using Python. In these cases, C, C++, or Fortran is recommended, although intelligent use of Python's array processing tools contained in the NumPy module can greatly speed up Python code. Another disadvantage is that compared to Matlab and IDL, Python is less well documented. This stems from the fact that it is public *open source* software and thus is dependent on volunteers from the community of developers and users for documentation. The documentation is freely available on the web but is scattered among a number of different sites and can be terse. This manual will acquaint you with the most commonly-used web sites. Search engines like Google can help you find others.

You are not assumed to have had any previous programming experience. However, the purpose of this manual isn't to teach you the principles of computer programming; it's to provide a practical guide to getting started with Python for scientific computing. Perhaps once you see some of the powerful tasks that you can accomplish with Python, you will be inspired to study computational science and engineering, as well as computer programming, in greater depth.

CHAPTER

TWO

LAUNCHING PYTHON

2.1 Installing Python on your computer

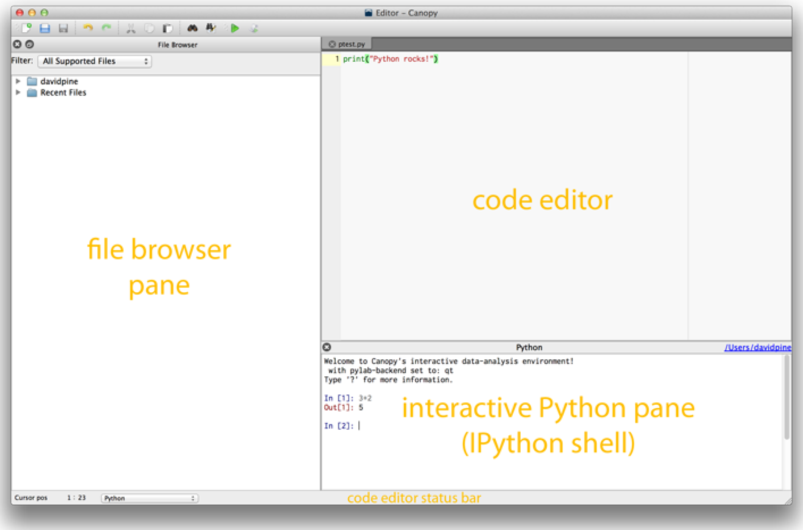
If you haven't already installed Python on your computer, see *Installing Python*, which includes instructions for installing Python on Macs running under MacOSX and on PCs running under Windows.

Once you have installed Python, find the Canopy icon on your computer and launch the application. Wait for the Canopy welcome screen to appear, and then click on the `Editor` icon. The Canopy window should appear, like the one shown below. This is the window you will generally use to work with Python.

2.2 The Canopy window

The default Canopy window has three panes: the code editor, the interactive Python pane, and the file browser pane. The **interactive Python pane** is the primary way that you interact with Python. You can use it to run Python computer programs, test snippets of Python code, navigate your computer file directories, and perform system tasks like creating, moving, and deleting files and directories. You will use the **code editor** to write and edit Python programs (or scripts), which are simply sequences of Python commands (code) stored in a file on your computer. The **file browser pane** allows you to navigate your computer's file directory system in order to view and retrieve files on your computer.

The individual panes in the Canopy window are reconfigurable and detachable but we will leave them pretty much as they are for now. However, you may want to adjust the overall size of the window to suit your computer screen. You can find more information about Canopy in the *Documentation Browser*, which you can access through the *Welcome to Canopy* window.



```
print("Python rocks!")
```

```
Welcome to Canopy's interactive data-analysis environment!
 with pylab-backend set to: qt
Type '?' for more information.

In [1]: 3+2
Out[1]: 5

In [2]:
```

Figure 2.1: Canopy window

2.3 The Interactive Python Pane

The default input prompt of the interactive Python pane looks like this:

```
In [1]:
```

This means that Canopy is running a particular version or "shell" of interactive Python called **IPython**. The IPython shell has been specifically designed for scientific and engineering use. The standard Python interactive shell uses the prompt >>>. You can pretty much do everything you want to do with either shell, but we will be using the IPython shell as we want to take advantage of some of its special features for scientific computing.

By typing short commands at the prompt, IPython can be used to perform various system tasks, such as running programs and creating and moving files around on your computer. This is a different kind of computer interface than the icon-based interface (or "graphical user interface" GUI) that you usually use to communicate with your computer. While it may seem more cumbersome for some tasks, it can be more powerful for other tasks, particularly those associated with programming.

Before getting started, we point out that like most modern computer languages, Python is *case sensitive*. That is, Python distinguishes between upper and lower case letters. Thus, two words spelled the same but having different letters capitalized are treated as different names in Python. Keep that in mind as we introduce different commands.

2.3.1 Magic Functions

IPython features a number of commands called "magic" commands that let you perform various useful tasks. There are two types of magic commands, line magic commands that begin with `%`—these are executed on a single line—and cell magic commands that begin with `%%`—these are executed on several lines. Here we will concern ourselves only with line magic commands.

The first thing to know about magic commands is that you can toggle (turn on and off) the need to use the `%` prefix for line magic commands by typing `%automagic`. By default, the `Automagic` switch is set to `ON` so you don't need the `%` prefix. To set `Automagic` to `OFF`, simply type `%automagic` at the IPython prompt. Cell magic commands always need the `%%` prefix.

In what follows below, we assume that the `Automagic` switch is set to `ON` so we omit the `%` sign.

Navigation Commands

IPython recognizes several common navigation commands that are used under the Unix/Linux operating systems. In the IPython shell, these few commands work on Macs, PCs, and Linux machines.

At the IPython prompt, type `cd ~` (*i.e.* "cd" – "space" – "tilde" , where tilde is found near the upper left part of most keyboards). This will set your computer to its home (default) directory. Next type `pwd` (**p**rint **w**orking **d**irectory) and press RETURN. The console should return the name of the current directory of your computer. It might look like this on a Mac:

```
In [2]: pwd
Out[2]: u'/Users/pine'
```

or this on a PC:

```
In [3]: pwd
Out[3]: C:\\Users\\pine
```

The responses `Out[2]: u'/Users/pine'` for the Mac and `Out[3]: C:\\Users\\pine` for the PC mean the the current directory is `pine`, which is a subdirectory of `Users`. Taken together `/Users/pine` on a Mac or `C:\\Users\\pine` on a PC is known as the *path* of the current directory. The path is just the name of a directory and the sequence of subdirectories in which it resides up to the *root* directory.

Typing `cd ..` (“cd” – “space” – two periods) moves the IPython shell up one directory in the directory tree, as illustrated by the set of commands below.

```
In [4]: cd ..
/Users

In [5]: pwd
Out[5]: u'/Users'
```

The directory moved up one from `/Users/pine` to `/Users`. Now type `ls` (**list**) and press `RETURN`. The console should list the names of the files and subdirectories in the current directory.

```
In [6]: ls
Shared/    pine/
```

In this case, there are only two directories (indicated by the slash) and not files. Type `cd ~` again to return to your home directory and then type `pwd` to verify where you are in your directory tree. [Technically, `ls` isn’t a magic command, but typing it without the `%` sign lists the contents of the current directory, irrespective of whether `Automagic` is `ON` or `OFF`.]

Let’s create a directory within your documents directory that you can use to store your Python programs. We will call it `PyProgs`. First, return to your home directory by typing `cd ~`. To create directory `PyProgs`, type `mkdir PyProgs` (**make dir**ectory). Type `ls` to confirm that you have created `PyProgs` and then type `cd PyProgs` to switch to that directory.

Now let’s say you want to return to the previous subdirectory, `Documents` or `My Documents`, which should be one up in the directory tree if you have followed along. Type `cd ..` and then type `pwd`. You should find that you are back in the previous directory, `Documents` or `My Documents`. If you type `ls`, you should see the new directory `PyProgs` that you just created.

More Magic Commands

The most important magic command is `%run` *filename* where *filename* is the name of a Python program you have created. We haven’t done this yet but include it here just for reference. We will come back to this later.

Some other useful magic commands include `%hist`, which lists the recent commands issued to the IPython terminal, and `%edit`, which opens a new empty file in the code editor window. Typing `%edit` *filename*, will open the file *filename* if it exists in the current directory, or it will create a new file by that name if it does not, and will open it as a blank file in the code editor window.

There are a number of other magic commands. You can get a list of them by typing `lsmagic`.

```
In [7]: lsmagic
Available line magics:
%alias  %alias_magic  %autocall  %automagic  %bookmark  %cd
%clear  %colors  %config  %connect_info  %debug  %dhist  %dirs
%doctest_mode  %ed  %edit  %env  %gui  %guiref  %hist  %history
%install_default_config  %install_ext  %install_profiles
%killbgscripts  %less  %load  %load_ext  %loadpy  %logoff  %logon
%logstart %logstate  %logstop  %lsmagic  %macro  %magic  %man
%more  %notebook  %page  %pastebin  %pdb  %pdef  %pdoc  %pfile
%pinfo  %pinfo2  %popd  %pprint  %precision  %profile  %prun
%psearch  %psource  %pushd  %pwd  %pycat  %pylab  %qtconsole
%quickref  %recall  %rehashx  %reload_ext  %rep  %rerun  %reset
%reset_selective  %run  %save  %sc  %store  %sx  %system  %tb
%time  %timeit  %unalias  %unload_ext  %who  %who_ls  %whos
%xdel  %xmode

Available cell magics:
%%!  %%bash  %%capture  %%file  %%javascript  %%latex  %%perl
%%prun  %%pypy %%python  %%python3  %%ruby  %%script  %%sh  %%svg
%%sx  %%system  %%timeit

Automagic is ON, % prefix IS NOT needed for line magics.
```

There are a lot of magic commands, most of which we don't need right now. We will introduce them in the text as needed.

2.3.2 System shell commands

You can also run system shell commands from the IPython shell by typing `!` followed by a system shell command. For Macs running OSX and for Linux machines, this means that Unix (or equivalently Linux) commands can be issued from the IPython prompt. For PCs, this means that Windows (DOS) commands can be issued from the IPython prompt. For example, typing `!ls` (**list**) and pressing RETURN lists all the files in the current directory on a Mac. Typing `!dir` on a PC does essentially the same thing (note that system shell commands in Windows are *not* case sensitive).

2.3.3 Tab completion

IPython also incorporates a number of shortcuts that make using the shell more efficient. One of the most useful is **tab completion**. Let's assume you have been following along and that your are in the directory `Documents` or `My Documents`. To switch to the directory `PyProgs`, you could type `cd PyProgs`. Instead of doing that, type `cd PyP` and then press the `TAB` key. This will complete the command, provided there is no ambiguity in how to finish the command. In the present case, that would mean that there was no other subdirectory beginning with `PyP`. Tab completion works with any command you type into the IPython terminal. Try it out! It will make your life more wonderful.

A related shortcut involves the ↑ key. If you type a command, say `cd` and then to press the ↑ key, IPython will complete the `cd` command with the last instance of that command. Thus, when you launch IPython, you can use this shortcut to take you to the directory you used when you last ran IPython.

You can also simply press the ↑ key, which will simply recall the most recent command. Repeated application of the ↑ key scrolls though the most recent commands in reverse order. The ↓ key can be used to scroll in the other direction.

2.3.4 Recap of commands

Let's recap the (magic) commands introduced above:

`pwd`: (**p**rint **w**orking **d**irectory) Prints the path of the current directory.

`ls`: (**lis**t) Lists the names of the files and directories located in the current directory.

`mkdir` *filename*: (**m**a**k**e **dir**ectory) Makes a new directory *filename*.

`cd` *directoryname*: (**c**hange **d**irectory) Changes the current directory to *directoryname*. Note: for this to work, *directoryname* must be a subdirectory in the current directory. Typing `cd ~` changes to the home directory of your computer. Typing `cd ..` moves the console one directory up in the directory tree.

`clear`: Clears the IPython screen of previous commands.

`run` *filename*: Runs (executes) a Python script. Described later in the section *Scripting Example 1*

Tab completion: Provides convenient shortcuts, with or without the arrow keys, for executing commands in the IPython shell.

2.4 Interactive Python as a calculator

You can use the IPython shell to perform simple arithmatic calculations. For example, to find the product 3×15, you type `3*15` at the `In` prompt and press `RETURN`:

```
In [1]: 3*15
Out[1]: 45
```

Python returns the correct product, as expected. You can do more complicated calculations:

```
In [2]: 6+21/3
Out[2]: 13.0
```

Let's try some more arithmetic:

```
In [3]: (6+21)/3
Out[3]: 9.0
```

Notice that the effect of the parentheses in `In [3]:  (6+21)/3` is to cause the addition to be performed first and then the division. Without the parentheses, Python will always perform the multiplication and division operations *before* performing the addition and subtraction operations. The order in which arithmetic operations are performed is the same as for most calculators: exponentiation first, then multiplication or division, then addition or subtraction, then left to right.

2.4.1 Binary arithmetic operations in Python

The table below lists the binary arithmatic operations in Python. It has all the standard binary operators for arithmetic, plus a few you may not have seen before.

Operation	Symbol	Example	Output
addition	`+`	`12+7`	`19`
subtraction	`-`	`12-7`	`5`
multiplication	`*`	`12*7`	`84`
division	`/`	`12/7`	`1.714285`
floor division	`//`	`12//7`	`1`
remainder	`%`	`12%7`	`5`
exponentiation	`**`	`12**7`	`35831808`

Floor division, designated by the symbols `//`, means divide and keep only the integer part without rounding. *Remainder*, designated by the symbols `%`, gives the remainder of after a floor division.

Warning: Integer division is different in Python 2 and 3

One peculiarity of all versions of Python prior to version 3 is that dividing two integers by each other yields the "floor division" result—another integer. Therefore `12/7` yields `1` whereas `12./7` or `12/7.` or `12./7.` all yield `1.714285`. Starting with version 3 of Python, all of the above expressions, including `3/2` yield `1.714285`. Unfortunately, we are using version 2.7 of Python so `12/7` yields `1`. You can force versions of Python prior to version 3 to divide integers like version 3 does by typing

```
from __future__ import division
```

at the beginning of an IPython session. You only need to type it once and it works for the entire session.

2.4.2 Types of numbers

There are four different types of numbers in Python: plain integers, long integers, floating point numbers, and complex numbers.

Plain integers, or simply **integers**, are 32 bits (binary digits) long, which means they extend from $-2^{31} = -2147483648$ to $2^{31} - 1 = 2147483647$. One bit is used to store the sign of the integer so there are only 31 bits left—hence, the power of 31. In Python, a number is automatically treated as an integer if is written without a decimal point and it is within the bounds given above. This means that `23`, written without a decimal point, is an integer and `23.`, written with a decimal point, is a floating point number. If an integer extends beyond the bounds of a simple integer, the it becomes a **long integer**, and is designated as such by an `L` following the last digit. Here are some examples of integer arithmetic:

```
In [4]: 12*3
Out[4]: 36

In [5]: 4+5*6-(21*8)
Out[5]: -134

In [6]: 11/5
Out[6]: 2.2

In [7]: 11//5
Out[7]: 2
```

```
In [8]: 9734828*79372     # product of these two large integers
Out[8]: 772672768016L     # is a long integer
```

For the binary operators +, -, *, and //, the output is an integer if the inputs are integers. The only exception is if the result of the calculation is out of the bounds of Python integers, in which case Python automatically converts the result to a long integer. The output of the division operator / is a floating point as of version 3 of Python. If an integer output is desired when two integers are divided, the floor division operator // must be used.

Floating point numbers are essentially rational numbers and can have a fractional part; integers, by their very nature, have no fractional part. In most versions of Python running on PCs or Macs, floating point numbers go between approximately $\pm 2 \times 10^{-308}$ and $\pm 2 \times 10^{308}$. Here are some examples of floating point arithmetic:

```
In [9]: 12.*3.
Out[9]: 36.0

In [10]: 123.4*(-53.9)/sqrt(5.)
Out[10]: -2974.5338992050501

In [11]: 11./5.
Out[11]: 2.2

In [12]: 11.//5.
Out[12]: 2.0

In [13]: 11.%5.
Out[13]: 1.0

In [14]: 6.022e23*300.
Out[14]: 1.8066e+26
```

Note that the result of any operation involving only floating point numbers as inputs is a real number, even in the cases where the floor division // or remainder % operators are used. The last output also illustrates an alternative way of writing floating point numbers as a mantissa followed by and e or E followed by a power of 10: so 1.23e-12 is equivalent to 1.23×10^{-12}.

We also sneaked into our calculations sqrt, the square root function. We will have more to say about functions in a few pages.

Complex numbers are written in Python as a sum of a real and imaginary part. For example, the complex number $3-2i$ is represented as 3-2j in Python where j represents $\sqrt{-1}$. Here are some examples of complex arithmetic:

```
In [15]: (2+3j)*(-4+9j)
Out[15]: (-35+6j)

In [16]: (2+3j)/(-4+9j)
Out[16]: (0.1958762886597938-0.3092783505154639j)

In [17]: sqrt(-3)
Out[17]: nan

In [18]: sqrt(-3+0j)
Out[18]: 1.7320508075688772j
```

Notice that to obtain the expected result or $\sqrt{-3}$, you must write the argument of the square root function as a complex number. Otherwise, Python returns `nan` (not a number).

If you multiply an integer by a floating point number, the result is a floating point number. Similarly, if you multiply a floating point number by a complex number, the result is a complex number. Python always promotes the result to the most complex of the inputs.

2.5 Python Modules

The Python computer language consists of a "core" language plus a vast collection of supplementary software that is contained in **modules**. Many of these modules come with the standard Python distribution and provide added functionality for performing computer system tasks. Other modules provide more specialized capabilities that not every user may want. You can think of these modules as a kind of library from which you can borrow according to your needs.

We will need three Python modules that are not part of the core Python distribution, but are nevertheless widely used for scientific computing. The three modules are

NumPy is the standard Python package for scientific computing with Python. It provides the all-important `array` data structure, which is at the very heart of NumPy. In also provides tools for creating and manipulating arrays, including indexing and sorting, as well as basic logical operations and element-by-element arithmetic operations like addition, subtraction, multiplication, division, and exponentiation. It includes the basic mathematical functions of trigonometry, exponentials, and logarithms, as well vast collection of special functions (Bessel functions, *etc.*), statistical functions, and random number generators. It also includes a large number of linear algebra routines that overlap with those in SciPy, although the SciPy routines tend to be more complete. You can find more information about NumPy at http://docs.scipy.org/doc/numpy/reference/index.html.

SciPy provides a wide spectrum of mathematical functions and numerical routines for Python. SciPy makes extensive use of NumPy arrays so when you import SciPy, you should always import NumPy too. In addition to providing basic mathematical functions, SciPy provides Python "wrappers" for numerical software written in other languages, like Fortran, C, or C++. A "wrapper" provides a transparent easy-to-use Python interface to standard numerical software, such as routines for doing curve fitting and numerically solving differential equations. SciPy greatly extends the power of Python and saves you the trouble of writing software in Python that someone else has already written and optimized in some other language. You can find more information about SciPy at http://docs.scipy.org/doc/scipy/reference/.

MatPlotLib is the standard Python package for making two and three dimensional plots. MatPlotLib makes extensive use of NumPy arrays. You will make all of your plots in Python using this package. You can find more information about MatPlotLib at http://MatPlotLib.sourceforge.net/.

We will use these three modules extensively and therefore will provide introductions to their capabilities as we develop Python in this manual. The links above provide much more extensive information and you will certainly want to refer to them from time to time.

These modules, NumPy, MatPlotLib, and SciPy, are built into the IPython shell so we can use them freely in that environment. Later, when we introduce Python programs (or scripts), we will see that in those cases you must explicitly load these modules using the `import` command to have access to them.

Finally, we note that you can write your own Python modules. They are a convenient way of packaging and storing Python code so that you can reuse it. We defer learning about how to write modules until after we have learned about Python.

2.6 Python functions: a first look

A function in Python is similar to a mathematical function. In consists of a name and one or more arguments contained inside parentheses, and it produces some output. For example, the NumPy function `sin(x)` calculates the sine of the number `x` (where `x` is expressed in radians). Let's try it out in the IPython shell:

```
In [1]: sin(0.5)
Out[1]: 0.479425538604203
```

The argument of the function can be a number or any kind of expression whose output produces a number. For example, the function `log(x)` calculates the natural logarithm of `x`. All of the following expressions are legal and produce the expected output:

```
In [2]: log(sin(0.5))
Out[2]: -0.73516668638531424

In [3]: log(sin(0.5)+1.0)
Out[3]: 0.39165386283471759

In [4]: log(5.5/1.2)
Out[4]: 1.5224265354444708
```

2.6.1 Some NumPy functions

NumPy includes an extensive library of mathematical functions. In the table below, we list some of the most useful ones. A much more complete list is available at http://docs.scipy.org/doc/numpy/reference/ufuncs.html#math-operations.

Function	Description
`sqrt(x)`	Square root of x
`exp(x)`	Exponential of x, *i.e.* e^x
`log(x)`	Natural log of x, *i.e.* $\ln x$
`log10(x)`	Base 10 log of x
`degrees(x)`	Converts x from radians to degrees
`radians(x)`	Converts x from degrees to radians
`sin(x)`	Sine of x (x in radians)
`cos(x)`	Cosine x (x in radians)
`tan(x)`	Tangent x (x in radians)
`arcsin(x)`	Arc sine (in radians) of x
`arccos(x)`	Arc cosine (in radians) of x
`arctan(x)`	Arc tangent (in radians) of x
`fabs(x)`	Absolute value of x
`round(x)`	Rounds a float to nearest integer
`floor(x)`	Rounds a float *down* to nearest integer
`ceil(x)`	Rounds a float *up* to nearest integer
`sign(x)`	-1 if $x < 0$, +1 if $x > 0$, 0 if $x = 0$

The functions discussed here all have one input and one output. Python functions can, in general, have multiple inputs and multiple outputs. We will discuss these and other features of functions later when we take up functions in the context of user-defined functions.

2.6.2 Keyword arguments

In addition to regular arguments, Python functions can have keyword arguments (`kwargs`). Keyword arguments are *optional* arguments that need not be specified when a function is called. See *Basic plotting* for examples of the use of keyword arguments. For the moment, we don't need them so we defer a full discussion of keyword arguments until we introduce them in the section on *User-defined functions*.

2.7 Variables

2.7.1 Names and the assignment operator

A variable is a name that is used to store data. It can be used to store different kinds of data, but here we consider the simplest case where the data is a single numerical value. Here are a few examples:

```
In [1]: a = 23

In [2]: p, q = 83.4, sqrt(2)
```

The equal sign "=" is the *assignment operator*. In the first statement, it assigns the value of 23 to the variable `a`. In the second statement it assigns a value of 83.4 to `p` and a value of 1.4142135623730951 to `q`. To be more precise, the name of a variable, such as `a`, is associated with a *memory location* in your computer; the assignment variable tells the computer to put a particular piece of data, in this case a numerical value, in that memory location. Note that Python stores the *numerical value*, not the expression used to generate it. Thus, `q` is assigned the 17-digit number 1.4142135623730951 generated by evaluating the expression `sqrt(2)`, *not* with $\sqrt{2}$. (Actually the value of `q` is stored as a binary, base 2, number using scientific notation with a mantissa and an exponent.)

Suppose we write

```
In [3]: b = a
```

In this case Python associates a new memory location with the name `b`, distinct from the one associated with `a`, and sets the value stored at that memory location to 23, the value of a. The following sequence of statements demonstrate that fact. Can you see how? Notice that simply typing a variable name and pressing Return prints out the value of the variable.

```
In [4]: a=23

In [5]: b=a

In [6]: a
Out[6]: 23

In [7]: b
Out[7]: 23

In [8]: a=12

In [9]: a
Out[9]: 12

In [10]: b
Out[10]: 23
```

The assignment variable works from right to left; that is, it assigns the value of the number on the right to the variable name on the left. Therefore, the statement "5=a" makes no sense in Python. The assignment operator "=" in Python is not equivalent to the equals sign "=" we are accustomed to in algebra.

The assignment operator can be used to increment or change the value of a variable

```
In [11]: b = b+1

In [12]: b
Out[12]: 24
```

The statement, `b = b+1` makes no sense in algebra, but in Python (and most computer languages), it makes perfect sense: it means "add 1 to the current value of b and assign the result to b." This construction appears so often in computer programming that there is a special set of operators to perform such changes to a variable: +=, -=, *=, and /=. Here are some examples of how they work:

```
In [13]: c , d = 4, 7.92

In [14]: c += 2

In [15]: c
Out[15]: 6
```

```
In [16]: c *= 3

In [16]: c
Out[16]: 18

In [17]: d /= -2

In [17]: d
Out[17]: -3.96

In [18]: d -= 4

In [19]: d
Out[19]: -7.96
```

Verify that you understand how the above operations work.

2.7.2 Legal and recommended variable names

Variable names in Python must start with a letter, and can be followed by as many alphanumeric characters as you like. Spaces are not allowed in variable names. However, the underscore character "_" is allowed, but no other character that is not a letter or a number is permitted.

Recall that Python is *case sensitive*, so the variable `a` is distinct from the variable `A`.

We recommend giving your variables descriptive names as in the following calculation:

```
In [20]: distance = 34.

In [21]: time_traveled = 0.59

In [22]: velocity = distance/time_traveled

In [23]: velocity
Out[23]: 57.6271186440678
```

The variable names `distance`, `time_traveled`, and `velocity` immediately remind you of what is being calculated here. This is good practice. But so is keeping variable names reasonably short, so don't go nuts!

2.7.3 Reserved words in Python

There are also some names or words that are reserved by Python for special purposes or functions. You must avoid using these names, which are provided here for your reference:

`and`	`del`	`from`	`not`	`while`
`as`	`elif`	`global`	`or`	`with`
`assert`	`else`	`if`	`pass`	`yield`
`break`	`except`	`import`	`print`	
`class`	`exec`	`in`	`raise`	
`continue`	`finally`	`is`	`return`	
`def`	`for`	`lambda`	`try`	

In addition, you should not use function names, like `sin`, `cos`, and `sqrt`, defined in the SciPy, NumPy, or any other library that you are using.

2.8 Script files and programs

Performing calculations in the IPython shell is handy if the calculations are short. But calculations quickly become tedious when they are more than a few lines long. If you discover you made a mistake at some early step, for example, you may have to go back and retype all the steps subsequent to the error. The solution to this problem is to save your code in a file. Saving code in a file means you can just correct the error and rerun the code without having to retype it. Saving code can also be useful if you want to reuse it later, perhaps with different inputs.

When we save code in a computer file, we call the sequence of commands stored in the file a *script* or a *program* or sometimes a *routine*. Programs can become quite sophisticated and complex. Here we are only going to introduce the simplest features of programming by writing a very simple script. Much later, we will introduce some of the more advanced features of programming.

To write a script you need a text editor. In principle, any text editor will do, but it's more convenient to use an editor that was designed for the task. We are going to use the **Code Editor** in the Canopy window that appears when you launch the Canopy application (see *Canopy window*). This editor, like most good programming editors, provides syntax highlighting, which color codes key words, comments, and other features of the Python syntax according to their function, and thus makes it easier to read the code and easier to spot programming mistakes. The Canopy code editor also provides syntax checking, much like a spell-checker in a word processing program, that identifies many coding errors. This can greatly speed the coding process. Tab completion also works in the Canopy Code Editor.

2.8.1 Scripting Example 1

Let's work through an example to see how scripting works. Suppose you are going on a road trip and you would like to estimate how long the drive will take, how much gas you will need, and the cost of the gas. It's a simple calculation. As inputs, you will need the distance of the trip, your average speed, the cost of gasoline, and the mileage of your car.

Writing a script to do these calculations is straightforward. First, launch Canopy and open the code editor. You should see a tab with the word `untitled` at the top left of the code editor pane (see *Canopy window*). If you don't, go to the `File` menu and select `New File`. Use the mouse to place your cursor at the top of the code editor pane. Enter the following code and *save the code* in a file called `myTrip.py` in the `PyProgs` folder you created earlier. This stores your script (or program) on your computer's disk. The exact name of the file is not important but the extension `.py` is essential. It tells the computer, and more importantly Python, that this is a Python program.

```
# Calculates time, gallons of gas used, and cost of gasoline for
# a trip
distance = 400.         # miles
mpg = 30.               # car mileage
speed = 60.             # average speed
costPerGallon = 4.10    # price of gas

time = distance/speed
gallons = distance/mpg
cost = gallons*costPerGallon
```

The number (or hash) symbol `#` is the "comment" character in Python; anything on a line following `#` is ignored when the code is executed. Judicious use of comments in your code will make your code much easier to understand days, weeks, or months after the time you wrote it. Use comments generously.

Python ignores blank spaces or "white space" as it is sometimes called. The statement `costPerGallon = 4.10` in the above program could equally well be written as `costPerGallon=4.10` without the spaces before and after the `=` assignment operator; either way the statement means the same thing. Similarly, the white space after `costPerGallon = 4.10` but before the comment (hash) symbol is also ignored by Python. The idea is that you should use white space to make your program more readable.

Now you are ready to run the code. Before doing so, you first need to use the IPython console to move to the `PyProgs` directory where the file containing the code resides. From the IPython console, use the `cd` command to move to the `PyProgs` directory. For example, you might type

```
In [1]: cd ~/Documents/PyProgs/
```

To *run* or *execute* a script, simply type `run` *filename*, which in this case means type `run myTrip.py`. When you run a script, Python simply executes the sequence of commands in the order they appear.

```
In [2]: run myTrip.py
```

Once you have run the script, you can see the values of the variables calculated in the script simply by typing the name of the variable. IPython responds with the value of that variable.

```
In [3]: time
Out[3]: 6.666666666666667

In [4]: gallons
Out[4]: 13.333333333333334

In [5]: cost
Out[5]: 54.666666666666664
```

You can change the number of digits IPython displays using the command `%precision`:

```
In [6]: %precision 2
Out[6]: u'%.2f'

In [7]: time
Out[7]: 6.67

In [8]: gallons
Out[8]: 13.33

In [9]: cost
Out[9]: 54.67
```

Typing `%precision` returns IPython to its default state; `%precision %e` causes IPython to display numbers in exponential format (scientific notation).

Note about printing

If you want your script to return the value of a variable (that is, print the value of the variable to your computer screen), use the `print` function. For example, at the end of our script, if we include the code

```
print(time)
print(gallons)
print(cost)
```

the script will return the values of the variables `time`, `gallons`, and `cost` that the script calculated. We will discuss the `print` function in much greater detail, as well as other methods for data output, in Chapter 4 on *Input and Output*.

2.8.2 Scripting Example 2

Let's try another problem. Suppose you want to find the distance between two Cartesian coordinates (x_1, y_1, z_1) and (x_2, y_2, z_2). The distance is given by the formula

$$\Delta r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Now let's write a script to do this calculation and save it in a file called `twoPointDistance.py`.

```
# Calculates the distance between two 3d Cartesian coordinates
import numpy as np

x1, y1, z1 = 23.7, -9.2, -7.8
x2, y2, z2 = -3.5, 4.8, 8.1

dr = np.sqrt( (x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2 )
```

We have introduced extra spaces into some of the expressions to improve readability. They are not necessary; where and whether you include them is largely a matter of taste.

There are two important differences between the code above and the commands we would have written into the IPython console to execute the same set of commands. The first is the statement on the second line

```
...
import numpy as np
...
```

and the second is the "`np.`" in front of the `sqrt` function on the last line. If you leave out the `import numpy as np` line and remove the `np.` in front of the `sqrt` function, you will get the following error message

```
----> 7 dr = sqrt( (x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2 )

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

The reason for the error is that the `sqrt` function is not a part of core Python. But it is a part of the NumPy module discussed earlier. To make the NumPy library available to the script, you need to add the statement `import numpy as np`. Then, when you call a NumPy function, you need to write the function with the `np.` prefix. Failure to do either will result in a error message. Now we can run the script.

```
In [10]: run twoPointDistance.py

In [11]: dr
Out[11]: 34.48
```

The script works as expected.

The reason we do not have to import NumPy when working in the IPython shell is that it is done automatically when the IPython shell is launched. Similarly, the package MatPlotLib is also automatically loaded (imported) when IPython is launched. However, when a script or program is executed, it is run on its own outside the IPython shell, even if the command to run the script is executed from the IPython shell.

Line continuation

From time to time, a line of code in a script will be unusually long, which can make the code difficult to read. In such cases, it is advisable to split the code onto several lines. For example, line 7 in the preceding script could be written as

```
dr = np.sqrt( (x2-x1)**2
            + (y2-y1)**2
            + (z2-z1)**2 )
```

You can generally continue an expression on another line in Python for code that is within a function argument, as it is here where the line is split inside the argument of the square root function. Note that the sub-expressions written on different lines are lined up. This is done solely to improve readability; Python does not require it. Nevertheless, as the whole point of splitting a line is to improve readability, it's best to line up expressions so as to maximize readability.

You can split any Python line inside of parentheses, brackets, and braces, as illustrated above. You can split it other places as well by using the backslash (\) character. For example, the code

```
a = 1 + 2 \
  + 3 + 4
```

is equivalent to

```
a = 1 + 2 + 3 + 4
```

So you can use backslash character (\) of explicit line continuation when implicit line continuation won't work.

2.9 Importing Modules

We saw in Example 2 in the last section that we needed to import the NumPy module in order to use the `sqrt` function. Indeed the NumPy library contains many useful functions, some of which are listed in section *Python functions: a first look*. Whenever any NumPy functions are used, the NumPy library must be loaded using an `import` statement.

There are a few ways to do this. The one we generally recommend is to use the `import as` implementation that we used in Example 2. For the main NumPy and MatPlotLib libraries, this is implemented as follows:

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

These statements import the entire library named in the `import` statement and associate a prefix with the imported library: `np` and `plt` in the above examples. Functions from within these libraries are then called by attaching the appropriate prefix with a period *before* the function name. Thus, the functions `sqrt` or `sin` from the NumPy library are called using the syntax `np.sqrt` or `np.sin`; the functions `plot` or `xlabel` from the `maplotlib.pyplot` would be called using `plt.plot` or `plt.xlabel`.

Alternatively, the NumPy and MatPlotLib libraries can be called simply by writing

```
import numpy
import maplotlib.pyplot
```

When loaded this way, the `sqrt` function would be called as `numpy.sqrt` and the `plot` function would be called as `MatPlotLib.pyplot.plot`. The `import as` syntax allows you to define nicknames for `numpy` and `maplotlib.pyplot`. Nearly any nickname can be chosen, but the Python community has settled on the nicknames `np` and `plt` for `numpy` and `maplotlib.pyplot`, so you are advised to stick with those. Using the standard nicknames makes your code more readable.

You can also import a single functions or subset of functions from a module without importing the entire module. For example, suppose you wanted to import just the natural log function `log` from NumPy. You could write

```
from numpy import log
```

To use the `log` function in a script, you would write

```
a = log(5)
```

which would assign the value `1.6094379124341003` to the variable `a`. If you wanted to import the three functions, `log`, `sin`, and `cos`, you would write

```
from numpy import log, sin, cos
```

and would similarly use them without an "`np.`" prefix. In general, we do not recommend using the the `from` *module* `import ...` way of importing functions. When reading code, it makes it harder to determine from which modules functions are imported, and can lead to clashes between similarly named functions from different modules. Nevertheless, you will see the form used in programs you encounter on the web and elsewhere so it is important to understand the syntax.

2.10 Getting help: documentation in IPython shell

Help is never far away when you are running the IPython shell. To obtain information on any valid Python or NumPy function, and many MatPlotLib functions, simply type `help(` *function* `)`, as illustrated here

```
In [1]: help(range)
range([start,] stop[, step]) -> list of integers

Return a list containing an arithmetic progression of integers.
range(i, j) returns [i, i+1, i+2, ..., j-1]; start (!) defaults
to 0.  When step is given, it specifies the increment (or
decrement).  For example, range(4) returns [0, 1, 2, 3].  The
end point is omitted! These are exactly the valid indices for a
list of 4 elements.
```

Often, the information provided can be quite extensive and you might find it useful to clear the IPython window with the `clear` command so you can easily scroll back to find the beginning of the documentation. You may have also noticed that when you type the name of a function plus the opening parenthesis, IPython displays a window showing the first dozen lines or so of the documentation on that function.

2.11 Programming is a detail-oriented activity

Now that you have a little experience with Python and computer programming, it's time for an important reminder: *Programming is a detail-oriented activity*. To be good at computer programming, to avoid frustration when programming, you must pay attention to details. A misplaced or forgotten comma or colon can keep your code from working. Note that I did not say it can "keep your code from working *well*"; it can keep your code from working at all! Worse still, little errors can make your code give erroneous answers, where your code appears to work, but in fact does not! So pay attention to the details!

This raises a second point: sometimes your code will run but give the wrong answer because of a programming error or because of a more subtle error in your algorithm. For this reason, it is important to test your code to make sure it is behaving properly. Test it to make sure it gives the correct answers for cases where you already know the correct answer or where you have some independent means of checking it. Test it in limiting cases, that is, for cases that are at the extremes of the sets of parameters you will employ. Always test your code; this is a cardinal rule of programming.

2.12 Exercises

1. A ball is thrown vertically up in the air from a height h_0 above the ground at an initial velocity v_0. Its subsequent height h and velocity v are given by the equations

$$h = h_0 + v_0 t - \tfrac{1}{2} g t^2$$
$$v = v_0 - gt$$

where $g = 9.8$ is the acceleration due to gravity in $\mathrm{m/s^2}$. Write a script that

finds the height h and velocity v at a time t after the ball is thrown. Start the script by setting $h_0 = 1.2$ (meters) and $v_0 = 5.4$ (m/s) and have your script print out the values of height and velocity (see *Note about printing*). Then use the script to find the height and velocity after 0.5 seconds. Then modify your script to find them after 2.0 seconds.

1. Write a script that defines the variables $V_0 = 10$, $a = 2.5$, and $z = 4\frac{1}{3}$, and then evaluates the expression

$$V = V_0 \left(1 - \frac{z}{\sqrt{a^2 + z^2}} \right) .$$

Then find V for $z = 8\frac{2}{3}$ and print it out (see *Note about printing*). Then find V for $z = 13$ by changing the value of z in your script.

2. Write a single Python script that calculates the following expressions:

   (a) $\dfrac{2 + e^{2.8}}{\sqrt{13} - 2}$

   (b) $\dfrac{1 - (1 + \ln 2)^{-3.5}}{1 + \sqrt{5}}$

   (c) $\sin \left(\dfrac{2 - \sqrt{2}}{2 + \sqrt{2}} \right)$

   After running your script in the IPython shell, typing `a`, `b`, or `c` at the IPython prompt should yield the value of the expressions in (a), (b), or (c), respectively.

3. A quadratic equation with the general form

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

has two solutions given by the quadratic formula

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

(a) Given a, b, and c as inputs, write a script that gives the numerical values of the two solutions. Write the constants a, b, and c as floats, and show that your script gives the correct solutions for a few test cases when the solutions are real numbers, that is, when the discriminant $b^2 - 4ac \geq 0$. Use the `print` function in your script, discussed at the end of Section 2.8.1 *Scripting Example 1*, to print out your two solutions.

(b) Written this way, however, your script gives an error message when the solutions are complex. For example, see what happens when $a = 1$, $b = 2$, and $c = 3$. You can fix this using statements in your script like `a = a+0j` after setting `a` to some float value. Thus, you can make the script work for any set of real inputs for a, b, and c. Again, use the `print` function to print out your two solutions.

CHAPTER THREE

STRINGS, LISTS, ARRAYS, AND DICTIONARIES

The most import data structure for scientific computing in Python is the **NumPy array**. NumPy arrays are used to store lists of numerical data and to represent vectors, matrices, and even tensors. NumPy arrays are designed to handle large data sets efficiently and with a minimum of fuss. The NumPy library has a large set of routines for creating, manipulating, and transforming NumPy arrays. NumPy functions, like `sqrt` and `sin`, are designed specifically to work with NumPy arrays. Core Python has an array data structure, but it's not nearly as versatile, efficient, or useful as the NumPy array. We will not be using Python arrays at all. Therefore, whenever we refer to an "array," we mean a "NumPy array."

Lists are another data structure, similar to NumPy arrays, but unlike NumPy arrays, lists are a part of core Python. Lists have a variety of uses. They are useful, for example, in various bookkeeping tasks that arise in computer programming. Like arrays, they are sometimes used to store data. However, lists do not have the specialized properties and tools that make arrays so powerful for scientific computing. So in general, we prefer arrays to lists for working with scientific data. For other tasks, lists work just fine and can even be preferable to arrays.

Strings are lists of keyboard characters as well as other characters not on your keyboard. They are not particularly interesting in scientific computing, but they are nevertheless necessary and useful. Texts on programming with Python typically devote a good deal of time and space to learning about strings and how to manipulate them. Our uses of them are rather modest, however, so we take a minimalist's approach and only introduce a few of their features.

Dictionaries are like lists, but the elements of dictionaries are accessed in a different way than for lists. The elements of lists and arrays are numbered consecutively, and to access an element of a list or an array, you simply refer to the number corresponding to its position in the sequence. The elements of dictionaries are accessed by "keys", which can be either strings or (arbitrary) integers (in no particular order). Dictionaries are an important part of core Python. However, we do not make much use of them in this introduction to scientific Python, so our discussion of them is limited.

3.1 Strings

Strings are lists of characters. Any character that you can type from a computer keyboard, plus a variety of other characters, can be elements in a string. Strings are created by enclosing a sequence of characters within a pair of single or double quotes. Examples of strings include `"Marylyn"`, `"omg"`, `"good_bad_#5f>!"`, `"{0:0.8g}"`, and `"We hold these truths ..."`. Caution: the quotes defining a given string must *both* be single or *both* be double quotes.

Strings can be assigned variable names

```
In [1]: a = "My dog's name is"
In [2]: b = "Bingo"
```

Strings can be concatenated using the "+" operator:

```
In [3]: c = a + " " + b
In [4]: c
Out[4]: "My dog's name is Bingo"
```

In forming the string `c`, we concatenated *three* strings, `a`, `b`, and a *string literal*, in this case a space `" "`, which is needed to provide a space to separate string `a` from `b`.

You will use strings for different purposes: labeling data in data files, labeling axes in plots, formatting numerical output, requesting input for your programs, as arguments in functions, *etc*.

Because numbers—digits—are also alpha numeric characters, strings can be made up of numbers:

```
In [5]: d = "927"
In [6]: e = 927
```

The variable `d` is a string while the variable `e` is an integer. What do you think happens if you try to add them by writing `d+e`? Try it out and see if you understand the result.

3.2 Lists

Python has two data structures, *lists* and *tuples*, that consist of a list of one or more elements. The elements of lists or tuples can be numbers or strings, or both. Lists (we will discuss tuples later) are defined by a pair of *square* brackets on either end with individual elements separated by commas. Here are two examples of lists:

```
In [1]: a = [0, 1, 1, 2, 3, 5, 8, 13]
In [2]: b = [5., "girl", 2+0j, "horse", 21]
```

We can access individual elements of a list using the variable name for the list with square brackets:

```
In [3]: b[0]
Out[3]: 5.0

In [4]: b[1]
Out[4]: 'girl'

In [5]: b[2]
Out[5]: (2+0j)
```

The first element of `b` is `b[0]`, the second is `b[1]`, the third is `b[2]`, and so on. Some computer languages index lists starting with 0, like Python and C, while others index lists (or things more-or-less equivalent) starting with 1 (like Fortran and Matlab). It's important to keep in mind that Python uses the former convention: lists are *zero-indexed*.

The last element of this array is `b[4]`, because `b` has 5 elements. The last element can also be accessed as `b[-1]`, no matter how many elements `b` has, and the next-to-last element of the list is `b[-2]`, *etc*. Try it out:

```
In [6]: b[4]
Out[6]: 21

In [7]: b[-1]
Out[7]: 21

In [8]: b[-2]
Out[8]: 'horse'
```

Individual elements of lists can be changed. For example:

```
In [9]: b
Out[9]: [5.0, 'girl', (2+0j), 'horse', 21]
```

```
In [10]: b[0] = b[0]+2

In [11]: b[3] = 3.14159

In [12]: b
Out[12]: [7.0, 'girl', (2+0j), 3.14159, 21]
```

Here we see that 2 was added to the previous value of `b[0]` and the string `'horse'` was replaced by the floating point number `3.14159`. We can also manipulate individual elements that are strings:

```
In [13]: b[1] = b[1] + "s & boys"

In [14]: b
Out[14]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]
```

You can also add lists, but the result might surprise you:

```
In [15]: a
Out[15]: [0, 1, 1, 2, 3, 5, 8, 13]

In [16]: a+a
Out[16]: [0, 1, 1, 2, 3, 5, 8, 13, 0, 1, 1, 2, 3, 5, 8, 13]

In [17]: a+b
Out[17]: [0, 1, 1, 2, 3, 5, 8, 13, 10.0, 'girls & boys', (2+0j),
          3.14159, 21]
```

Adding lists concatenates them, just as the "+" operator concatenates strings.

3.2.1 Slicing lists

You can access pieces of lists using the *slicing* feature of Python:

```
In [18]: b
Out[18]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]

In [19]: b[1:4]
Out[19]: ['girls & boys', (2+0j), 3.14159]

In [20]: b[3:5]
Out[20]: [3.14159, 21]
```

You access a subset of a list by specifying two indices separated by a colon "`:`". This is a powerful feature of lists that we will use often. Here are a few other useful slicing shortcuts:

```
In [21]: b[2:]
Out[21]: [(2+0j), 3.14159, 21]

In [22]: b[:3]
Out[22]: [10.0, 'girls & boys', (2+0j)]

In [23]: b[:]
Out[23]: [10.0, 'girls & boys', (2+0j), 3.14159, 21]
```

Thus, if the left slice index is `0`, you can leave it out; similarly, if the right slice index is the length of the list, you can leave it out also.

What does the following slice of an array give you?

