

---

# IPython: An Interactive Computing and Development Environment

Act without doing; work without effort. Think of the small as large and the few as many.  
Confront the difficult while it is still easy; accomplish the great task by a series of small acts.

—Laozi

People often ask me, “What is your Python development environment?” My answer is almost always the same, “IPython and a text editor”. You may choose to substitute an Integrated Development Environment (IDE) for a text editor in order to take advantage of more advanced graphical tools and code completion capabilities. Even if so, I strongly recommend making IPython an important part of your workflow. Some IDEs even provide IPython integration, so it’s possible to get the best of both worlds.

The *IPython* project began in 2001 as Fernando Pérez’s side project to make a better interactive Python interpreter. In the subsequent 11 years it has grown into what’s widely considered one of the most important tools in the modern scientific Python computing stack. While it does not provide any computational or data analytical tools by itself, IPython is designed from the ground up to maximize your productivity in both interactive computing and software development. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages. It also provides very tight integration with the operating system’s shell and file system. Since much of data analysis coding involves exploration, trial and error, and iteration, IPython will, in almost all cases, help you get the job done faster.

Of course, the IPython project now encompasses a great deal more than just an enhanced, interactive Python shell. It also includes a rich GUI console with inline plotting, a web-based interactive notebook format, and a lightweight, fast parallel computing engine. And, as with so many other tools designed for and by programmers, it is highly customizable. I’ll discuss some of these features later in the chapter.

Since IPython has interactivity at its core, some of the features in this chapter are difficult to fully illustrate without a live console. If this is your first time learning about IPython, I recommend that you follow along with the examples to get a feel for how things work. As with any keyboard-driven console-like environment, developing muscle-memory for the common commands is part of the learning curve.



Many parts of this chapter (for example: profiling and debugging) can be safely omitted on a first reading as they are not necessary for understanding the rest of the book. This chapter is intended to provide a standalone, rich overview of the functionality provided by IPython.

## IPython Basics

You can launch IPython on the command line just like launching the regular Python interpreter except with the `ipython` command:

```
$ ipython
Python 2.7.2 (default, May 27 2012, 21:26:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref  -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

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

You can execute arbitrary Python statements by typing them in and pressing `<return>`. When typing just a variable into IPython, it renders a string representation of the object:

```
In [541]: import numpy as np

In [542]: data = {i : randn() for i in range(7)}

In [543]: data
Out[543]:
{0: 0.6900018528091594,
 1: 1.0015434424937888,
 2: -0.5030873913603446,
 3: -0.6222742250596455,
 4: -0.9211686080130108,
 5: -0.726213492660829,
 6: 0.2228955458351768}
```

Many kinds of Python objects are formatted to be more readable, or *pretty-printed*, which is distinct from normal printing with `print`. If you printed a dict like the above in the standard Python interpreter, it would be much less readable:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
 3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
 6: 0.3308507317325902}
```

IPython also provides facilities to make it easy to execute arbitrary blocks of code (via somewhat glorified copy-and-pasting) and whole Python scripts. These will be discussed shortly.

## Tab Completion

On the surface, the IPython shell looks like a cosmetically slightly-different interactive Python interpreter. Users of Mathematica may find the enumerated input and output prompts familiar. One of the major improvements over the standard Python shell is *tab completion*, a feature common to most interactive data analysis environments. While entering expressions in the shell, pressing <Tab> will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and        an_example  any
```

In this example, note that IPython displayed both the two variables I defined as well as the Python keyword `and` and built-in function `any`. Naturally, you can also complete methods and attributes on any object after typing a period:

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append    b.extend    b.insert    b.remove    b.sort
b.count     b.index     b.pop       b.reverse
```

The same goes for modules:

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date        datetime.MAXYEAR    datetime.timedelta
datetime.datetime    datetime.MINYEAR    datetime.tzinfo
datetime.datetime_CAPI datetime.time
```



Note that IPython by default hides methods and attributes starting with underscores, such as magic methods and internal “private” methods and attributes, in order to avoid cluttering the display (and confusing new Python users!). These, too, can be tab-completed but you must first type an underscore to see them. If you prefer to always see such methods in tab completion, you can change this setting in the IPython configuration.

Tab completion works in many contexts outside of searching the interactive namespace and completing object or module attributes. When typing anything that looks like a file path (even in a Python string), pressing <Tab> will complete anything on your computer’s file system matching what you’ve typed:

