A. An Introduction to Scientific Computing with Python

"The world's a book, writ by the eternal art –
Of the great author printed in man's heart,
'Tis falsely printed, though divinely penned,
And all the errata will appear at the end."
(Francis Quarles)

In this appendix we aim to give a brief introduction to the Python language[1] and its use in scientific computing. It is intended for users who are familiar with programming in another language such as IDL, MATLAB, C, C++, or Java. It is beyond the scope of this book to give a complete treatment of the Python language or the many tools available in the modules listed below. The number of tools available is large, and growing every day. For this reason, no single resource can be complete, and even experienced scientific Python programmers regularly reference documentation when using familiar tools or seeking new ones. For that reason, this appendix will emphasize the best ways to access documentation both on the web and within the code itself.

We will begin with a brief history of Python and efforts to enable scientific computing in the language, before summarizing its main features. Next we will discuss some of the packages which enable efficient scientific computation: NumPy, SciPy, Matplotlib, and IPython. Finally, we will provide some tips on writing efficient Python code. Some recommended resources are listed in §A.10.

A.1. A brief history of Python

Python is an open-source, interactive, object-oriented programming language, created by Guido Van Rossum in the late 1980s. Python is loosely based on the earlier ABC, a language intended to be taught to scientists and other computer users with no formal background in computer science or software development. Python was created to emulate the strengths of ABC—its ease of use, uncluttered syntax, dynamic typing, and interactive execution —while also improving on some of ABC's weaknesses. It was designed primarily as a second language for experienced programmers—a full-featured scripting framework to "bridge the gap between the shell and C."[2] But because of its heritage in ABC, Python quickly became a popular language choice for introductory programming courses, as well as for nonprogrammers who require a computing tool. Increasingly, the same beauty and ease of use that

[1] http://python.org

[2] Guido Van Rossum, "The Making of Python," interview available at http://www.artima.com/intv/pythonP.html; accessed October 2012.

make Python appealing to beginning programmers have drawn more experienced converts to the language as well.

One important strength of Python is its extensible design: third-party users can easily extend Python's type system to suit their own applications. This feature is a key reason that Python has developed into a powerful tool for a large range of applications, from network control, to web design, to high-performance scientific computing. Unlike proprietary systems like MATLAB or IDL, development in the Python universe is driven by the users of the language, most of whom volunteer their time. Partly for this reason, the user base of Python has grown immensely since its creation, and the language has evolved as well. Guido Van Rossum is still actively involved in Python's development, and is affectionately known in the community as the "BDFL"—the Benevolent Dictator For Life. He regularly gives keynote talks at the Python Software Foundation's annual *PyCon* conferences, which now annually attract several thousand attendees from a wide variety of fields and backgrounds.

A.2. The SciPy universe

Though Python provides a sound linguistic foundation, the language alone would be of little use to scientists. Scientific computing with Python today relies primarily on the *SciPy* ecosystem, an evolving set of open-source packages built on Python which implement common scientific tasks, and are maintained by a large and active community.

A.2.1. NumPy

The central package of the SciPy ecosystem is *NumPy* (pronounced "Num-Pie"), short for "Numerical Python." NumPy's core object is an efficient N-dimensional array implementation, and it includes tools for common operations such as sorting, searching, elementwise operations, subarray access, random number generation, fast Fourier transforms, and basic linear algebra. NumPy was created by Travis Oliphant in 2005, when he unified the features of two earlier Python array libraries, *Numeric* and *NumArray*. NumPy is now at the core of most scientific tools written in Python. Find more information at http://www.numpy.org.

A.2.2. SciPy

One step more specialized than NumPy is *SciPy* (pronounced "Sigh-Pie"), short for "Scientific Python." SciPy is a collection of common computing tools built upon the NumPy array framework. It contains wrappers of much of the well-tested and highly optimized code in the NetLib archive, much of which is written in Fortran (e.g., BLAS, LAPACK, FFTPACK, FITPACK, etc.). SciPy contains routines for operations as diverse as numerical integration, spline interpolation, linear algebra, statistical analysis, tree-based searching, and much more. SciPy traces its roots to Travis Oliphant's *Multipack* package, a collection of Python interfaces to scientific modules written mainly in Fortran. In 2001, Multipack was combined with scientific toolkits created by Eric Jones and Pearu Peterson, and the expanded package was renamed SciPy. NumPy was created by the SciPy developers, and was originally envisioned to be a part of the SciPy package. For ease of maintenance and development, the two projects have different release cycles, but their development communities remain closely tied. More information can be found at http://scipy.org.

A.2.3. IPython

One popular aspect of well-known computing tools such as IDL, MATLAB, and Mathematica is the ability to develop code and analyze data in an interactive fashion. This is possible to an extent with the standard Python command-line interpreter, but *IPython* (short for Interactive Python) extends these capabilities in many convenient ways. It allows tab completion of Python commands, allows quick access to command history, features convenient tools for documentation, provides time-saving "magic" commands, and much more. Conceived by Fernando Perez in 2001, and building on functionality in the earlier *IPP* and *LazyPython* packages, IPython has developed from a simple enhanced command line into a truly indispensable tool for interactive scientific computing. The recently introduced parallel processing functionality and the notebook feature are already changing the way that many scientists share and execute their Python code. As of the writing of this book in 2013, the IPython team had just received a \$1.15 million grant from the Sloan Foundation to continue their development of this extremely useful research tool. Find more information at http://ipython.org.

A.2.4. Matplotlib

Scientific computing would be lost without a simple and powerful system for data visualization. In Python this is provided by *Matplotlib*, a multiplatform system for plotting and data visualization, which is built on the NumPy framework. Matplotlib allows quick generation of line plots, scatter plots, histograms, flow charts, three-dimensional visualizations, and much more. It was conceived by John Hunter in 2002, and originally intended to be a patch to enable basic MATLAB-style plotting in IPython. Fernando Perez, the main developer of IPython, was then in the final stretch of his PhD, and unable to spend the time to incorporate Hunter's code and ideas. Hunter decided to set out on his own, and version 0.1 of Matplotlib was released in 2003. It received a boost in 2004 when the Space Telescope Science Institute lent institutional support to the project, and the additional resources led to a greatly expanded package. Matplotlib is now the de facto standard for scientific visualization in Python, and is cleanly integrated with IPython. Find more information at http://matplotlib.org.

A.2.5. Other specialized packages

There are a host of available Python packages which are built upon a subset of these four core packages and provide more specialized scientific tools. These include *Scikit-learn* for machine learning, *scikits-image* for image analysis and manipulation, *statsmodels* for statistical tests and data exploration, *Pandas* for storage and analysis of heterogeneous labeled data, *Chaco* for enhanced interactive plotting, *MayaVi* for enhanced three-dimensional visualization, *SymPy* for symbolic mathematics, *NetworkX* for graph and network analysis, and many others which are too numerous to list here. A repository of Python modules can be found in the Python Package

Index at http://pypi.python.org. For more information on these and other packages, see the reference list in §A.10.

A.3. Getting started with Python

In this section we will give a brief overview of the main features of the Python syntax, geared toward a newcomer to the language. A complete introduction to the Python language is well beyond the scope of this small section; for this purpose many resources are available both online and in print. For some suggestions, see the references in §A.10.

A.3.1. Installation

Python is open source and available to download for free at http://python.org. One important note when installing Python is that there are currently two branches of Python available: Python 2 and Python 3. Python 3.x includes some useful enhancements, but is not backward compatible with Python 2.x. At the time of this book's publication, many of the numerical and computational packages available for Python have not yet moved to full compatibility with Python 3.x. For that reason, the code in this book is written using the syntax of Python 2.4+, the dependency for the NumPy package. Note, however, that the Matplotlib package currently requires Python 2.5+, and Scikit-learn currently requires Python 2.6+. We recommend installing Python 2.7 if it is available.

Third-party packages such as NumPy and SciPy can be installed by downloading the source code or binaries, but there are also tools which can help to automate the process. One option is `pip`, the Python Package Index installer, which streamlines the process of downloading and installing new modules: for example, NumPy can be installed on a computer with a working C compiler by simply typing `pip install numpy`. Find out more at http://pypi.python.org/. Refer to individual package documentation for installation troubleshooting.

Another good option is to use Linux's Advanced Packaging Tool (a core utility in many Linux distributions), which gives users access to a repository of software, including Python packages that are precompiled and optimized for the particular Linux system, and also keeps track of any dependencies needed for new packages. See your system's documentation for more information.