```
In [24]: b[1:-1]
```

You can get the length of a list using Python's `len` function:

```
In [25]: len(b)
Out[25]: 5
```

3.2.2 Creating and modifying lists

Python has functions for creating and augmenting lists. The most useful is the `range` function, which can be used to create a uniformly spaced sequence of integers. The general form of the function is `range([start,] stop[, step])`, where the arguments are all integers; those in square brackets are optional:

```
In [26]: range(10)       # makes a list of 10 integers from 0 to 9
Out[26]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [27]: range(3,10)     # makes a list of 10 integers from 3 to 9
Out[27]: [3, 4, 5, 6, 7, 8, 9]

In [28]: range(0,10,2)   # makes a list of 10 integers from 0 to 9
                         # with increment 2
Out[28]: [0, 2, 4, 6, 8]
```

You can add one or more elements to the beginning or end of a list using the "+" operator:

```
In [29]: a = range(1,10,3)
In [30]: a
Out[30]: [1, 4, 7]

In [31]: a += [16, 31, 64, 127]

In [32]: a
Out[32]: [1, 4, 7, 16, 31, 64, 127]

In [33]: a = [0, 0] + a

In [34]: a
Out[34]: [0, 0, 1, 4, 7, 16, 31, 64, 127]
```

You can insert elements into a list using slicing:

```
In [35]: b = a[:5] + [101, 102] + a[5:]

In [36]: b
Out[36]: [0, 1, 1, 4, 7, 101, 102, 16, 31, 64, 127]
```

3.2.3 Tuples

Finally, a word about tuples: tuples are lists that are *immutable*. That is, once defined, the individual elements of a tuple cannot be changed. Whereas a list is written as a sequence of numbers enclosed in *square* brackets, a tuple is written as a sequence of numbers enclosed in *round* parentheses. Individual elements of a tuple are addressed in the same way as individual elements of lists are addressed, but those individual elements cannot be changed. All of this illustrated by this simple example:

```
In [37]: c = (1, 1, 2, 3, 5, 8, 13)
In [37]: c[4]
Out[38]: 5

In [39]: c[4] = 7
---------------------------------------------------------------------------
TypeError: 'tuple' object does not support item assignment
```

When we tried to change `c[4]`, the system returned an error because we are prohibited from changing an element of a tuple. Tuples offer some degree of safety when we want to define lists of immutable constants.

3.2.4 Multidimensional lists and tuples

We can also make multidimensional lists, or lists of lists. Consider, for example, a list of three elements, where each element in the list is itself a list:

```
In [40]: a = [[3, 9], [8, 5], [11, 1]]
```

Here we have a three-element list where each element consists of a two-element list. Such constructs can be useful in making tables and other structures. They also become relevant later on in our discussion of NumPy arrays and matrices, which we introduce below.

We can access the various elements of a list with a straightforward extension of the indexing scheme we have been using. The first element of the list `a` above is `a[0]`, which is `[3, 9]`; the second is `a[1]`, which is `[8, 5]`. The first element of `a[1]` is accessed as `a[1][0]`, which is 8, as illustrated below:

```
In [41]: a[0]
Out[41]: [3, 9]

In [42]: a[1]
Out[42]: [8, 5]

In [43]: a[1][0]
Out[43]: 8

In [44]: a[2][1]
Out[44]: 1
```

Multidimensional tuples work exactly like multidimensional lists, except they are immutable.

3.3 NumPy arrays

The NumPy array is the real workhorse of data structures for scientific and engineering applications. The NumPy array, formally called `ndarray` in NumPy documentation, is similar to a list but where all the elements of the list are of the same type. The elements of a NumPy array, or simply an *array*, are usually numbers, but can also be boolians, strings, or other objects. When the elements are numbers, they must all be of the same type. For example, they might be all integers or all floating point numbers.

3.3.1 Creating arrays (1-d)

NumPy has a number of functions for creating arrays. We focus on four (or five or six, depending on how you count!). The **first** of these, the `array` function, converts a list to an array:

```
In [1]: a = [0, 0, 1, 4, 7, 16, 31, 64, 127]

In [2]: b = array(a)

In [3]: b
Out[3]: array([  0,   0,   1,   4,   7, 16, 31, 64, 127])

In [4]: c = array([1, 4., -2, 7])

In [5]: c
Out[5]: array([ 1.,  4., -2.,  7.])
```

Notice that `b` is an integer array, as it was created from a list of integers. On the other hand, `c` is a floating point array even though only one of the elements of the list from which it was made was a floating point number. The `array` function automatically promotes all of the numbers to the type of the most general entry in the list, which in this case is a floating point number. In the case that elements of the list is made up of numbers and strings, all the elements become strings when an array is formed from a list.

The **second** way arrays can be created is using the NumPy `linspace` or `logspace` functions. The `linspace` function creates an array of N evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, N)`. If the third argument `N` is omitted, then `N=50`.

```
In [6]: linspace(0, 10, 5)
Out[6]: array([  0. ,   2.5,   5. ,   7.5,  10. ])
```

The `linspace` function produced 5 evenly spaced points between 0 and 10 inclusive. NumPy also has a closely related function `logspace` that produces evenly spaced points on a logarithmically spaced scale. The arguments are the same as those for `linspace` except that `start` and `stop` refer to a power of 10. That is, the array starts at 10^{start} and ends at 10^{stop}.

```
In [7]: %precision 1        # display only 1 digit after decimal
Out[7]: u'%.1f'

In [8]: logspace(1, 3, 5)
Out[8]: array([   10. ,    31.6,   100. ,   316.2,  1000. ])
```

The `logspace` function created an array with 5 points evenly spaced on a logarithmic axis starting at 10^1 and ending at 10^3. The `logspace` function is particularly useful when you want to create a log-log plot.

The **third** way arrays can be created is using the NumPy `arange` function, which is similar to the Python `range` function for creating lists. The form of the function is `arange(start, stop, step)`. If the third argument is omitted `step=1`. If the first and third arguments are omitted, then `start=0` and `step=1`.

```
In [9]: arange(0, 10, 2)
Out[9]: array([0, 2, 4, 6, 8])

In [10]: arange(0., 10, 2)
Out[10]: array([ 0., 2., 4., 6., 8.])

In [11]: arange(0, 10, 1.5)
Out[11]: array([ 0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

The `arange` function produces points evenly spaced between 0 and 10 exclusive of the final point. Notice that `arange` produces an integer array in the first case but a floating point array in the other two cases. In general `arange` produces an integer array if the arguments are all integers; making any one of the arguments a float causes the array that is created to be a float.

A **fourth** way to create an array is with the `zeros` and `ones` functions. As their names imply, they create arrays where all the elements are either zeros or ones. They each take on mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array. Left unspecified, the data type is a float. Here are three examples

```
In [12]: zeros(6)
Out[12]: array([ 0., 0., 0., 0., 0., 0.])

In [13]ones(8)
Out[13]: array([ 1., 1., 1., 1., 1., 1., 1., 1.])

In [14]ones(8, dtype=int)
Out[14]: array([1, 1, 1, 1, 1, 1, 1, 1])
```

Recap of ways to create a 1-d array

`array(a)`: Creates an array from the list `a`.

`linspace(start, stop, num)`: Returns `num` evenly spaced numbers over an interval from `start` to `stop` inclusive. [`num=50` if omitted.]

logspace(start, stop, num): Returns num logarithmically spaced numbers over an interval from 10^{start} to 10^{stop} inclusive. [num=50 if omitted.]

arange([start,] stop[, step,], dtype=None): Returns data points from start to end, exclusive, evenly spaced by step. [step=1 if omitted. start=0 and step=1 if both are omitted.]

zeros(num, dtype=float): Returns an an array of 0s with num elements. Optional dtype argument can be used to set data type; left unspecified, a float array is made.

ones(num, dtype=float): Returns an an array of 1s with num elements. Optional dtype argument can be used to set data type; left unspecified, a float array is made.

3.3.2 Mathematical operations with arrays

The utility and power of arrays in Python comes from the fact that you can process and transform all the elements of an array in one fell swoop. The best way to see how this works is look at an example.

```
In [15]: a = linspace(-1., 5, 7)

In [16]: a
Out[16]: array([-1., 0., 1., 2., 3., 4., 5.])

In [17]: a*6
Out[17]: array([ -6.,  0.,  6., 12., 18., 24., 30.])
```

Here we can see that each element of the array has been multiplied by 6. This works not only for multiplication, but for any other mathematical operation you can imagine: division, exponentiation, *etc*.

```
In [18]: a/5
Out[18]: array([-0.2, 0. , 0.2, 0.4, 0.6, 0.8, 1. ])

In [19]: a**3
Out[19]: array([  -1.,    0.,    1.,    8.,   27.,   64., 125.])

In [20]: a+4
Out[20]: array([ 3., 4., 5., 6., 7., 8., 9.])

In [21]: a-10
```

```
Out[21]: array([-11., -10., -9., -8., -7., -6., -5.])

In [22]: (a+3)*2
Out[22]: array([  4.,   6.,   8., 10., 12., 14., 16.])

In [23]: sin(a)
Out[23]: array([-0.84147098, 0.        , 0.84147098, 0.90929743,
                 0.14112001, -0.7568025 , -0.95892427])

In [24]: exp(-a)
Out[24]: array([ 2.71828183, 1.        , 0.36787944, 0.13533528,
                 0.04978707, 0.01831564, 0.00673795])

In [25]: 1. + exp(-a)
Out[25]: array([ 3.71828183, 2.        , 1.36787944, 1.13533528,
                 1.04978707, 1.01831564, 1.00673795])

In [26]: b = 5*ones(8)

In [27]: b
Out[27]: array([ 5., 5., 5., 5., 5., 5., 5., 5.])

In [28] b += 4

In [29] b
Out[29]: array([ 9., 9., 9., 9., 9., 9., 9., 9.])
```

In each case, you can see that the same mathematical operations are performed individually on each element of each array. Even fairly complex algebraic computations can be carried out this way.

Let's say you want to create an x-y data set of $y = \cos x$ *vs.* x over the interval from -3.14 to 3.14. Here is how you might do it.

```
In [30]: x = linspace(-3.14, 3.14, 21)

In [31]: y = cos(x)

In [32]: x
Out[32]: array([-3.14 , -2.826, -2.512, -2.198, -1.884, -1.57 ,
                -1.256, -0.942, -0.628, -0.314, 0.   , 0.314,
                 0.628, 0.942, 1.256, 1.57 , 1.884, 2.198,
                 2.512, 2.826, 3.14 ])

In [33]: y
```

```
Out[33]: array([ -1.000e+00, -9.506e-01, -8.083e-01,
                 -5.869e-01, -3.081e-01,  7.963e-04,
                  3.096e-01,  5.882e-01,  8.092e-01,
                  9.511e-01,  1.000e+00,  9.511e-01,
                  8.092e-01,  5.882e-01,  3.096e-01,
                  7.963e-04, -3.081e-01, -5.869e-01,
                 -8.083e-01, -9.506e-01, -1.000e+00])
```

You can use arrays as inputs for any of the functions introduced in the section on *Python functions: a first look*. You might well wonder what happens if Python encounters an illegal operation. Here is one example.

```
In [34]: a
Out[34]: array([-1., 0., 1., 2., 3., 4., 5.])

In [35]: log(a)
-c:1: RuntimeWarning: divide by zero encountered in log
-c:1: RuntimeWarning: invalid value encountered in log
Out[35]: array([   nan,  -inf, 0.    , 0.693, 1.099, 1.386,
                 1.609])
```

We see that NumPy calculates the logarithm where it can, and returns `nan` (not a number) for an illegal operation, taking the logarithm of a negative number, and `-inf`, or $-\infty$ for the logarithm of zero. The other values in the array are correctly reported. NumPy also prints out a warning message to let you know that something untoward has occurred.

Arrays can also be added, subtracted, multiplied, and divided by each other on an element-by-element basis, provided the two arrays have the same size. Consider adding the two arrays `a` and `b` defined below:

```
In [36]: a = array([34., -12, 5.])

In [37]: b = array([68., 5.0, 20.])

In [38]: a+b
Out[38]: array([ 102.,   -7.,   25.])
```

The result is that each element of the two arrays are added. Similar results are obtained for subtraction, multiplication, and division:

```
In [39]: a-b
Out[39]: array([-34., -17., -15.])

In [40]: a*b
Out[40]: array([ 2312.,   -60.,   100.])
```

```
In [41]: a/b
Out[41]: array([ 0.5 , -2.4 ,  0.25])
```

These kinds of operations with arrays are called *vectorized* operations because the entire array, or "vector", is processed as a unit. Vectorized operations are much faster than processing each element of arrays one by one. Writing code that takes advantage of these kinds of vectorized operations is almost always to be preferred to other means of accomplishing the same task, both because it is faster and because it is usually syntactically simpler. You will see examples of this later on when we discuss loops in Chapter 6.

3.3.3 Slicing and addressing arrays

Arrays can be sliced in the same ways that strings and lists can be sliced—any way you slice it! Ditto for accessing individual array elements: 1-d arrays are addressed the same way as strings and lists. Slicing, combined with the vectorized operations can lead to some pretty compact and powerful code.

Suppose, for example, that we have two arrays `y`, and `t` for position *vs* time of a falling object, say a ball, and we want to use these data to calculate the velocity as a function of time:

```
In [42]: y = array([ 0. , 1.3,  5. , 10.9, 18.9, 28.7, 40. ])

In [43]: t = array([ 0. , 0.49, 1. , 1.5 , 2.08, 2.55, 3.2 ])
```

We can get find the average velocity for time interval i by the formula

$$v_i = \frac{y_i - y_{i-1}}{t_i - t_{i-1}}$$

We can easily calculate the entire array of of velocities using the slicing and vectorized subtraction properties of NumPy arrays by noting that we can create two `y` arrays displaced by one index

```
In [44]: y[:-1]
Out[44]: array([  0. ,   1.3,   5. ,  10.9,  18.9,  28.7])

In [45]: y[1:]
Out[45]: array([  1.3,   5. ,  10.9,  18.9,  28.7,  40. ])
```

The element-by-element difference of these two arrays is

```
In [46]: y[1:]-y[:-1]
Out[46]: array([  1.3,   3.7,   5.9,   8. ,   9.8,  11.3])
```

The element-by-element difference of the two arrays `y[1:]-y[:-1]` divided by `t[1:]-t[:-1]` gives the entire array of velocities

```
In [47]: v = (y[1:]-y[:-1])/(t[1:]-t[:-1])

In [48]: v
Out[48]: array([  2.65306122,   7.25490196,  11.8        ,
                 13.79310345,  20.85106383,  17.38461538])
```

Of course, these are the average velocities over each interval so the times best associated with each interval are the times halfway in between the original time array, which we can calculate using a similar trick of slicing:

```
In [49]: tv = (t[1:]+t[:-1])/2.

In [50]: tv
Out[50]: array([ 0.245,  0.745,  1.25 ,  1.79 ,  2.315,  2.875])
```

3.3.4 Multi-dimensional arrays and matrices

So far we have examined only one-dimensional NumPy arrays, that is, arrays that consist of a simple sequence of numbers. However, NumPy arrays can be used to represent multidimensional arrays. For example, you may be familiar with the concept of a *matrix*, which consists of a series of rows and columns of numbers. Matrices can be represented using two-dimensional NumPy arrays. Higher dimension arrays can also be created as the application demands.

Creating NumPy arrays

There are a number of ways of creating multidimensional NumPy arrays. The most straightforward way is to convert a list to an array using NumPy's `array` function, which we demonstrate here:

```
In [51]: b = array([[1., 4, 5], [9, 7, 4]])

In [52]: b
Out[52]: array([[1., 4., 5.],
                [9., 7., 4.]])
```

Notice the syntax used above in which two one-dimensional lists `[1., 4, 5]` and `[9, 7, 4]` are enclosed in square brackets to make a two-dimensional list. The `array` function converts the two-dimensional list, a structure we introduced earlier, to a two-dimensional array. When it makes the conversion from a list to an array, the array function makes all the elements have the same data type as the most complex entry, in this case a float. This points out an important difference between NumPy arrays and lists: all elements of a NumPy array must be of the same data type: floats, or integers, or complex numbers, *etc*.

There are a number of other functions for creating arrays. For example, a 3 row by 4 column array or 3×4 array with all the elements filled with 1 can be created using the `ones` function introduced earlier.

```
In [53]: a = ones((3,4), dtype=float)

In [54]: a
Out[54]: array([[ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.],
                [ 1.,  1.,  1.,  1.]])
```

Using a tuple to specify the size of the array in the first argument of the `ones` function creates a multidimensional array, in this case a two-dimensional array with the two elements of the tuple specifying the number of rows and columns, respectively. The `zeros` function can be used in the same way to create a matrix or other multidimensional array of zeros.

The `eye(N)` function creates an $N \times N$ two-dimensional identity matrix with ones along the diagonal:

```
In [55]: eye(4)
Out[55]: array([[ 1.,  0.,  0.,  0.],
                [ 0.,  1.,  0.,  0.],
                [ 0.,  0.,  1.,  0.],
                [ 0.,  0.,  0.,  1.]])
```

Multidimensional arrays can also be created from one-dimensional arrays using the `reshape` function. For example, a 2×3 array can be created as follows:

```
In [56]: c = arange(6)

In [57]: c
Out[57]: array([0, 1, 2, 3, 4, 5])

In [58]: c = reshape(c, (2,3))
In [59]: c
Out[59]: array([[0, 1, 2],
               [3, 4, 5]])
```

Indexing multidimensional arrays

The individual elements of arrays can be accessed in the same way as for lists:

```
In [60]: b[0][2]
Out[60]: 5.
```

You can also use the syntax

```
In [61]: b[0,2]
Out[61]: 5.
```

which means the same thing. Caution: both the `b[0][2]` and the `b[0,2]` syntax work for NumPy arrays and mean exactly the same thing; for lists only the `b[0][2]` syntax works.

Matrix operations

Addition, subtraction, multiplication, division, and exponentiation all work with multidimensional arrays the same way they work with one dimensional arrays, on an element-by-element basis, as illustrated below:

```
In [62]: b
Out[62]: array([[ 1., 4., 5.],
               [ 9., 7., 4.]])

In [63]: 2*b
Out[63]: array([[  2.,  8., 10.],
              [ 18., 14.,  8.]])

In [64]: b/4.
Out[64]: array([[ 0.25, 1.  , 1.25],
               [ 2.25, 1.75, 1.  ]])

In [65]: b**2
Out[65]: array([[  1., 16., 25.],
               [ 81., 49., 16.]])

In [66]: b-2
```

```
Out[66]: array([[-1., 2., 3.],
                [ 7., 5., 2.]])
```

Functions also act on an element-to-element basis

```
In [67]: sin(b)
Out[67]: array([[ 0.84147098, -0.7568025 , -0.95892427],
                [ 0.41211849, 0.6569866 , -0.7568025 ]])
```

Multiplying two arrays together is done on an element-by-element basis. Using the matrices b and c defined above, multiplying them together gives

```
In [68]: b
Out[68]: array([[ 1., 4., 5.],
                [ 9., 7., 4.]])

In [69]: c
Out[69]: array([[0, 1, 2],
                [3, 4, 5]])

In [70]: b*c
Out[70]: array([[  0.,   4., 10.],
                [ 27., 28., 20.]])
```

Of course, this requires that both arrays have the same shape. Beware: array multiplication, done on an element-by-element basis, is not the same as matrix multiplication as defined in linear algebra. Therefore, we distinguish between *array* multiplication and *matrix* multiplication in Python.

Normal matrix multiplication is done with NumPy's `dot` function. For example, defining `d` to be the transpose of `c` using using the `array` function's `T` transpose method creates an array with the correct dimensions that we can use to find the matrix product of `b` and `d`:

```
In [71]: d = c.T

In [72]: d
Out[72]: array([[0, 3],
                [1, 4],
                [2, 5]])

In [73]: dot(b,d)
Out[73]: array([[ 14., 44.],
                [ 15., 75.]])
```

3.3.5 Differences between lists and arrays

While lists and arrays are superficially similar—they are both multi-element data structures—they behave quite differently in a number of circumstances. First of all, lists are part of the core Python programming language; arrays are a part of the numerical computing package NumPy. Therefore, you have access to NumPy arrays only if you load the NumPy package using the `import` command.

Here we list some of the differences between Python lists and NumPy arrays, and why you might prefer to use one or the other depending on the circumstance.

- **The elements of a NumPy array must all be of the same type**, whereas the elements of a Python list can be of completely different types.
- **NumPy arrays support "vectorized" operations** like element-by-element addition and multiplication. This is made possible, in part, by the fact that all elements of the array have the same type, which allows array operations like element-by-element addition and multiplication to be carried out by very efficient C loops. Such "vectorized" operations on arrays, which includes operations by NumPy functions such as `numpy.sin` and `numpy.exp`, are much faster than operations performed by loops using the core Python `math` package functions, such as `math.sin` and `math.exp`, that act only on individual elements and not on whole lists or arrays.
- **Adding one or more additional elements to a NumPy array creates a new array and destroys the old one.** Therefore it can be very inefficient to build up large arrays by appending elements one by one, especially if the array is very large, because you repeatedly create and destroy large arrays. By contrast, elements can be added to a list without creating a whole new list. If you need to build an array element by element, it is usually better to build it as a list, and then convert it to an array when the list is complete. At this point, it may be difficult for you to appreciate how and under what circumstances you might want build up an array element by element. Examples are provided later on: for an example see the section on *Looping over arrays in user-defined functions*.

3.4 Dictionaries

A Python *list* is a collection of Python objects indexed by an ordered sequence of integers starting from zero. A **dictionary** is also collection of Python objects, just like a list, but one that is indexed by strings or numbers (not necessarily integers and not in any particular order) or even tuples! For example, suppose we want to make a dictionary of room numbers indexed by the name of the person who occupies each room. We create our dictionary using curly brackets `{...}`.

```
In [1]: room = {"Emma":309, "Jacob":582, "Olivia":764}
```

The dictionary above has three entries separated by commas, each entry consisting of a **key**, which in this case is a string, and a **value**, which in this case is a room number. Each key and its value are separated by a colon. The syntax for accessing the various entries is similar to a that of a list, with the key replacing the index number. For example, to find out the room number of Olivia, we type

```
In [2]: room["Olivia"]
Out[2]: 764
```

The key need not be a string; it can be any immutable Python object. So a key can be a string, an integer, or even a tuple, but it can't be a list. And the elements accessed by their keys need not be a string, but can be almost any legitimate Python object, just as for lists. Here is a weird example

```
In [3]: weird = {"tank":52, 846:"horse", "bones":[23,
   ...: "fox", "grass"], "phrase":"I am here"}

In [4]: weird["tank"]
Out[4]: 52

In [5]: weird[846]
Out[5]: 'horse'

In [6]: weird["bones"]
Out[6]: [23, 'fox', 'grass']

In [7]: weird["phrase"]
Out[7]: 'I am here'
```

Dictionaries can be built up and added to in a straightforward manner

```
In [8]: d = {}

In [9]: d["last name"] = "Alberts"

In [10]: d["first name"] = "Marie"

In [11]: d["birthday"] = "January 27"

In [12]: d
Out[12]: {'birthday': 'January 27', 'first name': 'Marie',
          'last name': 'Alberts'}
```

You can get a list of all the keys or values of a dictionary by typing the dictionary name followed by `.keys()` or `.values()`.

```
In [13]: d.keys()
Out[13]: ['last name', 'first name', 'birthday']

In [14]: d.values()
Out[14]: ['Alberts', 'Marie', 'January 27']
```

In other languages, data types similar to Python dictionaries may be called “hashmaps” or “associative arrays”, so you may see such term used if you read on the web about dictionaries.

3.5 Random numbers

Random numbers are widely used in science and engineering computations. They can be used to simulate noisy data, or to model physical phenomena like the distribution of velocities of molecules in a gas, or to act like the roll of dice in a game. There are even methods for numerically evaluating multi-dimensional integrals using random numbers.

The basic idea of a random number generator is that it should be able to produce a sequence of numbers that are distributed according to some predetermined distribution function. NumPy provides a number of such random number generators in its library `numpy.random`. Here we focus on three: `rand`, `randn`, and `randint`.

3.5.1 Uniformly distributed random numbers

The `rand(num)` function creates and array of `num` floats uniformly distributed on the interval from 0 to 1.

```
In [1]: rand()
Out[1]: 0.5885170150833566

In [2]: rand(5)
Out[2]: array([ 0.85586399,  0.21183612,  0.80235691,
                0.65943861,  0.25519987])
```

If `rand` has no argument, a single random number is generated. Otherwise, the argument specifies the number of size of the array of random numbers that is created.

If you want random numbers uniformly distributed over some other interval, say from a to b, then you can do that simply by stretching the interval so that it has a width of $b - a$

and displacing the lower limit from 0 to a. The following statements produce random numbers uniformly distributed from 10 to 20:

```
In [3]: a, b = 10, 20

In [4]: (b-a)*rand(20) + a
Out[4]: array([ 10.99031149, 18.11685555, 11.48302458,
                18.25559651, 17.55568817, 11.86290145,
                17.84258224, 12.1309852 , 14.30479884,
                12.05787676, 19.63135536, 16.58552886,
                19.15872073, 17.59104303, 11.48499468,
                10.16094915, 13.95534353, 18.21502143,
                19.61360422, 19.21058726])
```

3.5.2 Normally distributed random numbers

The function `randn(num)` produces a *normal* or *Gaussian* distribution of `num` random numbers with a mean of 0 and a standard deviation of 1. That is, they are distributed according to

$$P(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \; .$$

The figure below shows histograms for the distributions of 10,000 random numbers generated by the `np.random.rand` (blue) and `np.random.randn` (green) functions. As advertised, the `np.random.rand` function produces an array of random numbers that is uniformly distributed between the values of 0 and 1, while `np.random.randn` function produces an array of random numbers that follows a distribution of mean 0 and standard deviation 1.

If we want a random numbers with a Gaussian distribution of width σ centered about x_0, we stretch the interval by a factor of σ and displace it by x_0. The following code produces 20 random numbers normally distributed around 15 with a width of 10:

```
In [5]: x0, sigma = 15, 10

In [6]: sigma*randn(20) + x0
Out[6]: array([  9.36069244, 13.49260733,  6.12550102,
                18.50471781,  9.89499319, 14.09576728,
                12.45076637, 17.83073628,  2.95085564,
                18.2756275 , 14.781659  , 31.80264078,
                20.8457924 , 13.87890601, 25.41433678,
                15.44237582, 21.2385386 , -3.91668973,
                31.19120157, 26.24254326])
```

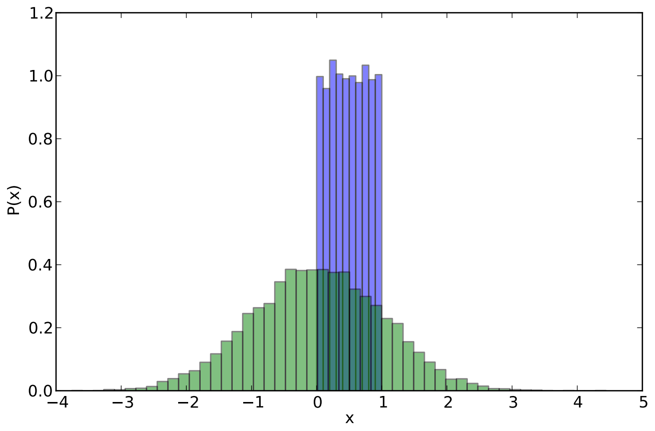


Figure 3.1: Histograms of random numbers.

3.5.3 Random distribution of integers

The function `randint(low, high, num)` produces a uniform random distribution of `num` integers between `low` (inculsive) and `high` (exclsusive). For example, we can simulate a dozen rolls a single die with the following statement

```
In [7]: randint(1, 7, 12)
Out[7]: array([6, 2, 1, 5, 4, 6, 3, 6, 5, 4, 6, 2])
```

3.5.4 Loading random number functions

When working within the IPython shell, you can use the random number functions simply by writing `rand(10)`, `randn(10)`, or, `randint(10)`, because the `np.random` library is loaded when IPython is launched. However, to use these functions in a script or program, you need to load them from the `numpy.random` library, as discussed in section on *Importing Modules*, and as illustrated in the above program for making the histogram in the above figure.

Recap of random number generators

Random number generators must be imported from the `numpy.random` library. For more information, see http://docs.scipy.org/doc/numpy/reference/routines.random.html

`rand(num)` generates an array of `num` random floats uniformly distributed on the interval from 0 to 1.

`randn(num)` generates an array of `num` random floats normally distributed with a width of 1.

`randint(low, high, num)` generates an array of `num` random integers between `low` (inclusive) and `high` exclusive.

3.6 Exercises

1. Create at array of 9 evenly spaced numbers going from 0 to 29 (inclusive) and give it the variable name `r`. Find the square of each element of the array (as simply as possible). Find twice the value of each element of the array in two different ways: (*i*) using addition and (*ii*) using multiplication.

2. Create the following arrays:

   (a) an array of 100 elements all equal to e, the base of the natural logarithm;

   (b) an array in 1-degree increments of all the angles in degrees from 0 to 360 degrees;

   (c) an array in 1-degree increments of all the angles in radians from 0 to 360 degrees;

   (d) an array from 12 to 17, not including 17, in 0.2 increments;

   (e) an array from 12 to 17, including 17, in 0.2 increments.

3. The position of a ball at time t dropped with zero initial velocity from a height h_0 is given by

   $$y = h_0 - \tfrac{1}{2}gt^2$$

   where $g = 9.8\ \mathrm{m/s}^2$. Suppose $h_0 = 10\ \mathrm{m}$. Find the sequence of times when the ball passes each half meter assuming the ball is dropped at $t = 0$. Hint: Create a NumPy array for y that goes from 10 to 0 in increments of -0.5 using the `arange` function. Solving the above equation for t, show that

   $$t = \sqrt{\frac{2(h_0 - y)}{g}}\ .$$

   Using this equation and the array you created, find the sequence of times when the ball passes each half meter. Save your code as a Python script. It should yield the following results for the `y` and `t` arrays:

```
In [2]: y
Out[2]: array([10. ,  9.5,  9. ,  8.5,  8. ,  7.5,  7. ,  6.5,
                6. ,  5.5,  5. ,  4.5,  4. ,  3.5,  3. ,  2.5,
                2. ,  1.5,  1. ,  0.5])

In [3]: t
Out[3]: array([ 0.        ,  0.31943828,  0.45175395,
0.55328334, 0.63887656, 0.71428571,
0.7824608 , 0.84515425, 0.9035079 ,
0.95831485, 1.01015254, 1.05945693,
1.10656667, 1.15175111, 1.19522861,
1.23717915, 1.27775313, 1.31707778,
1.35526185, 1.39239919])
```

4. Recalling that the average velocity over an interval Δt is defined as $\bar{v} = \Delta y/\Delta t$, find the average velocity for each time interval in the previous problem using NumPy arrays. Keep in mind that the number of time intervals is one less than the number of times. Hint: What are the arrays `y[1:20]` and `y[0:19]`? What does the array `y[1:20]-y[0:19]` represent? (Try printing out the two arrays from the IPython shell.) Using this last array and a similar one involving time, find the array of average velocities. Bonus: Can you think of a more elegant way of representing `y[1:20]-y[0:19]` that does not make explicit reference to the number of elements in the `y` array—one that would work for any length array?

   You should get the following answer for the array of velocities:

   ```
   In [5]: v
   Out[5]: array([-1.56524758,  -3.77884195,  -4.9246827 ,
                  -5.84158351,  -6.63049517,  -7.3340579 ,
                  -7.97531375,  -8.56844457,  -9.12293148,
                  -9.64549022,  -10.14108641, -10.61351563,
                  -11.06575711, -11.50020061, -11.91879801,
                  -12.32316816, -12.71467146, -13.09446421,
                  -13.46353913])
   ```

CHAPTER

FOUR

INPUT AND OUTPUT

A good relationship depends on good communication. In this chapter you learn how to communicate with Python. Of course, communicating is a two-way street: input and output. Generally, when you have Python perform some task, you need to feed it information—input. When it is done with that task, it reports back to you the results of its calculations—output.

There are two venues for input that concern us: the computer keyboard and the input data file. Similarly, there are two venues for output: the computer screen and the output data file. We start with input from the keyboard and output to the computer screen. Then we deal with data file input and output—or "io."

4.1 Keyboard input

Many computer programs need input from the user. In *Scripting Example 1*, the program needed the distance traveled as an input in order to determine the duration of the trip and the cost of the gasoline. As you might like to use this same script to determine the cost of several different trips, it would be useful if the program requested that input when it was run from the IPython shell.

Python has a function called `raw_input` (renamed `input` in Python 3) for getting input from the user and assigning it a variable name. It has the form

```
strname = raw_input("prompt to user")
```

When the `raw_input` statement is executed, it prints the text in the quotes to the computer screen and waits for input from the user. The user types a string of characters and presses the return key. The `raw_input` function then assigns that string to the variable name on the right of the assignment operator =.

Let's try it out this snippet of code in the IPython shell.

```
In [1]: distance = raw_input("Input distance of trip in miles: ")

Input distance of trip in miles:
```

Python prints out the string argument of the `raw_input` function and waits for a response from you. Let's go ahead and type `450` for "450 miles" and press return. Now type the variable name `distance` to see its value

```
In [2]: distance
Out[2]: u'450'
```

The value of the `distance` is `450` as expected, but it is a string (the `u` stands for "unicode" which refers to the string coding system Python uses). Because we want to use `450` as a number and not a distance, we need to convert it from a string to a number. We can do that with the `eval` function by writing

```
In [3]: distance = eval(distance)

In [4]: distance
Out[4]: 450
```

The `eval` function has converted `distance` to an integer. This is fine and we are ready to move on. However, we might prefer that `distance` be a float instead of an integer. There are two ways to do this. We could assume the user is very smart and will type "`450.`" instead of "`450`", which will cause distance to be a float when `eval` does the conversion. That is, the number 450 is dynamically typed to be a float or an integer depending on whether or not the user uses a decimal point. Alternatively, we could use the function `float` in place of `eval`, which would ensure that `distance` is a floating point variable. Thus, our code would look like this (including the user response):

```
In [5]: distance = raw_input("Input distance of trip in miles: ")

Input distance of trip in miles: 450

In [5]: distance
Out[5]: u'450'

In [7]: distance = float(distance)

In [8]: distance
Out[8]: 450.0
```

Now let's incorporate what we have learned into the code we wrote for *Scripting Example 1*

```
1   # Calculates time, gallons of gas used, and cost of gasoline for
2   # a trip
3
4   distance = raw_input("Input distance of trip in miles: ")
5   distance = float(distance)
6
7   mpg = 30.                   # car mileage
8   speed = 60.                 # average speed
9   costPerGallon = 4.10        # price of gas
10
11  time = distance/speed
12  gallons = distance/mpg
13  cost = gallons*costPerGallon
```

Lines 4 and 5 can be combined into a single line, which is a little more efficient:

```
distance = float(raw_input("Input distance of trip in miles: "))
```

Whether you use `float` or `int` or `eval` depends on whether you want a float, an integer, or a dynamically typed variable. In this program, it doesn't matter.

Now you can simply run the program and then type `time`, `gallons`, and `cost` to view the results of the calculations done by the program.

Before moving on to output, we note that sometimes you may want string input rather that numerical input. For example, you might want the user to input their name, in which case you would simply use the `raw_input` function without converting its output.

4.2 Screen output

It would be much more convenient if the program in the previous section would simply write its output to the computer screen, instead of requiring the user to type `time`, `gallons`, and `cost` to view the results. Fortunately, this can be accomplished very simply using Python's `print` function. For example, simply including the statement `print(time, gallons, cost)` after line 12, running the program would give the following result:

```
In [1]: run myTripIO.py

What is the distance of your trip in miles? 450
(7.5, 15.0, 61.49999999999999)
```

The program prints out the results as a tuple of time (in hours), gasoline used (in gallons), and cost (in dollars). Of course, the program doesn't give the user a clue as to which quantity is which. The user has to know.

4.2.1 Formatting output with `str.format()`

We can clean up the output of the example above and make it considerably more user friendly. The program below demonstrates how to do this.

```python
# Calculates time, gallons of gas used, and cost of gasoline for
# a trip

distance = float(raw_input("Input distance of trip in miles: "))
mpg = 30.               # car mileage
speed = 60.             # average speed
costPerGallon = 4.10    # price of gas

time = distance/speed
gallons = distance/mpg
cost = gallons*costPerGallon

print("\nDuration of trip = {0:0.1f} hours".format(time))
print("Gasoline used = {0:0.1f} gallons (@ {1:0.0f} mpg)"
      .format(gallons, mpg))
print("Cost of gasoline = ${0:0.2f} (@ ${1:0.2f}/gallon)"
      .format(cost, costPerGallon))
```

The final two `print` function calls in this script are continued on a second line in order to improve readability. Running this program, with the distance provided by the user, gives

```
In [9]: run myTripNiceIO.py

What is the trip distance in miles? 450

Duration of trip = 7.5 hours
Gasoline used = 15.0 gallons (@ 30 mpg)
Cost of gasoline = $61.50 (@ $4.10/gallon)
```

Now the output is presented in a way that is immediately understandable to the user. Moreover, the numerical output is formatted with an appropriate number of digits to the right of the decimal point. For good measure, we also included the assumed mileage (30 mpg) and the cost of the gasoline. All of this is controlled by the `str.format()` function within the `print` function.

The argument of the `print` function is of the form `str.format()` where `str` is a string that contains text that is written to be the screen, as well as certain format specifiers contained in curly braces `{}`. The `format` function contains the list of variables that are to be printed.