```
In [3]: book_scripts/<Tab>
book_scripts/cprof_example.py      book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py        book_scripts/prof_mod.py
```

```
In [3]: path = 'book_scripts/<Tab>
book_scripts/cprof_example.py      book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py        book_scripts/prof_mod.py
```

Combined with the %run command (see later section), this functionality will undoubtedly save you many keystrokes.

Another area where tab completion saves time is in the completion of function keyword arguments (including the = sign!).

## Introspection

Using a question mark (?) before or after a variable will display some general information about the object:

```
In [545]: b?
Type:      list
String Form:[1, 2, 3]
Length:    3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

This is referred to as *object introspection*. If the object is a function or instance method, the docstring, if defined, will also be shown. Suppose we’d written the following function:

```
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -----
    the_sum : type of arguments
```

```
"""
    return a + b
```

Then using `?` shows us the docstring:

```
In [547]: add_numbers?
Type:      function
String Form:<function add_numbers at 0x5fad848>
File:      book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Docstring:
Add two numbers together
Returns
-----
the_sum : type of arguments
```

Using `??` will also show the function's source code if possible:

```
In [548]: add_numbers??
Type:      function
String Form:<function add_numbers at 0x5fad848>
File:      book_scripts/<ipython-input-546-5473012eeb65>
Definition: add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
        Add two numbers together
        Returns
        -----
        the_sum : type of arguments
    """
    return a + b
```

`?` has a final usage, which is for searching the IPython namespace in a manner similar to the standard UNIX or Windows command line. A number of characters combined with the wildcard (`*`) will show all names matching the wildcard expression. For example, we could get a list of all functions in the top level NumPy namespace containing `load`:

```
In [549]: np.*load*?
np.load
np.loads
np.loadtxt
np.pkgload
```

## The `%run` Command

Any file can be run as a Python program inside the environment of your IPython session using the `%run` command. Suppose you had the following simple script stored in `ipython_script_test.py`:

```
def f(x, y, z):
    return (x + y) / z

a = 5
```

```
b = 6
c = 7.5

result = f(a, b, c)
```

This can be executed by passing the file name to `%run`:

```
In [550]: %run ipython_script_test.py
```

The script is run in an *empty namespace* (with no imports or other variables defined) so that the behavior should be identical to running the program on the command line using `python script.py`. All of the variables (imports, functions, and globals) defined in the file (up until an exception, if any, is raised) will then be accessible in the IPython shell:

```
In [551]: c
Out[551]: 7.5

In [552]: result
Out[552]: 1.4666666666666666
```

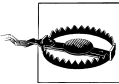
If a Python script expects command line arguments (to be found in `sys.argv`), these can be passed after the file path as though run on the command line.



Should you wish to give a script access to variables already defined in the interactive IPython namespace, use `%run -i` instead of plain `%run`.

## Interrupting running code

Pressing `<Ctrl-C>` while any code is running, whether a script through `%run` or a long-running command, will cause a `KeyboardInterrupt` to be raised. This will cause nearly all Python programs to stop immediately except in very exceptional cases.



When a piece of Python code has called into some compiled extension modules, pressing `<Ctrl-C>` will not cause the program execution to stop immediately in all cases. In such cases, you will have to either wait until control is returned to the Python interpreter, or, in more dire circumstances, forcibly terminate the Python process via the OS task manager.

## Executing Code from the Clipboard

A quick-and-dirty way to execute code in IPython is via pasting from the clipboard. This might seem fairly crude, but in practice it is very useful. For example, while developing a complex or time-consuming application, you may wish to execute a script piece by piece, pausing at each stage to examine the currently loaded data and results. Or, you might find a code snippet on the Internet that you want to run and play around with, but you'd rather not create a new `.py` file for it.

Code snippets can be pasted from the clipboard in many cases by pressing <Ctrl-Shift-V>. Note that it is not completely robust as this mode of pasting mimics typing each line into IPython, and line breaks are treated as <return>. This means that if you paste code with an indented block and there is a blank line, IPython will think that the indented block is over. Once the next line in the block is executed, an `IndentationError` will be raised. For example the following code:

```
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
```

will not work if simply pasted:

```
In [1]: x = 5

In [2]: y = 7

In [3]: if x > 5:
...:     x += 1
...:

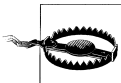
In [4]:     y = 8
IndentationError: unexpected indent
```

If you want to paste code into IPython, try the `%paste` and `%cpaste` magic functions.