There are also several third-party Python builds which contain many or all of these tools within a single installation. Using one of these can significantly streamline setting up the above tools on a new system. Some examples of these are PythonXY[3] and Extension Packages[4] for Windows, the SciPy Superpack[5] for Mac, and the Enthought Python Distribution (EPD),[6] and Anaconda[7] for multiple platforms. The latter two are proprietary package distributions, but have free versions available.

[3]http://www.pythonxy.com

[4]http://www.lfd.uci.edu/ gohlke/pythonlibs/

[5]http://fonnesbeck.github.com/ScipySuperpack/

[6]http://enthought.com/

[7]http://continuum.io/

A.3.2. Running a Python script

There are several ways to run a piece of Python code. One way is to save it in a text file and pass this filename as an argument to the Python executable. For example, the following command can be saved to the file hello_world.py:

```
# simple script
print 'hello world'
```

and then executed using the command-line argument `python hello_world.py`. (Note that comments in Python are indicated by the # character: anything after this character on a line will be ignored by the interpreter.) Alternatively, one can simply run `python` with no arguments and enter the interactive shell, where commands can be typed at the >>> prompt:

```
>>> print 'hello world'
hello world
```

In §A.4, we will also introduce the IPython interpreter, which is similar to Python's command-line interface but contains added features.

A.3.3. Variables and operators

Python, like many programming languages, centers around the use of **variables**. Variables can be of several built-in types, which are inferred dynamically (i.e., Python variables do not require any type information in declaration):

```
>>> x = 2                   # integer value
>>> pi = 3.1415             # floating-point variable
>>> label = "Addition:"     # string variable
>>> print label, x, '+', pi, "=", (x + pi)
Addition: 2 + 3.1415 = 5.1415
>>>
>>> label = 'Division:'
>>> print label, x, '/', pi, "=", (x / pi)
Division: 2 / 3.1415 = 0.636638548464
```

Variable names are case sensitive, so that `pi` and `PI` may refer to two different objects. Note the seamless use of integers, floating-point (decimal) values, strings (indicated by single quotes `'...'` or double-quotes `"..."`), and simple arithmetic expressions (`+`, `-`, `*`, and `/`) behave as expected.[8]

[8]One potential point of confusion is the division operator with integers: in Python 2.x, if two integers are divided, an integer is always returned, ignoring any remainder. This can be remedied by explicitly converting one value to floating point, using `float(x)`. Alternatively, to use the Python 3 semantics where integer division returns a float, you may use the statement `from future import __division__`

Also available are the ** operator for exponentiation, and the modulus operator % which finds the remainder between two numbers:

```
>>> 2 ** 3  # exponentiation
8
>>> 7 % 3  # modulus (remainder after division)
1
```

Python also provides the operators +=, -=, /=, *=, **=, and %=, which combine arithmetic operations with assignment:

```
>>> x = 4
>>> x += 2  # equivalent to x = x + 2
>>> x
6
```

A.3.4. Container objects

Besides numeric and string variables, Python also provides several built-in container types, which include **lists**, **tuples**, **sets**, and **dictionaries**.

Lists

A list holds an ordered sequence of objects, which can be accessed via the zero-based item index using square brackets [ ]:

```
>>> L = [1, 2, 3]
>>> L[0]
1
```

Lists items can be any mix of types, which makes them very flexible:

```
>>> pi = 3.14
>>> L = [5, 'dog', pi]
>>> L[-1]
3.14
```

Here we have used a negative index to access items at the end of the list. Another useful list operation is **slicing**, which allows access to sublists. Slices can start and end anywhere in the list, and can be contiguous or noncontiguous:

```
>>> L = ['Larry', 'Goran', 'Curly', 'Peter', 'Paul',
        'Mary']
>>> L[0:3]  # slice containing the first three items
['Larry', 'Goran', 'Curly']
```

```
>>> L[:3]  # same as above: the zero is implied
['Larry', 'Goran', 'Curly']
>>> L[-2:]  # last two items
['Paul', 'Mary']
>>> L[1:4]  # items 1 (inclusive) through 4
       # (non-inclusive)
['Goran', 'Curly', 'Peter']
>>> L[::2]  # every second item
['Larry', 'Curly', 'Paul']
>>> L[::-1]  # all items, in reverse order
['Mary', 'Paul', 'Peter', 'Curly', 'Goran', 'Larry']
```

A general slice is of the form `[start:stop:stride]`. `start` defaults to 0, `stop` defaults to -1, and `stride`, which indicates the number of steps to take between each new element, defaults to 1. Slicing will become even more important when working with N-dimensional NumPy arrays below.

Tuples

Tuples are similar to lists, but are indicated by parentheses `(1, 2, 3)` rather than square brackets `[1, 2, 3]`. They support item access and slicing using the same syntax as lists. The primary difference is that tuples are immutable: once they're created the items in them cannot be changed. Tuples are most commonly used in functions which return multiple values.

Sets

Sets, available since Python 2.6, act as unordered lists in which items are not repeated. They can be very convenient to use in circumstances where no repetition of elements is desired:

```
>>> S = set([1, 1, 3, 2, 1, 3])
>>> S
set([1, 2, 3])
```

Dictionaries

A dictionary is another container type, which stores an unordered sequence of key-value pairs. It can be defined using curly brackets `{}`, and like lists allows mixes of types:

```
>>> D = {'a':1, 'b':2.5, 'L':[1, 2, 3]}
>>> D['a']
1
>>> D['L']
[1, 2, 3]
```

Internally, Python uses dictionaries for many built-in aspects of the language: for example, the function `globals()` will return a dictionary object containing all currently defined global variables.

A.3.5. Functions

For operations which will be repeatedly executed, it is often convenient to define a **function** which implements the desired operation. A function can optionally accept one or several **parameters**, and can optionally return a computed value:

```
>>> def convert_to_radians(deg):
...     pi = 3.141592653
...     return deg * pi / 180.0
...
>>> convert_to_radians(90)
1.5707963265
```

Notice the key elements of a function definition: the `def` keyword, the function name, the arguments in parentheses `()`, the colon `:` marking the beginning of a code block, the **local variable** `pi` defined in the function, and the optional `return` statement which returns a computed value to the point of the function call. Here we see our first use of **indentation** in Python: unlike many languages, *white space in Python has meaning*. This can be difficult to become accustomed to for a programmer with background in other languages, but Python proponents point out that this feature can make code very easy to read and write. Indentation in Python can consist of tabs or any number of spaces; standard practice is to use four spaces, and to never use tabs.

Python has many built-in functions which implement common tasks. For example, it is often convenient to be able to quickly define a sequential list of integers: Python allows this through the built-in `range` function:

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

Notice that `range(N)` starts at zero and has `N` elements, and therefore does not include `N` itself. There are a large number of other useful built-in functions: for more information, refer to the references listed in §A.10.

A.3.6. Logical operators

Another of Python's built-in object types are the boolean values, `True` and `False` (case sensitive). A number of boolean operators are available, as specified in table A.1.

Nonboolean variables can be coerced into boolean types: for example, a nonzero integer evaluates to `True`, while a zero integer evaluates to `False`; an empty string

TABLE A.1.
Boolean operators.

Operation	Description
`x or y`	return `True` if either `x` or `y`, or both evaluate to `True`; otherwise return `False`
`x and y`	return `True` only if both `x` and `y` evaluate to `True`; otherwise return `False`
`not x`	return `True` only if `x` is `False`; otherwise return `False`.

TABLE A.2.
Comparison expressions.

Operator	Description	Operator	Description
<	Strictly less than	<=	Less than or equal
>	Strictly greater than	>=	Greater than or equal
==	Equal	!=	Not Equal

evaluates to `False` in a boolean expression, while a nonempty string evaluates to `True`:

```
>>> bool(''), bool('hello')
(False, True)
```

Hand in hand with the boolean operators are the comparison expressions, which are summarized in table A.2.

As a simple example, one can use comparisons and boolean expressions in combination:

```
>>> x = 2
>>> y = 4
>>> (x == 2) and (y >= 3)
True
```

These boolean expressions become very important when used with control flow statements, which we will explore next.

A.3.7. Control flow

Control flow statements include **conditionals** and **loops** which allow the programmer to control the order of code execution.

For conditional statements, Python implements the `if`, `elif`, and `else` commands:

```
>>> def check_value(x):
...     if x < 0:
...         return 'negative'
...     elif x == 0:
...         return 'zero'
...     else:
...         return 'positive'
...
>>> check_value(0)
zero
>>> check_value(123.4)
positive
```

The keyword elif is a contraction of else if, and allows multiple conditionals in series without excessive indentation. There can be any number of elif statements strung together, and the series may or may not end with else.