- The `\n` at the start of the string in the `print` statement on line 12 in the newline character. It creates the blank line before the output is printed.
- The positions of the curly braces determine where the variables in the `format` function at the end of the statement are printed.
- The format string inside the curly braces specifies how each variable in the `format` function is printed.
- The number before the colon in the format string specifies which variable in the list in the `format` function is printed. Remember, Python is zero-indexed, so 0 means the first variable is printed, 1 means the second variable, *etc*.
- The zero after the colon specifies the *minimum* number of spaces reserved for printing out the variable in the format function. A zero means that only as many spaces as needed will be used.
- The number after the period specifies the number of digits to the right of the decimal point that will be printed: `1` for `time` and `gallons` and `2` for `cost`.
- The `f` specifies that a number with a fixed number of decimal points. If the `f` format specifier is replaced with `e`, then the number is printed out in exponential format (scientific notation).

In addition to `f` and `e` format types, there are two more that are commonly used: `d` for integers (digits) and `s` for strings. There are, in fact, many more formatting possibilities. Python has a whole "Format Specification Mini-Language" that is documented at http://docs.python.org/library/string.html#formatspec. It's very flexible but arcane. You might find it simplest to look at the "Format examples" section further down the same web page.

The program below illustrates most of the formatting you will need for writing a few variables, be they strings, integers, or floats, to screen or to data files (which we discuss in the next section).

```
string1 = "How"
string2 = "are you my friend?"
int1 = 34
int2 = 942885
float1 = -3.0
float2 = 3.141592653589793e-14
```

```
print(' ***')
print(string1)
print(string1 + ' ' + string2)
print(' 1. {} {}'.format(string1, string2))
print(' 2. {0:s} {1:s}'.format(string1, string2))
print(' 3. {0:s} {0:s} {1:s} - {0:s} {1:s}'
      .format(string1, string2))
print(' 4. {0:10s}{1:5s}'
      .format(string1, string2))
print(' ***')
print(int1, int2)
print(' 6. {0:d} {1:d}'.format(int1, int2))
print(' 7. {0:8d} {1:10d}'.format(int1, int2))
print(' ***')
print(' 8. {0:0.3f}'.format(float1))
print(' 9. {0:6.3f}'.format(float1))
print('10. {0:8.3f}'.format(float1))
print(2*'11. {0:8.3f}'.format(float1))
print(' ***')
print('12. {0:0.3e}'.format(float2))
print('13. {0:10.3e}'.format(float2))
print('14. {0:10.3f}'.format(float2))
print(' ***')
print('15. 12345678901234567890')
print('16. {0:s}--{1:8d},{2:10.3e}'
      .format(string2, int1, float2))
```

Here is the output:

```
 ***
How
How are you my friend?
 1. How are you my friend?
 2. How are you my friend?
 3. How How are you my friend? - How are you my friend?
 4. How       are you my friend?
 ***
(34, 942885)
 6. 34 942885
 7.       34     942885
 ***
 8. -3.000
 9. -3.000
10.   -3.000
11.   -3.00011.   -3.000
```

```
 ***
12. 3.142e-14
13.  3.142e-14
14.       0.000
 ***
15. 12345678901234567890
16. are you my friend?--      34, 3.142e-14
```

Successive empty brackets `{}` like those that appear in the statement above `print(' 1.  {} {}'.format(string1, string2))` are numbered consecutively starting at 0 and will print out whatever variables appear inside the `format()` method using their default format.

Finally, note that the code starting on lines 14 and 16 each are split into two lines. We have done this so that the lines fit on the page without running off the edge. Python allows you to break lines up like this to improve readability.

4.2.2 Printing arrays

Formatting NumPy arrays for printing requires another approach. As an example, let's create an array and then format it in various ways. From the IPython terminal

```
In [10]: a = linspace(3, 19, 7)
In [11]: print(a)
[  3.          5.66666667   8.33333333  11.
   13.66666667  16.33333333  19.        ]
```

Simply using the `print` function does print out the array, but perhaps not in the format you desire. To control the output format, you use the NumPy function `set_printoptions`. For example, suppose you want to see no more than two digits to the right of the decimal point. Then you simply write

```
In [12]: set_printoptions(precision=2)
In [13]: print(a)
[  3.     5.67   8.33  11.    13.67  16.33  19.  ]
```

If you want to change the number of digits to the right of the decimal point to 4, you set the keyword argument `precision` to 4

```
In [14]: set_printoptions(precision=4)
In [15]: print(a)
[  3.       5.6667   8.3333  11.      13.6667  16.3333  19.    ]
```

Suppose you want to use scientific notation. The method for doing it is somewhat arcane, using something called a `lambda` function. For now, you don't need to understand how it works to use it. Just follow the examples shown below, which illustrate several different output formats using the `print` function with NumPy arrays.

```
In [16]: set_printoptions(
    ...: formatter={'float': lambda x: format(x, '6.2e')})

In [17]: print(a)
[3.00e+00 5.67e+00 8.33e+00 1.10e+01 1.37e+01 1.63e+01 1.90e+01]
```

To specify the format of the output, you use the `formatter` keyword argument. The first entry to the right of the curly bracket is a string that can be `'float'`, as it is above, or `'int'`, or `'str'`, or a number of other data types that you can look up in the online NumPy documentation. The only other thing you should change is the format specifier string. In the above example, it is `'6.2e'`, specifying that Python should allocate at least 6 spaces, with 2 digits to the right of the decimal point in scientific (exponential) notation. For fixed width floats with 3 digits to the right of the decimal point, use the `f` in place of the `e` format specifier, as follows

```
In [18]: set_printoptions(
    ...: formatter={'float': lambda x: format(x, '6.3f')})
In [19]: print(a)
[ 3.000  5.667  8.333 11.000 13.667 16.333 19.000]
```

To return to the default format, type the following

```
In [20]: set_printoptions(precision=8)
In [21]: print(a)
[  3.           5.66666667   8.33333333  11.
  13.66666667  16.33333333  19.        ]
```

The `set_printoptions` is a NumPy function, so if you use it in a script or program, you should call it by writing `np.set_printoptions`.

4.3 File input

4.3.1 Reading data from a text file

Often you would like to analyze data that you have stored in a text file. Consider, for example, the data file below for an experiment measuring the free fall of a mass.

```
Data for falling mass experiment
Date: 16-Aug-2013
Data taken by Lauren and John

data point      time (sec)      height (mm)     uncertainty (mm)
    0               0.0             180             3.5
    1               0.5             182             4.5
    2               1.0             178             4.0
    3               1.5             165             5.5
    4               2.0             160             2.5
    5               2.5             148             3.0
    6               3.0             136             2.5
    7               3.5             120             3.0
    8               4.0              99             4.0
    9               4.5              83             2.5
   10               5.0              55             3.6
   11               5.5              35             1.75
   12               6.0               5             0.75
```

We would like to read these data into a Python program, associating the data in each column with an appropriately named array. While there are a multitude of ways to do this in Python, the simplest by far is to use the NumPy `loadtxt` function, whose use we illustrate here. Suppose that the name of the text file is `MyData.txt`. Then we can read the data into four different arrays with the following statement

```
In [1]: dataPt, time, height, error = np.loadtxt("MyData.txt",
      skiprows=5 , unpack=True)
```

In this case, the `loadtxt` function takes three arguments: the first is a string that is the name of the file to be read, the second tells `loadtxt` to skip the first 5 lines at the top of file, sometimes called the *header*, and the third tells `loadtxt` to output the data (*unpack* the data) so that it can be directly read into arrays. `loadtxt` reads however many columns of data are present in the text file to the array names listed to the left of the "=" sign. The names labeling the columns in the text file are not used, but you are free to choose the same or similar names, of course, as long as they are legal array names. By the way, for the above `loadtxt` call to work, the file `MyData.txt` should be in the current working directory of the IPython shell. Otherwise, you need to specify the directory path with the file name.

It is critically important that the data file be a *text* file. It cannot be a MSWord file, for example, or an Excel file, or anything other than a plain text file. Such files can be created by a text editor programs like **Notepad** and **Notepad++** (for a PC) or **TextEdit** and **TextWrangler** (for a Mac). They can also be created by MSWord and Excel provided you explicitly save the files as text files. **Beware**: You should exit any text file you make and save it with a program that allows you to save the text file using **UNIX**-type formatting, which uses a *line feed* (LF) to end a line. Some programs, like MSWord under Windows, may include a carriage return (CR) character, which can confuse `loadtxt`. Note that we give the file name a `.txt` *extension*, which indicates to most operating systems that this is a *text* file, as opposed to an Excel file, for example, which might have a `.xlsx` or `.xls` extension.

If you don't want to read in all the columns of data, you can specify which columns to read in using the `usecols` key word. For example, the call

```
In [2]: time, height = loadtxt('MyData.txt', skiprows=5 ,
                                usecols = (1,2), unpack=True)
```

reads in only columns 1 and 2; columns 0 and 3 are skipped. As a consequence, only two array names are included to the left of the "=" sign, corresponding to the two column that are read. Writing `usecols = (0,2,3)` would skip column 1 and read in only the data in colums 0, 2, and 3. In this case, 3 array names would need to be provided on the left hand side of the "=" sign.

One convenient feature of the `loadtxt` function is that it recognizes any *white space* as a column separator: spaces, tabs, *etc.*

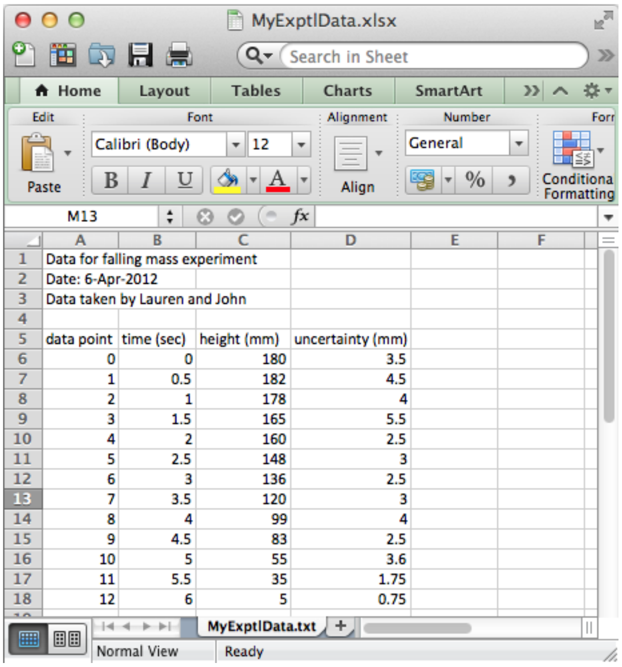
Finally you should remember that `loadtxt` is a NumPy function. So if you are using it in a Python module, you must be sure to include an "`import numpy as np`" statement before calling "`np.loadtxt`".

4.3.2 Reading data from a CSV file

Sometimes you have data stored in a spreadsheet program like Excel that you would like to read into a Python program. The *Excel data sheet* shown here contains the same data set we saw above in a text file.

While there are a number of different approaches one can use to reading such files, one of the simplest of most robust is to save the spreadsheet as a CSV ("comma separated value") file, a format which all common spreadsheet programs can create and read. So, if your Excel spreadsheet was called `MyData.xlsx`, the CSV file saved using Excel's `Save As` command would by default be `MyData.csv`. It would look like this

```
Data for falling mass experiment,,,
Date: 16-Aug-2013,,,
Data taken by Lauren and John,,,
,,,
data point,time (sec),height (mm),uncertainty (mm)
0,0,180,3.5
```



	A	B	C	D	E	F
1	Data for falling mass experiment					
2	Date: 6-Apr-2012					
3	Data taken by Lauren and John					
4						
5	data point	time (sec)	height (mm)	uncertainty (mm)		
6	0	0	180	3.5		
7	1	0.5	182	4.5		
8	2	1	178	4		
9	3	1.5	165	5.5		
10	4	2	160	2.5		
11	5	2.5	148	3		
12	6	3	136	2.5		
13	7	3.5	120	3		
14	8	4	99	4		
15	9	4.5	83	2.5		
16	10	5	55	3.6		
17	11	5.5	35	1.75		
18	12	6	5	0.75		

Figure 4.1: Excel data sheet

```
1,0.5,182,4.5
2,1,178,4
3,1.5,165,5.5
4,2,160,2.5
5,2.5,148,3
6,3,136,2.5
7,3.5,120,3
8,4,99,4
9,4.5,83,2.5
10,5,55,3.6
11,5.5,35,1.75
12,6,5,0.75
```

As its name suggests, the CSV file is simply a text file with the data that was formerly in spreadsheet columns now separated by commas. We can read the data in this file into a Python program using the `loadtxt` NumPy function once again. Here is the code

```
In [3]: dataPt, time, height, error = loadtxt("MyData.csv",
                            skiprows=5 , unpack=True, delimiter=',')
```

The form of the function is exactly the same as before except we have added the argument `delimiter=','` that tells `loadtxt` that the columns are separated by commas instead of white space (spaces or tabs), which is the default. Once again, we set the `skiprows` argument to skip the header at the beginning of the file and to start reading at the first row of data. The data are output to the arrays to the right of the assignment operator = exactly as in the previous example.

4.4 File output

4.4.1 Writing data to a text file

There is a plethora of ways to write data to a data file in Python. We will stick to one very simple one that's suitable for writing data files in text format. It uses the NumPy `savetxt` routine, which is the counterpart of the `loadtxt` routine introduced in the previous section. The general form of the routine is

```
savetxt(filename, array, fmt="%0.18e", delimiter=" ", newline="\n",
    header="", footer="", comments="# ")
```

We illustrate `savetext` below with a script that first creates four arrays by reading in the data file `MyData.txt`, as discussed in the previous section, and then writes that same data set to another file `MyDataOut.txt`.

```python
import numpy as np

dataPt, time, height, error = np.loadtxt("MyData.txt",
                                  skiprows=5, unpack=True)

np.savetxt('MyDataOut.txt',
    zip(dataPt, time, height, error), fmt="%12.1f")
```

The first argument of of `savetxt` is a string, the name of the data file to be created. Here we have chosen the name `MyDataOut.txt`, inserted with quotes, which designates it as a string literal. Beware, if there is already a file of that name on your computer, it will be overwritten—the old file will be destroyed and a new one will be created.

The second argument is the data array the is to be written to the data file. Because we want to write not one but four data arrays to the file, we have to package the four data arrays as one, which we do using the `zip` function, a Python function that combines returns a list of tuples, where the i^{th} tuple contains the i^{th} element from each of the arrays (or lists, or tuples) listed as its arguments. Since there are four arrays, each row will be a tuple with four entries, producing a table with four columns. Note that the first two arguments, the filename and data array, are regular arguments and thus must appear as the first and second arguments in the correct order. The remaining arguments are all keyword arguments, meaning that they are optional and can appear in any order, provided you use the keyword.

The next argument is a format string that determines how the elements of the array are displayed in the data file. The argument is optional and, if left out, is the format `0.18e`, which displays numbers as 18 digit floats in exponential (scientific) notation. Here we choose a different format, `12.1f`, which is a float displayed with 1 digit to the right of the decimal point and a minimum width of 12. By choosing 12, which is more digits than any of the numbers in the various arrays have, we ensure that all the columns will have the same width. It also ensures that the decimal points in column of numbers are aligned. This is evident in the data file below, *MyDataOut.txt*, which was produced by the above script.

```
         0.0         0.0       180.0         3.5
         1.0         0.5       182.0         4.5
         2.0         1.0       178.0         4.0
         3.0         1.5       165.0         5.5
         4.0         2.0       160.0         2.5
         5.0         2.5       148.0         3.0
         6.0         3.0       136.0         2.5
         7.0         3.5       120.0         3.0
         8.0         4.0        99.0         4.0
         9.0         4.5        83.0         2.5
```

```
10.0            5.0           55.0            3.6
11.0            5.5           35.0            1.8
12.0            6.0            5.0            0.8
```

We omitted the optional `delimiter` keyword argument, which leaves the delimiter as the default space.

We also omitted the optional `header` keyword argument, which is a string variable that allows you to write header text above the data. For example, you might want to label the data columns and also include the information that was in the header of the original data file. To do so, you just need to create a string with the information you want to include and then use the `header` keyword argument. The code below illustrates how to do this.

```python
import numpy as np

dataPt, time, height, error = np.loadtxt("MyData.txt",
                                 skiprows=5, unpack=True)

info = 'Data for falling mass experiment'
info += '\nDate: 16-Aug-2013'
info += '\nData taken by Lauren and John'
info += '\n\n   data point    time (sec) height (mm)   '
info += 'uncertainty (mm)'

np.savetxt('MyDataOut.txt',
      zip(dataPt, time, height, error), header=info, fmt="%12.1f")
```

Now the data file produces has a header preceding the data. Notice that the header rows all start with a # comment character, which is the default setting for the `savetxt` function. This can be changed using the keyword argument `comments`. You can find more information about `savetxt` using the IPython `help` function or from the online NumPy documentation.

```
# Data for falling mass experiment
# Date: 16-Aug-2013
# Data taken by Lauren and John
#
#    data point    time (sec) height (mm)  uncertainty (mm)
         0.0          0.0        180.0           3.5
         1.0          0.5        182.0           4.5
         2.0          1.0        178.0           4.0
         3.0          1.5        165.0           5.5
         4.0          2.0        160.0           2.5
         5.0          2.5        148.0           3.0
         6.0          3.0        136.0           2.5
```

```
 7.0          3.5          120.0          3.0
 8.0          4.0           99.0          4.0
 9.0          4.5           83.0          2.5
10.0          5.0           55.0          3.6
11.0          5.5           35.0          1.8
12.0          6.0            5.0          0.8
```

4.4.2 Writing data to a CSV file

To produce a CSV file, you would specify a comma as the delimiter. You might use the `0.1f` format specifier, which leaves no extra spaces between the comma data separators, as the file is to be read by a spreadsheet program, which will determine how the numbers are displayed. The code, which could be substituted for the `savetxt` line in the above code reads

```
np.savetxt('MyDataOut.csv',
        zip(dataPt, time, height, error), fmt="%0.1f",
        delimiter=",")
```

and produces the following data file

```
0.0,0.0,180.0,3.5
1.0,0.5,182.0,4.5
2.0,1.0,178.0,4.0
3.0,1.5,165.0,5.5
4.0,2.0,160.0,2.5
5.0,2.5,148.0,3.0
6.0,3.0,136.0,2.5
7.0,3.5,120.0,3.0
8.0,4.0,99.0,4.0
9.0,4.5,83.0,2.5
10.0,5.0,55.0,3.6
11.0,5.5,35.0,1.8
12.0,6.0,5.0,0.8
```

This data file, with a `csv` extension, can be directly read into a spreadsheet program like Excel.

4.5 Exercises

1. Write a Python program that calculates how much money you can spend each day for lunch for the rest of the month based on today's date and how much money you currently have in your lunch account. The program should ask you: (1) how much money you have in your account, (2) what today's date is, and (3) how many days there are in month. The program should return your daily allowance. The results of running your program should look like this:

   ```
   How much money (in dollars) in your lunch account? 118.39

   What day of the month is today? 17

   How many days in this month? 30

   You can spend $8.46 each day for the rest of the month.
   ```

   Extra: Create a dictionary (see *Dictionaries*) that stores the number of days in each month (forget about leap years) and have your program ask what month it is rather than the number of days in the month.

2. From the IPython terminal, create the following three NumPy arrays:

   ```
   a = array([1, 3, 5, 7])
   b = array([8, 7, 5, 4])
   c = array([0, 9,-6,-8])
   ```

   Now use the `zip` function to create the object `d` defined as

   ```
   d = zip(a, b, c)
   ```

   Print `d` out at the terminal prompt. What kind of object is `d`? Hint: It is not a NumPy array. Convert `d` into a NumPy array and call that array `e`. Type `e` at the terminal prompt so that `e` is printed out on the IPython terminal. One of the elements of `e` is `-8`. Show how to address and print out just that element of `e`. Show how to address that same element of `d`. What has become of the three original arrays `a`, `b`, and `c`, that is, how do they appear in `e`?

3. Create the following data file and then write a Python script to read it into a three NumPy arrays with the variable names `f`, `a`, `da` for the frequency, amplitude, and amplitude error.

   ```
   Date: 2013-09-16
   Data taken by Liam and Selena
   frequency (Hz) amplitude (mm)  amp error (mm)
   ```

```
 0.7500       13.52        0.32
 1.7885       12.11        0.92
 2.8269       14.27        0.73
 3.8654       16.60        2.06
 4.9038       22.91        1.75
 5.9423       35.28        0.91
 6.9808       60.99        0.99
 8.0192       33.38        0.36
 9.0577       17.78        2.32
10.0962       10.99        0.21
11.1346        7.47        0.48
12.1731        6.72        0.51
13.2115        4.40        0.58
14.2500        4.07        0.63
```

Show that you have correctly read in the data by having your script print out to your computer screen the three arrays. Format the printing so that it produces output like this:

```
f =
[  0.75      1.7885    2.8269    3.8654    4.9038    5.9423
   6.9808    8.0192    9.0577   10.0962   11.1346   12.1731
  13.2115   14.25    ]
a =
[ 13.52   12.11   14.27   16.6    22.91   35.28   60.99   33.38
  17.78   10.99    7.47    6.72    4.4     4.07]
da =
[ 0.32   0.92   0.73   2.06   1.75   0.91   0.99   0.36   2.32
  0.21   0.48   0.51   0.58   0.63]
```

Note that the array `f` is displayed with four digits to the right of the decimal point while the arrays `a` and `da` are displayed with only two. The columns of the displayed arrays need not line up as they do above.

4. Write a script to read the data from the previous problem into three NumPy arrays with the variable names `f`, `a`, `da` for the frequency, amplitude, and amplitude error and then, in the same script, write the data out to a data file, including the header, with the data displayed in three columns, just as its displayed in the problem above. It's ok if the header lines begin with the # comment character. Your data file should have the extension `.txt`.

5. Write a script to read the data from the previous problem into three NumPy arrays with the variable names `f`, `a`, `da` for the frequency, amplitude, and amplitude error and then, in the same script, write the data out to a csv data file, without the header, to a data file with the data displayed in three columns. Use a single format specifier

and set it to `"%0.16e"`. If you have access the spreadsheet program (like MS Excel), try opening the file you have created with your Python script and verify that the arrays are displayed in three columns. Note that your csv file should have the extension `.csv`.

CHAPTER

FIVE

PLOTTING

The graphical representation of data—plotting—is one of the most important tools for evaluating and understanding scientific data and theoretical predictions. However, plotting is not a part of core Python but is provided through one of several possible library modules. The most highly developed and widely used plotting package for Python is MatPlotLib (http://MatPlotLib.sourceforge.net/). It is a powerful and flexible program that has become the *de facto* standard for 2-d plotting with Python.

Because MatPlotLib is an external library—in fact it's a collection of libraries—it must be imported into any routine that uses it. MatPlotLib makes extensive use of NumPy so the two should be imported together. Therefore, for any program for which you would like to produce 2-d plots, you should include the lines

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

There are other MatPlotLib sub-libraries, but the `pyplot` library provides nearly everything that you need for 2-d plotting. The standard prefix for it is `plt`. The two statements above must appear before any calls to NumPy or MatPlotLib routines are made. MatPlotLib is automatically loaded with the IPython shell so you do not need to use import matplotlib.pyplot nor do you need to use the `plt` prefix when working in the IPython shell.

One final word before we get started: We only scratch the surface of what is possible using MatPlotLib and as you become familiar with it, you will surely want to do more than this manual describes. In that case, you need to go the the web to get more information. A good place to start is http://matplotlib.org/api/pyplot_summary.html. Another interesting web page is http://matplotlib.org/gallery.html.

5.1 An interactive session with `pyplot`

We begin with an interactive plotting session that illustrates some very basic features of MatPlotLib. Type in the `plot` command shown below and press the return key. Take care to follow the exact syntax.

```
In [1]: plot([1,2,3,2,3,4,3,4,5])
Out[1]: [<MatPlotLib.lines.Line2D at 0x94e1310>]
```

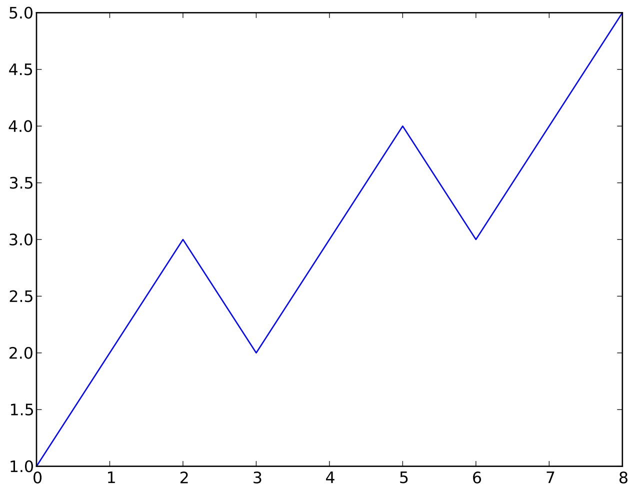


Figure 5.1: Interactive plot window

A window should appear with a plot that looks something like the *Interactive plot window* shown here. By default, the `plot` function draws a line between the data points that were entered. You can save this plot to an image file by clicking on the floppy disk icon at the top of the plot window. You can also zoom, pan, scroll through the plot, and return to the original view using the other icons in the plot window. Experimenting with them reveals their functions.

When you are finished, be sure to close the plot window.

Let's take a closer look at the `plot` function. It is used to plot x-y data sets and is written like this

```
plot(x, y)
```

where x and y are arrays (or lists) that have the same size. If the x array is missing, that is, if there is only a single array, as in our example above, the plot function uses 0, 1, ..., N-1 for the x array, where N is the size of the y array. Thus, the plot function provides a quick graphical way of examining a data set.

More typically, you supply both an x and a y data set to plot. Taking things a bit further, you may also want to plot several data sets on the same graph, use symbols as well as lines, label the axes, create a title and a legend, and control the color of symbols and lines. All of this is possible but requires calling a number of plotting functions. For this reason, plotting is usually done using a Python script or program.

5.2 Basic plotting

The quickest way to learn how to plot using the MatPlotLib library is by example. For our first task, let's plot the sine function over the interval from 0 to 4π. The main plotting function plot in MatPlotLib does not plot functions *per se*, it plots (x, y) data points. As we shall see, we can instruct the function plot either to just draw point—or dots—at each data point, or we can instruct it to draw straight lines between the data points. To create the illusion of the smooth function that the sine function is, we need to create enough (x, y) data points so that when plot draws straight lines between the data points, the function appears to be smooth. The sine function undergoes two full oscillations with two maxima and two minima between 0 and 4π. So let's start by creating an array with 33 data points between 0 and 4π, and then let MatPlotLib draw a straight line between them. Our code consists of four parts

- import the NumPy and MatPlotLib modules (lines 1-2 below)
- create the (x, y) data arrays (lines 3-4 below)
- have plot draw straight lines between the (x, y) data points (line 5 below)
- display the plot in a figure window using the show function (line 6 below)

Here is our code, which consists of only 6 lines:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 x = np.linspace(0, 4.*np.pi, 33)
4 y = np.sin(x)
5 plt.plot(x, y)
6 plt.show()
```

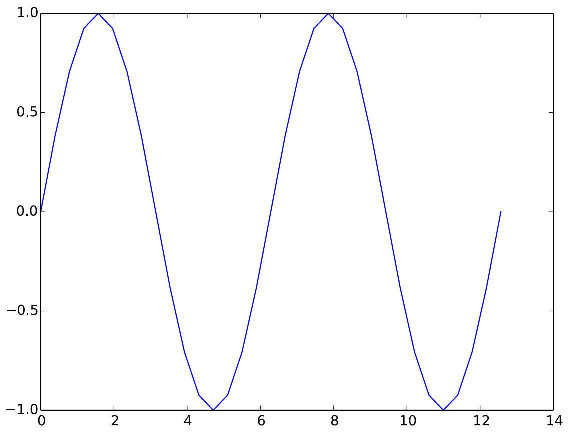


Figure 5.2: Sine function

Only 6 lines suffice to create the plot, which consists of the sine function over the interval from 0 to 4π, as advertised, as well as axes annotated with nice whole numbers over the appropriate interval. It's a pretty nice plot made with very little code.

One problem, however, is that while the plot oscillates like a sine wave, it is not smooth. This is because we did not create the (x, y) arrays with enough data points. To correct this, we need more data points. The plot below was created using the same program shown above but with 129 (x, y) data points instead of 33. Try it out your self by copying the above program and replacing 33 in line 3 with 129 so that the function `linspace` creates an array with 129 data points instead of 33.

The code above illustrates how plots can be made with very little code using the MatPlotLib module. In making this plot, MatPlotLib has made a number of choices, such as the size of the figure, the blue color of the line, even the fact that by default a line is drawn between successive data points in the (x, y) arrays. All of these choices can be changed by explicitly instructing MatPlotLib to do so. This involves including more arguments in the function calls we have used and using new functions that control other properties of the plot. The next example illustrates a few of the simpler embellishments that are possible.

In the *Wavy pulse* figure, we plot two (x, y) data sets: a smooth line curve and some data represented by red circles. In this plot, we label the x and y axes, create a legend, and draw lines to indicate where x and y are zero. The code that creates this plot is shown below.

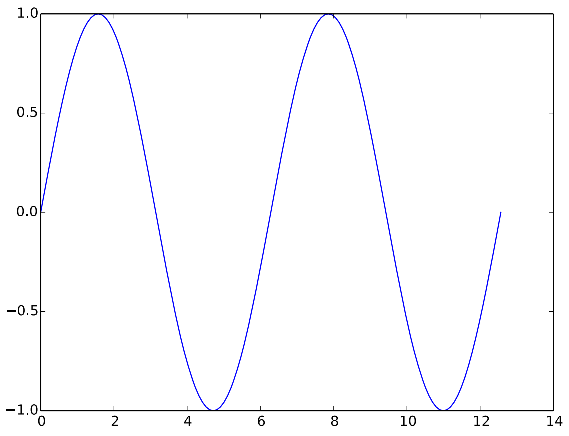


Figure 5.3: Sine function plotted using more data points

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

# read data from file
xdata, ydata = np.loadtxt('wavePulseData.txt', unpack=True)

# create x and y arrays for theory
x = np.linspace(-10., 10., 200)
y = np.sin(x) * np.exp(-(x/5.0)**2)

# create plot
plt.figure(1, figsize = (6,4) )
plt.plot(x, y, 'b-', label='theory')
plt.plot(xdata, ydata, 'ro', label="data")
plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')
plt.axhline(color = 'gray', zorder=-1)
plt.axvline(color = 'gray', zorder=-1)

# save plot to file
plt.savefig('WavyPulse.pdf')

# display plot on screen
```

```
25 plt.show()
```

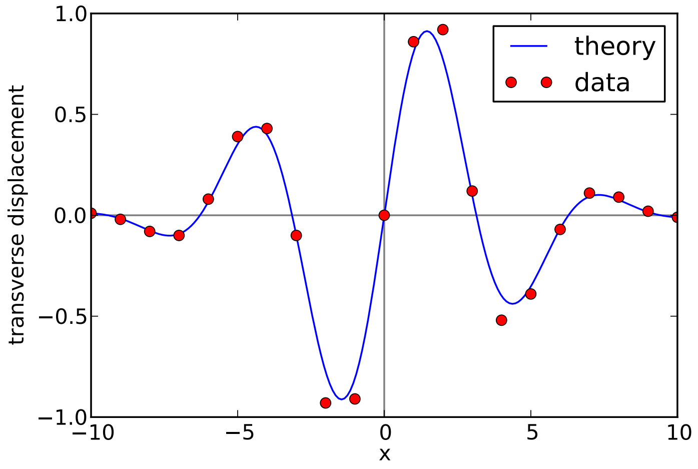


Figure 5.4: Wavy pulse

If you have read the first four chapters, the code in lines 1-9 in the above script should be familiar to you. Fist, the script loads the NumPy and MatPlotLib modules, then reads data from a data file into two arrays, `xdata` and `ydata`, and then creates two more arrays, `x` and `y`. The first pair or arrays, `xdata` and `ydata`, contain the x-y data that are plotted as red circles in the *Wavy pulse* figure; the arrays created in line 8 and 9 contain the x-y data that are plotted as a blue line.

The functions that do the plotting begin on line 12. Let's go through them one by one and see what they do. You will notice in several cases that *keyword arguments* (`kwargs`) are used in several cases. Keyword arguments are *optional* arguments that have the form `kwarg=` *data*, where *data* might be a number, a string, a tuple, or some other form of data.

`figure()` creates a blank figure window. If it has no arguments, it creates a window that is 8 inches wide and 6 inches high by default, although the size that appears on your computer depends on your screen's resolution. For most computers, it will be much smaller. You can create a window whose size differs from the default using the optional keyword argument `figsize`, as we have done here. If you use `figsize`, set it equal to a 2-element tuple where the elements are the width and height, respectively, of the plot. Multiple calls to `figure()` opens multiple windows: `figure(1)` opens up one window for plotting, `figure(2)` another, and `figure(3)` yet another.

`plot(x, y,` ***optional arguments*** **`)`** graphs the x-y data in the arrays `x` and `y`. The third argument is a format string that specifies the color and the type of line or symbol that is used to plot the data. The string `'ro'` specifies a red (`r`) circle (`o`). The string `'b-'` specifies a blue (`b`) solid line (`-`). The keyword argument `label` is set equal to a string that labels the data if the `legend` function is called subsequently.

`xlabel(` ***string*** **`)`** takes a string argument that specifies the label for the graph's x-axis.

`ylabel(` ***string*** **`)`** takes a string argument that specifies the label for the graph's y-axis.

`legend()` makes a legend for the data plotted. Each x-y data set is labeled using the string that was supplied by the `label` keyword in the `plot` function that graphed the data set. The `loc` keyword argument specifies the location of the legend.

`axhline()` draws a horizontal line across the width of the plot at `y=0`. The optional keyword argument `color` is a string that specifies the color of the line. The default color is black. The optional keyword argument `zorder` is an integer that specifies which plotting elements are in front of or behind others. By default, new plotting elements appear *on top of* previously plotted elements and have a value of `zorder=0`. By specifying `zorder=-1`, the horizontal line is plotted *behind* all existing plot elements that have not be assigned an explicit `zorder` less than -1.

`axvline()` draws a vertical line from the top to the bottom of the plot at `x=0`. See `axhline()` for explanation of the arguments.

`savefig(` ***string*** **`)`** saves the figure to data data file with a name specified by the string argument. The string argument can also contain path information if you want to save the file so some place other than the default directory.

`show()` displays the plot on the computer screen. No screen output is produced before this function is called.

To plot the solid blue line, the code uses the `'b-'` format specifier in the `plot` function call. It is important to understand that MatPlotLib draws *straight lines* between data points. Therefore, the curve will appear smooth only if the data in the NumPy arrays are sufficiently dense. If the space between data points is too large, the straight lines the `plot` function draws between data points will be visible. For plotting a typical function, something on the order of 100-200 data points usually produces a smooth curve, depending on just how curvy the function is. On the other hand, only two points are required to draw a smooth straight line.

Detailed information about the MatPlotLib plotting functions are available online, starting with the site http://matplotlib.org/api/pyplot_summary.html. The main MatPlotLib site is http://matplotlib.org/.

5.2.1 Specifying line and symbol types and colors

In the above example, we illustrated how to draw one line type (solid), one symbol type (circle), and two colors (blue and red). There are many more possibilities, which are specified in the tables below. The way it works is to specify a string consisting of one or more plotting format specifiers. There are two types of format specifiers, one for the line or symbol type and another for the color. It does not matter in which order the format specifiers are listed in the string. Examples are given following the two tables. Try them out to make sure you understand how these plotting format specifiers work.

The first table below shows the characters used to specify the line or symbol type that is used. If a line type is chosen, the lines are drawn between the data points. If a marker type is chosen, the a marker is plotted at each data point.

character	description	character	description
`-`	solid line style	`3`	tri_left marker
`--`	dashed line style	`4`	tri_right marker
`-.`	dash-dot line style	`s`	square marker
`:`	dotted line style	`p`	pentagon marker
`.`	point marker	`*`	star marker
`,`	pixel marker	`h`	hexagon1 marker
`o`	circle marker	`H`	hexagon2 marker
`v`	triangle_down marker	`+`	plus marker
`^`	triangle_up marker	`x`	x marker
`<`	triangle_left marker	`D`	diamond marker
`>`	triangle_right marker	`d`	thin_diamond marker
`1`	tri_down marker	`|`	vline marker
`2`	tri_up marker	`_`	hline marker

This second table gives the character codes for eight different colors. Many more are possible but the color specification becomes more complex. You can consult the web-based MatPlotLib documentation for further details.

character	color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Here are some examples of how these format specifiers can be used:

```
plot(x, y, 'ro')     # plots red circles
plot(x, y, 'ks-')    # plot black squares connected by black lines
plot(x, y, 'g^')     # plots green triangles that point up

plot(x, y, 'k-')     # plots a black line between the points
plot(x, y, 'ms')     # plots magenta squares
```

You can also make two calls sequentially for added versatility. For example, by sequentially calling the last two plot calls, the plot produces magenta squares on top of black lines connecting the data points.

These format specifiers give rudimentary control of the plotting symbols and lines. MatPlotLib provides much more precise and detailed control of the plotting symbol size, line types, and colors using optional keyword arguments instead of the plotting format strings introduced above. For example, the following command creates a plot of large yellow diamond symbols with blue edges connected by a green dashed line:

```
plot(x, y, color='green', linestyle='dashed', marker='d',
     markerfacecolor='yellow', markersize=12,
     markeredgecolor='blue')
```

Try it out! The online MatPlotLib documentation provides all the plotting format keyword arguments and their possible values.

5.2.2 Error bars

When plotting experimental data it is customary to include error bars that indicate graphically the degree of uncertainty that exists in the measurement of each data point. The MatPlotLib function `errorbar` plots data with error bars attached. It can be used in a way that either replaces or augments the `plot` function. Both vertical and horizontal error bars can be displayed. The figure below illustrates the use of error bars.

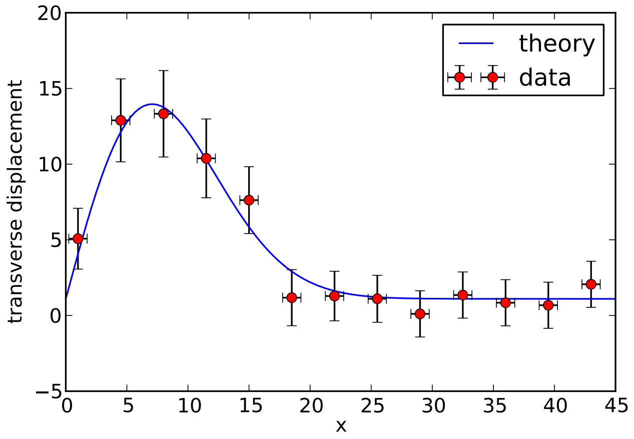


Figure 5.5: Error Bars

When error bars are desired, you typically replace the `plot` function with the `errorbar` function. The first two arguments of the `errorbar` function are the `x` and `y` arrays to be plotted, just as for the `plot` function. The keyword `fmt` *must be used* to specify the format of the points to be plotted; the format specifiers are the same as for `plot`. The keywords `xerr` and `yerr` are used to specify the x and y error bars. Setting one or both of them to a constant specifies one size for all the error bars. Alternatively, setting one or both of them equal to an array that has the same length as the `x` and `y` arrays allows you to give each data point an error bar with a different value. If you only want y error bars, then you should only specify the `yerr` keyword and omit the `xerr` keyword. The color of the error bars is set with the keyword `ecolor`.

The code and plot below illustrates how to make error bars and was used to make the above plot. Lines 14 and 15 contain the call to the `errorbar` function. The x error bars are all set to a constant value of 0.75, meaning that the error bars extend 0.75 to the left and 0.75 to the right of each data point. The y error bars are set equal to an array, which was read in from the data file containing the data to be plotted, so each data point has a different y error bar. By the way, leaving out the `xerr` keyword argument in the `errorbar` function call below would mean that only the y error bars would be plotted.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # read data from file
5 xdata, ydata, yerror = np.loadtxt('expDecayData.txt', unpack=True)
```

```python

# create theoretical fitting curve
x = np.linspace(0, 45, 128)
y = 1.1+ 3.0*x*np.exp(-(x/10.0)**2)

# create plot
plt.figure(1, figsize = (6,4) )
plt.plot(x, y, 'b-', label="theory")
plt.errorbar(xdata, ydata, fmt='ro', label="data",
             xerr=0.75, yerr=yerror, ecolor='black')
plt.xlabel('x')
plt.ylabel('transverse displacement')
plt.legend(loc='upper right')

# save plot to file
plt.savefig('ExpDecay.pdf')

# display plot on screen
plt.show()
```

We have more to say about the `errorbar` function in the sections on logarithmic plots. But the brief introduction given here should suffice for making most plots not involving logarithmic axes.

5.2.3 Setting plotting limits and excluding data

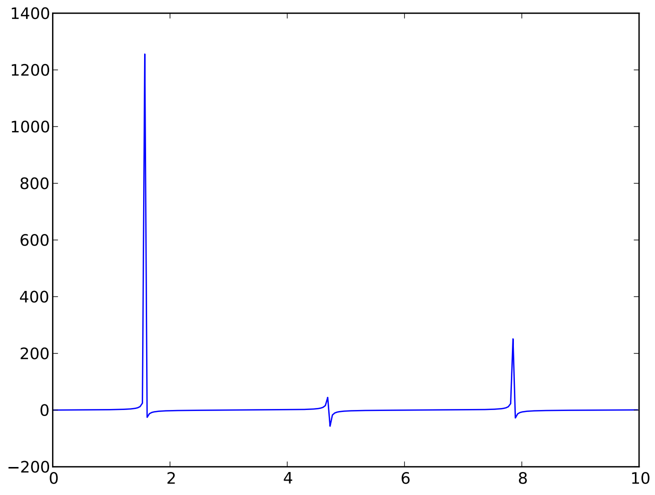
It turns out that you often want to restrict the range of numerical values over which you plot data or functions. In these cases you may need to manually specify the plotting window or, alternatively, you may wish to exclude data points that are outside some set of limits. Here we demonstrate methods for doing this.

Setting plotting limits

Suppose you want to plot the tangent function over the interval from 0 to 10. The following script offers an straightforward first attempt.

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

theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)
```

```
plt.figure()
plt.plot(theta, ytan)
plt.show()
```



The resulting plot, shown above, doesn't quite look like what you might have expected for $\tan\theta$ *vs* θ. The problem is that $\tan\theta$ diverges at $\theta = \pi/2, 3\pi/2, 5\pi/2, ...$, which leads to large spikes in the plots as values in the `theta` array come near those values. Of course, we don't want the plot to extend all the way out to $\pm\infty$ in the y direction, nor can it. Instead, we would like the plot to extend far enough that we get the idea of what is going on as $y \to \pm\infty$, but we would still like to see the behavior of the graph near $y = 0$. We can restrict the range of `ytan` values that are plotted using the MatPlotLib function `ylim`, as we demonstrate in the script below.