As the error message suggests, we should instead use the `%paste` and `%cpaste` magic functions. `%paste` takes whatever text is in the clipboard and executes it as a single block in the shell:

```
In [6]: %paste
x = 5
y = 7
if x > 5:
    x += 1

    y = 8
## -- End pasted text --
```



Depending on your platform and how you installed Python, there's a small chance that `%paste` will not work. Packaged distributions like EPDFree (as described in the intro) should not be a problem.

`%cpaste` is similar, except that it gives you a special prompt for pasting code into:

```
In [7]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
```

```

:    x += 1
:
:    y = 8
:--

```

With the `%cpaste` block, you have the freedom to paste as much code as you like before executing it. You might decide to use `%cpaste` in order to look at the pasted code before executing it. If you accidentally paste the wrong code, you can break out of the `%cpaste` prompt by pressing `<Ctrl-C>`.

Later, I'll introduce the IPython HTML Notebook which brings a new level of sophistication for developing analyses block-by-block in a browser-based notebook format with executable code cells.

### IPython interaction with editors and IDEs

Some text editors, such as Emacs and vim, have 3rd party extensions enabling blocks of code to be sent directly from the editor to a running IPython shell. Refer to the IPython website or do an Internet search to find out more.

Some IDEs, such as the PyDev plugin for Eclipse and Python Tools for Visual Studio from Microsoft (and possibly others), have integration with the IPython terminal application. If you want to work in an IDE but don't want to give up the IPython console features, this may be a good option for you.

## Keyboard Shortcuts

IPython has many keyboard shortcuts for navigating the prompt (which will be familiar to users of the Emacs text editor or the UNIX bash shell) and interacting with the shell's command history (see later section). [Table 3-1](#) summarizes some of the most commonly used shortcuts. See [Figure 3-1](#) for an illustration of a few of these, such as cursor movement.

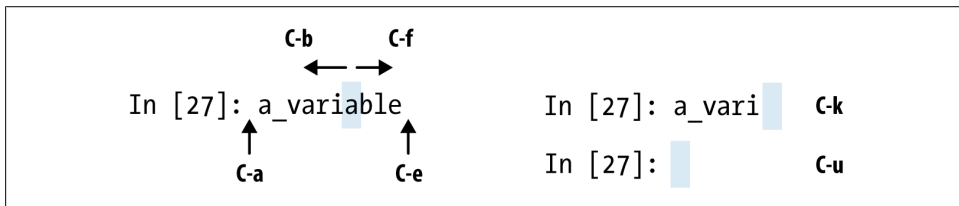


Figure 3-1. Illustration of some of IPython's keyboard shortcuts



Table 3-1. Standard IPython Keyboard Shortcuts

Command	Description
Ctrl-p or up-arrow	Search backward in command history for commands starting with currently-entered text
Ctrl-n or down-arrow	Search forward in command history for commands starting with currently-entered text
Ctrl-r	Readline-style reverse history search (partial matching)
Ctrl-Shift-v	Paste text from clipboard
Ctrl-c	Interrupt currently-executing code
Ctrl-a	Move cursor to beginning of line
Ctrl-e	Move cursor to end of line
Ctrl-k	Delete text from cursor until end of line
Ctrl-u	Discard all text on current line
Ctrl-f	Move cursor forward one character
Ctrl-b	Move cursor back one character
Ctrl-l	Clear screen

## Exceptions and Tracebacks

If an exception is raised while %run-ing a script or executing any statement, IPython will by default print a full call stack trace (traceback) with a few lines of context around the position at each point in the stack.

```
In [553]: %run ch03/ipython_bug.py
-----
AssertionError                                Traceback (most recent call last)
/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where)
    176         else:
    177             filename = fname
--> 178         __builtin__.execfile(filename, *where)
book_scripts/ch03/ipython_bug.py in <module>()
     13     throws_an_exception()
     14
----> 15 calling_things()
book_scripts/ch03/ipython_bug.py in calling_things()
     11 def calling_things():
     12     works_fine()
----> 13     throws_an_exception()
     14
     15 calling_things()
book_scripts/ch03/ipython_bug.py in throws_an_exception()
      7     a = 5
      8     b = 6
----> 9     assert(a + b == 10)
     10
     11 def calling_things():
AssertionError:
```

Having additional context by itself is a big advantage over the standard Python interpreter (which does not provide any additional context). The amount of context shown can be controlled using the `%xmode` magic command, from minimal (same as the standard Python interpreter) to verbose (which inlines function argument values and more). As you will see later in the chapter, you can step *into the stack* (using the `%debug` or `%pdb` magics) after an error has occurred for interactive post-mortem debugging.