Python contains two types of loop statements: for loops and while loops. The syntax of the for loop is as follows:

```
>>> for drink in ['coffee', 'slivovitz', 'water']:
...     print drink
...
coffee
slivovitz
water
```

In practice, for loops are often used with the built-in range function, which was introduced above:

```
>>> words = ['twinkle', 'twinkle', 'little', 'star']
>>> for i in range(4):
...     print i, words[i]
...
0 twinkle
1 twinkle
2 little
3 star
```

The second type of loop, while loops, operate similarly to while loops in other languages:

```
>>> i = 0
>>> while i < 10:
...     i += 3
...
>>> i
12
```

In loops, it can often be useful to skip the remainder of a loop iteration or to break out of the loop. These tasks can be accomplished using the continue and break statements:

```
>>> i = 0
>>> while True:
...     i += 3
...     if i > 10:
...         break  # break out of the loop
...
>>> print i
12
>>> for i in range(6):
...     if i % 2 == 0:
...         continue  # continue from beginning of
            # loop block
...     print i, 'only odd numbers get here'
...
1 only odd numbers get here
3 only odd numbers get here
5 only odd numbers get here
```

Another useful statement is the pass statement, which is a null operation that can be useful as a placeholder:

```
>>> word = 'python'
>>> for char in word:
...    if char == 'o':
...        pass  # this does nothing
...    else:
...        print char,

p y t h n
```

A.3.8. Exceptions and exception handling

When something goes wrong in a Python script, an **exception** is raised. This can happen, for example, when a list index is out of range, or when a mathematical expression is undefined:

```
>>> L = [1, 2, 3]
>>> L[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> 0 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

In these cases, it is often useful to **catch** the exceptions in order to decide what to do. This is accomplished with the `try`, `except` statement:

```
>>> def safe_getitem(L, i):
...     try:
...         return L[i]
...     except IndexError:
...         return 'index out of range'
...     except TypeError:
...         return 'index of wrong type'
...
>>> L = [1, 2, 3]
>>> safe_getitem(L, 1)
2
>>> safe_getitem(L, 100)
index out of range
>>> safe_getitem(L, 'cat')
index of wrong type
```

In addition to `try` and `except`, the `else` and `finally` keywords can be used to fine-tune the behavior of your exception handling blocks. For more information on the meaning of these keywords, see the language references in §A.10.

In your own code, you may wish to raise an exception under certain circumstances. This can be done with the `raise` keyword:

```
>>> def laugh(n):
...     if n <= 0:
...         raise ValueError('n must be positive')
...     else:
...         return n * 'ha! '
>>> laugh(6)
ha! ha! ha! ha! ha! ha!
>>> laugh(-2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: n must be positive
```

Python has numerous built-in exception types, and users may also define custom exception types for their own projects.

A.3.9. Modules and packages

A key component of the extensibility of Python is the existence of **modules** and **packages**, which may be built in or defined by the user. A module is a collection of functions and variables, and a package is a collection of modules. These can be accessed using the import statement. There are several useful built-in modules in Python, for example the math module can be accessed this way:

```
>>> import math
>>> math.sqrt(2)
1.41421356237
```

Alternatively, one can import specific variables, functions, or classes from modules using the from, import idiom:

```
>>> from math import pi, sin
>>> sin(pi / 2)
1.0
```

At times, you'll see the use of, for example, from math import *, which imports everything in the module into the global namespace. This should generally be avoided, because it can easily lead to name conflicts if multiple modules or packages are used. More information on any module can be found by calling the built-in help function with the module as the argument:

```
>>> help(math)
```

Another useful function is the dir() function, which lists all the attributes of a module or object. For example, typing the following will list all the operations available in the math module:

```
>>> dir(math)
```

A.3.10. Objects in Python

Python can be used as an object-oriented language, and the basic building blocks of the language are objects. Every variable in Python is an object. A variable containing an integer is simply an object of type int; a list is simply an object of type list. Any object may have **attributes**, **methods**, and/or **properties** associated with it. Attributes and methods are analogous to variables and functions, respectively, except that attributes and methods are associated with particular objects. Properties act as a hybrid of an attribute and a method, and we won't discuss them further here.

As an example of using attributes and methods, we'll look at the complex data type:

```
>>> c = complex(1.0, 2.0)
>>> c
(1+2j)
>>> c.real  # attribute
```

```
1.0
>>> c.imag  # attribute
2.0
>>> c.conjugate()  # method
(1-2j)
```

Here we have created a complex number c, and accessed two of its attributes, `c.real` (the real part) and `c.imag` (the imaginary part). These attributes are variables which are associated with the object c. Similarly, we call the method `c.conjugate()`, which computes the complex conjugate of the number. This method is a function which is associated with the object c.

Most built-in object types have methods which can be used to view or modify the objects. For example, the `append` method of `list` objects can be used to extend the array:

```
>>> L = [4, 5, 6]
>>> L.append(8)
>>> L
[4, 5, 6, 8]
```

and the `sort` method can be used to sort a list in place:

```
>>> numbers = [5, 2, 6, 3]
>>> numbers.sort()
>>> numbers
[2, 3, 5, 6]
>>> words = ['we', 'will', 'alphabetize', 'these',
    'words']
>>> words.sort()
>>> words
['alphabetize', 'these', 'we', 'will', 'words']
```

To learn more about other useful attributes and methods of built-in Python types, use the `help()` or `dir()` commands mentioned above, or see the references in §A.10.

A.3.11. Classes: user-defined objects

Python provides a syntax for users to create their own object types. These `class` objects can be defined in the following way:

```
>>> class MyObject(object):  # derive from the
    # base-class `object`
...     def __init__(self, x):
...         print 'initializing with x =', x
...         self.x = x
...
...     def x_squared(self):
```

```
...         return self.x ** 2
...
>>> obj = MyObject(4)
initializing with x = 4
>>> obj.x  # access attribute
4
>>> obj.x_squared()  # invoke method
16
```

The special method `__init__()` is what is called when the object is initialized. Its first argument is the object itself, which by convention is named `self`. The remaining arguments are up to the user. Special methods like `__init__` are marked by the double underscore, and have specific purposes. In addition, the user can define any number of custom attributes and methods.

This class interface in Python is very powerful: it allows Python scripts and applications to make use of *inheritance* and other object-oriented design principles, which if used well can make code both very flexible and very easy to write, read, and understand. An introduction to object-oriented design principles is beyond the scope of this appendix, but an excellent introduction can be found in several of the references in §A.10.

A.3.12. Documentation strings

One key feature of Python objects is that they can have built-in documentation strings, usually referred to as *docstrings*. Docstrings can be accessed by calling the `help()` function on the object, as we saw above. When writing your own code, it is good practice to write good docstrings. This is accomplished by creating a string (most often using the triple quote `"""` to indicate a multiline string) in the first line of the function:

```
>>> def radians(deg):
...     """
...     Convert an angle to radians
...     """
...     pi = 3.141592653
...     return deg * pi / 180.0
...
>>> help(radians)
```

This help command will open a text reader showing the docstring defined at the top of the function.

A.3.13. Summary

In this section, we have given a brief introduction to the basic syntax of Python. In the next section, we will introduce some of the powerful open-source scientific-computing tools which are built on this framework. First, though, we will highlight some features of IPython which will make our job easier.

A.4. IPython: the basics of interactive computing

In the example above, we used Python's standard interactive computing environment for line-by-line interpreted coding. For the remainder of this introduction, we'll switch to using IPython's enhanced interactive interface. There are many useful features of IPython, but we'll just mention a few of the most important ones here. The IPython environment can be started by running `ipython` with no arguments: the default prompt is different than that of the normal Python interpreter, but acts similarly:

```
In [1]: print 'hello world'
hello world
In [2]: 4 + 6
Out[2]: 10
```

A.4.1. Command history

IPython stores the command history in the special global variables `In` and `Out`. So, for example, to see the first command which was typed above, we can simply use `In`:

```
In [3]: print In[1]
print 'hello world'
```

Any output is also stored. So, the result of our input in line 2, can be recovered:

```
In [4]: print Out[2]
10
```

A.4.2. Command completion

IPython allows tab completion of commands, both for commands in the user history, and for arbitrary commands in the namespace. For example, if you type `for i in ran` and then type the tab key, IPython will fill in the rest of the command, giving you `for i in range`. If there are multiple completion possibilities, it will display a list of those possibilities:

```
In [5]: x = ra<TAB>
raise      range      raw_input
```

Similarly, using the up arrow allows you to cycle through your command history. Typing a partial command followed by the up arrow will cycle through any commands which start with those characters. For example, in the current session, typing the partial command `prin` followed by the up arrow will automatically fill in the command `print Out[2]`, the most recent statement which begins with those characters. Typing the up arrow again will change this to `print In[1]` and then `print ‘hello world’`. In interactive coding, when complicated commands are often repeated, this command completion feature is very helpful.

A.4.3. Help and documentation

Above we saw how the `help()` function can be used to access any objects docstring. In IPython, this can be accomplished using the special character `?`. For example, the documentation of the `range` function is simply displayed:

```
In [6]: range?
Type:       builtin_function_or_method
String Form:<built-in function range>
Namespace:  ipython builtin
Docstring:
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.
```

For pure Python functions or classes (i.e., ones which are not derived from compiled libraries), the double question mark `??` allows one to examine the source code itself:

```
In [7]: def myfunc(x):
   ...:     return 2 * x
   ...:
In [8]: myfunc??  # note: no space between object
        # name and ??
Type:       function
String Form:<function myfunc at 0x2a6c140>
File:       /home/<ipython-input-15-21fdc7f0ea27>
Definition: myfunc(x)
Source:
def myfunc(x):
    return 2 * x
```

This can be very useful when exploring functions and classes from third-party modules.

A.4.4. Magic functions

The IPython interpreter also includes **magic functions** which give shortcuts to useful operations. They are marked by a % sign, and are too numerous to list here. For example, the %run command allows the code within a file to be run from within IPython. We can run the hello_world.py file created above as follows:

```
In [9]: %run hello_world.py
hello world
```

There are many more such magic functions, and we will encounter a few of them below. You can see a list of what is available by typing % followed by the tab key. The IPython documentation features also work for magic functions, so that documentation can be viewed by typing, for example, %run?.

In addition to the interactive terminal, IPython provides some powerful tools for parallel processing; for creating interactive html notebooks combining code, formatted text, and graphics; and much more. For more information about IPython, see the references in §A.10, especially IPython's online documentation and tutorials. For a well-written practical introduction to IPython, we especially recommend the book *Python for Data Analysis*, listed in §A.10.

A.5. Introduction to NumPy

In this section, we will give an introduction to the core concepts of scientific computing in Python, which is primarily built around the NumPy array. Becoming fluent in the methods and operations of NumPy arrays is vital to writing effective and efficient Python code. This section is too short to contain a complete introduction, and the interested reader should make use of the references at the end of this appendix for more information. NumPy is contained in the numpy module, which by convention is often imported under the shortened name np.

A.5.1. Array creation

NumPy arrays are objects of the type np.ndarray, though this type specifier is rarely used directly. Instead there are several array creation routines that are often used:

```
In [1]: import numpy as np

In [2]: np.array([5, 2, 6, 7])  # create an array
        # from a python list
Out[2]: array([5, 2, 6, 7])

In [3]: np.arange(4)  # similar to the built-in
        # range() function
Out[3]: array([0, 1, 2, 3])
```

```
In [4]: np.linspace(1, 2, 5)  # 5 evenly-spaced steps
        # from 1 to 2
Out[4]: array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])

In [5]: np.zeros(5)  # array of zeros
Out[5]: array([ 0.,  0.,  0.,  0.,  0.])

In [6]: np.ones(6)  # array of ones
Out[6]: array([ 1.,  1.,  1.,  1.,  1.,  1.])
```

Arrays can also be multidimensional:

```
In [7]: np.zeros((2, 4))  # 2 X 4 array of zeros
Out[7]:
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

NumPy also has tools for random array creation:

```
In [9]: np.random.random(size=4)
       # uniform between 0 and 1
Out[9]: array([ 0.28565   ,  0.3614929 ,
        0.95431006,  0.24266193])

In [10]: np.random.normal(size=4) # standard-norm
         # distributed
Out[10]: array([-0.62332252,  0.09098354,
       0.40975753,  0.53407146])
```

Other array creation routines are available, but the examples above summarize the most useful options. With any of them, the ? functionality of IPython can be used to determine the usage and options.

A.5.2. Element access

Elements of arrays can be accessed using the same indexing and slicing syntax as in lists:

```
In [11]: x = np.arange(10)

In [12]: x[0]
Out[12]: 0

In [13]: x[::2]
Out[13]: array([0, 2, 4, 6, 8])
```

Note that array slices are implemented as **views** rather than copies of the original array, so that operations such as the following are possible:

```
In [14]: x = np.zeros(8)  # array of eight zeros

In [15]: x[::2] = np.arange(1, 5) # [::2] selects
        # every 2nd entry

In [16]: x
Out[16]: array([ 1.,  0.,  2.,  0.,  3.,  0.,  4.,
        0.])
```

For multidimensional arrays, element access uses multiple indices:

```
In [17]: X = np.zeros((2, 2))  # 2 X 2 array of zeros

In [18]: X[0, 0] = 1

In [19]: X[0, 1] = 2

In [20]: X
Out[20]:
array([[ 1.,  2.],
       [ 0.,  0.]])

In [21]: X[0, :]  # first row of X. X[0] is
         # equivalent
Out[21]: array([ 1.,  2.])

In [22]: X[:, 1]  # second column of X.
Out[22]: array([ 2.,  0.])
```

Combinations of the above operations allow very complicated arrays to be created easily and efficiently.

A.5.3. Array data type

NumPy arrays are typed objects, and often the type of the array is very important. This can be controlled using the `dtype` parameter. The following example is on a 64-bit machine:

```
In [23]: x = np.zeros(4)

In [24]: x.dtype
Out[24]: dtype('float64')

In [25]: y = np.zeros(4, dtype=int)
In [26]: y.dtype
Out[26]: dtype('int64')
```

Allocating arrays of the correct type can be very important; for example, a floating-point number assigned to an integer array will have its decimal part truncated:

```
In [27]: x = np.zeros(1, dtype=int)

In [28]: x[0] = 3.14  # converted to an integer!

In [29]: x[0]
Out[29]: 3
```

Other possible values for dtype abound: for example, you might use bool for boolean values or complex for complex numbers. When we discuss *structured arrays* below, we'll see that even more sophisticated data types are possible.

A.5.4. Universal functions and broadcasting

A powerful feature of NumPy is the concept of a ufunc, short for "universal function." These functions operate on every value of an array. For example, trigonometric functions are implemented in NumPy as **unary ufuncs**—ufuncs with a single parameter:

```
In [30]: x = np.arange(3)  # [0, 1, 2]

In [31]: np.sin(x)  # take the sine of each element
         # of x
Out[31]: array([ 0.        ,  0.84147098,  0.90929743])
```

Arithmetic operations like +, -, *, / are implemented as **binary ufuncs**—ufuncs with two parameters:

```
In [32]: x * x  # multiply each element of x by
         # itself
Out[32]: array([0, 1, 4])
```

Ufuncs can also operate between arrays and scalars, applying the operation and the scalar to each array value:

```
In [33]: x + 5  # add 5 to each element of x
Out[33]: array([5, 6, 7])
```

This is a simple example of **broadcasting**, where one argument of a ufunc has a shape upgraded to the shape of the other argument. A more complicated example of broadcasting is adding a vector to a matrix:

```
In [34]: x = np.ones((3, 3))  # 3 X 3 array of ones

In [35]: y = np.arange(3)  # [0, 1, 2]

In [36]: x + y  # add y to each row of x
Out[36]:
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Sometimes, both arrays are broadcast at the same time:

```
In [37]: x = np.arange(3)

In [38]: y = x.reshape((3, 1))  # create a 3 X 1
         # column array

In [39]: x + y
Out[39]:
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Here a row array and a column array are added together, and both are broadcast to complete the operation. A visualization of these broadcasting operations is shown in figure A.1.

Broadcasting can seem complicated, but it follows simple rules:

1. If the two arrays differ in their number of dimensions, the shape of the array with fewer dimensions is padded with ones on its leading (left) side.
2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is broadcast up to match the other shape.
3. If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

All of this takes place efficiently without actually allocating any extra memory for these temporary arrays. Broadcasting in conjunction with universal functions allows some very fast and flexible operations on multidimensional arrays.