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

theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)

plt.figure()
plt.plot(theta, ytan)
plt.ylim(-8, 8)          # restricts range of y axis from -8 to +8
plt.axhline(color="gray", zorder=-1)
plt.show()
```

The figure produced by this script is shown below. The plot now looks much more like the familiar $\tan\theta$ function we know. We have also include a call to the `axline` function to create an x axis.

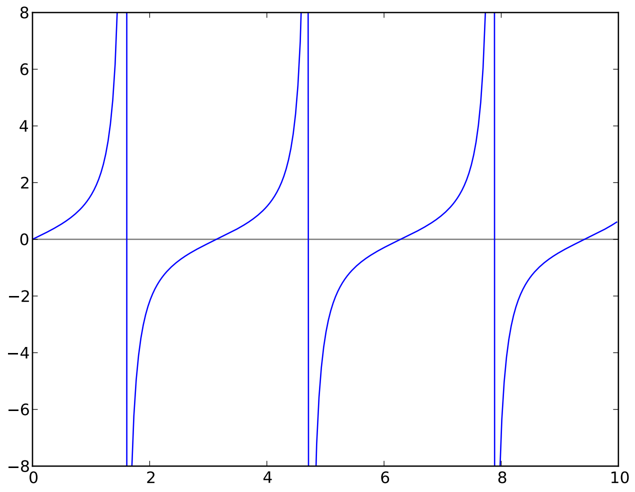


Figure 5.6: Tangent function (with spurious lines)

The vertical blue lines at $\theta = \pi/2, 3\pi/2, 5\pi/2$ should not appear in a plot of $\tan\theta$ *vs* θ. However, they do appear because the `plot` function simply draws lines between the data points in the `x`-`y` arrays provided in its arguments. Thus, `plot` draws a line between the very large positive and negative `ytan` values corresponding to the `theta` values on either side of $\pi/2$ where $\tan\theta$ diverges to $\pm\infty$. It would be nice to exclude that line.

Masked arrays

We can exclude the data points near $\theta = \pi/2, 3\pi/2, 5\pi/2$ in the above plot, and thus avoid drawing the nearly vertical lines at those points, using NumPy's *masked array* feature. The code below shows how this is done and produces the graph below. The masked array feature is implemented in line 6 with a call to NumPy's `masked_where` function in the sub-module `ma` (masked array). Therefore, it is called by writing `np.ma.masked_where`. The `masked_where` function works as follows. The first argument sets the condition for masking elements of the array, which is specified by the second argument. In this case, the function says to mask all elements of the array `ytan` (the second argument) where the absolute value of `ytan` is greater than 20. The result is set equal to `ytanM`. When `ytanM` is plotted, MatPlotLib's `plot` function omits all masked points from the plot. You can think of it as the `plot` function lifting the pen that is drawing the line in the plot when it comes to the masked points in the array `ytanM`.

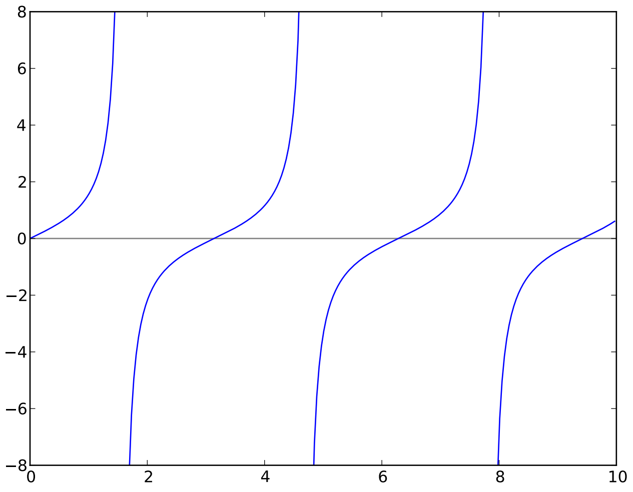


Figure 5.7: Tangent function

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

theta = np.arange(0.01, 10., 0.04)
ytan = np.tan(theta)
ytanM = np.ma.masked_where(np.abs(ytan)>20., ytan)

plt.figure()
plt.plot(theta, ytanM)
plt.ylim(-8, 8)
plt.axhline(color="gray", zorder=-1)

plt.show()
```

5.2.4 Subplots

Often you want to create two or more graphs and place them next to one another, generally because they are related to each other in some way. The plot below shows an example of such a plot. In the top graph, $\tan\theta$ and $\sqrt{(8/\theta)^2-1}$ *vs* θ are plotted. The two curves cross each other at the points where $\tan\theta = \sqrt{(8/\theta)^2-1}$. In the bottom $\cot\theta$ and $-\sqrt{(8/\theta)^2-1}$ *vs* θ are plotted. These two curves cross each other at the points where $\cot\theta = -\sqrt{(8/\theta)^2-1}$.

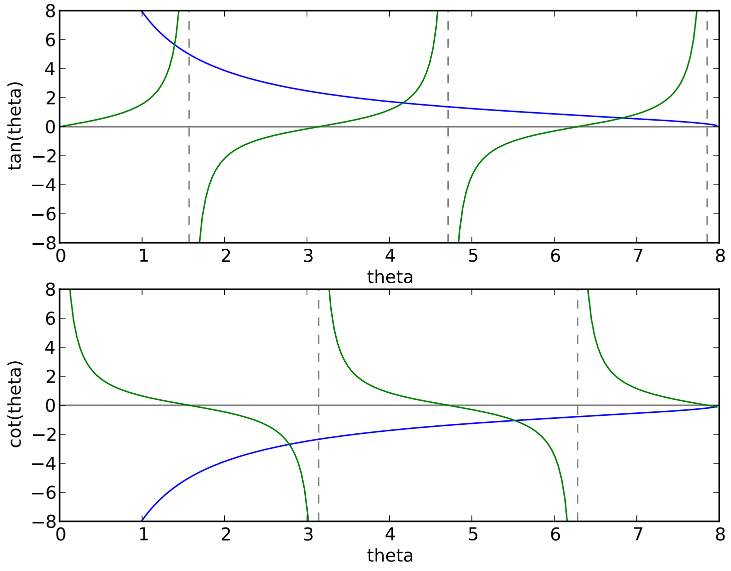


Figure 5.8: Crossing functions

The code that produces this plot is provided below.

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

theta = np.arange(0.01, 8., 0.04)
y = np.sqrt((8./theta)**2-1.)
ytan = np.tan(theta)
ytan = np.ma.masked_where(np.abs(ytan)>20., ytan)
ycot = 1./np.tan(theta)
```

```
9  ycot = np.ma.masked_where(np.abs(ycot)>20., ycot)
10 
11 plt.figure(1)
12 
13 plt.subplot(2, 1, 1)
14 plt.plot(theta, y)
15 plt.plot(theta, ytan)
16 plt.ylim(-8, 8)
17 plt.axhline(color="gray", zorder=-1)
18 plt.axvline(x=np.pi/2., color="gray", linestyle='--', zorder=-1)
19 plt.axvline(x=3.*np.pi/2., color="gray", linestyle='--', zorder=-1)
20 plt.axvline(x=5.*np.pi/2., color="gray", linestyle='--', zorder=-1)
21 plt.xlabel("theta")
22 plt.ylabel("tan(theta)")
23 
24 plt.subplot(2, 1, 2)
25 plt.plot(theta, -y)
26 plt.plot(theta, ycot)
27 plt.ylim(-8, 8)
28 plt.axhline(color="gray", zorder=-1)
29 plt.axvline(x=np.pi, color="gray", linestyle='--', zorder=-1)
30 plt.axvline(x=2.*np.pi, color="gray", linestyle='--', zorder=-1)
31 plt.xlabel("theta")
32 plt.ylabel("cot(theta)")
33 
34 plt.show()
```

The function `subplot`, called on lines 13 and 24, creates the two subplots in the above figure. `subplot` has three arguments. The first specifies the number of rows that the figure space is to be divided into; on line 13, it's two. The second specifies the number of columns that the figure space is to be divided into; on line 13, it's one. The third argument specifies which rectangle the will contain the plot specified by the following function calls. Line 13 specifies that the plotting commands that follow will be act on the first box. Line 24 specifies that the plotting commands that follow will be act on the second box.

We have also labeled the axes and included dashed vertical lines at the values of θ where $\tan\theta$ and $\cot\theta$ diverge.

5.3 Logarithmic plots

Data sets can span many orders of magnitude from fractional quantities much smaller than unity to values much larger than unity. In such cases it is often useful to plot the data on logarithmic axes.

5.3.1 Semi-log plots

For data sets that vary exponentially in the independent variable, it is often useful to use one or more logarithmic axes. Radioactive decay of unstable nuclei, for example, exhibits an exponential decrease in the number of particles emitted from the nuclei as a function of time. In the plot below, for example, we show the decay of the radioactive isotope Phosphorus-32 over a period of 6 months, where the radioactivity is measured once each week. Starting at a decay rate of nearly 10^4 electrons (counts) per second, the decay rate diminishes to only about 1 count per second after about 6 months or 180 days. If we plot counts per second as a function of time on a normal plot, as we have done in the plot on the left below, then the count rate is indistinguishable from zero after about 100 days. On the other hand, if we use a logarithmic axis for the count rate, as we have done in the plot on the right below, then we can follow the count rate well past 100 days and can readily distinguish it from zero. Moreover, if the data vary exponentially in time, then the data will fall along a straight line, as they do for the case of radioactive decay.

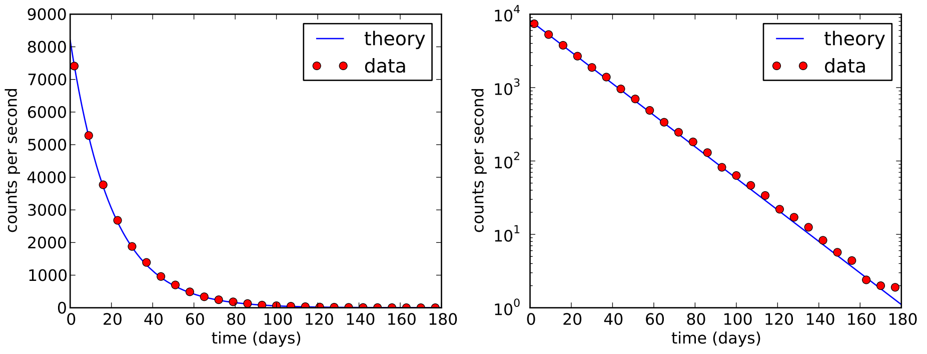


Figure 5.9: Semi-log plotting

MatPlotLib provides two functions for making semi-logarithmic plots, `semilogx` and `semilogy`, for creating plots with logarithmic x and y axes, with linear y and x axes, respectively. We illustrate their use in the program below, which made the above plots.

```
 1  import numpy as np
 2  import matplotlib.pyplot as plt
 3
 4  # read data from file
 5  time, counts, unc = np.loadtxt('SemilogDemo.txt', unpack=True)
 6
 7  # create theoretical fitting curve
 8  tau = 20.2      # Phosphorus-32 half life = 14 days; tau = t_half/ln(2)
 9  N0 = 8200.       # Initial count rate (per second)
10  t = np.linspace(0, 180, 128)
11  N = N0 * np.exp(-t/tau)
12
13  # create plot
14  plt.figure(1, figsize = (10,4) )
15
16  plt.subplot(1, 2, 1)
17  plt.plot(t, N, 'b-', label="theory")
18  plt.plot(time, counts, 'ro', label="data")
19  plt.xlabel('time (days)')
20  plt.ylabel('counts per second')
21  plt.legend(loc='upper right')
22
23  plt.subplot(1, 2, 2)
24  plt.semilogy(t, N, 'b-', label="theory")
25  plt.semilogy(time, counts, 'ro', label="data")
26  plt.xlabel('time (days)')
27  plt.ylabel('counts per second')
28  plt.legend(loc='upper right')
29
30  plt.tight_layout()
31
32  # display plot on screen
33  plt.show()
```

The `semilogx` and `semilogy` functions work the same way as the `plot` function. You just use one or the other depending on which axis you want to be logarithmic.

The `tight_layout()` function

You may have noticed the `tight_layout()` function, called without arguments on line 30 of the program. This is a convenience function that adjusts the sizes of the plots to make room for the axes labels. If it is not called, the y-axis label of the right plot runs into the left plot. The `tight_layout()` function can also be useful in graphics windows with only one plot sometimes.

5.3.2 Log-log plots

MatPlotLib can also make log-log or double-logarithmic plots using the function `loglog`. It is useful when both the x and y data span many orders of magnitude. Data that are described by a power law $y = Ax^b$, where A and b are constants, appear as straight lines when plotted on a log-log plot. Again, the `loglog` function works just like the `plot` function but with logarithmic axes.

5.4 More advanced graphical output

The plotting methods introduced in the previous sections are perfectly adequate for basic plotting and are therefore recommended for simple graphical output. Here, we introduce an alternative syntax that harnesses the full power of MatPlotLib. It gives the user more options and greater control. Perhaps the most efficient way to learn this alternative syntax is to look at an example. The figure below illustrating *Mulitple plots in the same window* is produced by the following code:

```python
# Demonstrates the following:
#     plotting logarithmic axes
#     user-defined functions
#     "where" function, NumPy array conditional

import numpy as np
import matplotlib.pyplot as plt

# Define the sinc function, with output for x=0 defined
# as a special case to avoid division by zero. The code
# below defining the sinc function is developed and
# explained in Chapter 7, Section 1.
def s(x):
  a = np.where(x==0., 1., np.sin(x)/x)
  return a

# create arrays for plotting
x = np.arange(0., 10., 0.1)
y = np.exp(x)

t = np.linspace(-10., 10., 100)
z = s(t)
```

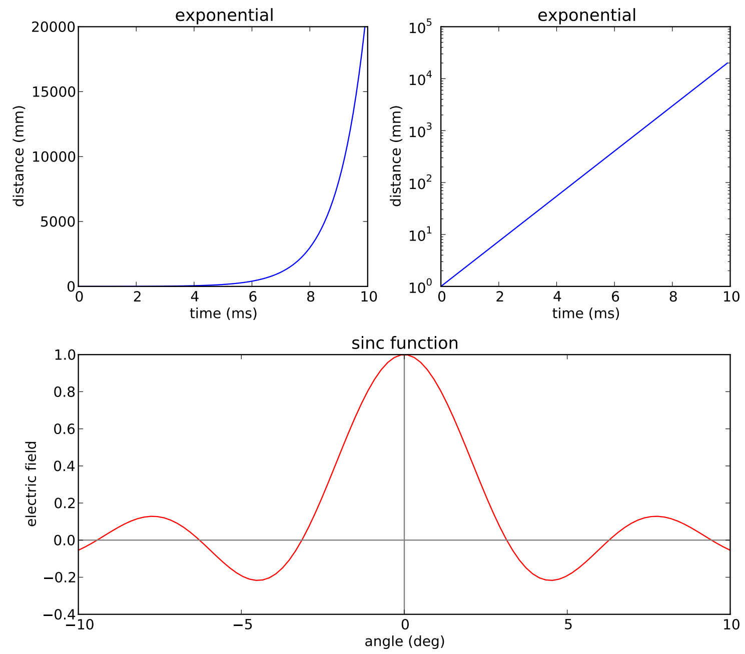


Figure 5.10: Mulitple plots in the same window

```
23
24  # create a figure window
25  fig = plt.figure(1, figsize=(9,8))
26
27  # subplot: linear plot of exponential
28  ax1 = fig.add_subplot(2,2,1)
29  ax1.plot(x, y)
30  ax1.set_xlabel('time (ms)')
31  ax1.set_ylabel('distance (mm)')
32  ax1.set_title('exponential')
33
34  # subplot: semi-log plot of exponential
35  ax2 = fig.add_subplot(2,2,2)
36  ax2.plot(x, y)
37  ax2.set_yscale('log')
38  ax2.set_xlabel('time (ms)')
39  ax2.set_ylabel('distance (mm)')
40  ax2.set_title('exponential')
41
42  # subplot: wide subplot of sinc function
43  ax3 = fig.add_subplot(2,1,2)
44  ax3.plot(t, z, 'r')
45  ax3.axhline(color='gray')
46  ax3.axvline(color='gray')
47  ax3.set_xlabel('angle (deg)')
48  ax3.set_ylabel('electric field')
49  ax3.set_title('sinc function')
50
51  # Adjusts white space to avoid collisions between subplots
52  fig.tight_layout()
53  plt.show()
```

After defining several arrays for plotting, the above program opens a figure window in line 23 with the statement

```
fig = plt.figure(figsize=(9,8))
```

The MatPlotLib statement above creates a **Figure** object, assigns it the name `fig`, and opens a blank figure window. Thus, just as we give lists, arrays, and numbers variable names (*e.g.* `a = [1, 2, 5, 7]`, `dd = np.array([2.3, 5.1, 3.9])`, or `st = 4.3`), we can give a figure object and the window in creates a name: here it is `fig`. In fact we can use the `figure` function to open up multiple figure objects with different figure windows. The statements

```
fig1 = plt.figure()
fig2 = plt.figure()
```

open up two separate windows, one named `fig1` and the other `fig2`. We can then use the names `fig1` and `fig2` to plot things in either window. The `figure` function need not take any arguments if you are satisfied with the default settings such as the figure size and the background color. On the other hane, by supplying one or more keyword arguments, you can customize the figure size, the background color, and a few other properties. For example, in the program listing (line 23), the keyword argument `figsize` sets the width and height of the figure window; the default size is `(8, 6)`; in our program we set it to `(9, 8)`, which is a bit wider and higher than the default size. In the example above, we also choose to open only a single window, hence the single `figure` call.

The `fig.add_subplot(2,2,1)` in line 30 is a MatPlotLib function that divides the figure window into 2 rows (the first argument) and 2 columns (the second argument). The third argument creates a subplot in the first of the 4 subregions (*i.e.* of the 2 rows × 2 columns) created by the `fig.add_subplot(2,2,1)` call. To see how this works, type the following code into a Python module and run it:

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

fig = plt.figure(figsize=(9,8))
ax1 = fig.add_subplot(2,2,1)

plt.show()
```

You should get a figure window with axes drawn in the upper left quadrant. The `fig.` prefix used with the `add_subplot(2,2,1)` function directs Python to draw these axes in the figure window named `fig`. If we had opened two figure windows, changing the prefix to correspond to the name of one or the other of the figure windows would direct the axes to be drawn in the appropriate window. Writing `ax1 = fig.add_subplot(2,2,1)` assigns the name ax1 to the axes in the upper left quadrant of the figure window.

The `ax1.plot(x, y)` in line 27 directs Python to plot the previously-defined `x` and `y` arrays onto the axes named `ax1`. The `ax2 = fig.add_subplot(2,2,2)` draws axes in the second, or upper right, quadrant of the figure window. The `ax3 = fig.add_subplot(2,1,2)` divides the figure window into 2 rows (first argument) and 1 column (second argument), creates axes in the second or these two sections, and assigns those axes (*i.e.* that subplot) the name `ax3`. That is, it divides the figure window into 2 halves, top and bottom, and then draws axes in the half number 2 (the third argument), or lower half of the figure window.

You may have noticed in above code that some of the function calls are a bit different from those used before: `xlabel('time (ms)')` becomes `set_xlabel('time (ms)')`, `title('exponential')` becomes `set_title('exponential')`, *etc.*

The call `ax2.set_yscale('log')` sets the y-axes in the second plot to be logarithmic, thus creating a semi-log plot. Creating properly-labeled logarthmic axes like this is more straightforward with the advanced syntax illustrated in the above example.

Using the prefixes `ax1`, `ax2`, or `ax3`, direct graphical instructions to their respective subplots. By creating and specifying names for the different figure windows and subplots within them, you access the different plot windows more efficiently. For example, the following code makes four identical subplots in a single figure window using a `for` loop.

```
In [1]: fig = figure()

In [2]: ax1 = fig.add_subplot(221)

In [3]: ax2 = fig.add_subplot(222)

In [4]: ax3 = fig.add_subplot(223)

In [5]: ax4 = fig.add_subplot(224)

In [6]: for ax in [ax1, ax2, ax3, ax4]:
   ...:     ax.plot([3,5,8],[6,3,1])

In [7]: show()
```

5.5 Exercises

1. Plot the function $y = 3x^2$ for $-1 \leq x \leq 3$ as a continuous line. Include enough points so that the curve you plot appears smooth. Label the axes x and y.
2. Plot the following function for $-15 \leq x \leq 15$:

   $$y = \frac{\cos x}{1 + \frac{1}{5}x^2}$$

   Include enough points so that the curve you plot appears smooth. Label the axes x and y.
3. Plot the functions $\sin x$ and $\cos x$ *vs* x on the same plot with x going from $-\pi$ to π. Make sure the limits of x-axis do not extend beyond the limits of the data. Plot $\sin x$ in the color green and $\cos x$ in the color black and include a legend to label the two curves. Place the legend within the plot, but such that it does not cover either of the sine or cosine traces.
4. Create a data file with the data shown below.

   (a) Read the data into Python program and plot t *vs* y using circles for data points with error bars. Use the data in the `dy` column as the error estimates for the y data. Label the horizontal and vertical axes "time (s)" and "position (cm)".

   (b) On the same graph, plot the function below as a smooth line. Make the line pass *behind* the data points.

   $$y(t) = \left[3 + \frac{1}{2}\sin\frac{\pi t}{5}\right] t\, e^{-t/10}$$

```
Data for Exercise 4
Date: 16-Aug-2013
Data taken by Lauren and John

 t       d       dy
 1.0     2.94    0.7
 4.5     8.29    1.2
 8.0     9.36    1.2
11.5    11.60    1.4
15.0     9.32    1.3
18.5     7.75    1.1
22.0     8.06    1.2
25.5     5.60    1.0
```

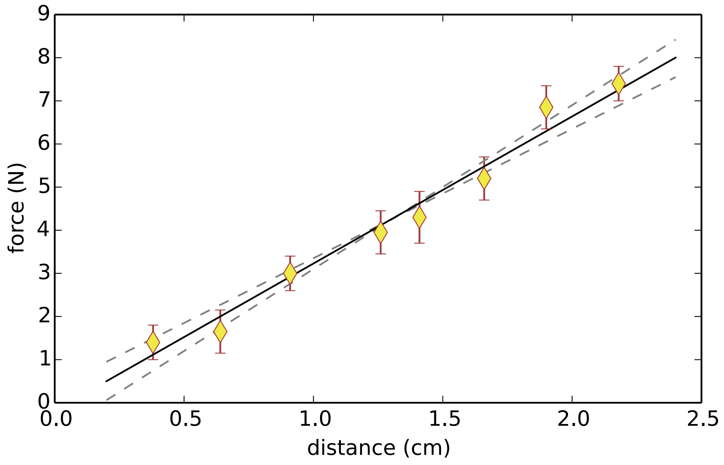
```
29.0     4.50     0.8
32.5     4.01     0.8
36.0     2.62     0.7
39.5     1.70     0.6
43.0     2.03     0.6
```

5. Use MatPlotLib's function `hist` along with NumPy's function's `random.rand` and `random.randn` to create the histogram graphs shown in Fig. *Histograms of random numbers.*

6. Plot force *vs* distance with error bars using the following data:

```
d=np.array([0.38, 0.64, 0.91, 1.26, 1.41, 1.66, 1.90, 2.18])
f=np.array([1.4, 1.65, 3.0, 3.95, 4.3, 5.20, 6.85, 7.4])
df=np.array([ 0.4, 0.5, 0.4, 0.5, 0.6, 0.5, 0.5, 0.4])
```

Your plot should also include a visual straight "best fit" to the data as well as visual "fits" that give the smallest and largest slopes consistent with the data. Note, you only need two points to define a straight line so the straight lines you draw on the plot should be arrays of length 2 and no longer. All of your fitted lines should lie *behind* the data. Try to make your plot look like the one below. *In addition*, add a legend to your plot the gives the slope with its uncertainty obtained from your visual fits to the data.



The web page http://matplotlib.org/api/pyplot_summary.html gives a summary of the main plotting commands available in MatPlotLib. The two important ones here are `plot` and `errorbar`, which make regular plots and plots with error bars, respectively. You will find the following keyword arguments useful: `yerr`, `ls`, `marker`, `mfc`, `mec`, `ms`, and `ecolor`, which you can find described by clicking on the `errorbar` function link on the web page cited above.

7. The data file below shows data obtained for the displacement (position) *vs* time of a falling object, together with the estimated uncertainty in the displacement.

```
Measurements of fall velocity vs time
Taken by A.P. Crawford and S.M. Torres
19-Sep-13
time (s)    position (m)    uncertainty (m)
 0.0            0.0              0.04
 0.5            1.3              0.12
 1.0            5.1              0.2
 1.5           10.9              0.3
 2.0           18.9              0.4
 2.5           28.7              0.4
 3.0           40.3              0.5
 3.5           53.1              0.6
 4.0           67.5              0.6
 4.5           82.3              0.6
 5.0           97.6              0.7
 5.5          113.8              0.7
 6.0          131.2              0.7
 6.5          148.5              0.7
 7.0          166.2              0.7
 7.5          184.2              0.7
 8.0          201.6              0.7
 8.5          220.1              0.7
 9.0          238.3              0.7
 9.5          256.5              0.7
10.0          275.6              0.8
```

   (a) Use these data to calculate the velocity and acceleration (in a Python program `.py` file), together with their uncertainties propagated from the displacement *vs* time uncertainties. Be sure to calculate time arrays corresponding the mid-point in time between the two displacements or velocities for the velocity and acceleration arrays, respectively.

   (b) In a single window frame, make three vertically stacked plots of the displacement, velocity, and acceleration *vs* time. Show the error bars on the different plots. Make sure that the time axes of all three plots cover the same range of times. Why do the relative sizes of the error bars grow progressively greater as one progresses from displacement to velocity to acceleration?

CHAPTER

SIX

CONDITIONALS AND LOOPS

Computer programs are useful for performing repetitive tasks. Without complaining, getting bored, or growing tired, they can repetitively perform the same calculations with minor, but important, variations over and over again. Humans share with computers none of these qualities. And so we humans employ computers to perform the massive repetitive tasks we would rather avoid. However, we need efficient ways of telling the computer to do these repetitive tasks; we don't want to have stop to tell the computer each time it finishes one iteration of a task to do the task again, but for a slightly different case. We want to tell it once, "Do this task 1000 times with slightly different conditions and report back to me when you are done." This is what **loops** were made for.

In the course of doing these repetitive tasks, computers often need to make decisions. In general, we don't want the computer to stop and ask us what it should do if a certain result is obtained from its calculations. We might prefer to say, "Look, if you get result A during your calculations, do this, otherwise, do this other thing." That is, we often want to tell the computer ahead of time what to do if it encounters different situations. This is what **conditionals** were made for.

Conditionals and loops control the flow of a program. They are essential to performing virtually any significant computational task. Python, like most computer languages, provides a variety of ways of implementing loops and conditionals.

6.1 Conditionals

Conditional statements allow a computer program to take different actions based on whether some condition, or set of conditions is true or false. In this way, the programmer can control the flow of a program.

6.1.1 `if`, `elif`, and `else` statements

The `if`, `elif`, and `else` statements are used to define conditionals in Python. We illustrate their use with a few examples.

`if-elif-else` example

Suppose we want to know if the solutions to the quadratic equation

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

are real, imaginary, or complex for a given set of coefficients a, b, and c. Of course, the answer to that question depends on the value of the discriminant $d = b^2 - 4ac$. The solutions are real if $d \geq 0$, imaginary if $b = 0$ and $d < 0$, and complex if $b \neq 0$ and $d < 0$. The program below implements the above logic in a Python program.

```
a = float(raw_input("What is the coefficients a? "))
b = float(raw_input("What is the coefficients b? "))
c = float(raw_input("What is the coefficients c? "))

d = b*b - 4.*a*c

if d >= 0.0:
    print("Solutions are real")          # block 1
elif b == 0.0:
    print("Solutions are imaginary")     # block 2
else:
    print("Solutions are complex")       # block 3

print("Finished!")
```

After getting the inputs of from the user, the program evaluates the discriminant d. The code `d >= 0.0` has a boolean truth value of either `True` or `False` depending on whether or not $d \geq 0$. You can check this out in the interactive IPython shell by typing the following set of commands

```
In [2]: d = 5

In [3]: d >= 2
Out[3]: True

In [4]: d >= 7
Out[4]: False
```

Therefore, the `if` statement in line 7 is simply testing to see if the statement `d >= 0.0` if `True` or `False`. If the statement is `True`, Python executes the indented block of statements following the `if` statement. In this case, there is only one line in indented block. Once it executes this statement, Python skips past the `elif` and `else` blocks and executes the `print("Finished!")` statement.

If the `if` statement in line 7 is `False`, Python skips the indented block directly below the `if` statement and executes the `elif` statement. If the condition `b == 0.0` is `True`, it executes the indented block immediately below the `elif` statement and then skips the `else` statement and the indented block below it. It then executes the `print("Finished!")` statement.

Finally, if the `elif` statement is `False`, Python skips to the `else` statement and executes the block immediately below the `else` statement. Once finished with that indented block, it then executes the `print("Finished!")` statement.

As you can see, each time a `False` result is obtained in an `if` or `elif` statement, Python skips the indented code block associated with that statement and drops down to the next conditional statement, that is, the next `elif` or `else`. A flowchart of the if-elif-else code is shown below.

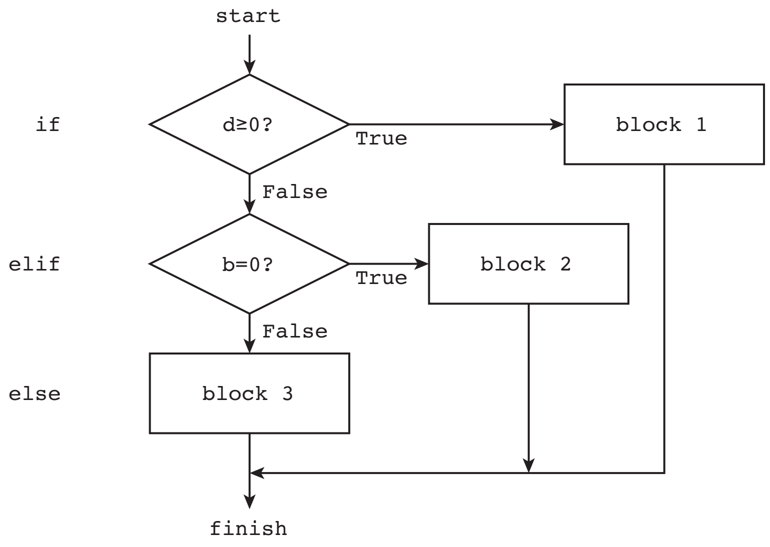


Figure 6.1: Flowchart for `if`-`elif`-`else` code.

At the outset of this problem we stated that the solutions to the quadratic equation are imaginary only if $b = 0$ and $d < 0$. In the `elif b == 0.0` statement on line 9, however, we only check to see if $b = 0$. The reason that we don't have to check if $d < 0$ is that the `elif` statement is executed only if the condition `if d >= 0.0` on line 7 is `False`. Similarly, we don't have to check if if $b = 0$ and $d < 0$ for the final `else` statement because this part of the `if`, `elif`, and `else` block will only be executed if the preceding `if` and `elif` statements are `False`. This illustrates a key feature of the `if`, `elif`, and `else` statements: these statements are executed sequentially until one of the `if` or `elif` statements is found to be `True`. Therefore, Python reaches an `elif` or `else` statement only if all the preceding `if` and `elif` statements are `False`.

The `if-elif-else` logical structure can accomodate as many `elif` blocks as desired. This allows you to set up logic with more than the three possible outcomes illustrated in the example above. When designing the logical structure you should keep in mind that once Python finds a true condition, it skips all subsequent `elif` and `else` statements in a given `if`, `elif`, and `else` block, irrespective of their truth values.

`if-else` example

You will often run into situations where you simply want the program to execute one of two possible blocks based on the outcome of an `if` statement. In this case, the `elif` block is omitted and you simply use an `if-else` structure. The following program testing whether an integer is even or odd provides a simple example.

```
a = int(raw_input("Please input an integer: "))
if a%2 == 0:
    print("{0:0d} is an even number.".format(a))
else:
    print("{0:0d} is an odd number.".format(a))
```

The flowchart below shows the logical structure of an `if-else` structure.

`if` example

The simplest logical structure you can make is a simple `if` statement, which executes a block of code if some condition is met but otherwise does nothing. The program below, which takes the absolute value of a number, provides a simple example of such a case.

```
a = eval(raw_input("Please input a number: "))
if a < 0:
    a = -a
print("The absolute value is {0}".format(a))
```

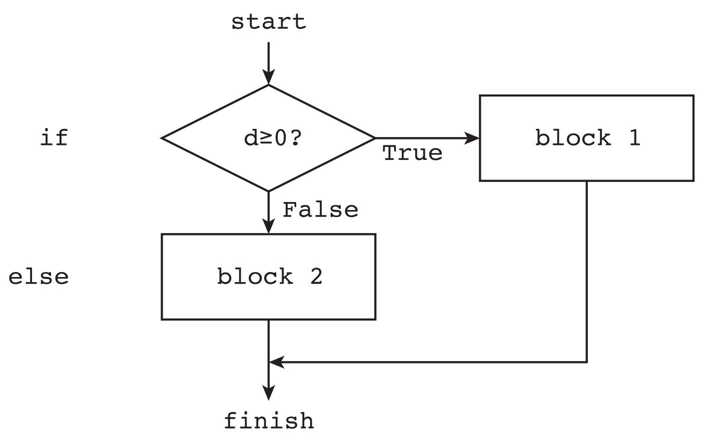


Figure 6.2: Flowchart for `if`-`else` code.

When the block of code in an `if` or `elif` statement is only one line long, you can write it on the same line as the `if` or `elif` statement. For example, the above code can be written as follows.

```
a = eval(raw_input("Please input a number: "))
if a < 0: a = -a
print("The absolute value is {0}".format(a))
```

This works exactly as the preceding code. Note, however, that if the block of code associated with an `if` or `elif` statement is more than one line long, the entire block of code should be written as indented text below the `if` or `elif` statement.

The flowchart below shows the logical structure of a simple `if` structure.

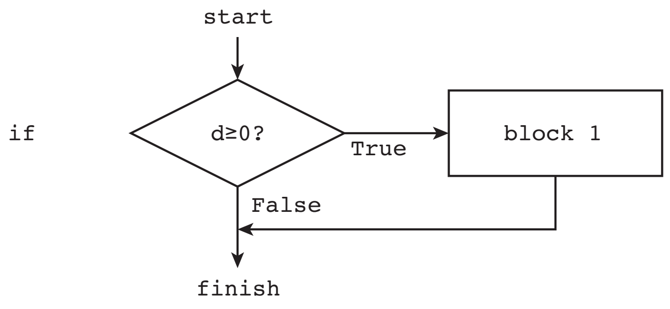


Figure 6.3: Flowchart for `if` code.

6.1.2 Logical operators

It is important to understand that "==" in Python is not the same as "=". The operator "=" is the assignment operator: `d = 5` assigns the value of 5 to the valiable `d`. On the other hand "==" is the *logical equals operator* and `d == 5` is a *logical truth statement*. It tells Python to check to see if `d` is equal to `5` or not, and assigns a value of `True` or `False` to the statement `d == 5` depending on whether or not `d` is equal to `5`. The table below summarizes the various logical operators available in Python.

operator	function
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
`and`	both must be true
`or`	one or both must be true
`not`	reverses the truth value

Logical operators in Python

The table above list three logical operators, `and`, `or`, and `not`, that we haven't encountered before. There are useful for combining different logical conditions. For example, suppose you want to check if $a > 2$ and $b < 10$ simultaneously. To do so, you would write `a>2 and b<10`. The code below illustrates the use of the logical operators `and`, `or`, and `not`.

```
In [5]: a = 5

In [6]: b = 10

In [7]: a != 5          # a is not equal to 5
Out[7]: False

In [8]: a>2 and b<20
Out[8]: True

In [9]: a>2 and b>10
Out[9]: False

In [10]: a>2 or b>10
Out[10]: True
```

```
In [11]: a>2
Out[11]: True

In [12]: not a>2
Out[12]: False
```

Logical statements like those above can be used in `if`, `elif`, and, as we shall see below, `while` statements, according to your needs.

6.2 Loops

In computer programming a *loop* is statement or block of statements that is executed repeatedly. Python has two kinds of loops, a `for` loop and a `while` loop. We first introduce the `for` loop and illustrate its use for a variety of tasks. We then introduce the `while` loop and, after a few illustrative examples, compare the two kinds of loops and discuss when to use one or the other.

6.2.1 `for` loops

The general form of a `for` loop in Python is

```
for <intervar> in <sequence>:
    <body>
```

where `<intervar>` is a variable, `<sequence>` is a sequence such as list or string or array, and `<body>` is a series of Python commands to be executed repeatedly for each element in the `<sequence>`. The `<body>` is indented from the rest of the text, which difines the extent of the loop. Let's look at a few examples.

```
for dogname in ["Max", "Molly", "Buster", "Maggie", "Lucy"]:
    print(dogname)
    print("    Arf, arf!")
print("All done.")
```

Running this program produces the following output.

```
In [1]: run doggyloop.py
Max
    Arf, arf!
Molly
    Arf, arf!
```

```
Buster
    Arf, arf!
Maggie
    Arf, arf!
Lucy
    Arf, arf!
All done.
```

The `for` loop works as follows: the *iteration variable* or *loop index* `dogname` is set equal to the first element in the list, `"Max"`, and then the two lines in the indented body are executed. Then `dogname` is set equal to second element in the list, `"Molly"`, and the two lines in the indented body are executed. The loop cycles through all the elements of the list, and then moves on to the code that follows the `for` loop and prints `All done.`

When indenting a block of code in a Python `for` loop, it is critical that every line be indented by the same amount. Using the **<tab>** key causes the Code Editor to indent 4 spaces. Any amount of indentation works, as long as it is the same for all lines in a `for` loop. While code editors designed to work with Python (including Canopy and Spyder) translate the **<tab>** key to 4 spaces, not all text editors do. In those cases, 4 spaces are not equivalent to a **<tab>** character even if they appear the same on the display. Indenting some lines by 4 spaces and other lines by a **<tab>** character will produce an error. So beware!

The figure below shows the flowchart for a `for` loop. It starts with an implicit conditional asking if there are any more elements in the sequence. If there are, it sets the iteration variable equal to the next element in the sequence and then executes the body—the indented text—using that value of the iteration variable. It then returns to the beginning to see if there are more elements in the sequence and continues the loop until there is none remaining.

6.2.2 Accumulators

Let's look at another application of Python's `for` loop. Suppose you want to calculate the sum of all the odd numbers between 1 and 100. Before writing a computer program to do this, let's think about how you would do it by hand. You might start by adding 1+3=4. Then take the result 4 and add the next odd integer, 5, to get 4+5=9; then 9+7=16, then 16+9=25, and so forth. You are doing repeated additions, starting with 1+3, while keeping track of the running sum, until you reach the last number 99.

In developing an algorithm for having the computer sum the series of numbers, we are going to do the same thing: add the numbers one at a time while keeping track of the running sum, until we reach the last number. We will keep track of the running sum with the variable `s`, which is called the *accumulator*. Initially `s=0`, since we haven't adding

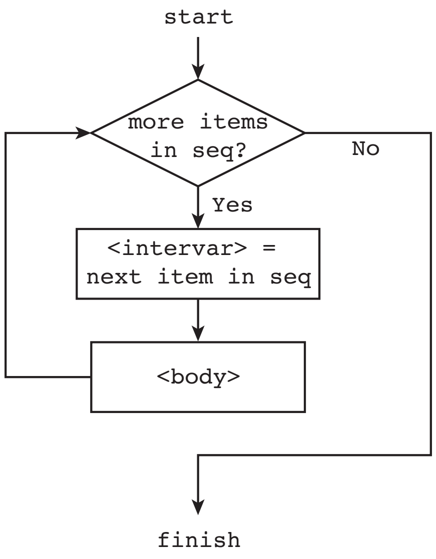


Figure 6.4: Flowchart for `for`-loop.

any numbers yet. Then we add the first number, 1, to `s` and `s` becomes 1. Then we add then next number, 3, in our sequence of odd numbers to `s` and `s` becomes 4. We continue doing this over and over again using a `for` loop while the variable `s` accumulates the running sum until we reach the final number. The code below illustrates how to do this.

```python
s = 0
for i in range(1, 100, 2):
    s = s+i
print(s)
```

The `range` function defines the list `[1, 3, 5, ..., 97, 99]`. The `for` loop successively adds each number in the list to the running sum until it reaches the last element in the list and the sum is complete. Once the `for` loop finishes, the program exits the loop and the final value of `s`, which is the sum of the odd numbers from 1 to 99, is printed out. Copy the above program and run it. You should get an answer of 2500.