## Magic Commands

IPython has many special commands, known as “magic” commands, which are designed to facilitate common tasks and enable you to easily control the behavior of the IPython system. A magic command is any command prefixed by the percent symbol `%`. For example, you can check the execution time of any Python statement, such as a matrix multiplication, using the `%timeit` magic function (which will be discussed in more detail later):

```
In [554]: a = np.random.randn(100, 100)

In [555]: %timeit np.dot(a, a)
10000 loops, best of 3: 69.1 us per loop
```

Magic commands can be viewed as command line programs to be run within the IPython system. Many of them have additional “command line” options, which can all be viewed (as you might expect) using `?`:

```
In [1]: %reset?
Resets the namespace by removing all names defined by the user.

Parameters
-----
-f : force reset without asking for confirmation.

-s : 'Soft' reset: Only clears your namespace, leaving history intact.
References to objects may be kept. By default (without this option),
we do a 'hard' reset, giving you a new session and removing all
references to objects from the current session.

Examples
-----
In [6]: a = 1

In [7]: a
Out[7]: 1

In [8]: 'a' in _ip.user_ns
Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns
Out[1]: False
```

Magic functions can be used by default without the percent sign, as long as no variable is defined with the same name as the magic function in question. This feature is called *automagic* and can be enabled or disabled using `%automagic`.

Since IPython’s documentation is easily accessible from within the system, I encourage you to explore all of the special commands available by typing `%quickref` or `%magic`. I will highlight a few more of the most critical ones for being productive in interactive computing and Python development in IPython.

Table 3-2. Frequently-used IPython Magic Commands

Command	Description
<code>%quickref</code>	Display the IPython Quick Reference Card
<code>%magic</code>	Display detailed documentation for all of the available magic commands
<code>%debug</code>	Enter the interactive debugger at the bottom of the last exception traceback
<code>%hist</code>	Print command input (and optionally output) history
<code>%pdb</code>	Automatically enter debugger after any exception
<code>%paste</code>	Execute pre-formatted Python code from clipboard
<code>%cpaste</code>	Open a special prompt for manually pasting Python code to be executed
<code>%reset</code>	Delete all variables / names defined in interactive namespace
<code>%page OBJECT</code>	Pretty print the object and display it through a pager
<code>%run script.py</code>	Run a Python script inside IPython
<code>%prun statement</code>	Execute <i>statement</i> with <code>cProfile</code> and report the profiler output
<code>%time statement</code>	Report the execution time of single statement
<code>%timeit statement</code>	Run a statement multiple times to compute an ensemble average execution time. Useful for timing code with very short execution time
<code>%who</code> , <code>%who_ls</code> , <code>%whos</code>	Display variables defined in interactive namespace, with varying levels of information/verbosity
<code>%xdel variable</code>	Delete a variable and attempt to clear any references to the object in the IPython internals

## Qt-based Rich GUI Console

The IPython team has developed a Qt framework-based GUI console, designed to wed the features of the terminal-only applications with the features provided by a rich text widget, like embedded images, multiline editing, and syntax highlighting. If you have either PyQt or PySide installed, the application can be launched with inline plotting by running this on the command line:

```
ipython qtconsole --pylab=inline
```

The Qt console can launch multiple IPython processes in tabs, enabling you to switch between tasks. It can also share a process with the IPython HTML Notebook application, which I’ll highlight later.