A.5.5. Structured arrays

Often data sets contain a mix of variables of different types. For this purpose, NumPy contains **structured arrays**, which store compound data types. For example, imagine we have a set of data where each object has an integer ID, a five-character string

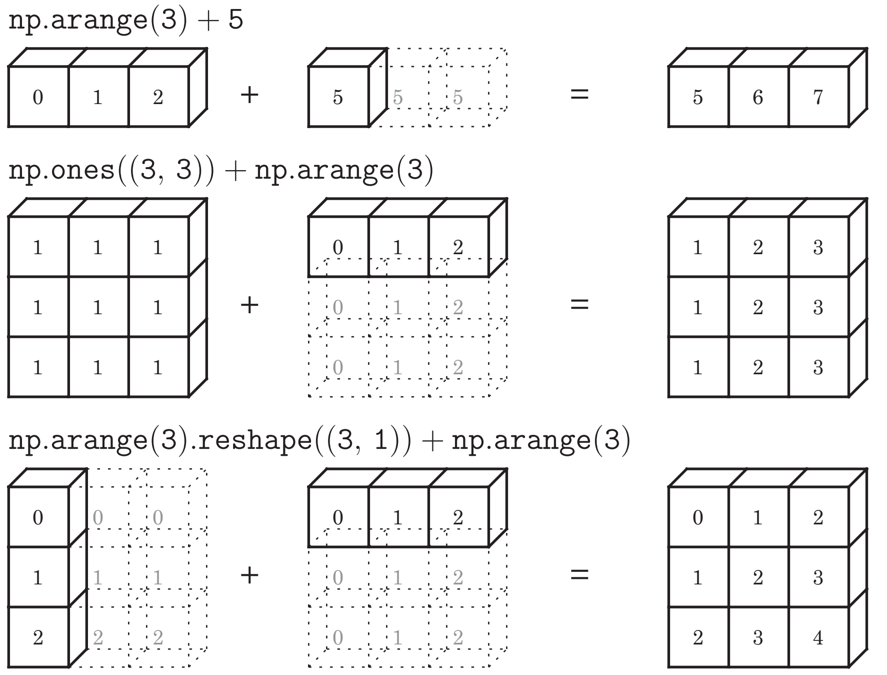

Figure A.1. A visualization of NumPy array broadcasting. Note that the extra memory indicated by the dotted boxes is never allocated, but it can be convenient to think about the operations as if it is.

name,[9] and a floating-point measurement. We can store this all in one array:

```
In [40]: dtype = [('ID', int), ('name', (str, 5)),
                  ('value', float)]

In [41]: data = np.zeros(3, dtype=dtype)

In [42]: print data
[(0, '', 0.0) (0, '', 0.0) (0, '', 0.0)]
```

The fields of the array (it may be convenient to think of them as "columns") can be accessed via the field name:

```
In [43]: data['ID' ] = [154, 207, 669]

In [44]: data['name'] = ['obj_1', 'obj_2', 'obj_3']

In [45]: data['value'] = [0.1, 0.2, 0.3]
```

[9]Because NumPy arrays use fixed memory blocks, we will also need to specify the maximum length of the string argument.

```
In [46]: print data[0]
(154, 'obj_1', 0.1)

In [47]: print data['value']
[ 0.1  0.2  0.3]
```

The data sets used in this book are loaded into structured arrays for convenient data manipulation.

A.5.6. Summary

This has been just a brief overview of the fundamental object for most scientific computing in Python: the NumPy array. The references at the end of the appendix include much more detailed tutorials on all the powerful features of NumPy array manipulation.

A.6. Visualization with Matplotlib

Matplotlib offers a well-supported framework for producing publication-quality plots using Python and NumPy. Every figure in this book has been created using Matplotlib, and the source code for each is available on the website http://www.astroML.org. Here we will show some basic plotting examples to get the reader started.

IPython is built to interact cleanly with Matplotlib. The magic command `%pylab` allows figures to be created and manipulated interactively, and we will use the `pylab` environment in the examples below. Here is how we'll prepare the session:

```
In [1]: %pylab
Welcome to pylab, a matplotlib-based Python
environment [backend: TkAgg].
For more information, type 'help(pylab)'.

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt
```

Lines 2 and 3 above are not strictly necessary: the `pylab` environment includes these, but we explicitly show them here for clarity.

Matplotlib figures center around the `figure` and `axes` objects. A figure is essentially a plot window, and can contain any number of axes. An axes object is a subwindow which can contain lines, images, shapes, text, and other graphical elements. Figures and axes can be created by the `plt.figure` and `plt.axes` objects, but the `pylab` interface takes care of these details in the background.

A useful command to be aware of is the `plt.show()` command. It should be called once at the end of a script to tell the program to keep the plot open before terminating the program. It is not needed when using interactive plotting in IPython.

Let's start by creating and labeling a simple plot:[10]

```
In [4]: x = np.linspace(0, 2 * np.pi, 1000)
        # 1000 values from 0 to 2pi

In [5]: y = np.sin(x)

In [6]: ax = plt.axes()

In [7]: ax.plot(x, y)

In [8]: ax.set_xlim(0, 2 * np.pi)  # set x limits

In [9]: ax.set_ylim(-1.3, 1.3)  # set y limits

In [10]: ax.set_xlabel('x')

In [11]: ax.set_ylabel('y')

In [12]: ax.set_title('Simple Sinusoid Plot')
```

The resulting plot is shown in figure A.2. The style of the line (dashed, dotted, etc.) or the presence of markers at each point (circles, squares, triangles, etc.) can be controlled using parameters in the `plt.plot()` command. Use `plt.plot?` in IPython to see the documentation, and experiment with the possibilities.

Another useful plot type is the error bar plot. Matplotlib implements this using `plt.errorbar`:

```
In [13]: x_obs = 2 * np.pi * np.random.random(50)

In [14]: y_obs = np.sin(x_obs)

In [15]: y_obs += np.random.normal(0, 0.1, 50)
        # add some error

In [16]: ax.errorbar(x_obs, y_obs, 0.1, fmt='.',
         color='black')
```

The result is shown in figure A.3. We have used the same axes object to overplot these points on the previous line. Notice that we have set the format to `'.'`, indicating a single point for each item, and set the color to `'black'`.

[10]Here we will use Matplotlib's object/method interface to plotting. There is also a MATLAB-style interface available, where for example `plt.plot(x, y)` could be called directly. This is sometimes convenient, but is less powerful. For more information, see the references in §A.10.

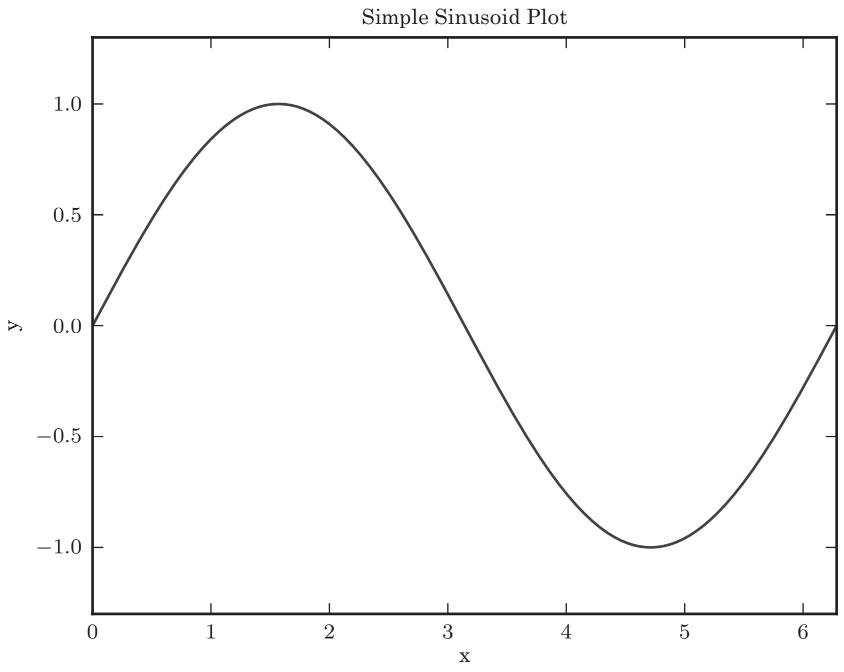

Figure A.2. Output of the simple plotting example.

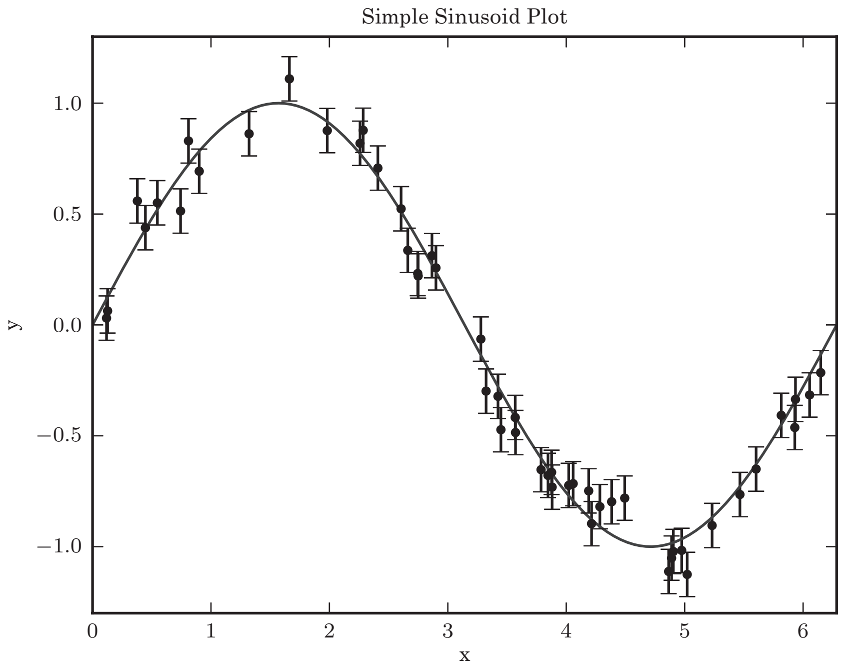

Figure A.3. Output of the error bar plotting example.