6.2.3 `while` loops

The general form of a `while` loop in Python is

```
while <condition>:
    <body>
```

where `<condition>` is a statement that can be either `True` or `False` and `<body>` is a series of Python commands that is executed repeatedly until `<condition>` becomes false. This means that somewhere in `<body>`, the truth value of <condition> must be changed so that it becomes false after a finite number of iterations. Consider the following example.

Suppose you want to calculate all the Fibonacci numbers smaller than 1000. The Fibonacci numbers are determined by starting with the integers 0 and 1. The next number in the sequence is the sum of the previous two. So, starting with 0 and 1, the next Fibonacci number is $0 + 1 = 1$, giving the sequence $0, 1, 1$. Continuing this process, we obtain $0, 1, 1, 2, 3, 5, 8, ...$ where each element in the list is the sum of the previous two. Using a `for` loop to calculate the Fibonacci numbers is impractical because we do not know ahead of time how many Fibonacci numbers there are smaller than 1000. By contrast a `while` loop is perfect for calculating all the Fibonacci numbers because it keeps calculating Fibonacci numbers until it reaches the desired goal, in this case 1000. Here is the code using a `while` loop.

```
x, y = 0, 1
while x < 1000:
    print(x)
    x, y = y, x+y
```

We have used the multiple assignment feature of Python in this code. Recall that all the values on the left are assigned using the original values of `x` and `y`.

The figure below shows the flowchart for the `while` loop. The loop starts with the evaluation of a condition. If the condition is `False`, the code in the body is skipped, the flow exits the loop, and then continues with the rest of the program. If the condition is `True`, the code in the body—the indented text—is executed. Once the body is finished, the flow returns to the condition and proceeds along the `True` or `False` branches depending on the truth value of the condition. Implicit in this loop is the idea that somewhere during the execution of the body of the while loop, the variable that is evaluated in the condition is changed in some way. Eventually that change will cause the condition to return a value of `False` so that the loop will end.

One danger of a `while` loop is that it entirely possible to write a loop that never terminates—an *infinite loop*. For example, if we had written `while y > 0:`, in place of `while x < 1000:`, the loop would never end. If you execute code that has an infinite loop, you can often terminate the program from the keyboard by typing **ctrl-C** a couple of times. If that doesn't work, you may have to terminate and then restart Python.

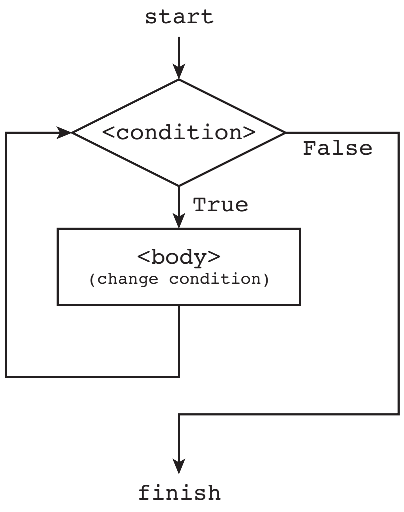


Figure 6.5: Flowchart for `while` loop.

For the kind of work we do in science and engineering, we generally find that the `for` loop is more useful than the `while` loop. Nevertheless, there are times when using a `while` loop is better suited to a task than is a `for` loop.

6.2.4 Loops and array operations

Loops are often used to sequentially modify the elements of an array. For example, suppose we want to square each element of the array `a = np.linspace(0, 32, 1e7)`. This is a hefty array with 10 million elements. Nevertheless, the following loop does the trick.

```
import numpy as np
a = np.linspace(0, 32, 1e7)
print(a)
for i in range(len(a)):
    a[i] = a[i]*a[i]
print(a)
```

Running this on my computer returns the result in about 8 seconds—not bad for having performed 10 million multiplications. Of course we could have performed the same calculation using the array multiplication we learned in Chapter 3 (*Strings, Lists, Arrays, and Dictionaries*). Here is the code.

```
import numpy as np
a = np.linspace(0, 32, 1e7)
print(a)
a = a*a
print(a)
```

Running this on my computer returns the results faster than I can discern, but certainly much less than a second. This illustrates an important point: **for loops are slow**. Array operations run much faster and are therefore to be preferred in any case where you have a choice. Sometimes finding an array operation that is equivalent to a loop can be difficult, especially for a novice. Nevertheless, doing so pays rich rewards in execution time. Moreover, the array notation is usually simpler and clearer, providing further reasons to prefer array operations over loops.

6.3 List Comprehensions

List comprehensions are a special feature of core Python for processing and constructing lists. We introduce them here because they use a looping process. They are used quite commonly in Python coding and they often provide elegant compact solutions to some common computing tasks.

Consider, for example the 3×3 matrix

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

Suppose we want to construct a vector from the diagonal elements of this matrix. We could do so with a `for` loop with an accumulator as follows

```
In [2]: diag = []
   ...: for i in [0, 1, 2]:
   ...:     diag.append(x[i][i])
   ...:

In [3]: diag
Out[3]: [1, 5, 9]
```

List comprehensions provide a simpler, cleaner, and faster way of doing the same thing

```
In [4]: diagLC = [x[i][i] for i in [0, 1, 2]]

In [5]: diagLC
Out[5]: [1, 5, 9]
```

A one-line list comprehension replaces a three-line accumulator plus loop code.

Suppose we now want the square of this list:

```
In [6]: [y*y for y in diagLC]
Out[6]: [1, 25, 81]
```

Notice here how `y` serves as a dummy variable accessing the various elements of the list `diagLC`.

Extracting a column from a 2-dimensaional array such as `x` is quite easy. For example the second row is obtained quite simply in the following fashion

```
In [7]: x[1]
Out[7]: [4, 5, 6]
```

Obtaining a column is not as simple, but a list comprehension makes it quite straightforward:

```
In [7]: c1 = [a[1] for a in x]
In [8]: c1
Out[8]: [2, 5, 8]
```

Another, slightly less elegant way to accomplish the same thing is

```
In [9]: [x[i][1] for i in range(3)]
Out[9]: [2, 5, 8]
```

Suppose you have a list of numbers and you want to extract all the elements of the list that are divisible by three. A slightly fancier list comprehension accomplishes the task quite simply and demonstrates a new feature:

```
In [10]:  y = [-5, -3, 1, 7, 4, 23, 27, -9, 11, 41]
In [14]: [a for a in y if a%3==0]
Out[14]: [-3, 27, -9]
```

As we see in this example, a conditional statement can be added to a list comprehension. Here it serves as a filter to select out only those elements that are divisible by three.

6.4 Exercises

1. Write a program to calculate the factorial of a positive integer input by the user. Recall that the factorial function is given by $x! = x(x-1)(x-2)...(2)(1)$ so that 1! = 1, 2! = 2, 3! = 6, 4! = 24, ...

   (a) Write the factorial function using a Python `while` loop.

   (b) Write the factorial function using a Python `for` loop.

   Check your programs to make sure they work for 1, 2, 3, 5, and beyond, but especially for the first 5 integers.

2. The following Python program finds the smallest non-trivial (not 1) prime factor of a positive integer.

```
n = int(raw_input("Input an integer > 1: "))
i = 2

while (n % i) != 0:
    i += 1

print("The smallest factor of n is:", i )
```

   (a) Type this program into your computer and verify that it works as advertised. Then briefly explain how it works and why the while loop always terminates.

   (b) Modify the program so that it tells you if the integer input is a prime number or not. If it is not a prime number, write your program so that it prints out the smallest prime factor. Using your program verify that the following integers are prime numbers: 101, 8191, 947431.

3. Consider the matrix list `x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`. Write a list comprehension to extract the last column of the matrix [3, 6, 9]. Write another list comprehension to create a vector of twice the square of the middle column `[8, 50, 128]`.

4. Write a program that calculates the value of an investment after some number of years specified by the user if

   (a) the principal is compounded annually

   (b) the principle is compounded monthly

   (c) the principle is compounded daily

Your program should ask the user for the initial investment (principal), the interest rate in percent, and the number of years the money will be invested (allow for fractional years). For an initial investment of \$1000 at an interest rate of 6%, after 10 years I get \$1790.85 when compounded annually, \$1819.40 when compounded monthly, and \$1822.03 when compounded daily, assuming 12 months in a year and 365.24 days in a year, where the monthly interest rate is the annual rate divided by 12 and the daily rate is the annual rate divided by 365 (don't worry about leap years).

5. Write a program that determines the day of the week for any given calendar date after January 1, 1900, which was a Monday. Your program will need to take into account leap years, which occur in every year that is divisible by 4, except for years that are divisible by 100 but are not divisible by 400. For example, 1900 was not a leap year, but 2000 was a leap year. Test that your program gives the following answers: Monday 1900 January 1, Tuesday 1933 December 5, Wednesday 1993 June 23, Thursday 1953 January 15, Friday 1963 November 22, Saturday 1919 June 28, Sunday 2005 August 28.

CHAPTER

SEVEN

FUNCTIONS

As you develop more complex computer code, it becomes increasingly important to organize your code into modular blocks. One important means for doing so is *user-defined* Python functions. User-defined functions are a lot like built-in functions that we have encountered in core Python as well as in NumPy and Matplotlib. The main difference is that user-defined functions are written by you. The idea is to define functions to simplify your code and to allow you to reuse the same code in different contexts.

The number of ways that functions are used in programming is so varied that we cannot possibly enumerate all the possibilities. As our use of Python functions in scientific program is somewhat specialized, we introduce only a few of the possible uses of Python functions, ones that are the most common in scientific programming.

7.1 User-defined functions

The NumPy package contains a plethora of mathematical functions. You can find a listing of the mathematical functions available through NumPy on the web page http://docs.scipy.org/doc/numpy/reference/routines.math.html. While the list may seem pretty exhaustive, you may nevertheless find that you need a function that is not available in the NumPy Python library. In those cases, you will want to write your own function.

In studies of optics and signal processing one often runs into the sinc function, which is defined as

$$\operatorname{sinc} x \equiv \frac{\sin x}{x} \ .$$

Let's write a Python function for the sinc function. Here is our first attempt:

```
def sinc(x):
    y = np.sin(x)/x
    return y
```

Every function definition begins with the word `def` followed by the name you want to give to the function, `sinc` in this case, then a list of arguments enclosed in parentheses, and finally terminated with a colon. In this case there is only one argument, `x`, but in general there can be as many arguments as you want, including no arguments at all. For the moment, we will consider just the case of a single argument.

The indented block of code following the first line defines what the function does. In this case, the first line calculates $\operatorname{sinc} x = \sin x/x$ and sets it equal to `y`. The `return` statement of the last line tells Python to return the value of `y` to the user.

We can try it out in the IPython shell. First we type in the function definition.

```
In [1]: def sinc(x):
   ...:     y = sin(x)/x
   ...:     return y
```

Because we are doing this from the IPython shell, we don't need to import NumPy; it's preloaded. Now the function $\operatorname{sinc} x$ is available to be used from the IPython shell

```
In [2]: sinc(4)
Out[2]: -0.18920062382698205

In [3]: a = sinc(1.2)

In [4]: a
Out[4]: 0.77669923830602194

In [5]: sin(1.2)/1.2
Out[5]: 0.77669923830602194
```

Inputs and outputs 4 and 5 verify that the function does indeed give the same result as an explicit calculation of $\sin x/x$.

You may have noticed that there is a problem with our definition of $\operatorname{sinc} x$ when `x=0.0`. Let's try it out and see what happens

```
In [6]: sinc(0.0)
Out[6]: nan
```

IPython returns `nan` or "not a number", which occurs when Python attempts a division by zero, which is not defined. This is not the desired response as $\operatorname{sinc} x$ is, in fact, perfectly well defined for $x = 0$. You can verify this using L'Hopital's rule, which you may have learned in your study of calculus, or you can ascertain the correct answer by calculating the Taylor series for $\operatorname{sinc} x$. Here is what we get

$$\operatorname{sinc} x = \frac{\sin x}{x} = \frac{x - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots}{x} = 1 - \frac{x^2}{3!} + \frac{x^4}{5!} + \ldots .$$

From the Taylor series, it is clear that $\operatorname{sinc} x$ is well-defined at and near $x = 0$ and that, in fact, $\operatorname{sinc}(0) = 1$. Let's modify our function so that it gives the correct value for `x=0`.

```
In [7]: def sinc(x):
   ...:     if x==0.0:
   ...:         y = 1.0
   ...:     else:
   ...:         y = sin(x)/x
   ...:     return y

In [8]: sinc(0)
Out[8]: 1.0

In [9]: sinc(1.2)
Out[9]: 0.77669923830602194
```

Now our function gives the correct value for `x=0` as well as for values different from zero.

7.1.1 Looping over arrays in user-defined functions

The code for $\operatorname{sinc} x$ works just fine when the argument is a single number or a variable that represents a single number. However, if the argument is a NumPy array, we run into a problem, as illustrated below.

```
In [10]: x = arange(0, 5., 0.5)

In [11]: x
Out[11]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,
                 4. ,  4.5])

In [12]: sinc(x)
----------------------------------------------------------
ValueError                  Traceback (most recent call last)
----> 1 sinc(x)

      1 def sinc(x):
----> 2     if x==0.0:
      3         y = 1.0
      4     else:
      5         y = np.sin(x)/x

ValueError: The truth value of an array with more than one
            element is ambiguous.
```

The `if` statement in Python is set up to evaluate the truth value of a single variable, not of multielement arrays. When Python is asked to evaluate the truth value for a multi-element array, it doesn't know what to do and therefore returns an error.

An obvious way to handle this problem is to write the code so that it processes the array one element at a time, which you could do using a `for` loop, as illustrated below.

```
1  def sinc(x):
2      y = []              # creates an empty list to store results
3      for xx in x:        # loops over all elements in x array
4          if xx==0.0:     # adds result of 1.0 to y list if
5              y += [1.0]  # xx is zero
6          else:           # adds result of sin(xx)/xx to y list if
7              y += [np.sin(xx)/xx]  # xx is not zero
8      return np.array(y)  # converts y to array and returns array
9
10 import numpy as np
11 import matplotlib.pyplot as plt
12
13 x = np.linspace(-10, 10, 256)
14 y = sinc(x)
15
16 plt.plot(x, y)
17 plt.axhline(color="gray", zorder=-1)
18 plt.axvline(color="gray", zorder=-1)
19 plt.show()
```

The `for` loop evaluates the elements of the `x` array one by one and appends the results to the list `y` one by one. When it is finished, it converts the list to an array and returns the array. The code following the function definition plots $\operatorname{sinc} x$ as a function of x.

In the program above, you may have noticed that the NumPy library is imported *after* the `sinc(x)` function definition. As the function uses the NumPy functions `sin` and `array`, you may wonder how this program can work. Doesn't the `import numpy` statement have to be called before any NumPy functions are used? The answer it an emphatic "YES". What you need to understand is that the function definition is *not executed* when it is defined, nor can it be as it has no input `x` data to process. That part of the code is just a definition. The first time the code for the `sinc(x)` function is actually executed is when it is called on line 14 of the program, which occurs after the NumPy library is imported in line 10. The figure below shows the plot of the $\operatorname{sinc} x$ function generated by the above code.

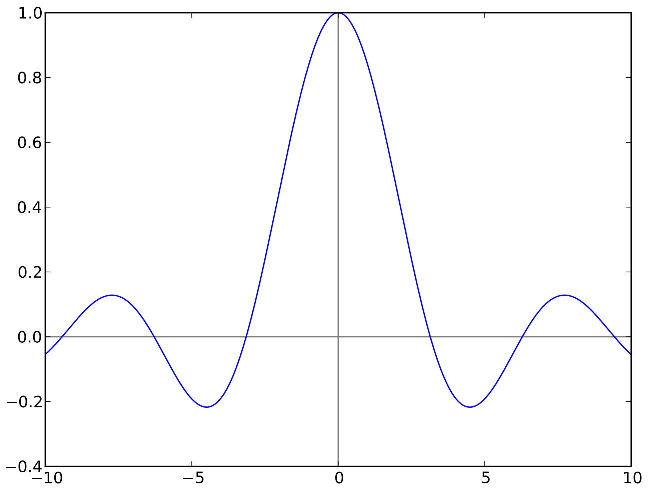


Figure 7.1: Plot of user-defined `sinc(x)` function.

7.1.2 Fast array processing in user-defined functions

While using loops to process arrays works just fine, it is usually not the best way to accomplish the task in Python. The reason is that loops in Python are executed rather slowly. To deal with this problem, the developers of NumPy introduced a number of functions designed to process arrays quickly and efficiently. For the present case, what we need is a conditional statement or function that can process arrays directly. The function we want is called `where` and it is a part of the NumPy library. There `where` function has the form

```
where(condition, output if True, output if False)
```

The first argument of the `where` function is a conditional statement involving an array. The `where` function applies the condition to the array element by element, and returns the second argument for those array elements for which the condition is `True`, and returns the third argument for those array elements that are `False`. We can apply it to the `sinc(x)` function as follows

```
def sinc(x):
    z = np.where(x==0.0, 1.0, np.sin(x)/x)
    return z
```

The `where` function creates the array `y` and sets the elements of `y` equal to 1.0 where the corresponding elements of `x` are zero, and otherwise sets the corresponding elements to `sin(x)/x`. This code executes much faster, 25 to 100 times, depending on the size of the array, than the code using a `for` loop. Moreover, the new code is much simpler to write and read. An additional benefit of the `where` function is that it can handle single variables and arrays equally well. The code we wrote for the sinc function with the `for` loop cannot handle single variables. Of course we could rewrite the code so that it did, but the code becomes even more clunky. It's better just to use NumPy's `where` function.

The moral of the story

The moral of the story is that you should avoid using `for` and `while` loops to process arrays in Python programs whenever an array-processing method is available. As a beginning Python programmer, you may not always see how to avoid loops, and indeed, avoiding them is not always possible, but you should look for ways to avoid loops, especially loops that iterate a large number of times. As you become more experienced, you will find that using array-processing methods in Python becomes more natural. Using them can greatly speed up the execution of your code, especially when working with large arrays.

7.1.3 Functions with more (or less) than one input or output

Python functions can have any number of input arguments and can return any number of variables. For example, suppose you want a function that outputs n (x, y) coordinates around a circle of radius r centered at the point (x_0, y_0). The inputs to the function would be r, x_0, y_0, and n. The outputs would be the n (x, y) coordinates. The following code implements this function.

```
def circle(r, x0, y0, n):
    theta = np.linspace(0., 2.*np.pi, n, endpoint=False)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x0+x, y0+y
```

This function has four inputs and two outputs. In this case, the four inputs are simple numeric variables and the two outputs are NumPy arrays. In general, the inputs and outputs can be any combination of data types: arrays, lists, strings, *etc*. Of course, the body of the function must be written to be consistent with the prescribed data types.

Functions can also return nothing to the calling program but just perform some task. For example, here is a program that clears the terminal screen

```
import subprocess
import platform

def clear():
    subprocess.Popen( "cls" if platform.system() ==
                      "Windows" else "clear", shell=True)
```

The function is invoked by typing `clear()`. It has no inputs and no outputs but it performs a useful task. This function uses two standard Python libraries, `subprocess` and `platform` that are useful for performing computer system tasks. It's not important that you know anything about them at this point. We simply use them here to demonstrate a useful cross-platform function that has no inputs and returns no values.

7.1.4 Positional and keyword arguments

It is often useful to have function arguments that have some default setting. This happens when you want an input to a function to have some standard value or setting most of the time, but you would like to reserve the possibility of giving it some value other than the default value.

For example, in the program `circle` from the previous section, we might decide that under most circumstances, we want `n=12` points around the circle, like the points on a clock face, and we want the circle to be centered at the origin. In this case, we would rewrite the code to read

```
def circle(r, x0=0.0, y0=0.0, n=12):
    theta = np.linspace(0., 2.*np.pi, n, endpoint=False)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x0+x, y0+y
```

The default values of the arguments `x0`, `y0`, and `n` are specified in the argument of the function definition in the `def` line. Arguments whose default values are specified in this manner are called *keyword arguments*, and they can be omitted from the function call if the user is content using those values. For example, writing `circle(4)` is now a perfectly legal way to call the `circle` function and it would produce 12 (x, y) coordinates centered about the origin $(x, y) = (0, 0)$. On the other hand, if you want the values of `x0`, y0, and n to be something different from the default values, you can specify their values as you would have before.

If you want to change only some of the keyword arguments, you can do so by using the keywords in the function call. For example, suppose you are content with have the circle centered on $(x, y) = (0, 0)$ but you want only 6 points around the circle rather than 12. Then you would call the circle function as follows:

```
circle(2, n=6)
```

The unspecified keyword arguments keep their default values of zero but the number of points n around the circle is now 6 instead of the default value of 12.

The normal arguments without keywords are called *positional arguments*; they have to appear *before* any keyword arguments and, when the function is called, must be supplied values in the same order as specified in the function definition. The keyword arguments, if supplied, can be supplied in any order providing they are supplied with their keywords. If supplied without their keywords, they too must be supplied in the order they appear in the function definition. The following function calls to circle both give the same output.

```
In [13]: circle(3, n=3, y0=4, x0=-2)
Out[13]: (array([ 1. , -3.5, -3.5]),
          array([ 4.        ,  6.59807621,  1.40192379]))

In [14]: circle(3, -2, 4, 3)     # w/o keywords, arguments
                                 # supplied in order
Out[14]: (array([ 1. , -3.5, -3.5]), array([ 4.        ,
                    6.59807621,  1.40192379]))
```

By now you probably have noticed that we used the keyword argument endpoint in calling linspace in our definition of the circle function. The default value of endpoint is True, meaning that linspace includes the endpoint specified in the second argument of linspace. We set it equal to False so that the last point was not included. Do you see why?

7.1.5 Variable number of arguments

While it may seem odd, it is sometimes useful to leave the number of arguments unspecified. A simple example is a function that computes the product of an arbitrary number of numbers:

```
def product(*args):
    print("args = {}".format(args))
    p = 1
    for num in args:
        p *= num
    return p

In [15]: product(11., -2, 3)
args = (11.0, -2, 3)
Out[15]: -66.0

In [16]: product(2.31, 7)
args = (2.31, 7)
Out[16]: 16.17
```

The `print("args...)` statement in the function definition is not necessary, of course, but is put in to show that the argument `args` is a tuple inside the function. Here it used because one does not know ahead of time how many numbers are to be multiplied together.

The `*args` argument is also quite useful in another context: when passing the name of a function as an argument in another function. In many cases, the function name that is passed may have a number of parameters that must also be passed but aren't known ahead of time. If this all sounds a bit confusing—functions calling other functions—a concrete example will help you understand.

Suppose we have the following function that numerically computes the value of the derivative of an arbitrary function $f(x)$:

```
def deriv(f, x, h=1.e-9, *params):
    return (f(x+h, *params)-f(x-h, *params))/(2.*h)
```

The argument `*params` is an optional positional argument. We begin by demonstrating the use of the function `deriv` without using the optional `*params` argument. Suppose we want to compute the derivative of the function $f_0(x) = 4x^5$. First, we define the function

```
def f0(x):
    return 4.*x**5
```

Now let's find the derivative of $f_0(x) = 4x^5$ at $x = 3$ using the function `deriv`:

```
In [17]: deriv(f0, 3)
Out[17]: 1620.0001482502557
```

The exact result, given by evaluating $f_0'(x) = 20x^4$ at $x = 3$ is 1620, so our function to numerically calculate the derivative works pretty well.

Suppose we had defined a more general function $f_1(x) = ax^p$ as follows:

```
def f1(x, a, p):
    return a*x**p
```

Suppose we want to calculate the derivative of this function for a particular set of parameters a and p. Now we face a problem, because it might seem that there is no way to pass the parameters a and p to the `deriv` function. Moreover, this is a generic problem for functions such as `deriv` that use a function as an input, because different functions you want to use as inputs generally come with different parameters. Therefore, we would like to write our program `deriv` so that it works, irrespective of how many parameters are needed to specify a particular function.

This is what the optional positional argument `*params` defined in `deriv` is for: to pass parameters of `f1`, like a and b, through `deriv`. To see how this works, let's set a and b to be 4 and 5, respectively, the same values we used in the definition of `f0`, so that we can compare the results:

```
In [16]: deriv(f1, 3, 1.e-9, 4, 5)
Out[16]: 1620.0001482502557
```

We get the same answer as before, but this time we have used `deriv` with a more general form of the function $f_1(x) = ax^p$.

The order of the parameters is important. The function `deriv` uses `x`, the first argument of `f1`, as its principal argument, and then uses `a` and `p`, in the same order that they are defined in the function `f1`, to fill in the additional arguments—the parameters—of the function `f1`.

Optional arguments must appear after the regular positional and keyword arguments in a function call. The order of the arguments must adhere to the following convention:

```
def func(pos1, pos2, ..., keywd1, keywd2, ..., *args, **kwargs):
```

That is, the order of arguments is: positional arguments first, then keyword arguments, then optional positional arguments (`*args`), then optional keyword arguments (`**kwargs`). Note that to use the `*params` argument, we had to explicitly include the keyword argument `h` even though we didn't need to change it from its default value.

Python also allows for a variable number of keyword arguments—`**kwargs`—in a function call. While `*args` is a tuple, `kwargs` is a dictionary, so that the value of an optional keyword argument is accessed through its dictionary key.

7.1.6 Passing data to and from functions

Functions are like mini-programs within the larger programs that call them. Each function has a set of variables with certain names that are to some degree or other isolated from the calling program. We shall get more specific about just how isolated those variables are below, but before we do, we introduce the concept of a *namespace*. Each function has its own namespace, which is essentially a mapping of variable names to objects, like numerics, strings, lists, and so forth. It's a kind of dictionary. The calling program has its own namespace, distinct from that of any functions it calls. The distinctiveness of these namespaces plays an important role in how functions work, as we shall see below.

Variables and arrays created entirely within a function

An important feature of functions is that variables and arrays created *entirely within* a function cannot be seen by the program that calls the function unless the variable or array is explicitly passed to the calling program in the `return` statement. This is important because it means you can create and manipulate variables and arrays, giving them any name you please, without affecting any variables or arrays outside the function, even if the variables and arrays inside and outside a function share the same name.

To see what how this works, let's rewrite our program to plot the sinc function using the sinc function definition that uses the `where` function.

```python
def sinc(x):
    z = np.where(x==0.0, 1.0, np.sin(x)/x)
    return z

import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-10, 10, 256)
y = sinc(x)

plt.plot(x, y)
plt.axhline(color="gray", zorder=-1)
plt.axvline(color="gray", zorder=-1)
plt.show()
```

Running this program produces a plot like the plot of sinc shown in the previous section. Notice that the array variable `z` is only defined within the function definition of sinc. If we run the program from the IPython terminal, it produces the plot, of course. Then if we ask IPython to print out the arrays, `x`, `y`, and `z`, we get some interesting and informative results, as shown below.

```
In [15]: run sinc3.py

In [16]: x
Out[16]: array([-10.        ,  -9.99969482,  -9.99938964, ...,
          9.99938964,   9.99969482,  10.        ])

In [17]: y
Out[17]: array([-0.05440211, -0.05437816, -0.0543542 , ...,
               -0.0543542 , -0.05437816, -0.05440211])

In [18]: z
---------------------------------------------------------
NameError                 Traceback (most recent call last)

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

When we type in `x` at the `In [16]:` prompt, IPython prints out the array `x` (some of the output is suppressed because the array `x` has many elements); similarly for `y`. But when we type `z` at the `In [18]:` prompt, IPython returns a `NameError` because `z` is not defined. The IPython terminal is working in the same *namespace* as the program. But the namespace of the sinc function is isolated from the namespace of the program that calls it, and therefore isolated from IPython. This also means that when the sinc function ends with `return z`, it doesn't return the name `z`, but instead assigns the values in the array `z` to the array `y`, as directed by the main program in line 9.

Passing variables and arrays to functions: mutable and immutable objects

What happens to a variable or an array passed to a function when the variable or array is *changed* within the function? It turns out that the answers are different depending on whether the variable passed is a simple numeric variable, string, or tuple, or whether it is an array or list. The program below illustrates the different ways that Python handles single variables *vs* the way it handles lists and arrays.

```python
def test(s, v, t, l, a):
    s = "I am doing fine"
    v = np.pi**2
    t = (1.1, 2.9)
    l[-1] = 'end'
    a[0] = 963.2
    return s, v, t, l, a

import numpy as np

```

```
s = "How do you do?"
v = 5.0
t = (97.5, 82.9, 66.7)
l = [3.9, 5.7, 7.5, 9.3]
a = np.array(l)

print('*************')
print("s = {0:s}".format(s))
print("v = {0:5.2f}".format(v))
print("t = {0:s}".format(t))
print("l = {0:s}".format(l))
print("a = "),                  # comma suppresses line feed
print(a)
print('*************')
print('*call "test"*')

s1, v1, t1, l1, a1 = test(s, v, t, l, a)

print('*************')
print("s1 = {0:s}".format(s1))
print("v1 = {0:5.2f}".format(v1))
print("t1 = {0:s}".format(t1))
print("l1 = {0:s}".format(l1))
print("a1 = "),
print(a1)
print('*************')
print("s = {0:s}".format(s))
print("v = {0:5.2f}".format(v))
print("t = {0:s}".format(t))
print("l = {0:s}".format(l))
print("a = "),                  # comma suppresses line feed
print(a)
print('*************')
```

The function `test` has five arguments, a string `s`, a numerical variable `v`, a tuple `t`, a list `l`, and a NumPy array `a`. `test` modifies each of these arguments and then returns the modified `s`, `v`, `t`, `l`, `a`. Running the program produces the following output.

```
In [17]: run passingVars.py
*************
s = How do you do?
v =  5.00
t = (97.5, 82.9, 66.7)
l = [3.9, 5.7, 7.5, 9.3]
a =  [ 3.9  5.7  7.5  9.3]
```

```
*************
*call "test"*
*************
s1 = I am doing fine
v1 =  9.87
t1 = (1.1, 2.9)
l1 = [3.9, 5.7, 7.5, 'end']
a1 =  [ 963.2    5.7    7.5    9.3]
*************
s = How do you do?
v =  5.00
t = (97.5, 82.9, 66.7)
l = [3.9, 5.7, 7.5, 'end']
a =  [ 963.2    5.7    7.5    9.3]
*************
```

The program prints out three blocks of variables separated by asterisks. The first block merely verifies that the contents of `s`, `v`, `t`, `l`, and `a` are those assigned in lines 10-13. Then the function `test` is called. The next block prints the output of the call to the function `test`, namely the variables `s1`, `v1`, `t1`, `l1`, and `a1`. The results verify that the function modified the inputs as directed by the `test` function.

The third block prints out the variables `s`, `v`, `t`, `l`, and `a` from the calling program *after* the function `test` was called. These variables served as the inputs to the function `test`. Examining the output from the third printing block, we see that the values of the string `s`, the numeric variable `v`, and the contents of `t` are unchanged after the function call. This is probably what you would expect. On the other hand, we see that the list `l` and the array `a` are changed after the function call. This might surprise you! But these are important points to remember, so important that we summarize them in two bullet points here:

- Changes to string, variable, and tuple arguments of a function within the function do not affect their values in the calling program.
- Changes to values of elements in list and array arguments of a function within the function are reflected in the values of the same list and array elements in the calling function.

The point is that simple numerics, strings and tuples are immutable while lists and arrays are mutable. Because immutable objects can't be changed, changing them within a function creates new objects with the same name inside of the function, but the old immutable objects that were used as arguments in the function call remain unchanged in the calling program. On the other hand, if elements of mutable objects like those in lists or arrays are changed, then those elements that are changed inside the function are also changed in the calling program.

7.2 Methods and attributes

You have already encountered quite a number of functions that are part of either NumPy or Python or Matplotlib. But there is another way in which Python implements things that act like functions. To understand what they are, you need to understand that variables, strings, arrays, lists, and other such data structures in Python are not merely the numbers or strings we have defined them to be. They are *objects*. In general, an object in Python has associated with it a number of *attributes* and a number of specialized functions called *methods* that act on the object. How attributes and methods work with objects is best illustrated by example.

Let's start with the NumPy array. A NumPy array is a Python object and therefore has associated with it a number of attributes and methods. Suppose, for example, we write `a = random.random(10)`, which creates an array of 10 uniformly distributed random numbers between 0 and 1. An example of an attribute of an array is the size or number of elements in the array. An attribute of an object in Python is accessed by typing the object name followed by a period followed by the attribute name. The code below illustrates how to access two different attributes of an array, it's size and its data type.

```
In [18]: a = random.random(10)

In [19]: a.size
Out[19]: 10

In [20]: a.dtype
Out[20]: dtype('float64')
```

Any object in Python can and in general does have a number of attributes that are accessed in just the way demonstrated above, with a period and the attribute name following the name of the particular object. In general, attributes involve properties of the object that are stored by Python with the object and require no computation. Python just looks up the attribute and returns its value.

Objects in Python also have associated with them a number of specialized functions called *methods* that act on the object. In contrast to attributes, methods generally involve Python performing some kind of computation. Methods are accessed in a fashion similar to attributes, by appending a period followed the method's name, which is followed by a pair of open-close parentheses, consistent with methods being a kind of function that acts on the object. Often methods are used with no arguments, as methods by default act on the object whose name they follow. In some cases. however, methods can take arguments. Examples of methods for NumPy arrays are sorting, calculating the mean, or standard deviation of the array. The code below illustrates a few array methods.

```
In [21]: a
Out[21]:
array([ 0.859057  ,  0.27228037,  0.87780026,  0.14341207,
        0.05067356,  0.83490135,  0.54844515,  0.33583966,
        0.31527767,  0.15868803])

In [22]: a.sum()              # sum
Out[22]: 4.3963751104791005

In [23]: a.mean()             # mean or average
Out[23]: 0.43963751104791005

In [24]: a.var()              # variance
Out[24]: 0.090819477333711512

In [25]: a.std()              # standard deviation
Out[25]: 0.30136270063448711

In [26]: a.sort()             # sort small to large

In [27]: a
Out[27]:
array([ 0.05067356,  0.14341207,  0.15868803,  0.27228037,
        0.31527767,  0.33583966,  0.54844515,  0.83490135,
        0.859057  ,  0.87780026])

In [28]: a.clip(0.3, 0.8)
Out[29]:
array([ 0.3       ,  0.3       ,  0.3       ,  0.3       ,
        0.31527767,  0.33583966,  0.54844515,  0.8       ,
        0.8       ,  0.8       ])
```

The `clip()` method provides an example of a method that takes an argument, in this case the arguments are the lower and upper values to which array elements are cutoff if their values are outside the range set by these values.

7.3 Example: linear least squares fitting

In this section we illustrate how to use functions and methods in the context of modeling experimental data.

In science and engineering we often have some theoretical curve or *fitting function* that we would like to fit to some experimental data. In general, the fitting function is of the form

$f(x; a, b, c, ...)$, where x is the independent variable and a, b, c, ... are parameters to be adjusted so that the function $f(x; a, b, c, ...)$ best fits the experimental data. For example, suppose we had some data of the velocity *vs* time for a falling mass. If the mass falls only a short distance such that its velocity remains well below its terminal velocity, we can ignore air resistance. In this case, we expect the acceleration to be constant and the velocity to change linearly in time according to the equation

$$v(t) = v_0 - gt \ , \tag{7.1}$$

where g is the local gravitational acceleration. We can fit the data graphically, say by plotting it as shown below in Fig. 4.6 and then drawing a line through the data. When we draw a straight line through a data, we try to minimize the distance between the points and the line, globally averaged over the whole data set.

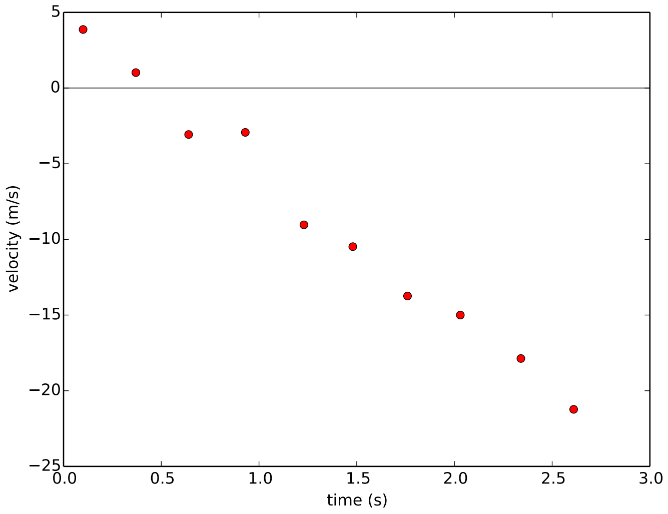


Figure 7.2: Velocity *vs* time for falling mass.

While this can give a reasonable estimate of the best fit to the data, the procedure is rather *ad hoc*. We would prefer to have a more well-defined analytical method for determining what constitutes a "best fit". One way to do that is to consider the sum

$$S = \sum_i^n [y_i - f(x_i; a, b, c, ...)]^2 \ , \tag{7.2}$$

where y_i and $f(x_i; a, b, c, ...)$ are the values of the experimental data and the fitting function, respectively, at x_i, and S is the square of their difference summed over all n data points. The quantity S is a sort of global measure of how much the the fit $f(x_i; a, b, c, ...)$ differs from the experimental data y_i.

Notice that for a given set of data points $\{x_i, y_i\}$, S is a function only of the fitting parameters $a, b, ...$, that is, $S = S(a, b, c, ...)$. One way of defining a *best* fit, then, is to find the values of the fitting parameters $a, b, ...$ that minimize the S.

In principle, finding the values of the fitting parameters $a, b, ...$ that minimize the S is a simple matter. Just set the partial derivatives of S with respect to the fitting parameter equal to zero and solve the resulting system of equations:

$$\frac{\partial S}{\partial a} = 0 \, , \quad \frac{\partial S}{\partial b} = 0 \, , ... \tag{7.3}$$

Because there are as many equations as there are fitting paramters, we should be able to solve the system of equations and find the values of the fitting parameters that minimize S. Solving those systems of equations is straightforward if the fitting function $f(x; a, b, ...)$ is linear in the fitting parameters. Some examples of fitting functions linear in the fitting parameters are:

$$\begin{aligned} f(x; a, b) &= a + bx \\ f(x; a, b, c) &= a + bx + cx^2 \\ f(x; a, b, c) &= a \sin x + be^x + ce^{-x^2} \, . \end{aligned} \tag{7.4}$$

For fitting functions such as these, taking the partial derivatives with respect to the fitting parameters, as proposed in (7.3), results in a set of algebraic equations that are linear in the fitting paramters $a, b, ...$ Because they are linear, these equations can be solved in a straightforward manner.

For cases in which the fitting function is not linear in the fitting parameters, one can generally still find the values of the fitting parameters that minimize S but finding them requires more work, which goes beyond our immediate interests here.

7.3.1 Linear regression

We start by considering the simplest case, fitting a straight line to a data set, such as the one shown in Fig. *4.6* above. Here the fitting function is $f(x) = a + bx$, which is linear in the fitting parameters a and b. For a straight line, the sum in (7.2) becomes

$$S(a, b) = \sum_i (y_i - a - bx_i)^2 \, . \tag{7.5}$$

Finding the best fit in this case corresponds to finding the values of the fitting parameters a and b for which $S(a,b)$ is a minimum. To find the minimum, we set the derivatives of $S(a,b)$ equal to zero:

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_i -2(y_i - a - bx_i) = 2\left(na + b\sum_i x_i - \sum_i y_i\right) = 0 \\ \frac{\partial S}{\partial b} &= \sum_i -2(y_i - a - bx_i)\, x_i = 2\left(a\sum_i x_i + b\sum_i x_i^2 - \sum_i x_i y_i\right) = 0\end{aligned} \tag{7.6}$$

Dividing both equations by $2n$ leads to the equations

$$\begin{aligned}a + b\bar{x} &= \bar{y} \\ a\bar{x} + b\frac{1}{n}\sum_i x_i^2 &= \frac{1}{n}\sum_i x_i y_i\end{aligned} \tag{7.7}$$

where

$$\begin{aligned}\bar{x} &= \frac{1}{n}\sum_i x_i \\ \bar{y} &= \frac{1}{n}\sum_i y_i \;.\end{aligned} \tag{7.8}$$

Solving Eq. (7.7) for the fitting parameters gives

$$\begin{aligned}b &= \frac{\sum_i x_i y_i - n\bar{x}\bar{y}}{\sum_i x_i^2 - n\bar{x}^2} \\ a &= \bar{y} - b\bar{x} \;.\end{aligned} \tag{7.9}$$

Noting that $n\bar{y} = \sum_i y$ and $n\bar{x} = \sum_i x$, the results can be written as

$$\begin{aligned}b &= \frac{\sum_i (x_i - \bar{x})\, y_i}{\sum_i (x_i - \bar{x})\, x_i} \\ a &= \bar{y} - b\bar{x} \;.\end{aligned} \tag{7.10}$$

While Eqs. (7.9) and (7.10) are equivalent analytically, Eq. (7.10) is preferred for numerical calculations because Eq. (7.10) is less sensitive to roundoff errors. Here is a Python function implementing this algorithm:

```python
def LineFit(x, y):
    ''' Returns slope and y-intercept of linear fit to (x,y)
    data set'''
    xavg = x.mean()
    slope = (y*(x-xavg)).sum()/(x*(x-xavg)).sum()
    yint = y.mean()-slope*xavg
    return slope, yint
```

It's hard to imagine a simpler implementation of the linear regression algorithm.

7.3.2 Linear regression with weighting: χ^2

The linear regression routine of the previous section weights all data points equally. That is fine if the absolute uncertainty is the same for all data points. In many cases, however, the uncertainty is different for different points in a data set. In such cases, we would like to weight the data that has smaller uncertainty more heavily than those data that have greater uncertainty. For this case, there is a standard method of weighting and fitting data that is known as χ^2 (or *chi-squared*) fitting. In this method we suppose that associated with each (x_i, y_i) data point is an uncertainty in the value of y_i of $\pm\sigma_i$. In this case, the "best fit" is defined as the the one with the set of fitting parameters that minimizes the sum

$$\chi^2 = \sum_i \left(\frac{y_i - f(x_i)}{\sigma_i}\right)^2 \ . \tag{7.11}$$

Setting the uncertainties $\sigma_i = 1$ for all data points yields the same sum S we introduced in the previous section. In this case, all data points are weighted equally. However, if σ_i varies from point to point, it is clear that those points with large σ_i contribute less to the sum than those with small σ_i. Thus, data points with large σ_i are weighted less than those with small σ_i.

To fit data to a straight line, we set $f(x) = a + bx$ and write

$$\chi^2(a,b) = \sum_i \left(\frac{y_i - a - bx_i}{\sigma_i}\right)^2 \ . \tag{7.12}$$

Finding the minimum for $\chi^2(a,b)$ follows the same procedure used for finding the minimum of $S(a,b)$ in the previous section. The result is

$$\begin{aligned} b &= \frac{\sum_i (x_i - \hat{x})\, y_i/\sigma_i^2}{\sum_i (x_i - \hat{x})\, x_i/\sigma_i^2} \\ a &= \hat{y} - b\hat{x} \ . \end{aligned} \tag{7.13}$$

where

$$\begin{aligned}\hat{x} &= \frac{\sum_i x_i/\sigma_i^2}{\sum_i 1/\sigma_i^2} \\ \hat{y} &= \frac{\sum_i y_i/\sigma_i^2}{\sum_i 1/\sigma_i^2}\ .\end{aligned} \tag{7.14}$$

For a fit to a straight line, the overall quality of the fit can be measured by the reduced chi-squared parameter

$$\chi_r^2 = \frac{\chi^2}{n-2} \tag{7.15}$$

where χ^2 is given by Eq. (7.11) evaluated at the optimal values of a and b given by Eq. (7.13). A good fit is characterized by $\chi_r^2 \approx 1$. This makes sense because if the uncertainties σ_i have been properly estimated, then $[y_i - f(x_i)]^2$ should on average be roughly equal to σ_i^2, so that the sum in Eq. (7.11) should consist of n terms approximately equal to 1. Of course, if there were only 2 terms (*n=2*), then χ^2 would be zero as the best straight line fit to two points is a perfect fit. That is essentially why χ_r^2 is normalized using $n-2$ instead of n. If χ_r^2 is significantly greater than 1, this indicates a poor fit to the fitting function (or an underestimation of the uncertainties σ_i). If χ_r^2 is significantly less than 1, then it indicates that the uncertainties were probably overestimated (the fit and fitting function may or may not be good).

We can also get estimates of the uncertainties in our determination of the fitting parameters a and b, although deriving the formulas is a bit more involved that we want to get into here. Therefore, we just give the results:

$$\begin{aligned}\sigma_b^2 &= \frac{1}{\sum_i (x_i - \hat{x})\, x_i/\sigma_i^2} \\ \sigma_a^2 &= \sigma_b^2 \frac{\sum_i x_i^2/\sigma_i^2}{\sum_i 1/\sigma_i^2}\ .\end{aligned} \tag{7.16}$$

The estimates of uncertainties in the fitting parameters depend explicitly on $\{\sigma_i\}$ and will only be meaningful if (*i*) $\chi_r^2 \approx 1$ and (*ii*) the estimates of the uncertainties σ_i are accurate.

You can find more information, including a derivation of Eq. (7.16), in *Data Reduction and Error Analysis for the Physical Sciences, 3rd ed* by P. R. Bevington & D. K. Robinson, McGraw-Hill, New York, 2003.

7.4 Anonymous functions (lambda)

Python provides another way to generate functions called *lambda* expressions. A lambda expression is a kind of in-line function that can be generated on the fly to accomplish

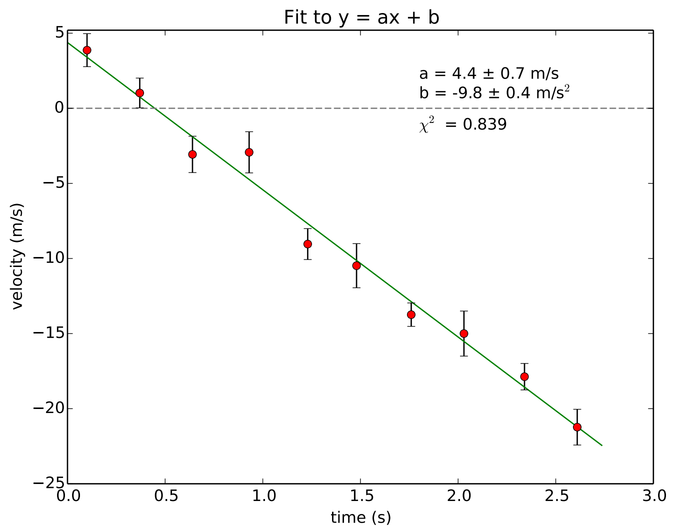


Figure 7.3: Fit using χ^2 least squares fitting routine with data weighted by error bars.

some small task. You can assign lambda functions a name, but you don't need to; hence, they are often called *anonymous* functions. A lambda uses the keyword `lambda` and has the general form

```
lambda arg1, arg2, ... : output
```

The arguments `arg1, arg2, ...` are inputs to a lambda, just as for a functions, and the output is an expression using the arguments.

While lambda expressions need not be named, we illustrate their use by comparing a conventional Python function definition to a lambda expression to which we give a name. First, we define a conventional python function

```
In [1]: def f(a, b):
   ...:     return 3*a+b**2

In [2]: f(2,3)
Out[2]: 15
```

Next, we define a lambda that does the same thing

```
In [3]: g = lambda a, b : 3*a+b**2

In [4]: g(2,3)
Out[4]: 15
```

The `lambda` defined by `g` does the same thing as the function `f`. Such `lambda` expressions are useful when you need a very short function definition, usually to be used locally only once or a few times.

Sometimes lambda expressions are used in function arguments that call for a function *name*, as opposed to the function itself. Moreover, in cases where a the function to be integrated is already defined but is a function one independent variable and several parameters, the lambda expression can be a convenient way of fashioning a single variable function. Don't worry if this doesn't quite make sense to you right now. You will see examples of lambda expressions used in just this way in the section *Numerical integration*.

There are also a number of nifty programming tricks that can be implemented using `lambda` expressions, but we will not go into them here. Look up `lambdas` on the web if you are curious about their more exotic uses.

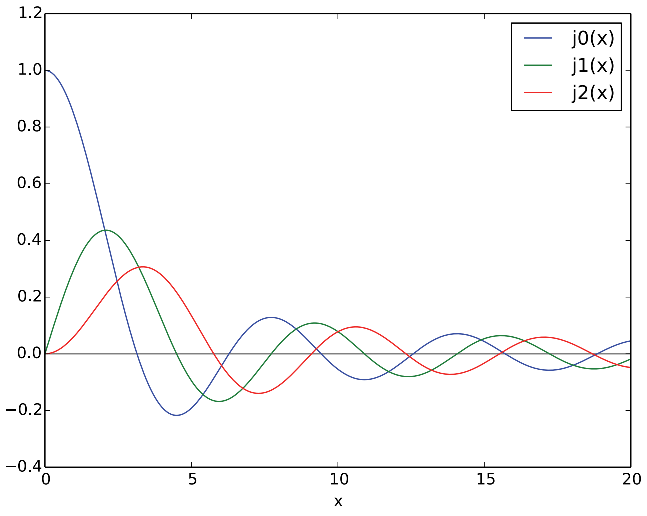
7.5 Exercises

1. Write a function that can return each of the first three spherical Bessel functions $j_n(x)$:

$$\begin{aligned} j_0(x) &= \frac{\sin x}{x} \\ j_1(x) &= \frac{\sin x}{x^2} - \frac{\cos x}{x} \\ j_2(x) &= \left(\frac{3}{x^2} - 1\right)\frac{\sin x}{x} - \frac{3\cos x}{x^2} \end{aligned} \tag{7.17}$$

   Your function should take as arguments a NumPy array x and the order n, and should return an array of the designated order n spherical Bessel function. Take care to make sure that your functions behave properly at $x = 0$.

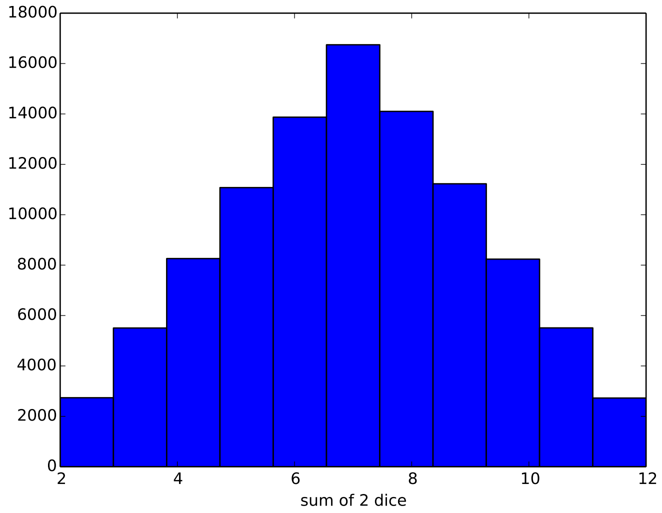
   Demonstrate the use of your function by writing a Python routine that plots the three Bessel functions for $0 \leq x \leq 20$. Your plot should look like the one below. Something to think about: You might note that $j_1(x)$ can be written in terms of $j_0(x)$, and that $j_2(x)$ can be written in terms of $j_1(x)$ and $j_0(x)$. Can you take advantage of this to write a more efficient function for the calculations of $j_1(x)$ and $j_2(x)$?



2. (a) Write a function that simulates the rolling of n dice. Use the NumPy function `random.random_integers(6)`, which generates a random integer between 1 and 6 with equal probability (like rolling fair dice). The input of your function should be the number of dice thrown each roll and the output should be the sum of the n dice.

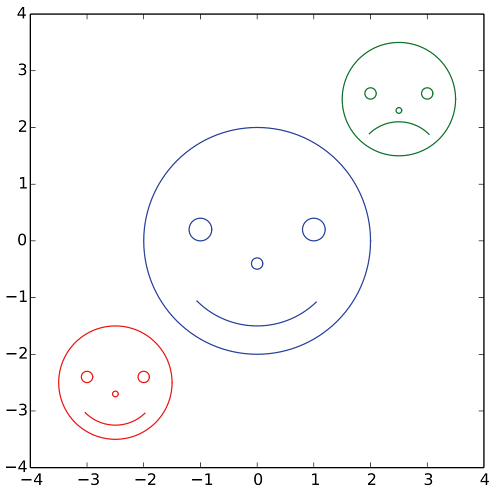
   (b) "Roll" 2 dice 10,000 times keeping track of all the sums of each set of rolls in a list. Then use your program to generate a histogram summarizing the rolls of two dice 10,000 times. The result should look like the histogram plotted below. Use the MatPlotLib function `hist` (see http://matplotlib.org/api/pyplot_summary.html) and set the number of bins in the histogram equal to the number of different possible outcomes of a roll of your dice. For example, the sum of two dice can be anything between 2 and 12, which corresponds to 11 possible outcomes. You should get a histogram that looks like the one below.

   (c) "Repeat part (b) using 3 dice and plot the resulting histogram.



3. Write a function to draw a circular smiley face with eyes, a nose, and a mouth. One argument should set the overall size of the face (the circle radius). Optional arguments should allow the user to specify the (x, y) position of the face, whether the face is smiling or frowning, and the color of the lines. The default should be a smiling blue face centered at $(0, 0)$. Once you write your function, write a program that calls it several times to produce a plot like the one below (cre-

ative improvisation is encouraged!). In producing your plot, you may find the call `plt.axes().set_aspect(1)` useful so that circles appear as circles and not ovals. You should only use MatPlotLib functions introduced in this text. To create a circle you can create an array of angles that goes from 0 to 2π and then produce the x and y arrays for your circle by taking the cosine and sine, respectively, of the array. Hint: You can use the same (x, y) arrays to make the smile and frown as you used to make the circle by plotting appropriate slices of those arrays. You do not need to create new arrays.



4. In the section *Example: linear least squares fitting*, we showed that the best fit of a line $y = a + bx$ to a set of data $\{(x_i, y_i)\}$ is obtained for the values of a and b given by Eq. (7.10). Those formulas were obtained by finding the values of a and b that minimized the sum in Eq. (7.5). This approach and these formulas are valid when the uncertainties in the data are the same for all data points. The Python function `LineFit(x, y)` in the section *Example: linear least squares fitting* implements Eq. (7.10).

   (a) Write a new fitting function `LineFitWt(x, y)` that implements the formulas given in Eq. (7.14) that minimize the χ^2 function give by Eq. (7.12). This more general approach is valid when the individual data points have different weightings *or* when they all have the same weighting. You should also write a function to calculate the reduced chi-squared χ_r^2 defined by Eq. (7.12).

(b) Write a Python program that reads in the data below, plots it, and fits it using the two fitting functions `LineFit(x, y)` and `LineFitWt(x, y)`. Your program should plot the data with error bars and with *both* fits with and without weighting, that is from `LineFit(x, y)` and `LineFitWt(x, y, dy)`. It should also report the results for both fits on the plot, similar to the output of the supplied program above, as well as the values of χ_r^2, the reduce chi-squared value, for both fits. Explain why weighting the data gives a steeper or less steep slope than the fit without weighting.

```
Velocity vs time data
for a falling mass
time (s)   velocity (m/s)   uncertainty (m/s)
  2.23          139                16
  4.78          123                16
  7.21          115                 4
  9.37           96                 9
 11.64           62                17
 14.23           54                17
 16.55           10                12
 18.70           -3                15
 21.05          -13                18
 23.21          -55                10
```

5. Modify the function `LineFitWt(x, y)` you wrote in Exercise 4 above so that in addition to returning the fitting parameters a and b, it also returns the uncertainties in the fitting parameters σ_a and σ_b using the formulas given by Eq. (7.16). Use your new fitting function to find the uncertainties in the fitted slope and y-intercept for the data provided with Exercise 4.

CHAPTER

EIGHT

CURVE FITTING

One of the most important tasks in any experimental science is modeling data and determining how well some theoretical function describes experimental data. In the last chapter, we illustrated how this can be done when the theoretical function is a simple straight line in the context of learning about Python functions and methods. Here we show how this can be done for a arbitrary fitting functions, including linear, exponential, power law, and other nonlinear fitting functions.

8.1 Using linear regression for fitting non-linear functions

We can use our results for linear regression with χ^2 weighting that we developed in Chapter 7 to fit functions that are nonlinear in the fitting parameters, *provided* we can transform the fitting function into one that is linear in the fitting parameters and in the independent variable (x).

8.1.1 Linear regression for fitting an exponential function

To illustrate this approach, let's consider some experimental data taken from a radioactive source that was emitting beta particles (electrons). We notice that the number of electrons emitted per unit time is decreasing with time. Theory suggests that the number of electrons N emitted per unit time should decay exponentially according to the equation

$$N(t) = N_0 e^{-t/\tau} \ . \tag{8.1}$$

This equation is nonlinear in t and in the fitting parameter τ and thus cannot be fit using the method of the previous chapter. Fortunately, this is a special case for which the fitting function can be transformed into a linear form. Doing so will allow us to use the fitting routine we developed for fitting linear functions.

We begin our analysis by transforming our fitting function to a linear form. To this end we take the logarithm of Eq. (8.1):

$$\ln N = \ln N_0 - \frac{t}{\tau} \ .$$

With this tranformation our fitting function is linear in the independent variable t. To make our method work, however, our fitting function must be linear in the *fitting parameters*, and our transformed function is still nonlinear in the fitting parameters τ and N_0. Therefore, we define new fitting parameters as follows

$$\begin{aligned} a &= \ln N_0 \\ b &= -1/\tau \end{aligned} \tag{8.2}$$

Now if we define a new dependent variable $y = \ln N$, then our fitting function takes the form of a fitting function that is linear in the fitting parameters a and b

$$y = a + bx$$

where the independent variable is $x = t$ and the dependent variable is $y = \ln N$.

We are almost ready to fit our transformed fitting function, with transformed fitting parameters a and b, to our transformed independent and dependent data, x and y. The last thing we have to do is to transform the estimates of the uncertainties δN in N to the uncertainties δy in y $(= \ln N)$. So how much does a given uncertainty in N translate into an uncertainty in y? In most cases, the uncertainty in y is much smaller than y, *i.e.* $\delta y \ll y$; similarly $\delta N \ll N$. In this limit we can use differentials to figure out the relationship between these uncertainties. Here is how it works for this example:

$$\begin{aligned} y &= \ln N \\ \delta y &= \left|\frac{\partial y}{\partial N}\right| \delta N \\ \delta y &= \frac{\delta N}{N} \ . \end{aligned} \tag{8.3}$$

Equation (8.3) tells us how a small change δN in N produces a small change δy in y. Here we identify the differentials dy and dN with the uncertainties δy and δN. Therefore, an uncertainty of δN in N corresponds, or translates, to an uncertainty δy in y.

Let's summarize what we have done so far. We started with the some data points $\{t_i, N_i\}$ and some addition data $\{\delta N_i\}$ where each datum δN_i corresponds to the uncertainty in the experimentally measured N_i. We wish to fit these data to the fitting function

$$N(t) = N_0 e^{-t/\tau} \ .$$

We then take the natural logarithm of both sides and obtain the linear equation

$$\begin{aligned} \ln N &= \ln N_0 - \frac{t}{\tau} \\ y &= a + bx \end{aligned} \tag{8.4}$$

with the obvious correspondences

$$\begin{aligned} x &= t \\ y &= \ln N \\ a &= \ln N_0 \\ b &= -1/\tau \end{aligned} \tag{8.5}$$

Now we can use the linear regression routine with χ^2 weighting that we developed in the previous section to fit (8.4) to the transformed data $x_i(=t_i)$ and $y_i(=\ln N_i)$. The inputs are the tranformed data $x_i, y_i, \delta y_i$. The outputs are the fitting parameters a and b, as well as the estimates of their uncertainties δa and δb along with the value of χ^2. You can obtain the values of the original fitting parameters N_0 and τ by taking the differentials of the last two equations in Eq. (8.5):

$$\begin{aligned} \delta a &= \left|\frac{\partial a}{\partial N_0}\right| \delta N_0 = \frac{\delta N_0}{N_0} \\ \delta b &= \left|\frac{\partial b}{\partial \tau}\right| \delta \tau = \frac{\delta \tau}{\tau^2} \end{aligned} \tag{8.6}$$

The Python routine below shows how to implement all of this for a set of experimental data that is read in from a data file.

Figure *8.1* shows the output of the fit to simulated beta decay data obtained using the program below. Note that the error bars are large when the number of counts N are small. This is consistent with what is known as *shot noise* (noise that arises from counting discrete events), which obeys *Poisson* statistics. You will study sources of noise, including shot noise, later in your lab courses. The program also prints out the fitting parameters of the transformed data as well as the fitting parameters for the exponential fitting function.

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

def LineFitWt(x, y, sig):
    """
    Fit to straight line.
    Inputs: x and y arrays and uncertainty array (unc) for y data.
    Ouputs: slope and y-intercept of best fit to data.
```

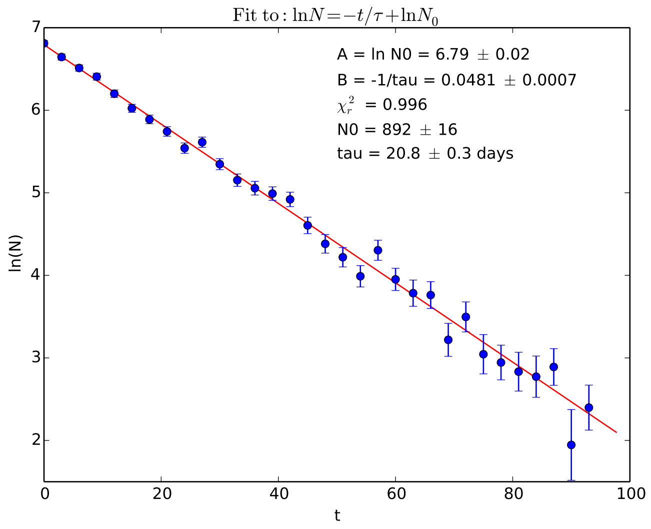


Figure 8.1: Semi-log plot of beta decay measurements from Phosphorus-32.

```
    """
    sig2 = sig**2
    norm = (1./sig2).sum()
    xhat = (x/sig2).sum() / norm
    yhat = (y/sig2).sum() / norm
    slope = ((x-xhat)*y/sig2).sum()/((x-xhat)*x/sig2).sum()
    yint = yhat - slope*xhat
    sig2_slope = 1./((x-xhat)*x/sig2).sum()
    sig2_yint = sig2_slope * (x*x/sig2).sum() / norm
    return slope, yint, np.sqrt(sig2_slope), np.sqrt(sig2_yint)

def redchisq(x, y, dy, slope, yint):
    chisq = (((y-yint-slope*x)/dy)**2).sum()
    return chisq/float(x.size-2)

# Read data from data file
t, N, dN = np.loadtxt("betaDecay.txt", skiprows=2, unpack=True)

########## Code to tranform & fit data starts here ##########

# Transform data and parameters to linear form: Y = A + B*X
```

```
X = t          # transform t data for fitting (trivial)
Y = np.log(N)  # transform N data for fitting
dY = dN/N      # transform uncertainties for fitting

# Fit transformed data X, Y, dY to obtain fitting parameters A & B
# Also returns uncertainties in A and B
B, A, dB, dA = LineFitWt(X, Y, dY)
# Return reduced chi-squared
redchisqr = redchisq(X, Y, dY, B, A)

# Determine fitting parameters for original exponential function
# N = N0 exp(-t/tau) ...
N0 = np.exp(A)
tau = -1.0/B
# ... and their uncertainties
dN0 = N0 * dA
dtau = tau**2 * dB

####### Code to plot transformed data and fit starts here #######

# Create line corresponding to fit using fitting parameters
# Only two points are needed to specify a straight line
Xext = 0.05*(X.max()-X.min())
Xfit = np.array([X.min()-Xext, X.max()+Xext])
Yfit = A + B*Xfit

plt.errorbar(X, Y, dY, fmt="bo")
plt.plot(Xfit, Yfit, "r-", zorder=-1)
plt.xlim(0, 100)
plt.ylim(1.5, 7)
plt.title("$\mathrm{Fit\\ to:}\\ \ln N = -t/\\tau + \ln N_0$")
plt.xlabel("t")
plt.ylabel("ln(N)")

plt.text(50, 6.6, "A = ln N0 = {0:0.2f} $\pm$ {1:0.2f}"
         .format(A, dA))
plt.text(50, 6.3, "B = -1/tau = {0:0.4f} $\pm$ {1:0.4f}"
         .format(-B, dB))
plt.text(50, 6.0, "$\chi_r^2$ = {0:0.3f}"
         .format(redchisqr))

plt.text(50, 5.7, "N0 = {0:0.0f} $\pm$ {1:0.0f}"
         .format(N0, dN0))
plt.text(50, 5.4, "tau = {0:0.1f} $\pm$ {1:0.1f} days"
         .format(tau, dtau))
```

```
plt.show()
```

8.1.2 Linear regression for fitting a power-law function

You can use a similar approach to the one outlined above to fit experimental data to a power law fitting function of the form

$$P(s) = P_0 s^\alpha \ . \tag{8.7}$$

We follow the same approach we used for the exponential fitting function and first take the logarithm of both sides of (8.7)

$$\ln P = \ln P_0 + \alpha \ln s \ . \tag{8.8}$$

We recast this in the form of a linear equation $y = a + bx$ with the following identifications:

$$\begin{aligned} x &= \ln s \\ y &= \ln P \\ a &= \ln P_0 \\ b &= \alpha \end{aligned} \tag{8.9}$$

Following a procedure similar to that used to fit using an exponential fitting function, you can use the tranformations given by (8.9) as the basis for a program to fit a power-law fitting function such as (8.8) to experimental data.

8.2 Nonlinear fitting

The method introduced in the previous section for fitting nonlinear fitting functions can be used only if the fitting function can be transformed into a fitting function that is linear in the fitting parameters a, b, c... When we have a nonlinear fitting function that cannot be transformed into a linear form, we need another approach.

The problem of finding values of the fitting parameters that minimize χ^2 is a nonlinear optimization problem to which there is quite generally no analytical solution (in contrast to the linear optimization problem). We can gain some insight into this nonlinear optimization problem, namely the fitting of a nonlinear fitting function to a data set, by considering a fitting function with only two fitting parameters. That is, we are trying to fit some data set $\{x_i, y_i\}$, with uncertainties in $\{y_i\}$ of $\{\sigma_i\}$, to a fitting function is $f(x; a, b)$ where a and b are the two fitting parameters. To do so, we look for the minimum in

$$\chi^2(a,b) = \sum_i \left(\frac{y_i - f(x_i)}{\sigma_i}\right)^2 \; .$$

Note that once the data set, uncertainties, and fitting function are specified, $\chi^2(a, b)$ is simply a function of a and b. We can picture the function $\chi^2(a, b)$ as a of landscape with peaks and valleys: as we vary a and b, $\chi^2(a, b)$ rises and falls. The basic idea of all nonlinear fitting routines is to start with some initial guesses for the fitting parameters, here a and b, and by scanning the $\chi^2(a, b)$ landscape, find values of a and b that minimize $\chi^2(a, b)$.

There are a number of different methods for trying to find the minimum in χ^2 for nonlinear fitting problems. Nevertheless, the method that is most widely used goes by the name of the *Levenberg-Marquardt* method. Actually the Levenberg-Marquardt method is a combination of two other methods, the *steepest descent* (or gradient) method and *parabolic extrapolation*. Roughly speaking, when the values of a and b are not too near their optimal values, the gradient descent method determines in which direction in (a, b)-space the function $\chi^2(a, b)$ decreases most quickly—the direction of steepest descent—and then changes a and b accordingly to move in that direction. This method is very efficient unless a and b are very near their optimal values. Near the optimal values of a and b, parabolic extrapolation is more efficient. Therefore, as a and b approach their optimal values, the Levenberg-Marquardt method gradually changes to the parabolic extrapolation method, which approximates $\chi^2(a, b)$ by a Taylor series second order in a and b and then computes directly the analytical minimum of the Taylor series approximation of $\chi^2(a, b)$. This method is only good if the second order Taylor series provides a good approximation of $\chi^2(a, b)$. That is why parabolic extrapolation only works well very near the minimum in $\chi^2(a, b)$.

Before illustrating the Levenberg-Marquardt method, we make one important cautionary remark: the Levenberg-Marquardt method can fail if the initial guesses of the fitting parameters are too far away from the desired solution. This problem becomes more serious the greater the number of fitting parameters. Thus it is important to provide reasonable initial guesses for the fitting parameters. Usually, this is not a problem, as it is clear from the physical situation of a particular experiment what reasonable values of the fitting parameters are. But beware!

The `scipy.optimize` module provides routines that implement the Levenberg-Marquardt non-linear fitting method. One is called `scipy.optimize.leastsq`. A somewhat more user-friendly version of the same method is accessed through another routine in the same `scipy.optimize` module: it's called `scipy.optimize.curve_fit` and it is the one we demonstrate here. The function call is

```
import scipy.optimize
[... insert code here ...]
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None,
                         **kwargs)
```

The arguments of `curve_fit` are

- `f(xdata, a, b, ...)`: is the fitting function where `xdata` is the data for the independent variable and `a, b, ...` are the fitting parameters, however many there are, listed as separate arguments. Obviously, `f(xdata, a, b, ...)` should return the y value of the fitting function.
- `xdata`: is the array containing the x data.
- `ydata`: is the array containing the y data.
- `p0`: is a tuple containing the initial guesses for the fitting parameters. The guesses for the fitting parameters are set equal to 1 if they are left unspecified. It is almost always a good idea to specify the initial guesses for the fitting parameters.
- `sigma`: is the array containing the uncertainties in the y data.
- `**kwargs`: are keyword arguments that can be passed to the fitting routine `scipy.optimize.leastsq` that `curve_fit` calls. These are usually left unspecified.

We demonstrate the use of `curve_fit` to fit the data plotted in the figure below:

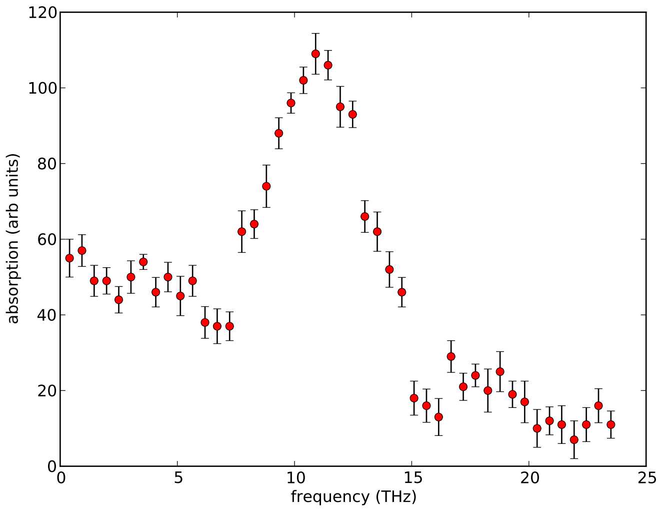
We model the data with the fitting function that consists of a quadratic polynomial background with a Gaussian peak:

$$A(f) = a + bf + cf^2 + Pe^{-\frac{1}{2}[(f-f_p)/f_w]^2}.$$

Lines 7 and 8 define the fitting functions. Note that the independent variable `f` is the first argument, which is followed by the six fitting parameters a, b, c, P, f_p, and f_w.

To fit the data with $A(f)$, we need good estimates of the fitting parameters. Setting $f = 0$, we see that $a \approx 60$. An estimate of the slope of the baseline gives $b \approx -60/20 = -3$. The curvature in the baseline is small so we take $c \approx 0$. The amplitude of the peak above the baseline is $P \approx 110 - 30 = 80$. The peak is centered at $f_p \approx 11$, while width of peak is about $f_w \approx 2$. We use these estimates to set the initial guesses of the fitting parameters in lines 14 and 15 in the code below.

The function that performs the Levenverg-Marquardt algorithm, *scipy.optimize.curve_fit*, is called in lines 19-20 with the output set equal to the one and two-dimensional arrays `nlfit` and `nlpcov`, respectively. The array `nlfit`, which gives the optimal values of



the fitting parameters, is unpacked in line 23. The square root of the diagonal of the two-dimensional array `nlpcov`, which gives the estimates of the uncertainties in the fitting parameters, is unpacked in lines 26-27 using a list comprehension.

The rest of the code plots the data, the fitting function using the optimal values of the fitting parameters found by `scipy.optimize.curve_fit`, and the values of the fitting parameters and their uncertainties.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import matplotlib.gridspec as gridspec  # for unequal plot boxes
4  import scipy.optimize
5  
6  # define fitting function
7  def GaussPolyBase(f, a, b, c, P, fp, fw):
8      return a + b*f + c*f*f + P*np.exp(-0.5*((f-fp)/fw)**2)
9  
10 # read in spectrum from data file
11 # f=frequency, s=signal, ds=s uncertainty
12 f, s, ds = np.loadtxt("Spectrum.txt", skiprows=4, unpack=True)
13 
14 # initial guesses for fitting parameters
15 a0, b0, c0 = 60., -3., 0.
16 P0, fp0, fw0 = 80., 11., 2.
```

```python

# fit data using SciPy's Levenberg-Marquart method
nlfit, nlpcov = scipy.optimize.curve_fit(GaussPolyBase,
                f, s, p0=[a0, b0, c0, P0, fp0, fw0], sigma=ds)

# unpack fitting parameters
a, b, c, P, fp, fw = nlfit
# unpack uncertainties in fitting parameters from diagonal
# of covariance matrix
da, db, dc, dP, dfp, dfw = \
          [np.sqrt(nlpcov[j,j]) for j in range(nlfit.size)]

# create fitting function from fitted parameters
f_fit = np.linspace(0.0, 25., 128)
s_fit = GaussPolyBase(f_fit, a, b, c, P, fp, fw)

# Calculate residuals and reduced chi squared
resids = s - GaussPolyBase(f, a, b, c, P, fp, fw)
redchisqr = ((resids/ds)**2).sum()/float(f.size-6)

# Create figure window to plot data
fig = plt.figure(1, figsize=(8,8))
gs = gridspec.GridSpec(2, 1, height_ratios=[6, 2])

# Top plot: data and fit
ax1 = fig.add_subplot(gs[0])
ax1.plot(f_fit, s_fit)
ax1.errorbar(f, s, yerr=ds, fmt='or', ecolor='black')
ax1.set_xlabel('frequency (THz)')
ax1.set_ylabel('absorption (arb units)')
ax1.text(0.7, 0.95, 'a = {0:0.1f}$\pm${1:0.1f}'
         .format(a, da), transform = ax1.transAxes)
ax1.text(0.7, 0.90, 'b = {0:0.2f}$\pm${1:0.2f}'
         .format(b, db), transform = ax1.transAxes)
ax1.text(0.7, 0.85, 'c = {0:0.2f}$\pm${1:0.2f}'
         .format(c, dc), transform = ax1.transAxes)
ax1.text(0.7, 0.80, 'P = {0:0.1f}$\pm${1:0.1f}'
         .format(P, dP), transform = ax1.transAxes)
ax1.text(0.7, 0.75, 'fp = {0:0.1f}$\pm${1:0.1f}'
         .format(fp, dfp), transform = ax1.transAxes)
ax1.text(0.7, 0.70, 'fw = {0:0.1f}$\pm${1:0.1f}'
         .format(fw, dfw), transform = ax1.transAxes)
ax1.text(0.7, 0.60, '$\chi_r^2$ = {0:0.2f}'
         .format(redchisqr),transform = ax1.transAxes)
ax1.set_title('$s(f) = a+bf+cf^2+P\,e^{-(f-f_p)^2/2f_w^2}$')
```

```
62
63  # Bottom plot: residuals
64  ax2 = fig.add_subplot(gs[1])
65  ax2.errorbar(f, resids, yerr = ds, ecolor="black", fmt="ro")
66  ax2.axhline(color="gray", zorder=-1)
67  ax2.set_xlabel('frequency (THz)')
68  ax2.set_ylabel('residuals')
69  ax2.set_ylim(-20, 20)
70  ax2.set_yticks((-20, 0, 20))
71
72  plt.show()
```

The above code also plots the difference between the data and fit, known as the *residuals* in the subplot below the plot of the data and fit. Plotting the residuals in this way gives a graphical representation of the goodness of the fit. To the extent that the residuals vary randomly about zero and do not show any overall upward or downward curvature, or any long wavelength oscillations, the fit would seem to be a good fit.

Finally, we note that we have used the MatPlotLib package `gridspec` to create the two subplots with different heights. The `gridspec` are made in lines 3 (where the package is imported), 36 (where 2 rows and 1 column are specified with relative heights of 6 to 2), 39 (where the first `gs[0]` height is specified), and 54 (where the second `gs[1]` height is specified). More details about the `gridspec` package can be found at the MatPlotLib web site.

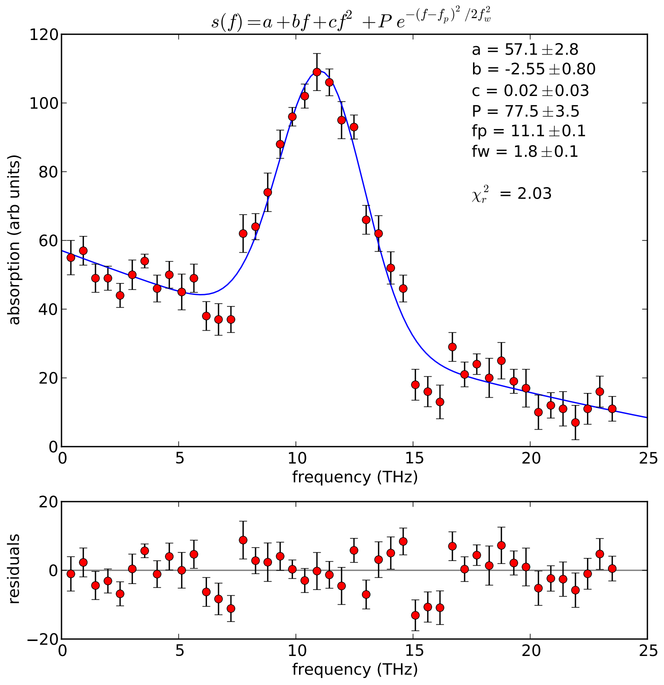


Figure 8.2: Fit to Gaussian with quadratic polynomial background.

8.3 Exercises

1. When a voltage source is connected across a resistor and inductor in series, the voltage across the inductor $V_i(t)$ is predicted to obey the equation

$$V(t) = V_0 e^{-\Gamma t} \tag{8.10}$$

where t is the time and the decay rate $\Gamma = R/L$ is the ratio of the resistance R to the inductance L of the circuit. In this problem, you are to write a Python routine that fits the above equation to the data below for the voltage measured across an inductor after it is connected in series with a resistor to a voltage source. Following the example in the text, linearize the (8.10) and use a linear fitting routine, either the one you wrote from the previous chapter or one from NumPy or SciPy.

   (a) Find the best values of Γ and V_0 and the uncertainties in their values σ_Γ and σ_{V_0}.

   (b) Find the value of χ_r^2 for your fit. Does it make sense?

   (c) Make a semi-log plot of the data using symbols with error bars (no line) and of the fit (line only). The fit should appear as a straight line that goes through the data points.

   (d) If the resistor has a value of 10.0 kΩ, what is the value of the inductance and its uncertainty according to your fit, assuming that the error in the resistance is negligibly small.

```
Data for decay of voltage across an inductor
in an RL circuit
Date: 24-Oct-2012
Data taken by D. M. Blantogg and T. P. Chaitor

time (ns)   voltage (volts)  uncertainty (volts)
    0.0        5.08e+00         1.12e-01
   32.8        3.29e+00         9.04e-02
   65.6        2.23e+00         7.43e-02
   98.4        1.48e+00         6.05e-02
  131.2        1.11e+00         5.25e-02
  164.0        6.44e-01         4.00e-02
  196.8        4.76e-01         3.43e-02
  229.6        2.73e-01         2.60e-02
  262.4        1.88e-01         2.16e-02
  295.2        1.41e-01         1.87e-02
  328.0        9.42e-02         1.53e-02
  360.8        7.68e-02         1.38e-02
```

```
    393.6          3.22e-02          8.94e-03
    426.4          3.22e-02          8.94e-03
    459.2          1.98e-02          7.01e-03
    492.0          1.98e-02          7.01e-03
```

2. Small nanoparticles of soot suspended in water start to aggregate when salt is added. The average radius r of the aggregates is predicted to grow as a power law in time t according to the equation $r = r_0 t^n$. Taking the logarithm of this equation gives $\ln r = n \ln t + \ln r_0$. Thus the data should fall on a straight line if $\ln r$ is plotted *vs* $\ln t$.

   (a) Plot the data below on a graph of $\ln r$ *vs* $\ln t$ to see if the data fall approximately on a straight line.