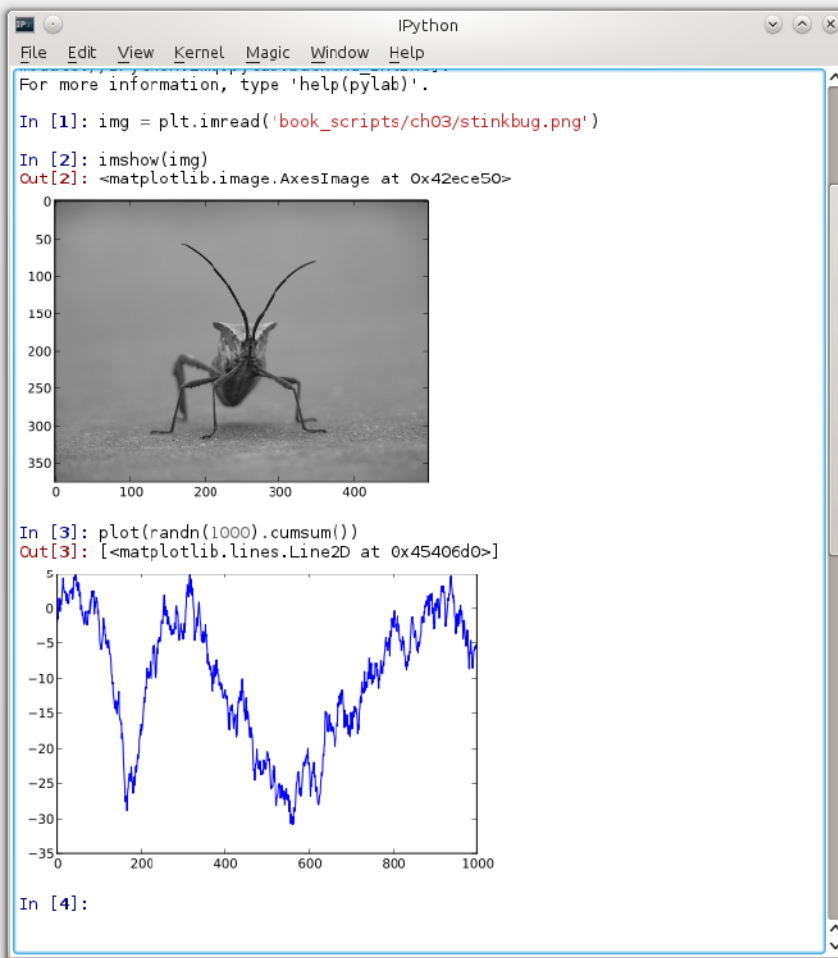


Figure 3-2. IPython Qt Console

## Matplotlib Integration and Pylab Mode

Part of why IPython is so widely used in scientific computing is that it is designed as a companion to libraries like matplotlib and other GUI toolkits. Don't worry if you have never used matplotlib before; it will be discussed in much more detail later in this book. If you create a matplotlib plot window in the regular Python shell, you'll be sad to find that the GUI event loop "takes control" of the Python session until the plot window is closed. That won't work for interactive data analysis and visualization, so IPython has

implemented special handling for each GUI framework so that it will work seamlessly with the shell.

The typical way to launch IPython with matplotlib integration is by adding the `--pylab` flag (two dashes).

```
$ ipython --pylab
```

This will cause several things to happen. First IPython will launch with the default GUI backend integration enabled so that matplotlib plot windows can be created with no issues. Secondly, most of NumPy and matplotlib will be imported into the top level interactive namespace to produce an interactive computing environment reminiscent of MATLAB and other domain-specific scientific computing environments. It's possible to do this setup by hand by using `%gui`, too (try running `%gui?` to find out how).