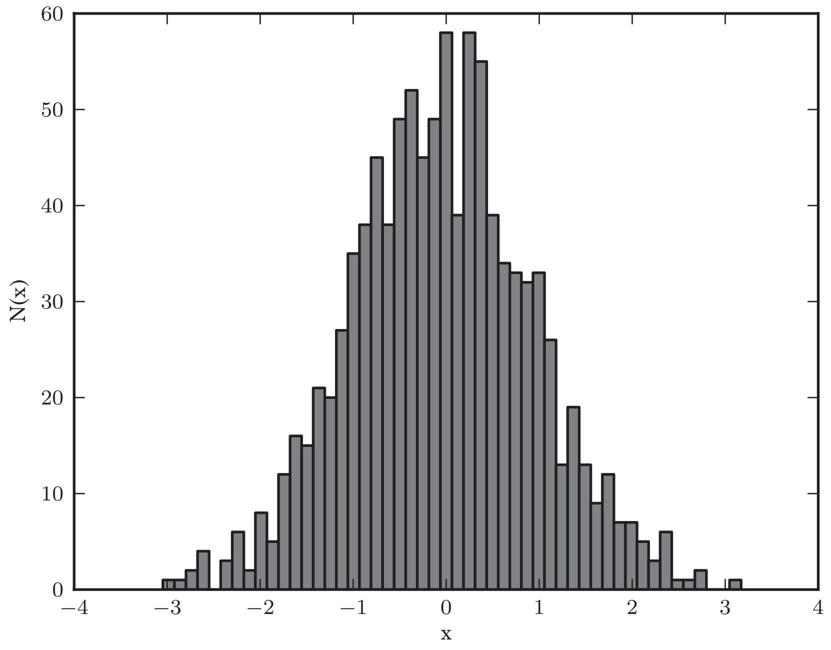

Figure A.4. Output of the histogram plotting example.

Another basic type of plot that is very useful is the histogram. This is implemented in Matplotlib through the `ax.hist()` command:

```
In [17]: fig = plt.figure()  # create a new figure
        # window

In [18]: ax = plt.axes()  # create a new axes

In [19]: x = np.random.normal(size=1000)

In [20]: ax.hist(x, bins=50)

In [21]: ax.set_xlabel('x')

In [22]: ax.set_ylabel('N(x)')
```

The result is shown in figure A.4. Again, there are many options to control the appearance of the histogram. Use `plt.hist?` in IPython to access the histogram documentation for details.

Matplotlib has many more features, including tools to create scatter plots (`plt.scatter`), display images (`plt.imshow`), display contour plots (`plt.contour` and `plt.contourf`), create hexagonal tessellations (`plt.hexbin`), draw filled regions (`plt.fill` and `plt.fill_between`), create multiple subplots (`plt.subplot` and `plt.axes`), and much more. For information about these routines, use the help feature in IPython. For additional documentation and examples, the main Matplotlib website (especially the *gallery* section) is a very useful resource: http://matplotlib.org. Additionally, the reader may wish to browse the source code of the figures in this book, available at http://www.astroML.org.

A.7. Overview of useful NumPy/SciPy modules

NumPy and SciPy contain a large collection of useful routines and tools, and we will briefly summarize some of them here. More information can be found in the references at the end of this appendix.

A.7.1. Reading and writing data

NumPy has tools for reading and writing both text-based data and binary format data. To save an array to an ASCII file, use `np.savetxt`; text files can be loaded using `np.loadtxt`:

```
In [1]: import numpy as np

In [2]: x = np.random.random((10, 3))

In [3]: np.savetxt('x.txt', x)

In [4]: y = np.loadtxt('x.txt')

In [5]: np.all(x == y)
Out[5]: True
```

A more customizable text loader can be found in `np.genfromtxt`. Arrays can similarly be written to binary files using `np.save` and `np.load` for single arrays, or `np.savez` to store multiple arrays in a single zipped file. More information is available in the documentation of each of these functions.

A.7.2. Pseudorandom number generation

NumPy and SciPy provide powerful tools for pseudorandom number generation. They are based on the Mersenne twister algorithm [5], one of the most sophisticated and well-tested pseudorandom number algorithms available. The `np.random` submodule contains routines for random number generation from a variety of distributions. For reproducible results, the random seed can explicitly be set:

```
In [6]: np.random.seed(0)

In [7]: np.random.random(4) # uniform between 0 and 1
Out[7]: array([ 0.5488135 ,  0.71518937,
        0.60276338,  0.54488318])
```

```
In [8]: np.random.normal(loc=0, scale=1, size=4)
        # standard norm
Out[8]: array([ 1.86755799, -0.97727788,
        0.95008842, -0.15135721])

In [9]: np.random.randint(low=0, high=10, size=4)
        # random integers
Out[9]: array([8, 1, 6, 7])
```

Many other distributions are available as well, and are listed in the documentation of `np.random`. Another convenient way to generate random variables is using the `distributions` submodule in `scipy.stats`:

```
In [10]: from scipy.stats import distributions

In [11]: distributions.poisson.rvs(10, size=4)
Out[11]: array([10, 16,  5, 13])
```

Here we have used the `poisson` object to generate random variables from a Poisson distribution with $\mu = 10$. The distribution objects in `scipy.stats` offer much more than just random number generation: they contain tools to compute the probability density, the cumulative probability density, the mean, median, standard deviation, *n*th moments, and much more. For more information, type `scipy.distributions?` in IPython to view the module documentation. See also the many distribution examples in chapter 3.

A.7.3. Linear algebra

NumPy and SciPy contain some powerful tools for efficient linear algebraic computations. The basic dot product is implemented in NumPy:

```
In [12]: M = np.range(9).reshape((3, 3))

In [13]: M
Out[13]:
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

In [14]: x = np.arange(1, 4)  # [1, 2, 3]

In [15]: np.dot(M, x)
Out[15]: array([ 8, 26, 44])
```

Many common linear algebraic operations are available in the submodules `numpy.linalg` and `scipy.linalg`. It is possible to compute matrix inverses (`linalg.inv`), singular value decompositions (`linalg.svd`), eigenvalue decompositions (`linalg.eig`), least-squares solutions (`linalg.lstsq`) and much more. The linear algebra submodules in NumPy and SciPy have many duplicate routines, and have a common interface. The SciPy versions of the routines often have more options to control output, and `scipy.linalg` contains a number of less common algorithms that are not a part of `numpy.linalg`. For more information, enter `numpy.linalg?` or `scipy.linalg?` in IPython to see the documentation.

A.7.4. Fast Fourier transforms

The fast Fourier transform is an important algorithm in many aspects of data analysis, and these routines are available in both NumPy (`numpy.fft` submodule) and SciPy (`scipy.fftpack` submodule). Like the linear algebra routines discussed above, the two implementations differ slightly, and the SciPy version has more options to control the results.

A simple example is as follows:

```
In [16]: x = np.ones(4)

In [17]: np.fft.fft(x)  # forward transform
Out[17]: array([ 4.+0.j,  0.+0.j,  0.+0.j,  0.+0.j])

In [18]: np.fft.ifft(Out[17])  # inverse recovers
         # input
Out[18]: array([ 1.+0.j,  1.+0.j,  1.+0.j,  1.+0.j])
```

The forward transform is implemented with `fft()`, while the inverse is implemented with `ifft`. To learn more, use `scipy.fftpack?` or `numpy.fft?` to view the documentation. See appendix E for some more discussion and examples of Fourier transforms.

A.7.5. Numerical integration

Often it is useful to perform numerical integration of numerical functions. The literature on numerical integration techniques is immense, and SciPy implements many of the more popular routines in the module `scipy.integrate`. Here we'll use a function derived from QUADPACK, an optimized numerical integration package written in Fortran, to integrate the function $\sin x$ from 0 to π:

```
In [19]: from scipy import integrate

In [20]: result, error = integrate.quad(np.sin, 0,
         np.pi)

In [21]: result, error
Out[21]: (2.0, 2.220446049250313e-14)
```

As expected, the algorithm finds that $\int_0^\pi \sin x\,dx = 2$. The integrators accept any function of one variable as the first argument, even one defined by the user. Other integration options (including double and triple integration) are also available. Use `integrate?` in IPython to see the documentation for more details.