```
Size of growing aggregate
Date: 19-Nov-2013
Data taken by M. D. Gryart and M. L. Waites
time (m)   size (nm)    unc (nm)
  0.12        115          10
  0.18        130          12
  0.42        202          14
  0.90        335          18
  2.10        510          20
  6.00        890          30
 18.00       1700          40
 42.00       2600          50
```

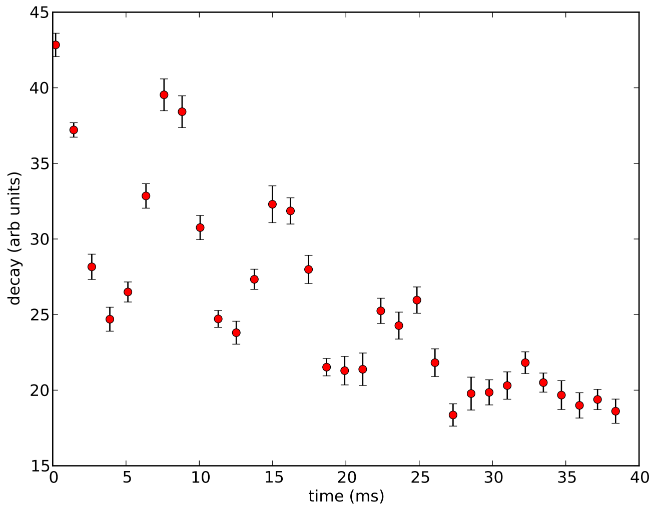
   (b) Defining $y = \ln r$ and $x = \ln t$, use the linear fitting routine you wrote for the previous problem to fit the data and find the optimal values for the slope and y intercept, as well as their uncertainties. Use these fitted values to find the optimal values of the the amplitude r_0 and the power n in the fitting function $r = r_0 t^n$. What are the fitted values of r_0 and n? What is the value of χ_r^2? Does a power law provide an adequate model for the data?

3. In this problem you explore using a non-linear least square fitting routine to fit the data shown in the figure below. The data, including the uncertainties in the y values, are provided in the table below. Your task is to fit the function

$$d(t) = A(1 + B\,\cos\omega t)e^{-t^2/2\tau^2} + C \tag{8.11}$$

   to the data, where the fitting parameters are A, B, C, ω, and τ.

   (a) Write a Python program that (*i*) reads the data in from a data file, (*ii*) defines a function `oscDecay(t, A, B, C, tau, omega)` for the function $d(t)$ above, and (*iii*) produces a plot of the data and the function $d(t)$. Choose the



fitting parameters `A`, `B`, `C`, `tau`, and `omega` to produce an approximate fit "by eye" to the data. You should be able estimate reasonable values for these parameters just by looking at the data and thinking about the behavior of $d(t)$. For example, $d(0) = A(1+B)+C$ while $d(\infty) = C$. What parameter in $d(t)$ controls the period of the peaks observed in the data? Use that information to estimate the value of that parameter.

(b) Following the example in section *Nonlinear fitting*, write a program using the SciPy function `scipy.optimize.curve_fit` to fit Eq. (8.11) to the data and thus find the optimal values of the fitting parameters A, B, C, ω, and τ. Your program should plot the data along with the fitting function using the optimal values of the fitting parameters. Write a function to calculate the reduced χ^2. Print out the value of the reduced χ^2 on your plot along with the optimal values of the fitting parameters. You can use the results from part (a) to estimate good starting values of the fitting parameters

(c) Once you have found the optimal fitting parameters, run your fitting program again using for starting values the optimal values of the fitting parameters A, B, C, and τ, but set the starting value of ω to be 3 times the optimal value. You should find that the program converges to a different set of fitting parameters than the ones you found in part (b). Using the program you wrote for part (b) make a plot of the data and the fit like the one you did for part (a). The fit should be noticeably worse. What is the value of the reduced χ^2 for this fit; it should be much larger than the one you found for part (c). The program has found a local minimum in χ^2—one that is obviously is not the best fit!

(d) Setting the fitting parameters A, B, C, and τ to the optimal values you found in part (b), plot χ_r^2 as a function of ω for ω spanning the range from 0.05 to 3.95. You should observe several local minima for different values of χ_r^2; the global minimum in χ_r^2 should occur for the optimal value of ω you found in part (b).

```
Data for absorption spectrum
Date: 21-Nov-2012
Data taken by P. Dubson and M. Sparks
time (ms)  signal  uncertainty
  0.2      41.1       0.9
  1.4      37.2       0.9
  2.7      28.3       0.9
  3.9      24.8       1.1
  5.1      27.8       0.8
  6.4      34.5       0.7
  7.6      39.0       0.9
  8.8      37.7       0.8
 10.1      29.8       0.9
 11.3      22.2       0.7
 12.5      22.3       0.6
 13.8      26.7       1.1
 15.0      30.4       0.7
 16.2      32.6       0.8
 17.5      28.9       0.8
 18.7      22.9       1.3
 19.9      21.7       0.9
 21.1      22.1       1.0
 22.4      22.3       1.0
 23.6      26.3       1.0
 24.8      26.2       0.8
 26.1      21.4       0.9
 27.3      20.0       1.0
 28.5      20.1       1.2
 29.8      21.2       0.5
 31.0      22.0       0.9
 32.2      21.6       0.7
 33.5      21.0       0.7
 34.7      19.7       0.9
 35.9      17.9       0.9
 37.2      18.1       0.8
 38.4      18.9       1.1
```

CHAPTER

NINE

NUMERICAL ROUTINES: SCIPY AND NUMPY

SciPy is a Python library of mathematical routines. Many of the SciPy routines are Python "wrappers", that is, Python routines that provide a Python interface for numerical libraries and routines originally written in Fortran, C, or C++. Thus, SciPy lets you take advantage of the decades of work that has gone into creating and optimizing numerical routines for science and engineering. Because the Fortran, C, or C++ code that Python accesses is compiled, these routines typically run very fast. Therefore, there is no real downside—no speed penalty—for using Python in these cases.

We have already encountered one of SciPy's routines, `scipy.optimize.leastsq`, for fitting nonlinear functions to experimental data, which was introduced in the the chapter on *Curve Fitting*. Here we will provide a further introduction to a number of other SciPy packages, in particular those on special functions, numerical integration, including routines for numerically solving ordinary differential equations (ODEs), discrete Fourier transforms, linear algebra, and solving non-linear equations. Our introduction to these capabilities does not include extensive background on the numerical methods employed; that is a topic for another text. Here we simply introduce the SciPy routines for performing some of the more frequently required numerical tasks.

One final note: SciPy makes extensive use of NumPy arrays, so NumPy should always be imported with SciPy

9.1 Special functions

SciPy provides a plethora of special functions, including Bessel functions (and routines for finding their zeros, derivatives, and integrals), error functions, the gamma function, Legendre, Laguerre, and Hermite polynomials (and other polynomial functions), Mathieu functions, many statistical functions, and a number of other functions. Most are contained in the `scipy.special` library, and each has its own special arguments and syntax, depending on the vagaries of the particular function. We demonstrate a number of them in the code below that produces a plot of the different functions called. For more information, you should consult the SciPy web site on the `scipy.special` library.

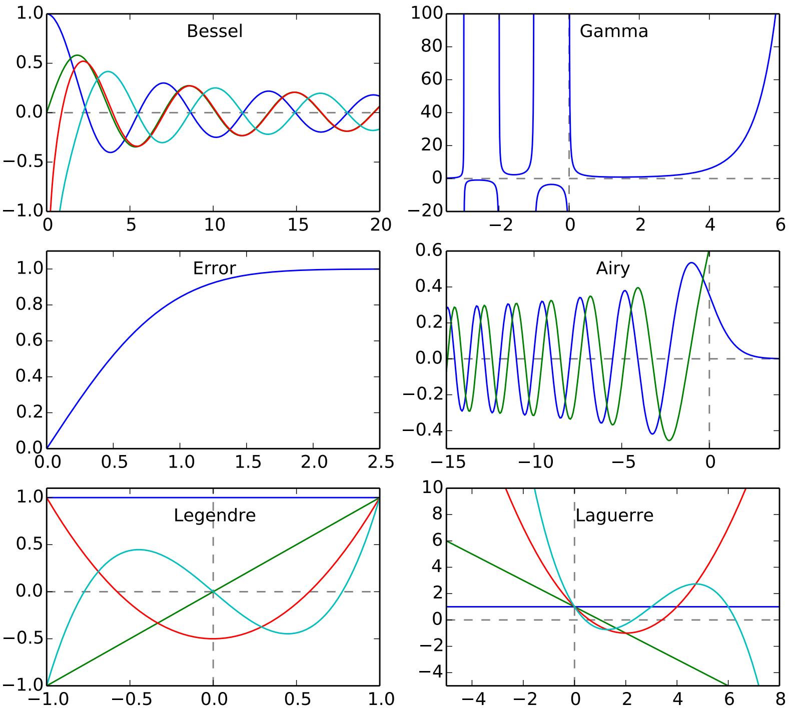


Figure 9.1: Plots of a few selected special functions

```python
import numpy as np
import scipy.special
import matplotlib.pyplot as plt

# create a figure window
fig = plt.figure(1, figsize=(9,8))
```

```python

# create arrays for a few Bessel functions and plot them
x = np.linspace(0, 20, 256)
j0 = scipy.special.jn(0, x)
j1 = scipy.special.jn(1, x)
y0 = scipy.special.yn(0, x)
y1 = scipy.special.yn(1, x)
ax1 = fig.add_subplot(321)
ax1.plot(x,j0, x,j1, x,y0, x,y1)
ax1.axhline(color="grey", ls="--", zorder=-1)
ax1.set_ylim(-1,1)
ax1.text(0.5, 0.95,'Bessel', ha='center', va='top',
     transform = ax1.transAxes)

# gamma function
x = np.linspace(-3.5, 6., 3601)
g = scipy.special.gamma(x)
g = np.ma.masked_outside(g, -100, 400)
ax2 = fig.add_subplot(322)
ax2.plot(x,g)
ax2.set_xlim(-3.5, 6)
ax2.axhline(color="grey", ls="--", zorder=-1)
ax2.axvline(color="grey", ls="--", zorder=-1)
ax2.set_ylim(-20, 100)
ax2.text(0.5, 0.95,'Gamma', ha='center', va='top',
     transform = ax2.transAxes)

# error function
x = np.linspace(0, 2.5, 256)
ef = scipy.special.erf(x)
ax3 = fig.add_subplot(323)
ax3.plot(x,ef)
ax3.set_ylim(0,1.1)
ax3.text(0.5, 0.95,'Error', ha='center', va='top',
     transform = ax3.transAxes)

# Airy function
x = np.linspace(-15, 4, 256)
ai, aip, bi, bip = scipy.special.airy(x)
ax4 = fig.add_subplot(324)
ax4.plot(x,ai, x,bi)
ax4.axhline(color="grey", ls="--", zorder=-1)
ax4.axvline(color="grey", ls="--", zorder=-1)
ax4.set_xlim(-15,4)
ax4.set_ylim(-0.5,0.6)
```

```
52      ax4.text(0.5, 0.95,'Airy', ha='center', va='top',
53           transform = ax4.transAxes)
54
55      # Legendre polynomials
56      x = np.linspace(-1, 1, 256)
57      lp0 = np.polyval(scipy.special.legendre(0),x)
58      lp1 = np.polyval(scipy.special.legendre(1),x)
59      lp2 = np.polyval(scipy.special.legendre(2),x)
60      lp3 = np.polyval(scipy.special.legendre(3),x)
61      ax5 = fig.add_subplot(325)
62      ax5.plot(x,lp0, x,lp1, x,lp2, x,lp3)
63      ax5.axhline(color="grey", ls="--", zorder=-1)
64      ax5.axvline(color="grey", ls="--", zorder=-1)
65      ax5.set_ylim(-1,1.1)
66      ax5.text(0.5, 0.9,'Legendre', ha='center', va='top',
67           transform = ax5.transAxes)
68
69      # Laguerre polynomials
70      x = np.linspace(-5, 8, 256)
71      lg0 = np.polyval(scipy.special.laguerre(0),x)
72      lg1 = np.polyval(scipy.special.laguerre(1),x)
73      lg2 = np.polyval(scipy.special.laguerre(2),x)
74      lg3 = np.polyval(scipy.special.laguerre(3),x)
75      ax6 = fig.add_subplot(326)
76      ax6.plot(x,lg0, x,lg1, x,lg2, x,lg3)
77      ax6.axhline(color="grey", ls="--", zorder=-1)
78      ax6.axvline(color="grey", ls="--", zorder=-1)
79      ax6.set_xlim(-5,8)
80      ax6.set_ylim(-5,10)
81      ax6.text(0.5, 0.9,'Laguerre', ha='center', va='top',
82           transform = ax6.transAxes)
83
84      plt.show()
```

The arguments of the different functions depend, of course, on the nature of the particular function. For example, the first argument of the two types of Bessel functions called in lines 10-13 is the so-called *order* of the Bessel function, and the second argument is the independent variable. The Gamma and Error functions take one argument each and produce one output. The Airy function takes only one input argument, but returns four outputs, which correspond the two Airy functions, normally designated $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$, and their derivatives $\mathrm{Ai}'(x)$ and $\mathrm{Bi}'(x)$. The plot shows only $\mathrm{Ai}(x)$ and $\mathrm{Bi}(x)$.

The polynomial functions shown have a special syntax that uses NumPy's `polyval` function for generating polynomials. If `p` is a list or array of `N` numbers and `x` is an array, then

```
polyval(p, x) = p[0]*x**(N-1) + p[1]*x**(N-2) + ... + p[N-2]*x +
                 p[N-1]
```

For example, if `p = [2.0, 5.0, 1.0]`, `polyval(p, x)` generates the following quadratic polynomial: $2x^2 + 5x + 1$.

SciPy's `special.legendre(n)` and `special.laguerre(n)` functions output the coefficients `p` needed in `polyval` to produce the $n^{\rm th}$-order Legendre and Laguerre polynomials, respectively. The `scipy.special` library has functions that specify many other polynomial functions in this same way.

9.2 Numerical integration

When a function cannot be integrated analytically, or is very difficult to integrate analytically, one generally turns to numerical integration methods. SciPy has a number of routines for performing numerical integration. Most of them are found in the same `scipy.integrate` library. We list them here for reference.

Function	**Description**
`quad`	single integration
`dblquad`	double integration
`tplquad`	triple integration
`nquad`	n-fold multiple integration
`fixed_quad`	Gaussian quadrature, order n
`quadrature`	Gaussian quadrature to tolerance
`romberg`	Romberg integration
`trapz`	trapezoidal rule
`cumtrapz`	trapezoidal rule to cumulatively compute integral
`simps`	Simpson's rule
`romb`	Romberg integration
`polyint`	Analytical polynomial integration (NumPy)
`poly1d`	Helper function for `polyint` (NumPy)

9.2.1 Single integrals

The function `quad` is the workhorse of SciPy's integration functions. Numerical integration is sometimes called *quadrature*, hence the name. It is normally the default choice for performing single integrals of a function $f(x)$ over a given fixed range from a to b

$$\int_a^b f(x)\,dx$$

The general form of quad is `scipy.integrate.quad(f, a, b)`, where f is the name of the function to be integrated and a and b are the lower and upper limits, respectively. The routine uses *adaptive quadrature* methods to numerically evaluate integrals, meaning it successively refines the subintervals (makes them smaller) until a desired level of numerical precision is achieved. For the quad routine, this is about 10^{-8}, although it usually does even better.

As an example, let's integrate a Gaussian function over the range from 0 to 1

$$\int_0^1 e^{-x^2}\,dx$$

We first need to define the function $f(x) = e^{-x^2}$, which we do using a lambda expression, and then we call the function quad to perform the integration.

```
In [1]: import scipy.integrate

In [2]: f = lambda x : exp(-x**2)

In [3]: scipy.integrate.quad(f, 0, 1)
Out[3]: (0.7468241328124271, 8.291413475940725e-15)
```

The function call `scipy.integrate.quad(f, 0, 1)` returns two numbers. The first is `0.7468...`, which is the value of the integral, and the second is `8.29...e-15`, which is an estimate of the absolute error in the value of the integral, which we see is quite small compared to `0.7468`.

Because quad requires a function *name* as its first argument, we can't simply use the expression `exp(-x**2)`. On the other hand, we could use the usual def statement to create a normal function, and then use the name of that function in quad. However, it's simpler here to use a lambda expression. In fact, we can just put the lambda expression directly into the first argument, as illustrated here

```
In [4]: scipy.integrate.quad(lambda x : exp(-x**2), 0, 1)
Out[4]: (0.7468241328124271, 8.291413475940725e-15)
```

That works too! Thus we see a lambda expression used as an *anonymous function*, a function with no name, as promised in the section *Anonymous functions (lambda)*.

Note: The quad function accepts positive and negative infinity as limits.

```
In [5]: scipy.integrate.quad(lambda x : exp(-x**2), 0, inf)
Out[5]: (0.8862269254527579, 7.101318390472462e-09)

In [6]: scipy.integrate.quad(lambda x : exp(-x**2), -inf, 1)
Out[6]: (1.6330510582651852, 3.669607414547701e-11)
```

The `quad` function handles infinite limits just fine. The absolute errors are somewhat larger but still well within acceptable bounds for practical work.

The `quad` function can integrate standard predefined NumPy functions of a single variable, like `exp`, `sin`, and `cos`.

```
In [7]: scipy.integrate.quad(exp, 0, 1)
Out[7]: (1.7182818284590453, 1.9076760487502457e-14)

In [8]: scipy.integrate.quad(sin, -0.5, 0.5)
Out[8]: (0.0, 2.707864644566304e-15)

In [9]: scipy.integrate.quad(cos, -0.5, 0.5)
Out[9]: (0.9588510772084061, 1.0645385431034061e-14)
```

Let's integrate the first order Bessel function of the first kind, usually denoted $J_1(x)$, over the interval from 0 to 5. Here is how we do it, using `scipy.special.jn(v,x)` where `v` is the (real) order of the Bessel function:

```
In [10]: import scipy.special

In [11]: scipy.integrate.quad(lambda x: scipy.special.jn(1,x),0,5)
Out[11]: (1.177596771314338, 1.8083362065765924e-14)
```

Because the SciPy function `scipy.special.jn(v, x)` is a function of two variables, `v` and `x`, we cannot use the function name `scipy.special.jn` in `quad`. So we use a `lambda` expression, which is a function of only one variable, `x`, because we have set the `v` argument equal to 1.

Integrating polynomials

Working in concert with the NumPy `poly1d`, the NumPy function `polyint` takes the n^{th} antiderivative of a polynomial and can be used to evaluate definite integrals. The function `poly1d` essentially does the same thing as `polyval` that we encountered in the section *Special functions*, but with a different syntax. Suppose we want to make the polynomial function $p(x) = 2x^2 + 5x + 1$. Then we write

```
In [12]: p = np.poly1d([2, 5, 1])

In [13]: p
Out[13]: poly1d([2, 5, 1])
```

The polynomial $p(x) = 2x^2 + 5x + 1$ is evaluated using the syntax `p(x)`. Below, we evaluate the polynomial at three different values of `x`.

```
In [14]: p(1), p(2), p(3.5)
Out[14]: (8, 19, 43.0)
```

Thus `polyval` allows us to define the function $p(x) = 2x^2 + 5x + 1$. Now the antiderivative of $p(x) = 2x^2 + 5x + 1$ is $P(x) = \frac{2}{3}x^3 + \frac{5}{2}x^2 + x + C$ where C is the integration constant. The NumPy function `polyint`, which takes the n^{th} antiderivative of a polynomial, works as follows

```
In [15]: P = polyint(p)

In [16]: P
Out[16]: poly1d([ 0.66666667,  2.5       ,  1.        ,  0.        ])
```

When `polyint` has a single input, `p` is this case, `polyint` returns the coefficients of the antiderivative with the integration constant set to zero, as `Out[16]` illustrates. It is then an easy matter to determine any definite integral of the polynomial $p(x) = 2x^2 + 5x + 1$ since

$$q \equiv \int_a^b p(x)\, dx = P(b) - P(a)\ .$$

For example, if $a = 1$ and $b = 5$,

```
In [17]: q=P(5)-P(1)

In [18]: q
Out[18]: 146.66666666666666
```

or

$$\int_1^5 \left(2x^2 + 5x + 1\right)\, dx = 146\tfrac{2}{3}\ .$$

9.2.2 Double integrals

The `scipy.integrate` function `dblquad` can be used to numerically evaluate double integrals of the form

$$\int_{y=a}^{y=b} dy \int_{x=g(y)}^{x=h(y)} dx\, f(x,y)$$

The general form of `dblquad` is

```
scipy.integrate.dblquad(func, a, b, gfun, hfun)
```

where `func` if the name of the function to be integrated, `a` and `b` are the lower and upper limits of the `x` variable, respectively, and `gfun` and `hfun` are the *names* of the functions that define the lower and upper limits of the `y` variable.

As an example, let's perform the double integral

$$\int_0^{1/2} dy \int_0^{\sqrt{1-4y^2}} 16xy\, dx$$

We define the functions *f*, *g*, and *h*, using lambda expressions. Note that even if *g*, and *h* are constants, as they may be in many cases, they must be defined as functions, as we have done here for the lower limit.

```
In [19]: f = lambda x, y : 16*x*y

In [20]: g = lambda x : 0

In [21]: h = lambda y : sqrt(1-4*y**2)

In [22]: scipy.integrate.dblquad(f, 0, 0.5, g, h)
Out[22]: (0.5, 5.551115123125783e-15)
```

Once again, there are two outputs: the first is the value of the integral and the second is its absolute uncertainty.

Of course, the lower limit can also be a function of y, as we demonstrate here by performing the integral

$$\int_0^{1/2} dy \int_{1-2y}^{\sqrt{1-4y^2}} 16xy\, dx$$

The code for this is given by

```
In [23]: g = lambda y : 1-2*y

In [24]: scipy.integrate.dblquad(f, 0, 0.5, g, h)
Out[24]: (0.33333333333333326, 3.700743415417188e-15)
```

Other integration routines

In addition to the routines described above, `scipy.integrate` has a number of other integration routines, including `nquad`, which performs n-fold multiple integration, as well as other routines that implement other integration algorithms. You will find, however, that `quad` and `dblquad` meet most of your needs for numerical integration.

9.3 Solving ODEs

The `scipy.integrate` library has two powerful powerful routines, `ode` and `odeint`, for numerically solving systems of coupled first order ordinary differential equations (ODEs). While `ode` is more versatile, `odeint` (ODE integrator) has a simpler Python interface works very well for most problems. It can handle both stiff and non-stiff problems. Here we provide an introduction to `odeint`.

A typical problem is to solve a second or higher order ODE for a given set of initial conditions. Here we illustrate using `odeint` to solve the equation for a driven damped pendulum. The equation of motion for the angle θ that the pendulum makes with the vertical is given by

$$\frac{d^2\theta}{dt^2} = -\frac{1}{Q}\frac{d\theta}{dt} + \sin\theta + d\cos\Omega t$$

where t is time, Q is the quality factor, d is the forcing amplitude, and Ω is the driving frequency of the forcing. Reduced variables have been used such that the natural (angular) frequency of oscillation is 1. The ODE is nonlinear owing to the $\sin\theta$ term. Of course, it's precisely because there are no general methods for solving nonlinear ODEs that one employs numerical techniques, so it seems appropriate that we illustrate the method with a nonlinear ODE.

The first step is always to transform any n^{th}-order ODE into a system of n first order

ODEs of the form:

$$\begin{aligned}
\frac{dy_1}{dt} &= f_1(t, y_1, ..., y_n) \\
\frac{dy_2}{dt} &= f_2(t, y_1, ..., y_n) \\
\vdots &= \vdots \\
\frac{dy_n}{dt} &= f_n(t, y_1, ..., y_n) \ .
\end{aligned}$$

We also need n initial conditions, one for each variable y_i. Here we have a second order ODE so we will have two coupled ODEs and two initial conditions.

We start by transforming our second order ODE into two coupled first order ODEs. The transformation is easily accomplished by defining a new variable $\omega \equiv d\theta/dt$. With this definition, we can rewrite our second order ODE as two coupled first order ODEs:

$$\begin{aligned}
\frac{d\theta}{dt} &= \omega \\
\frac{d\omega}{dt} &= -\frac{1}{Q}\,\omega + \sin\theta + d\cos\Omega t \ .
\end{aligned}$$

In this case the functions on the right hand side of the equations are

$$\begin{aligned}
f_1(t, \theta, \omega) &= \omega \\
f_2(t, \theta, \omega) &= -\frac{1}{Q}\,\omega + \sin\theta + d\cos\Omega t \ .
\end{aligned}$$

Note that there are no explicit derivatives on the right hand side of the functions f_i; they are all functions of t and the various y_i, in this case θ and ω.

The initial conditions specify the values of θ and ω at $t = 0$.

SciPy's ODE solver `scipy.integrate.odeint` has three required arguments and many optional keyword arguments, of which we only need one, `args`, for this example. So in this case, `odeint` has the form

```
odeint(func, y0, t, args=())
```

The first argument `func` is the name of a Python function that returns a list of values of the n functions $f_i(t, y_1, ..., y_n)$ at a given time t. The second argument `y0` is an array (or list) of the values of the initial conditions of $y_1, ..., y_n$). The third argument is the array of times at which you want `odeint` to return the values of $y_1, ..., y_n$). The keyword argument `args` is a tuple that is used to pass parameters (besides `y0` and `t`) that are needed to evaluate `func`. Our example should make all of this clear.

After having written the n^{th}-order ODE as a system of n first-order ODEs, the next task is to write the function `func`. The function `func` should have three arguments: (1) the list (or array) of current `y` values, the current time `t`, and a list of any other parameters `params` needed to evaluate `func`. The function `func` returns the values of the derivatives $dy_i/dt = f_i(t, y_1, ..., y_n)$ in a list (or array). Lines 5-11 illustrate how to write `func` for our example of a driven damped pendulum. Here we name the function simply `f`, which is the name that appears in the call to `odeint` in line 33 below.

The only other tasks remaining are to define the parameters needed in the function, bundling them into a list (see line 22 below), and to define the initial conditions, and bundling them into another list (see line 25 below). After defining the time array in lines 28-30, the only remaining task is to call `odeint` with the appropriate arguments and a variable, `psoln` in this case to store output. The output `psoln` is an n element array where each element is itself an array corresponding the the values of y_i for each time in the time `t` array that was an argument of `odeint`. For this example, the first element `psoln[:,0]` is the y_0 or `theta` array, and the second element `psoln[:,1]` is the y_1 or `omega` array. The remainder of the code simply plots out the results in different formats. The resulting plots are shown in the figure *Pendulum trajectory* after the code.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from scipy.integrate import odeint
4
5  def f(y, t, params):
6      theta, omega = y      # unpack current values of y
7      Q, d, Omega = params  # unpack parameters
8      derivs = [omega,      # list of dy/dt=f functions
9                -omega/Q + np.sin(theta) + d*np.cos(Omega*t)]
10     return derivs
11
12 # Parameters
13 Q = 2.0          # quality factor (inverse damping)
14 d = 1.5          # forcing amplitude
15 Omega = 0.65     # drive frequency
16
17 # Initial values
18 theta0 = 0.0     # initial angular displacement
19 omega0 = 0.0     # initial angular velocity
20
21 # Bundle parameters for ODE solver
22 params = [Q, d, Omega]
23
24 # Bundle initial conditions for ODE solver
25 y0 = [theta0, omega0]
```

```

# Make time array for solution
tStop = 200.
tInc = 0.05
t = np.arange(0., tStop, tInc)

# Call the ODE solver
psoln = odeint(f, y0, t, args=(params,))

# Plot results
fig = plt.figure(1, figsize=(8,8))

# Plot theta as a function of time
ax1 = fig.add_subplot(311)
ax1.plot(t, psoln[:,0])
ax1.set_xlabel('time')
ax1.set_ylabel('theta')

# Plot omega as a function of time
ax2 = fig.add_subplot(312)
ax2.plot(t, psoln[:,1])
ax2.set_xlabel('time')
ax2.set_ylabel('omega')

# Plot omega vs theta
ax3 = fig.add_subplot(313)
twopi = 2.0*np.pi
ax3.plot(psoln[:,0]%twopi, psoln[:,1], '.', ms=1)
ax3.set_xlabel('theta')
ax3.set_ylabel('omega')
ax3.set_xlim(0., twopi)

plt.tight_layout()
plt.show()
```

The plots above reveal that for the particular set of input parameters chosen, `Q = 2.0`, `d = 1.5`, and `Omega = 0.65`, the pendulum trajectories are chaotic. Weaker forcing (smaller d) leads to what is perhaps the more familiar behavior of sinusoidal oscillations with a fixed frequency which, at long times, is equal to the driving frequency.

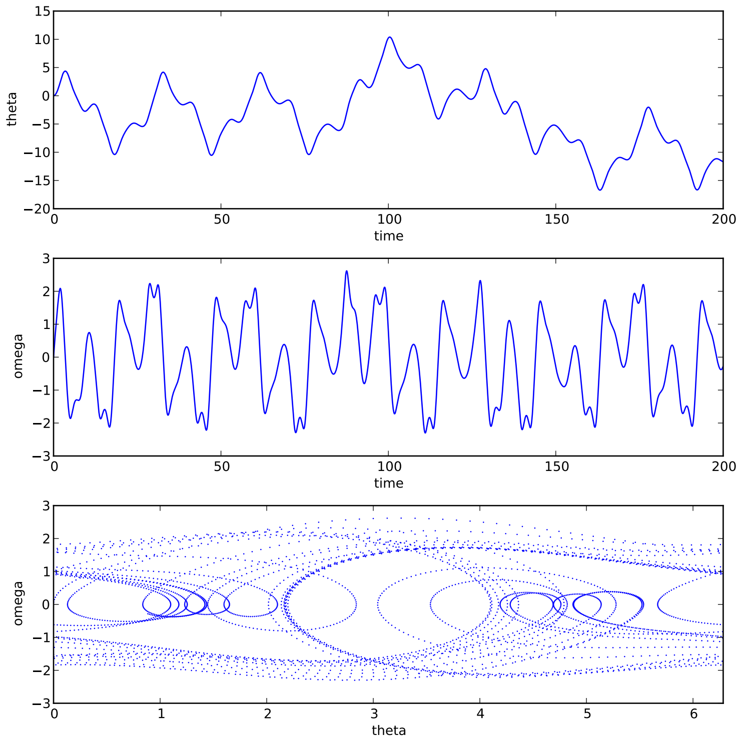


Figure 9.2: Pendulum trajectory

9.4 Discrete (fast) Fourier transforms

The SciPy library has a number of routines for performing discrete Fourier transforms. Before delving into them, we provide a brief review of Fourier transforms and discrete Fourier transforms.

9.4.1 Continuous and discrete Fourier transforms

The Fourier transform of a function $g(t)$ is given by

$$G(f) = \int_{-\infty}^{\infty} g(t)\, e^{-i\,2\pi f t}\, dt \ , \tag{9.1}$$

where f is the Fourier transform variable; if t is time, then f is frequency. The inverse transform is given by

$$g(t) = \int_{-\infty}^{\infty} G(f)\, e^{i\,2\pi f t}\, df \tag{9.2}$$

Here we define the Fourier transform in terms of the frequency f rather than the angular frequency $\omega = 2\pi f$.

The conventional Fourier transform is defined for continuous functions, or at least for functions that are dense and thus have an infinite number of data points. When doing numerical analysis, however, you work with *discrete* data sets, that is, data sets defined for a finite number of points. The discrete Fourier transform (DFT) is defined for a function g_n consisting of a set of N discrete data points. Those N data points must be defined at *equally-spaced* times $t_n = n\Delta t$ where Δt is the time between successive data points and n runs from 0 to $N-1$. The discrete Fourier transform (DFT) of g_n is defined as

$$G_l = \sum_{n=0}^{N-1} g_n\, e^{-i\,(2\pi/N)\,ln} \tag{9.3}$$

where l runs from 0 to $N-1$. The inverse discrete Fourier transform (iDFT) is defined as

$$g_n = \frac{1}{N} \sum_{l=0}^{N-1} G_l\, e^{i\,(2\pi/N)\,ln} \ . \tag{9.4}$$

The DFT is usually implemented on computers using the well-known Fast Fourier Transform (FFT) algorithm, generally credited to Cooley and Tukey who developed it at AT&T Bell Laboratories during the 1960s. But their algorithm is essentially one of many independent rediscoveries of the basic algorithm dating back to Gauss who described it as early as 1805.

9.4.2 The SciPy FFT library

The SciPy library `scipy.fftpack` has routines that implement a souped-up version of the FFT algorithm along with many ancillary routines that support working with DFTs. The basic FFT routine in `scipy.fftpack` is appropriately named `fft`. The program below illustrates its use, along with the plots that follow.

```python
import numpy as np
from scipy import fftpack
import matplotlib.pyplot as plt

width = 2.0
freq = 0.5

t = np.linspace(-10, 10, 101)    # linearly space time array
g = np.exp(-np.abs(t)/width) * np.sin(2.0*np.pi*freq*t)

dt = t[1]-t[0]        # increment between times in time array

G = fftpack.fft(g)    # FFT of g
f = fftpack.fftfreq(g.size, d=dt) # frequenies f[i] of g[i]
f = fftpack.fftshift(f)     # shift frequencies from min to max
G = fftpack.fftshift(G)     # shift G order to coorespond to f

fig = plt.figure(1, figsize=(8,6), frameon=False)
ax1 = fig.add_subplot(211)
ax1.plot(t, g)
ax1.set_xlabel('t')
ax1.set_ylabel('g(t)')

ax2 = fig.add_subplot(212)
ax2.plot(f, np.real(G), color='dodgerblue', label='real part')
ax2.plot(f, np.imag(G), color='coral', label='imaginary part')
ax2.legend()
ax2.set_xlabel('f')
ax2.set_ylabel('G(f)')

plt.show()
```

The DFT has real and imaginary parts, both of which are plotted in the figure.

The `fft` function returns the N Fourier components of G_n starting with the zero-frequency component G_0 and progressing to the maximum positive frequency component $G_{(N/2)-1}$ (or $G_{(N-1)/2}$ if N is odd). From there, `fft` returns the maximum *negative* component $G_{N/2}$ (or $G_{(N-1)/2}$ if N is odd) and continues upward in frequency

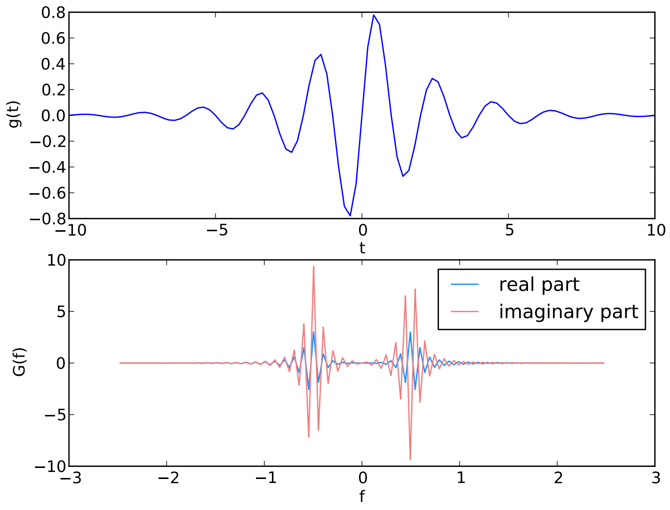


Figure 9.3: Function $g(t)$ and its DFT $G(f)$.

until it reaches the minimum negative frequency component G_{N-1}. This is the standard way that DFTs are ordered by most numerical DFT packages. The `scipy.fftpack` function `fftfreq` creates the array of frequencies in this non-intuitive order such that `f[n]` in the above routine is the correct frequency for the Fourier component `G[n]`. The arguments of `fftfreq` are the size of the the orignal array `g` and the keyword argument `d` that is the spacing between the (equally spaced) elements of the time array (`d=1` if left unspecified). The package `scipy.fftpack` provides the convenience function `fftshift` that reorders the frequency array so that the zero-frequency occurs at the middle of the array, that is, so the frequencies proceed monotonically from smallest (most negative) to largest (most positive). Applying `fftshift` to both `f` and `G` puts the frequencies `f` in ascending order and shifts `G` so that the frequency of `G[n]` is given by the shifted `f[n]`.

The `scipy.fftpack` module also contains routines for performing 2-dimensional and n-dimensional DFTs, named `fft2` and `fftn`, respectively, using the FFT algorithm.

As for most FFT routines, the `scipy.fftpack` FFT routines are most efficient if N is a power of 2. Nevertheless, the FFT routines are able to handle data sets where N is not a power of 2.

`scipy.fftpack` also supplies an inverse DFT function `ifft`. It is written to act on the *unshifted* FFT so take care! Note also that `ifft` returns a *complex* array. Because of machine roundoff error, the imaginary part of the function returned by `ifft` will, in general, be very near zero but not exactly zero even when the original function is a purely real function.

9.5 Linear algebra

Python's mathematical libraries, NumPy and SciPy, have extensive tools for numerically solving problems in linear algebra. Here we focus on two problems that arise commonly in scientific and engineering settings: (1) solving a system of linear equations and (2) eigenvalue problems. In addition, we also show how to perform a number of other basic computations, such as finding the determinant of a matrix, matrix inversion, and *LU* decomposition. The SciPy package for linear algebra is called `scipy.linalg`.

9.5.1 Basic computations in linear algebra

SciPy has a number of routines for performing basic operations with matrices. The determinant of a matrix is computed using the `scipy.linalg.det` function:

```
In [1]: import scipy.linalg
In [2]: a = array([[-2, 3], [4, 5]])
In [3]: a
Out[4]: array([[-2,  3],
               [ 4,  5]])

In [5]: scipy.linalg.det(a)
Out[5]: -22.0
```

The inverse of a matrix is computed using the `scipy.linalg.inv` function, while the product of two matrices is calculated using the NumPy `dot` function:

```
In [6]: b = scipy.linalg.inv(a)

In [6]: b
Out[6]: array([[-0.22727273,  0.13636364],
               [ 0.18181818,  0.09090909]])

In [7]: dot(a,b)
Out[7]: array([[ 1.,  0.],
               [ 0.,  1.]])
```

9.5.2 Solving systems of linear equations

Solving systems of equations is nearly as simple as constructing a coefficient matrix and a column vector. Suppose you have the following system of linear equations to solve:

$$\begin{aligned}2x_1 + 4x_2 + 6x_3 &= 4\\ x_1 - 3x_2 - 9x_3 &= -11\\ 8x_1 + 5x_2 - 7x_3 &= 1\end{aligned}$$

The first task is to recast this set of equations as a matrix equation of the form $\mathsf{A}\,\mathbf{x} = \mathbf{b}$. In this case, we have:

$$\mathsf{A} = \begin{pmatrix} 2 & 4 & 6\\ 1 & -3 & -9\\ 8 & 5 & -7\end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1\\ x_2\\ x_3\end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 4\\ -11\\ 1\end{pmatrix}.$$

Next we construct the array A and vector $\mathbf{b}$ as NumPy arrays:

```
In [8]: A = array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
In [9]: b = array([4, -11, 2])
```

Finally we use the SciPy function `scipy.linalg.solve` to find x_1, x_2, and x_3.

```
In [10]: scipy.linalg.solve(A,b)
Out[10]: array([ -8.91304348,  10.2173913 ,  -3.17391304])
```

which gives the results: $x_1 = -8.91304348$, $x_2 = 10.2173913$, and $x_3 = -3.17391304$. Of course, you can get the same answer by noting that $\mathbf{x} = \mathsf{A}^{-1}\mathbf{b}$. Following this approach, we can use the *scipy.linalg.inv* introduced in the previous section:

```
Ainv = scipy.linalg.inv(A)
```

```
In [10]: dot(Ainv, b)
Out[10]: array([ -8.91304348,  10.2173913 ,  -3.17391304])
```

which is the same answer we obtained using `scipy.linalg.solve`. Using `scipy.linalg.solve` is numerically more stable and a faster than using $\mathbf{x} = \mathsf{A}^{-1}\mathbf{b}$, so it is the preferred method for solving systems of equations.

You might wonder what happens if the system of equations are not all linearly independent. For example if the matrix A is given by

$$\mathsf{A} = \begin{pmatrix} 2 & 4 & 6\\ 1 & -3 & -9\\ 1 & 2 & 3\end{pmatrix}$$

where the third row is a multiple of the first row. Let's try it out and see what happens. First we change the bottom row of the matrix A and then try to solve the system as we did before.

```
In [11]: A[2] = array([1, 2, 3])

In [12]: A
Out[12]: array([[ 2,  4,  6],
                [ 1, -3, -9],
                [ 1,  2,  3]])

In [13]: scipy.linalg.solve(A,b)
LinAlgError: Singular matrix

In [14]: Ainv = scipy.linalg.inv(A)
LinAlgError: Singular matrix
```

Whether we use `scipy.linalg.solve` or `scipy.linalg.inv`, SciPy raises an error because the matrix is singular.

9.5.3 Eigenvalue problems

One of the most common problems in science and engineering is the eigenvalue problem, which in matrix form is written as

$$\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$$

where A is a square matrix, $\mathbf{x}$ is a column vector, and λ is a scalar (number). Given the matrix A, the problem is to find the set of eigenvectors $\mathbf{x}$ and their corresponding eigenvalues λ that solve this equation.

We can solve eigenvalue equations like this using `scipy.linalg.eig`. the outputs of this function is an array whose entries are the eigenvalues and a matrix whose rows are the eigenvectors. Let's return to the matrix we were using previously and find its eigenvalues and eigenvectors.

```
A = array([[2, 4, 6],[1, -3, -9],[8, 5, -7]])

In [15]: A
Out[15]: array([[ 2,  4,  6],
                [ 1, -3, -9],
                [ 8,  5, -7]])

In [16]: lam, evec = scipy.linalg.eig(A)
In [17]: lam
Out[17]: array([ 2.40995356+0.j, -8.03416016+0.j,
                -2.37579340+0.j])

In [18]: evec
Out[18]: array([[-0.77167559, -0.52633654,  0.57513303],
                [ 0.50360249,  0.76565448, -0.80920669],
                [-0.38846018,  0.36978786,  0.12002724]])
```

The first eigenvalue and its corresponding eigenvector are given by

```
In [19]: lam[0]
Out[19]: (2.4099535647625494+0j)

In [20]: evec[:,0]
Out[20]: array([-0.77167559,  0.50360249, -0.38846018])
```

We can check that they satisfy the $\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$:

```
In [21]: dot(A,evec[:,0])
Out[21]: array([-1.85970234,  1.21365861, -0.93617101])

In [22]: lam[0]*evec[:,0]
Out[22]: array([-1.85970234+0.j,  1.21365861+0.j,
                -0.93617101+0.j])
```

Thus we see by direct substitution that the left and right sides of $\mathsf{A}\mathbf{x} = \lambda\mathbf{x}$: are equal. In general, the eigenvalues can be complex, so their values are reported as complex numbers.