A.7.6. Optimization

Numerical optimization (i.e., function minimization) is an important aspect of many data analysis problems. Like numerical integration, the literature on optimization methods is vast. SciPy implements many of the most common and powerful techniques in the submodule `scipy.optimize`. Here we'll find the minimum of a simple function using `fmin`, which implements a Nelder–Mead simplex algorithm:

```
In [22]: from scipy import optimize

In [23]: def simple_quadratic(x):
    ...:     return x ** 2 + x

In [24]: optimize.fmin(simple_quadratic, x0=100)
Optimization terminated successfully.
         Current function value: -0.250000
         Iterations: 24
         Function evaluations: 48
Out[24]: array([-0.50003052])
```

The optimization converges to near the expected minimum value of $x = -0.5$. There are several more sophisticated optimizers available which can take into account gradients and other information: for more details use `optimize?` in IPython to see the documentation for the module.

A.7.7. Interpolation

Another common necessity in data analysis is interpolation discretely sampled data. There are many algorithms in the literature, and SciPy contains the submodule `scipy.interpolate` which implements many of these, from simple linear and polynomial interpolation to more sophisticated spline-based techniques.

Here is a brief example of using a univariate spline to interpolate between points. We'll start by enabling the interactive plotting mode in IPython:

```
In [26]: %pylab
Welcome to pylab, a matplotlib-based Python
environment [backend: TkAgg].
For more information, type 'help(pylab)'.

In [27]: from scipy import interpolate

In [28]: x = np.linspace(0, 16, 30)
         # coarse grid: 30 pts

In [29]: y = np.sin(x)
```

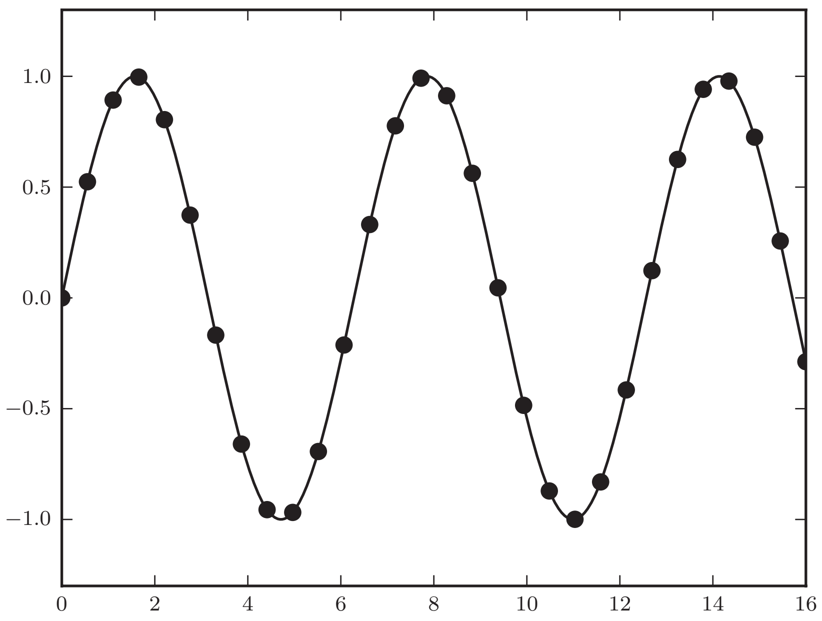

Figure A.5. Output of the interpolation plotting example.

```
In [30]: x2 = np.linspace(0, 16, 1000)
        # fine grid: 1000 pts

In [31]: spl = interpolate.UnivariateSpline(x, y, s=0)

In [32]: ax = plt.axes()

In [33]: ax.plot(x, y, 'o') # 'o' means draw points
        as circles

In [34]: ax.plot(x2, spl(x2), '-')
         # '-' means draw a line
```

The result is shown in figure A.5. The spline fit has interpolated the sampled points to create a smooth curve. There are many additional options for spline interpolation, including the adjustment of smoothing factors, number of knots, weighting of points, spline degree, and more. Linear and polynomial interpolation is also available, using the function `interpolate.interp1d`. For more information, type `interpolate?` in IPython to view the documentation.

A.7.8. Other submodules

There is much more functionality in SciPy, and it can be explored using the online documentation as well as the IPython help functions. Here is a partial list of remaining packages that readers might find useful:

- `scipy.spatial`: distance and spatial functions, nearest neighbor, Delaunay tessellation
- `scipy.sparse`: sparse matrix storage, sparse linear algebra, sparse solvers, sparse graph traversal
- `scipy.stats`: common statistical functions and distributions
- `scipy.special`: special functions (e.g., Airy functions, Bessel functions, orthogonal polynomials, gamma functions, and much more)
- `scipy.constants`: numerical and dimensional constants

As above, we recommend importing these modules and using IPython's `?` functionality to explore what is available.

A.8. Efficient coding with Python and NumPy

As discussed above, Python is a dynamically typed, interpreted language. This leads to several advantages: for example, the ability to write code quickly and the ability to execute code line by line in the IPython interpreter. This benefit comes at a cost, however: it means that certain code blocks may run much more slowly than equivalent code written in a compiled language like C. In data analysis settings, many of these disadvantages can be remedied by efficient use of NumPy arrays. Here we will show a few common mistakes that are made by beginners using Python and NumPy, and solutions which can lead to orders-of-magnitude improvement in computation times. Throughout this section we'll make use of IPython's `%timeit` magic command, which provides quick benchmarks for Python execution.

A.8.1. Data structures

The Python list object is very convenient to use, but not necessarily suitable for large data sets. It functions well for small sequences, but using large lists for computation can lead to slow execution:

```
In [1]: L = range(10000000)  # a large list
In [2]: %timeit sum(L)
1 loops, best of 3: 1.37 s per loop
```

This slow execution is due both to the data layout (in this case, a linked list) and to the overhead required for dynamic typing. As in other languages, the operation is much faster if the data is stored as a contiguous array. NumPy arrays provide this sort of type-specific, contiguous data container, as well as the tools to efficiently manipulate these arrays:

```
In [3]: import numpy as np
In [4]: x = np.array(L, dtype=float)
In [5]: %timeit np.sum(x)
100 loops, best of 3: 17.1 ms per loop
```

This is a factor of nearly 100 improvement in execution time, simply by using NumPy rather than the built-in Python list. This suggests our first guideline for data processing with Python:

Guideline 1: store data in NumPy arrays, not Python lists. Any time you have a sequence or list of data larger than a few dozen items, it should be stored and manipulated as a NumPy array.[11]

A.8.2. Loops

Even when using NumPy arrays, several pitfalls remain that can lead to inefficient code. A Python user who has a C or Java background may be used to using `for` loops or `while` loops to implement certain tasks. If an algorithm seems to require loops, it is nearly always better in Python to implement it using **vectorized** operations in NumPy (recall our discussion of ufuncs in §A.5.4). For example, the following code takes much longer to execute than the equivalent C code:

```
In [6]: x = np.random.random(1000000)

In [7]: def loop_add_one(x):
   ...:     for i in range(len(x)):
   ...:         x[i] += 1

In [8]: %timeit loop_add_one(x)
1 loops, best of 3: 1.67 s per loop

In [9]: def vectorized_add_one(x):
   ...:     x += 1

In [10]: timeit vectorized_add_one(x)
100 loops, best of 3: 4.13 ms per loop
```

Using vectorized operations—enabled by NumPy's ufuncs—for repeated operations leads to much faster code. This suggests our second guideline for data processing with Python:

Guideline 2: avoid large loops in favor of vectorized operations. If you find yourself applying the same operation to a sequence of many items, vectorized methods within NumPy will likely be a better choice.

A.8.3. Slicing, masks, and fancy indexing

Above we covered **slicing** to access subarrays. This is generally very fast, and is preferable to looping through the arrays. For example, we can quickly copy the first

[11]One exception to this rule is when building an array dynamically: because NumPy arrays are fixed length, they cannot be expanded efficiently. One common pattern is to build a list using the `append` operation, and convert it using `np.array` when the list is complete.

half of an array to the second half with a slicing operation:

```
In [11]: x[:len(x) / 2] = x[len(x) / 2:]
```

Sometimes when coding one must check the values of an array, and perform different operations depending on the value. For example, suppose we have an array and we want every value greater than 0.5 to be changed to 999. The first impulse might be to write something like the following:

```
In [12]: def check_vals(x):
    ....:     for i in range(len(x)):
    ....:         if x[i] > 0.5:
    ....:             x[i] = 999

In [13]: %timeit check_vals(x)
1 loops, best of 3: 1.03 s per loop
```

This same operation can be performed much more quickly using a boolean mask:

```
In [14]: %timeit x[(x > 0.5)] = 999
         # vectorized version
100 loops, best of 3: 3.79 ms per loop
```

In this implementation, we have used the boolean mask array generated by the expression `(x > 0.5)` to access only the portions of `x` that need to be modified. The operation in line 14 gives the same result as the operation in line 12, but is orders of magnitude faster.

Masks can be combined using the bitwise operators & for AND, | for OR, ~ for NOT, and ^ for XOR. For example, you could write the following:

```
In [15]: x[(x < 0.1) | (x > 0.5)] = 999
```

This will result in every item in the array `x` which is less than 0.1 or greater than 0.5 being replaced by 999. Be careful to use parentheses around boolean statements when combining operations, or operator precedence may lead to unexpected results.

Another similar concept is that of **fancy indexing**, which refers to indexing with lists. For example, imagine you have a two-dimensional array, and would like to create a new matrix of random rows (e.g., for bootstrap resampling). You may be tempted to proceed like this:

```
In [16]: X = np.random.random((1000000, 3))

In [17]: def get_random_rows(X):
    ....:     X_new = np.empty(X.shape)
    ....:     for i in range(X_new.shape[0]):
```

```
   ....:           X_new[i] = X[np.random.randint
                   (X.shape[0])]
   ....:       return X_new

In [18]: %timeit get_random_rows(X)
1 loops, best of 3: 1.63 s per loop
```

This can be sped up through fancy indexing, by generating the array of indices all at once and vectorizing the operation:

```
In [19]: def get_random_rows_fast(X):
   ....:       ind = np.random.randint(0, X.shape[0],
               X.shape[0])
   ....:       return X[ind]

In [20]: %timeit get_random_rows_fast(X)
1 loops, best of 3: 226 ms per loop
```

The list of indices returned by the call to `randint` is used directly in the indices. Note that fancy indexing is generally much slower than slicing for equivalent operations.

Slicing, masks, and fancy indexing can be used together to accomplish a wide variety of tasks. Along with ufuncs and broadcasting, discussed in §A.5.4, most common array manipulation tasks can be accomplished without writing a loop. This leads to our third guideline:

Guideline 3: use array slicing, masks, fancy indexing, and broadcasting to eliminate loops. If you find yourself looping over indices to select items on which an operation is performed, it can probably be done more efficiently with one of these techniques.

A.8.4. Summary

A common theme can be seen here: Python loops are slow, and NumPy array tricks can be used to sidestep this problem. As with most guidelines in programming, there are imaginable situations in which these suggestions can (and should) be ignored, but they are good to keep in mind for most situations. Be aware also that there are some algorithms for which loop elimination through vectorization is difficult or impossible. In this case, it can be necessary to interface Python to compiled code. This will be explored in the next section.

A.9. Wrapping existing code in Python

At times, you may find an algorithm which cannot benefit from the vectorization methods discussed in the previous section (a good example is in many tree-based search algorithms). Or you may desire to use within your Python script some legacy code which is written in a compiled language, or that exists in a shared library. In this case, a variety of approaches can be used to wrap compiled Fortran, C, or C++ code for use within Python. Each has its advantages and disadvantages, and can be appropriate for certain situations. Note that packages like NumPy, SciPy, and Scikit-learn use several of these tools both to implement efficient algorithms, and to make use of library packages written in Fortran, C, and C++.

Cython is a superset of the Python programming language that allows users to wrap external C and C++ packages, and also to write Python-like code which can be automatically translated to fast C code. The resulting compiled code can then be imported into Python scripts in the familiar way. Cython is very flexible, and within the scientific Python community, has gradually become the de facto standard tool for writing fast, compilable code. More information, including several helpful tutorials, is available at http://www.cython.org/.

f2py is a Fortran to Python interface generator, which began as an independent project before eventually becoming part of NumPy. f2py automates the generation of Python interfaces to Fortran code. If the Fortran code is designed to be compiled into a shared library, the interfacing process can be very smooth, and work mostly out of the box. See http://www.scipy.org/F2py for more information.

Ctypes, included in Python since version 2.5, is a built-in library that defines C data types in order to interface with compiled dynamic or shared libraries. Wrapper functions can be written in Python, which prepare arguments for library function calls, find the correct library, and execute the desired code. It is a nice tool for calling system libraries, but has the disadvantage that the resulting code is usually very platform dependent. For more information, refer to http://docs.python.org/library/ctypes.html

Python C-API Python is implemented in C, and therefore the C Application Programming Interface (API) can be used directly to wrap C libraries, or write efficient C code. Please be aware that this method is not for the faint of heart! Even writing a simple interface can take many lines of code, and it is very easy to inadvertently cause segmentation faults, memory leaks, or other nasty errors. For more information, see http://docs.python.org/c-api/, but be careful!

Two other tools to be aware of are **SWIG**, the Simplified Wrapper and Interface Generator (http://www.swig.org/), and **Weave**, a package within SciPy that allows incorporation of snippets of C or C++ code within Python scripts (http://www.scipy.org/weave). Cython has largely superseded the use of these packages in the scientific Python community.

All of these tools have use cases to which they are suited. For implementation of fast compiled algorithms and wrapping of C and C++ libraries or packages, we recommend Cython as a first approach. Cython's capabilities have greatly expanded during the last few years, and it has a very active community of developers. It has emerged as the favored approach by many in the scientific Python community, in particular the NumPy, SciPy, and Scikit-learn development teams.

A.10. Other resources

The discussion above is a good start, but there is much that has gone unmentioned. Here we list some books and websites that are good resources to learn more.

A.10.1. Books

- *Python for Data Analysis: Agile Tools for Real World Data* by McKinney [6]. This book was newly published at the time of this writing, and offers an excellent introduction to interactive scientific computing with IPython. The book is well written, and the first few chapters are extremely useful for scientists interested in Python. Several chapters at the end are more specialized, focusing on specific tools useful in fields with labeled and date-stamped data.
- *SciPy and NumPy: An Overview for Developers* by Bressert [2]. This is a short handbook written by an astronomer for a general scientific audience. It covers many of the tools available in NumPy and SciPy, assuming basic familiarity with the Python language. It focuses on practical examples, many of which are drawn from astronomy.
- *Object Oriented Programming in Python* by Goldwasser and Letscher [3]. This is a full-length text that introduces the principles of object-oriented programming in Python. It is designed as an introductory text for computer science or engineering students, and assumes no programming background.
- *Python Scripting for Computational Science* by Langtangen [4]. This is a well-written book that introduces Python computing principles to scientists who are familiar with computing, focusing on using Python as a scripting language to tie together various tools within a scientific workflow.

A.10.2. Websites

- **Learning Python**
  - http://www.python.org/ General info on the Python language. The site includes documentation and tutorials.
  - http://www.diveintopython.net/ An online Python book which introduces the Python syntax to experienced programmers.
  - http://software-carpentry.org/ Excellent tutorials and lectures for scientists to learn computing. The tutorials are not limited to Python, but cover other areas of computing as well.
  - http://scipy-lectures.github.com/ Lectures and notes on Python for scientific computing, including NumPy, SciPy, Matplotlib, Scikit-learn, and more, covering material from basic to advanced.
- **Python packages**
  - http://pypi.python.org Python package index. An easy way to find and install new Python packages.
  - http://ipython.org/ Documentation and tutorials for the IPython project. See especially the notebook functionality: this is quickly becoming a standard tool for sharing code and data in Python.

  - http://www.numpy.org Documentation and tutorials for NumPy.
  - http://www.scipy.org/ Documentation and tutorials for SciPy and scientific computing with Python.
  - http://scikit-learn.org/ Documentation for Scikit-learn. This site includes very well written and complete narrative documentation describing machine learning in Python on many types of data. See especially the tutorials.
  - http://matplotlib.org/ Documentation and tutorials for Matplotlib. See especially the example gallery: it gives a sense of the wide variety of plot types that can be created with Matplotlib.

- **Python and astronomy**
  - http://www.astropython.org/ Python for astronomers. This is a great resource for both beginners and experienced programmers.
  - http://www.astropy.org/ Community library for astronomy-related Python tools. These are the "nuts and bolts" of doing astronomy with Python: tools for reading FITS files and VOTables, for converting between coordinate systems, computing cosmological integrals, and much more.
  - http://www.astroML.org/ Code repository associated with this text. Read more in the following section.