Generalized eigenvalue problem

The `scipy.linalg.eig` function can also solve the *generalized* eigenvalue problem

$$\mathsf{A}\mathbf{x} = \lambda\mathsf{B}\mathbf{x}$$

where B is a square matrix with the same size as A. Suppose, for example, that we have

```
In [22]: A = array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
Out[22]: B = array([[5, 9, 1], [-3, 1, 6], [4, 2, 8]])
```

Then we can solve the generalized eigenvalue problem by entering B as the optional second argument to `scipy.linalg.eig`

```
In [23]: lam, evec = scipy.linalg.eig(A,B)
```

The solutions are returned in the same fashion as before, as an array `lam` whose entries are the eigenvalues and a matrix `evac` whose rows are the eigenvectors.

```
In [24]: lam
Out[24]: array([-1.36087907+0.j,  0.83252442+0.j,
                -0.10099858+0.j])

In [25]: evec
Out[25]: array([[-0.0419907 , -1.        ,  0.93037493],
                [-0.43028153,  0.17751302, -1.        ],
                [ 1.        , -0.29852465,  0.4226201 ]])
```

Hermitian and banded matrices

SciPy has a specialized routine for solving eigenvalue problems for Hermitian (or real symmetric) matrices. The routine for hermitian matrices is `scipy.linalg.eigh`. It is more efficient (faster and uses less memory) than `scipy.linalg.eig`. The basic syntax of the two routines is the same, although some of the *optional* arguments are different. Both routines can solve generalized as well as standard eigenvalue problems.

SciPy also has a specialized routine `scipy.linalg.eig_banded` for solving eigenvalue problems for real symmetric or complex hermitian banded matrices.

9.6 Solving non-linear equations

SciPy has many different routines for numerically solving non-linear equations or systems of non-linear equations. Here we will introduce only a few of these routines, the ones that are relatively simple and appropriate for the most common types of nonlinear equations.

9.6.1 Single equations of a single variable

Solving a single nonlinear equation is enormously simpler than solving a system of nonlinear equations, so that is where we start. A word of caution: solving non-linear equations can be a tricky business so it is important that you have a good sense of the behavior of the function you are trying to solve. The best way to do this is to plot the function over the domain of interest before trying to find the solutions. This will greatly assist you in finding the solutions you seek and avoiding spurious solutions.

We begin with a concrete example. Suppose we want to find the solutions to the equation

$$\tan x = \sqrt{(8/x)^2 - 1}$$

Plots of $\tan x$ and $\sqrt{(8/x)^2 - 1}$ *vs* x are shown in the top plot in the figure *Crossing functions*, albeit with x replaced by θ. The solutions to this equation are those x values where the two curves $\tan x$ and $\sqrt{(8/x)^2 - 1}$ cross each other. The first step towards obtaining a numerical solution is to rewrite the equation to be solved in the form $f(x) = 0$. Doing so, the above equation becomes

$$\tan x - \sqrt{(8/x)^2 - 1} = 0$$

Obviously the two equations above have the same solutions for x. Parenthetically we mention that the problem of finding the solutions to equations of the form $f(x) = 0$ is often referred to as *finding the roots* of $f(x)$.

Next, we plot $f(x)$ over the domain of interest, in this case from $x = 0$ to 8. We are only interested in positive solutions and for $x > 8$, the equation has no real solutions as the argument of the square root becomes negative. The solutions, the points where $f(x) = 0$ are indicated by green circles; there are three of them. Another notable feature of the function is that it diverges to $\pm\infty$ at $x = \{0, \pi/2, 3\pi/2, 5\pi/2\}$.

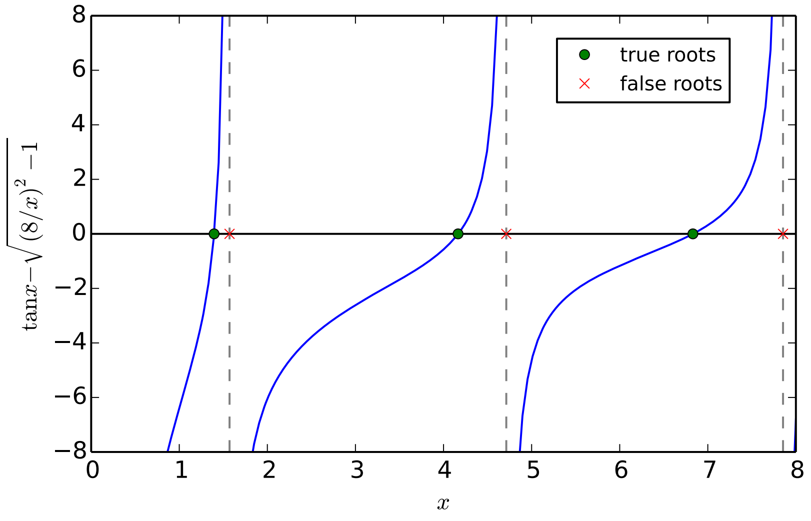


Figure 9.4: Roots of a nonlinear function

Brent method

One of the workhorses for finding solutions to a single variable nonlinear equation is the method of Brent, discussed in many texts on numerical methods. SciPy's implementation of the Brent algorithm is the function `scipy.optimize.brentq(f, a, b)`, which has three required arguments. The first `f` is the name of the user-defined function to be solved. The next two, `a` and `b` are the x values that bracket the solution you are looking for. You should choose `a` and `b` so that there is only one solutions in the interval between `a` and `b`. Brent's method also requires that `f(a)` and `f(b)` have opposite signs; an error message is returned if they do not. Thus to find the three solutions to $\tan x - \sqrt{(8/x)^2 - 1} = 0$, we need to run `scipy.optimize.brentq(f, a, b)` three times using three different values of `a` and `b` that bracket each of the three solutions. The program below illustrates the how to use `scipy.optimize.brentq`

```python
import numpy as np
import scipy.optimize
import matplotlib.pyplot as plt

def tdl(x):
    y = 8./x
    return np.tan(x) - np.sqrt(y*y-1.0)

# Find true roots

rx1 = scipy.optimize.brentq(tdl, 0.5, 0.49*np.pi)
rx2 = scipy.optimize.brentq(tdl, 0.51*np.pi, 1.49*np.pi)
rx3 = scipy.optimize.brentq(tdl, 1.51*np.pi, 2.49*np.pi)
rx = np.array([rx1, rx2, rx3])
ry = np.zeros(3)
# print using a list comprehension
print('\nTrue roots:')
print('\n'.join('f({0:0.5f}) = {1:0.2e}'.format(x, tdl(x)) for x in rx))

# Find false roots

rx1f = scipy.optimize.brentq(tdl, 0.49*np.pi, 0.51*np.pi)
rx2f = scipy.optimize.brentq(tdl, 1.49*np.pi, 1.51*np.pi)
rx3f = scipy.optimize.brentq(tdl, 2.49*np.pi, 2.51*np.pi)
rxf = np.array([rx1f, rx2f, rx3f])
# print using a list comprehension
print('\nFalse roots:')
print('\n'.join('f({0:0.5f}) = {1:0.2e}'.format(x, tdl(x)) for x in rxf))

# Plot function and various roots
```

```
x = np.linspace(0.7, 8, 128)
y = tdl(x)
# Create masked array for plotting
ymask = np.ma.masked_where(np.abs(y)>20., y)

plt.figure(figsize=(6, 4))
plt.plot(x, ymask)
plt.axhline(color='black')
plt.axvline(x=np.pi/2., color="gray", linestyle='--', zorder=-1)
plt.axvline(x=3.*np.pi/2., color="gray", linestyle='--', zorder=-1)
plt.axvline(x=5.*np.pi/2., color="gray", linestyle='--', zorder=-1)
plt.xlabel(r'$x$')
plt.ylabel(r'$\tan x - \sqrt{(8/x)^2-1}$')
plt.ylim(-8, 8)

plt.plot(rx, ry, 'og', ms=5, label='true roots')

plt.plot(rxf, ry, 'xr', ms=5, label='false roots')
plt.legend(numpoints=1, fontsize='small', loc = 'upper right',
           bbox_to_anchor = (0.92, 0.97))
plt.tight_layout()
plt.show()
```

Running this code generates the following output:

```
In [1]: run rootbrentq.py

True roots:
f(1.39547) = -6.39e-14
f(4.16483) = -7.95e-14
f(6.83067) = -1.11e-15

False roots:
f(1.57080) = -1.61e+12
f(4.71239) = -1.56e+12
f(7.85398) = 1.16e+12
```

The Brent method finds the three true roots of the equation quickly and accurately when you provide values for the brackets `a` and `b` that are valid. However, like many numerical methods for finding roots, the Brent method can produce spurious roots as it does in the above example when `a` and `b` bracket singularities like those at $x = \{\pi/2, 3\pi/2, 5\pi/2\}$. Here we evaluated the function at the purported roots found by `brentq` to verify that the values of $x$ found were indeed roots. For the true roots, the values of the function were very near zero, to within an acceptable roundoff error of less than 10^{-13}. For the false roots, exceedingly large numbers on the order of 10^{12} were obtained, indicating a possible problem with these roots. These results, together with the plots, allow you to unambiguously identify the true solutions to this nonlinear function.

The `brentq` function has a number of optional keyword arguments that you may find useful. One keyword argument causes `brentq` to return not only the solution but the value of the function evaluated at the solution. Other arguments allow you to specify a tolerance to which the solution is found as well as a few other parameters possibly of interest. Most of the time, you can leave the keyword arguments at their default values. See the `brentq` entry online on the SciPy web site for more information.

Other methods for solving equations of a single variable

SciPy provides a number of other methods for solving nonlinear equations of a single variable. It has an implementation of the Newton-Raphson method called `scipy.optimize.newton`. It's the racecar of such methods; its super fast but less stable that the Brent method. To fully realize its speed, you need to specify not only the function to be solved, but also its first derivative, which is often more trouble than its worth. You can also specify its second derivative, which may further speed up finding the solution. If you do not specify the first or second derivatives, the method uses the secant method, which is usually slower than the Brent method.

Other methods, including the Ridder (`scipy.optimize.ridder`) and bisection (`scipy.optimize.bisect`), are also available, although the Brent method is generally superior. SciPy let's you use your favorite.

9.6.2 Solving systems of nonlinear equations

Solving systems of nonlinear equations is not for the faint of heart. It is a difficult problem that lacks any general purpose solutions. Nevertheless, SciPy provides quite an assortment of numerical solvers for nonlinear systems of equations. However, because of the complexity and subtleties of this class of problems, we do not discuss their use here.

9.7 Exercises

1. Use NumPy's `polyval` function together with SciPy to plot the following functions:

   (a) The first four Chebyshev polynomials of first kind. Plot these over the interval from -1 to +1.

   (b) The first four Hermite polynomials *multiplied* by $e^{-x^2/2}$. Plot these on the interval from -5 to +5. These are the first four wave functions of the quantum mechanical simple harmonic oscillator.

APPENDIX

A

INSTALLING PYTHON

For scientific programming with Python, you need to install Python and three scientific Python libraries: NumPy, SciPy, and MatPlotLib. There are many more libraries you can install, but Python along with NumPy, SciPy, and MatPlotLib are those that are essential for scientific programming.

A.1 Installing Python

There are a number of ways to install Python and the scientific libraries you will need on your computer. Some are easier than others. You can install Python and the scientific libraries you need from “source” and compile them yourself. This is not recommended unless you are an expert in Python, in which case you have little need for this manual.

For most people, the simplest way to install Python and all the scientific libraries you need is to install either *Canopy* or *Spyder*. Canopy and Spyder are integrated development environments (IDEs) for Python. They have a number of very useful features and tools. First, they have syntax highlighting, which colors different parts Python syntax according to function, making code easier to read. Second, and more importantly, they run a program in the background called *PyFlakes* that checks the validity of the Python syntax as you write it. It’s like a spelling and grammar checker all rolled into one, and it is extremely useful, for novice and expert alike. The Canopy and Spyder IDEs have a number of other useful features, which we do not go into here, but expect you will learn about as you become more familiar with Python. Canopy is a simpler IDE than Spyder, and easier for novices to learn and maintain. Spyder has more advanced features, which you may find useful as you become more expert in Python programming.

Canopy is written, maintained, and distributed by the software company Enthought (http://www.enthought.com/). There are two versions of Canopy. One version, *Canopy Express*, is completely free to everybody and contains all the libraries you will need for this manual. The other, *Canopy Basic*, contains nearly every library you are ever likely to need for scientific computing. It is free to academic users; others pay a fee. Go to https://www.enthought.com/products/canopy/ and press the "Get Canopy" button, which will take you to a page where you can either download *Canopy Express* or request and academic license, which will allow you to download *Canopy Basic*.

Spyder provides a completely open source programming environment for Python. The entire Spyder distribution is free to all and can be found at https://code.google.com/p/spyderlib/. It also includes nearly all the scientific libraries you are likely to need for scientific computing.

In this manual, we assume you are using Canopy, but the Spyder interface is very similar to Canopy so that most users should have no difficulty using Spyder with this manual. If you choose to use Spyder, launch Spyder and then go to the Preferences menu and then under the `Console` menu, select the `Advanced settings` tab; tick the box `Start an IPython kernel at startup` (it may already be selected, in which case you need to do nothing). You only need to do this once, which sets up the IPython console when Spyder is launched. Once that is done, you should be able to follow everything written in this manual.

A.2 Testing your installation of Python

Running the Python program below tests your installation of Python to verify that the installation was successful. In particular, it tests that the NumPy, SciPy, and MatPlotLib libraries that are needed for this manual are properly installed.

If you are a student, you should input your first and last names inside the single quotes on lines 15 and 16, respectively.

Instructors can modify lines 21-23 to suit the needs of the course.

```
1  # This code tests that your Python installation worked.
2  # It generates a png image file that you should e-mail
3  # to the address shown on the plot
4  import scipy
5  import numpy
6  import matplotlib
7  import matplotlib.pyplot as plt
8  import platform
9  import socket
10
11 # If you are a student, please fill in your first and last
12 # names inside the quotes in the two lines below.  Do not
```

```python
# modify anything else in this file

your_first_name = 'Dana'
your_last_name = 'Martin'

# If you are an instructor, modify the next 3 lines.
# You do not need to modify anything else in this file.

classname = 'Intro Phys I'
term = 'Fall_2014'      # must contain no spaces
email = 'hmwkemail@univX.edu'

plt.plot([0,1], 'r', [1,0], 'b')
plt.text( 0.5, 0.8, '{0:s} {1:s}'
        .format(your_first_name, your_last_name),
        horizontalalignment='center',
        size = 'x-large',
        bbox=dict(facecolor='purple', alpha=0.4))
plt.text( 0.5, 0.1,
    '{1:s}\nscipy {2:s}\nnumpy {3:s}\nmatplotlib {4:s}\non {5:s}\n{6:s}'
        .format(
        classname,
        term,
        scipy.__version__,
        numpy.__version__,
        matplotlib.__version__,
        platform.platform(),
        socket.gethostname()
        ) ,
    horizontalalignment='center'
    )
filename = your_last_name + '_' + your_first_name + '_' + term + '.png'
plt.title('*** E-mail the saved version of this plot, ***\n' +
    '"{0:s}" to {1:s}'.format(filename, email), fontsize=12)
plt.savefig(filename)
plt.show()
```

APPENDIX

B

IPYTHON NOTEBOOKS

IPython notebooks are useful for logging your work. Here we suggest that you use them for logging and turning in homework assignments. You may also find them useful in other contexts such as laboratory work. The IPython Notebook interface is similar to Mathematica or Maple.

B.1 An interactive interface

An IPython notebook is a web-based environment for interactive computing. You can work in this environment interactively just as you would using the IPython shell (the interactive Python pane shown in the *Canopy window* figure). In addition, you can also store and run programs in an IPython notebook just like you would from the Code Editor window (see *Canopy window*). Thus, it would seem that an IPython notebook and the Canopy environments do essentially the same thing. Up to a point, that is true. In the final analysis, however, Canopy (or Spyder or some similar Python development environment) is generally more useful for developing, storing, and running code. An IPython Notebook, on the other hand, is excellent for logging your work in Python. That is why we suggest using them for homework assignments. You may find them useful in other contexts as well.

B.2 Installing and launching an IPython notebook

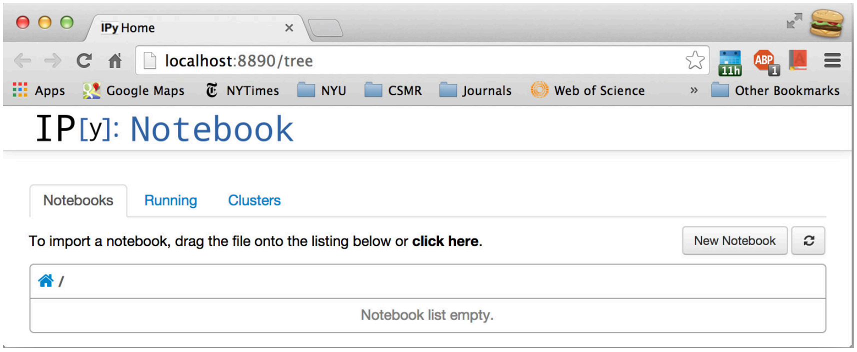
If you have installed Canopy or Spyder, then you have already installed all the software you need to use IPython notebooks. IPython notbooks are stored in files with a `.ipynb` extension. To create an IPython notebook, launch the Terminal (Mac) or the Command

Prompt (PC) application. On the Mac, the Terminal application is found in the Applications/Utilies folder. On the PC, the Command Prompt application is found under the Start/All Programs/Accessories menu. Here we will refer the the Terminal or Command Prompt applications at the System Console. Type `pwd` (Mac) or `chdir` (PC) to determine the current directory (folder) of the System Console. Type `ls` (Mac) or `dir` (PC) to list all the files and subdirectories in the current directory. Using the `cd` change directory command, move the System Console to the directory in which you want to store your IPython notebook.

At the System Console prompt, type

```
ipython notebook --matplotlib inline &
```

This will launch the IPython notebook web application and will display the *IPython Notebook Home Page* as a page in your default web browser.



To import a notebook, drag the file onto the listing below or **click here**.

Notebook list empty.

Figure B.1: IPython Notebook Home Page

To create a new IPython notebook, click the `New Notebook` button. That will open a new *IPython Notebook* with the provisional title `Untitled0` in a new tab in your browser. To give the notebook a more meaningful name, click on `Untitled0` in your browser window to the right of `IP[y]: Notebook` and rename your notebook `FirstNotebook`. The Name `FirstNotebook` will replace `Untitled0` in your Notebook browser window and a file named `FirstNotebook.ipynb` will appear in the directory from which you launched IPython Notebook. That file will contain all the work you do in the IPython notebook. Next time you launch IPython Notebook from this same directory, all the IPython notebooks in that directory will appear in a list on the IPython Notebook Dashboard. Clicking on one of them will launch that notebook.

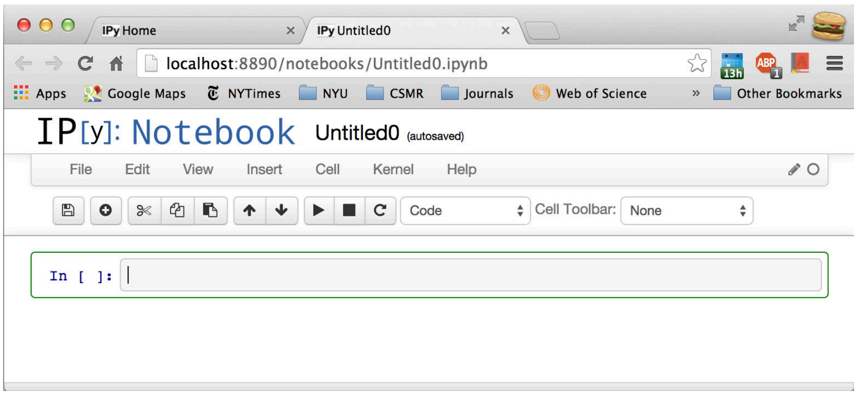


Figure B.2: IPython Notebook

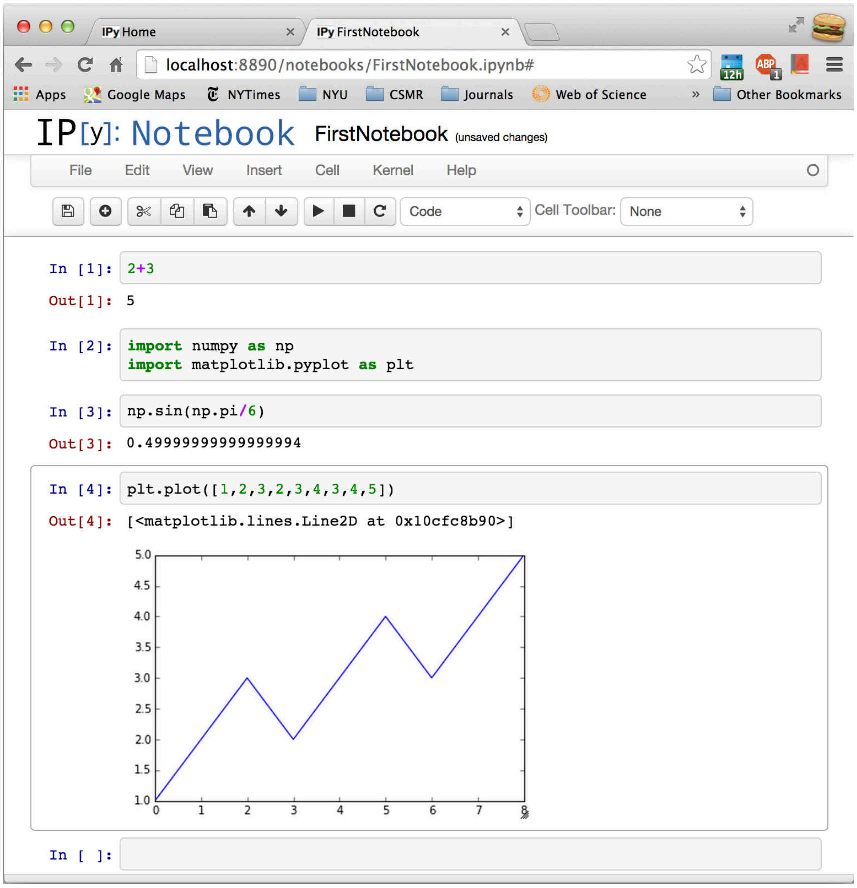
B.3 Using an IPython Notebook

When you open a new IPython Notebook, an IPython interactive cell with the prompt `In[ ]:` to the left, appears. You can type code into this cell just as you would in the IPython shell of the *Canopy window*. For example, typing `2+3` into the cell and pressing `Shift-Enter` (or `Shift-Return`) to execute the cell yields the expected result. Try it out.

Below the result a new IPython interactive cell appears ready for your next entry. In the next cell type the commands to import `numpy` and `matplotlib.pyplot` as shown in the *IPython Notebook demo* figure. These two commands are separated by the `Shift` (without `Enter` or `Return`) so that they appear in the same cell. They are then both executed by pressing `Shift-Enter` (or `Shift-Return`). After importing numpy, typing `sin(np.pi/6.)` and pressing `Shift-Enter` produces the expected result (to nearly 1 part in 10^{16}).

You can also create plots in an IPython notebook. For example, typing `plt.plot([1,2,3,2,3,4,3,4,5])` and pressing `Shift-Enter` produces the same plot shown in the *Interactive plot window* figure. The plot is produced “in line” and not in a separate window, because we used the `--matplotlib inline` switch when we launched IPython Notebook. If you have followed along, your IPython notebook should look something like that shown in the figure *IPython Notebook demo*.

When importing NumPy and MatPlotLib, you can import the entire libraries using the commands “`from numpy import *`” and “`from matplotlib.pyplot import *`”. Importing the two libraries in this way means that you don’t have to use the `np.` and `plt.` prefixes when calling NumPy and MatPlotLib functions, just as in



```
In [1]: 2+3
Out[1]: 5

In [2]: import numpy as np
        import matplotlib.pyplot as plt

In [3]: np.sin(np.pi/6)
Out[3]: 0.49999999999999994

In [4]: plt.plot([1,2,3,2,3,4,3,4,5])
Out[4]: [<matplotlib.lines.Line2D at 0x10cfc8b90>]

In [ ]:
```

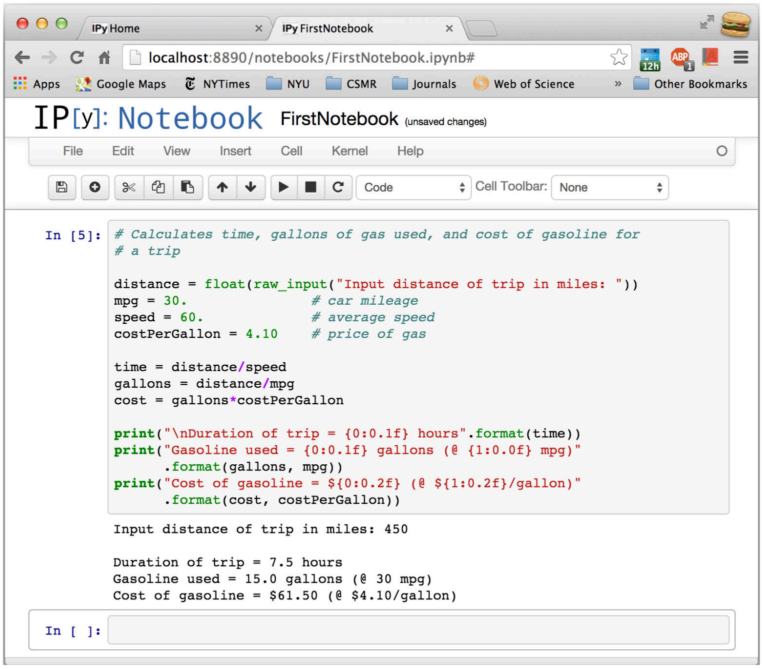
Figure B.3: IPython Notebook demo

the IPython console in Canopy or Spyder. While this could lead to some confusion in the namespaces, it's usually harmless.

Be sure to press the `Save and Checkpoint` icon at the far left near the top of the IPython Notebook window from time to time to **save your work**.

B.4 Running programs in an IPython Notebook

You can also run programs in an IPython notebook. As an example, we run the program introduced in the section on *Screen output*. The program is input into a single notebook cell and then executed by pressing `Shift-Enter`.



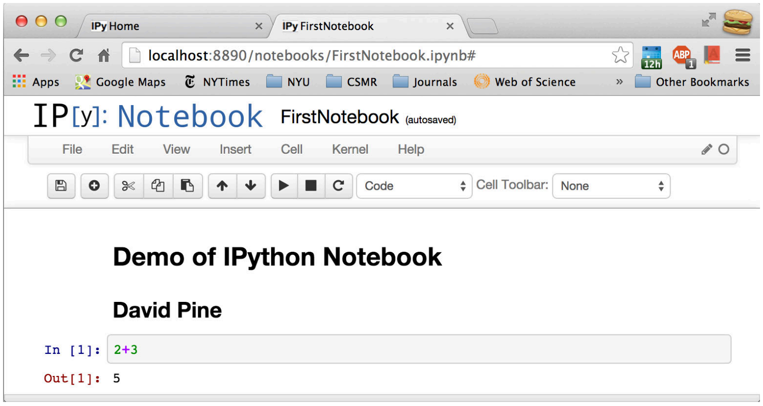
```
In [5]: # Calculates time, gallons of gas used, and cost of gasoline for
        # a trip

        distance = float(raw_input("Input distance of trip in miles: "))
        mpg = 30.                   # car mileage
        speed = 60.                 # average speed
        costPerGallon = 4.10        # price of gas

        time = distance/speed
        gallons = distance/mpg
        cost = gallons*costPerGallon

        print("\nDuration of trip = {0:0.1f} hours".format(time))
        print("Gasoline used = {0:0.1f} gallons (@ {1:0.0f} mpg)"
              .format(gallons, mpg))
        print("Cost of gasoline = ${0:0.2f} (@ ${1:0.2f}/gallon)"
              .format(cost, costPerGallon))

        Input distance of trip in miles: 450

        Duration of trip = 7.5 hours
        Gasoline used = 15.0 gallons (@ 30 mpg)
        Cost of gasoline = $61.50 (@ $4.10/gallon)

In [ ]:
```

Figure B.4: Running a program

In this example, the program requests input from the user: the distance of the trip. The program runs up to the point where it needs input from the user, and then pauses until the user responds and presses the `Enter` or `Return` key. The program then completes its execution. Thus the IPython notebook provides a complete log of the session.

B.5 Annotating an IPython Notebook

An IPython notebook will be more easily comprehended if it includes annotations of the session. In addition to logging the inputs and outputs of computations, IPython Notebook allows the user to embed headings, explanatory notes, mathematics, and images.



Demo of IPython Notebook

David Pine

```
In [1]: 2+3
Out[1]: 5
```

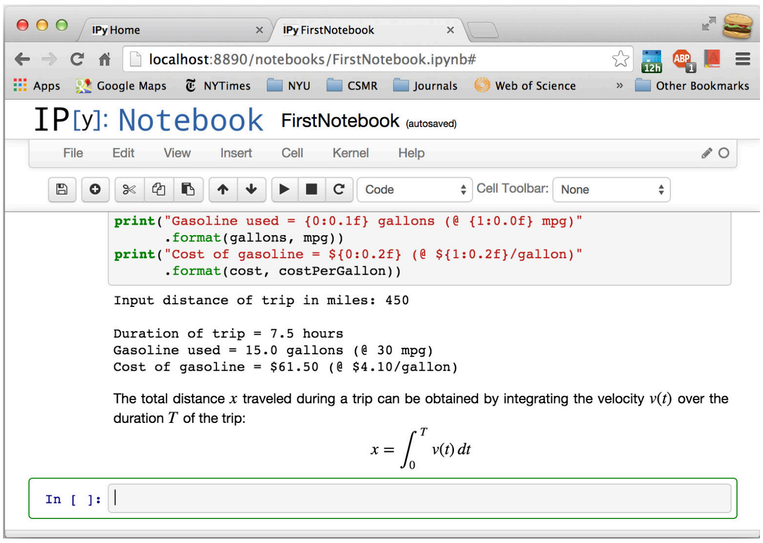
Figure B.5: Annotating a notebook

Suppose, for example, that we want to have a title at the top of the IPython notebook we have been working with, and we want to include the name of the author of the session. To do this, we scroll the IPython notebook back up to the top and place the cursor and click in the very first input cell, the one that contained `2+3`. We then open the `Insert` menu near the top center of the window and click on `Insert Cell Above`, which opens up a new input cell above the first cell. Next, we click on the box in the Toolbar that says `Code`. A list of cell types appears: `Code` (currently checked), `Markdown`, `Raw Text`, `Heading 1`, `Heading 2`, ..., `Heading 6`. Select `Heading 1`; immediately the `In [ ]:` prompt disappears, indicating that this box is no longer meant for inputing and executing Python code. Type "`Demo of IPython Notebook`" and press `Shift-Enter` (or `Shift-Return`). A heading in large print appears before the first IPython code cell. Place the cursor back in the first Ipython code cell (`2+3`). Once again, select the `Insert` menu and click on `Insert Cell Above`. Again, click on the Toolbar that says `Code`, but this time select `Heading 2`. Type your name into the newly created cell and press `Shift-Enter`. Your name is printed in the cell in slightly smaller print than for the previous case.

You can also write comments, including mathematical expressions, into an IPython Notebook cell. Let's include a comment after the program we ran that calculated the cost of gasoline for a road trip. First we place the cursor in the open formula cell below program we ran and then click on the box in the Toolbar that says `Code` and change it to `Markdown`. Returning to the cell, we enter the text of our comment. We can enter any text we wish, including mathematical expressions using the markup language Latex. (If you do not already know Latex, you can get a brief introduction at these sites: http://en.wikibooks.org/wiki/LaTeX/Mathematics or ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf.) Here we enter the following text:

```
The total distance $x$ traveled during a trip can be
obtained by integrating the velocity $v(t)$ over the
duration $T$ of the trip:
\begin{align}
    x = \int_0^T v(t)\, dt
\end{align}
```

After entering the text, pressing `Shift-Enter` yields the result shown in *Annotation using a Markdown cell*.



```
print("Gasoline used = {0:0.1f} gallons (@ {1:0.0f} mpg)"
      .format(gallons, mpg))
print("Cost of gasoline = ${0:0.2f} (@ ${1:0.2f}/gallon)"
      .format(cost, costPerGallon))
```

```
Input distance of trip in miles: 450

Duration of trip = 7.5 hours
Gasoline used = 15.0 gallons (@ 30 mpg)
Cost of gasoline = $61.50 (@ $4.10/gallon)
```

The total distance x traveled during a trip can be obtained by integrating the velocity $v(t)$ over the duration T of the trip:

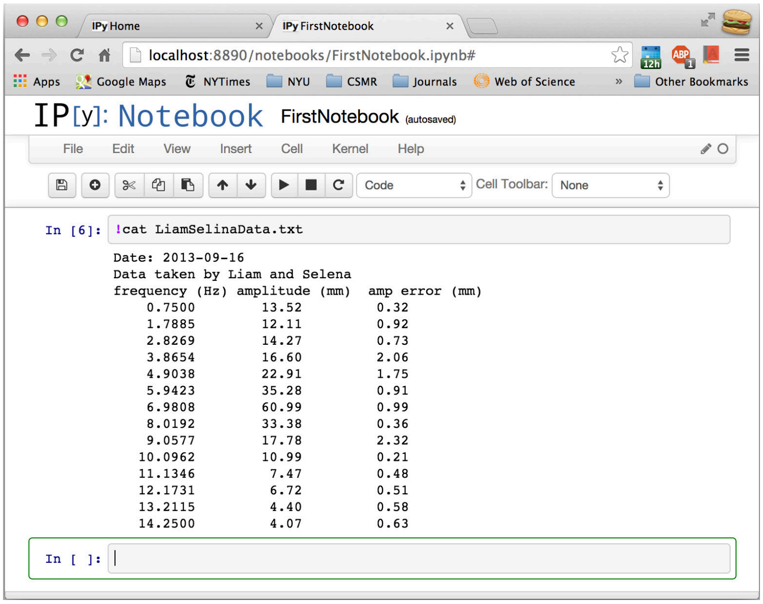
$$x = \int_0^T v(t)\, dt$$

```
In [ ]:
```

Figure B.6: Annotation using a Markdown cell

The `$` symbol brackets inline mathematical expressions in Latex, while the `\begin{align}` and `\end{align}` expressions bracket displayed expressions. You only need to use Latex if you want to have fancy mathematical expressions in your notes. Otherwise, they are not necessary.

Suppose you were importing a data (`.txt`) file from your hard disk and you wanted to print it out in one of the notebook cells. If you were in the `Terminal` (Mac) or `Command Prompt` (PC), you could write the contents of any text file using the command `cat` *filename* (Mac) or `type` *filename* (PC). You can execute the same operation from the IPython prompt using the Unix (Mac) or DOS (PC) command preceded by an exclamation point, as described in the section on *System shell commands*.



```
In [6]: !cat LiamSelinaData.txt
        Date: 2013-09-16
        Data taken by Liam and Selena
        frequency (Hz) amplitude (mm)  amp error (mm)
            0.7500        13.52         0.32
            1.7885        12.11         0.92
            2.8269        14.27         0.73
            3.8654        16.60         2.06
            4.9038        22.91         1.75
            5.9423        35.28         0.91
            6.9808        60.99         0.99
            8.0192        33.38         0.36
            9.0577        17.78         2.32
           10.0962        10.99         0.21
           11.1346         7.47         0.48
           12.1731         6.72         0.51
           13.2115         4.40         0.58
           14.2500         4.07         0.63

In [ ]:
```

Figure B.7: Displaying a text file from disk

B.6 Editing and rerunning a notebook

In working with an IPython notebook, you may find that you want to move some cells around, or delete some cells, or simply change some cells. All of these tasks are possible. You can cut and paste cells, as in a normal document editor, using the `Edit` menu. You can also freely edit cells and re-execute them by pressing `Shift-Enter`. Sometimes you may find that you would like to re-execute the entire notebook afresh. You can do this by going to the `Kernel` menu and selecting `Restart`. A warning message will appear asking you if you really want to restart. Answer in the affirmative. Then open the `Cell` menu and select `Run All`, which will re-execute the notebook starting with the first cell.

You will have to re-enter any screen input requested by the notebook scripts.

B.7 Quitting an IPython notebook

It goes almost without saying that before quitting an IPython notebook, you should make sure you have saved the notebook by pressing the `Save and Checkpoint` item in the `File` menu or its icon in the Toolbar.

When you are ready to quit working with a notebook, click on the `Close and halt` item in the `File` menu. Your notebook browser tab will close and you will return to the IPython Notebook Dashboard. Just close the IPython Notebook Dashboard tab in your browser to end the session.

Finally, return to the `Terminal` or `Command Prompt` application, hold down the `control` key and press `c` twice in rapid succession. This stops the IPython Notebook session. You should see the normal system prompt. You can then close the `Terminal` (Mac) or `Command Prompt` (PC) session if you wish.

B.8 Working with an existing IPython notebook

To work with an existing IPython notebook, open the `Terminal` (Mac) or `Command Prompt` (PC) application and navigate to the directory in which the notebook you want to work with resides. Recall that IPython notebooks have the `.ipynb` extension. Launch the IPython Notebook Dashboard as you did previously by issuing the command

```
ipython notebook --matplotlib inline &
```

This will open the IPython Notebook Dashboard in your web browser, where you should see a list of all the IPython notebooks in that directory (folder). Click on the name of the notebook you want to open. It will appear in a new tab on your web browser as before.

Note that while all the input and output from the previous saved session is present, none of it has been run. That means that none of the variables or other objects has been defined in this new session. To initialize all the objects in the file, you must rerun the file. To rerun the file, press the `Cell` menu and select `Run All`, which will re-execute all the cells. You will have to re-enter any screen input requested by the notebook scripts. Now you are ready to pick up where you left off the last time.

APPENDIX

C

PYTHON RESOURCES

This text provides an introduction to Python for science and engineering applications but is hardly exhaustive. There are many other resources that you will want to tap. Here I point out several that you may find useful.

C.1 Web resources

The best web resource for Python is a good search engine like Google. Nevertheless, I list a few web sites here that you might find useful. I have successfully resisted any attempt to be exhaustive so that the list is actually useful. Nevertheless, if you find a really cool site that you think should be on this list, please let me know.

http://www.python.org/ The official Python web site. I almost never look here.

http://docs.python.org/2/reference/ Sometimes I look here for detailed information about Python 2, which is the version used in this manual. Someday, when all the most useful packages are available for Python 3, we will switch to Python 3 and use http://docs.python.org/3/reference/ instead.

http://docs.scipy.org/doc/numpy/reference/ I usually start here when I need information about NumPy. It has links to just about all the NumPy documentation I need. By the way, I say “num-pee”, which rhymes with “bumpy”—a lot of people say “num-pie”, which doesn’t sound like English to me.

http://docs.scipy.org/doc/scipy/reference/ I start here when I need information about SciPy, its various packages and their functions. I say "psy-pi" for SciPy, like everyone else. Who says I have to be consistent? (see Emerson)

http://matplotlib.org/api/pyplot_summary.html The *Plotting Commands Summary* page for MatPlotLib. It has a search feature and links to all the MatPlotLib documentation, which I use a lot. You can go the the main MatPlotLib page, http://matplotlib.org/, but frankly, it's less useful. The site http://www.loria.fr/~rougier/teaching/matplotlib/ is also useful for learning some MatPlotLib tricks.

http://ipython.org/ I go to this page mostly to learn about IPython Notebook (http://ipython.org/notebook.html) but it's also useful if you need information about the IPython interpreter, especially if you want to find out more about IPython magic commands.

http://scipy-lectures.github.io/ This link provides a web-based quick introduction to scientific Python that assumes you have some experience programming (not necessarily in Python). It's terser than my introduction, but covers some useful topics that I do not cover.

http://www.enthought.com/ I get my latest version of Python and all the packages I need for scientific computing here. One stop shopping and everything is free for academic users. They offer three distributions: Express, Basic, and Professional. Express is free to all users and contains all the Python libraries, NumPy, MatPlotlib, SciPy, *etc*, that are described in this manual. Basic includes all the packages Enthought supports, which is likely to be everything you will ever need. Professional adds support services. Basic is free to academic users. One nice feature of Canopy is its package manager, which makes it child's play to update or add Python packages. This is a very nice feature, especially for beginners. Canopy displaces the older Enthought EPD packages.

https://code.google.com/p/spyderlib/ Get the latest version of Spyder, the alternative IDE to Enthought's Canopy. Sypder is completely open source and has a number of nice features, like introspection, not currently available in Canopy. Spyder is very popular and can be installed easily on all platforms. For an effective package manager, see Anaconda (next entry).

http://continuum.io/ Anaconda provides a free distribution of Spyder with a comprehensive package manager, and may be the best choice for power users of Python for scientific programming. The package manager is run by line commands from a terminal window, which may put off beginners. Anaconda can be downloaded from https://store.continuum.io/cshop/anaconda/.

http://www.scipy.org/Mailing_Lists Go here if you want to sign up for a mailing list for NumPy or SciPy, or if you want to report a bug. Mailing lists give you access to a community of developers and users that can often provide expert help. Just remember to be polite and respectful of those helping you and also to those posting questions.

https://lists.sourceforge.net/lists/listinfo/matplotlib-users The mailing list for MatPlotLib. See paragraph immediately above.

http://stackoverflow.com/ StackOverflow may just be your best friend when it comes to solving problems using Python. You pose your question about whatever problem you are having with Python and other people write answers. The answers are monitored and the best survive. In fact, most of the questions you will pose have already been posed by someone else and answered. So you can just look up the answer to most of your questions. StackOverflow is a wonderful resource.

C.2 Books

There are a lot of books on Python and there is no way I can provide reviews for all of them. I have found that the book by Mark Lutz, *Learning Python*, published by O'Reilly Media does the trick for most people. It doesn't have anything special for scientific programming, and thus does not cover the NumPy, SciPy, or MatPlotLib packages, but for just about everything else, it's an excellent resource. It gives a good introduction to object oriented programming, or OOP, which I say little about in this text. The 3rd edition of the book covers Python 2 while the 4th and 5th (current) editions cover Python 3. You are probably better off getting the latest edition as everybody will soon be using Python 3. If you are using Python 2, as we do in this text, you can easily enough figure out the differences between Python 3 and 2.

INDEX