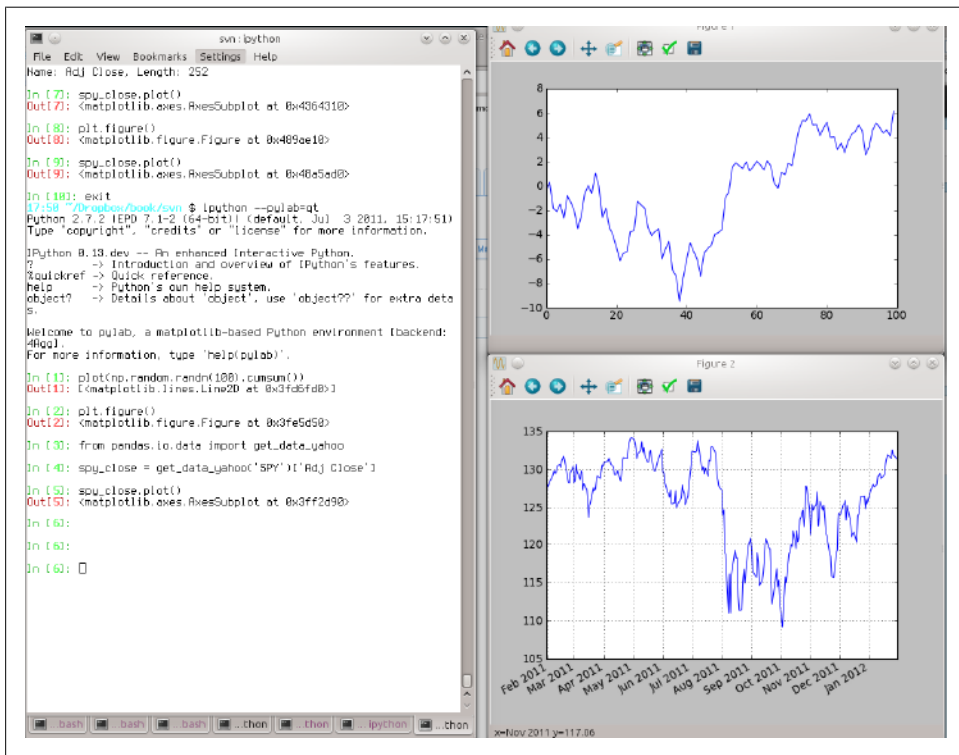


Figure 3-3. Pylab mode: IPython with matplotlib windows

# Using the Command History

IPython maintains a small on-disk database containing the text of each command that you execute. This serves various purposes:

- Searching, completing, and executing previously-executed commands with minimal typing
- Persisting the command history between sessions.
- Logging the input/output history to a file

## Searching and Reusing the Command History

Being able to search and execute previous commands is, for many people, the most useful feature. Since IPython encourages an iterative, interactive code development workflow, you may often find yourself repeating the same commands, such as a `%run` command or some other code snippet. Suppose you had run:

```
In[7]: %run first/second/third/data_script.py
```

and then explored the results of the script (assuming it ran successfully), only to find that you made an incorrect calculation. After figuring out the problem and modifying `data_script.py`, you can start typing a few letters of the `%run` command then press either the `<Ctrl-P>` key combination or the `<up arrow>` key. This will search the command history for the first prior command matching the letters you typed. Pressing either `<Ctrl-P>` or `<up arrow>` multiple times will continue to search through the history. If you pass over the command you wish to execute, fear not. You can move *forward* through the command history by pressing either `<Ctrl-N>` or `<down arrow>`. After doing this a few times you may start pressing these keys without thinking!

Using `<Ctrl-R>` gives you the same partial incremental searching capability provided by the `readline` used in UNIX-style shells, such as the bash shell. On Windows, `readline` functionality is emulated by IPython. To use this, press `<Ctrl-R>` then type a few characters contained in the input line you want to search for:

```
In [1]: a_command = foo(x, y, z)
```

```
(reverse-i-search)`com': a_command = foo(x, y, z)
```

Pressing `<Ctrl-R>` will cycle through the history for each line matching the characters you've typed.

## Input and Output Variables

Forgetting to assign the result of a function call to a variable can be very annoying. Fortunately, IPython stores references to *both* the input (the text that you type) and output (the object that is returned) in special variables. The previous two outputs are stored in the `_` (one underscore) and `__` (two underscores) variables, respectively:

```
In [556]: 2 ** 27
Out[556]: 134217728
```

```
In [557]: _
Out[557]: 134217728
```

Input variables are stored in variables named like `_iX`, where `X` is the input line number. For each such input variables there is a corresponding output variable `_X`. So after input line 27, say, there will be two new variables `_27` (for the output) and `_i27` for the input.

```
In [26]: foo = 'bar'
```

```
In [27]: foo
Out[27]: 'bar'
```

```
In [28]: _i27
Out[28]: u'foo'
```

```
In [29]: _27
Out[29]: 'bar'
```

Since the input variables are strings, that can be executed again using the Python `exec` keyword:

```
In [30]: exec _i27
```

Several magic functions allow you to work with the input and output history. `%hist` is capable of printing all or part of the input history, with or without line numbers. `%reset` is for clearing the interactive namespace and optionally the input and output caches. The `%xdel` magic function is intended for removing all references to a *particular* object from the IPython machinery. See the documentation for both of these magics for more details.



When working with very large data sets, keep in mind that IPython's input and output history causes any object referenced there to not be garbage collected (freeing up the memory), even if you delete the variables from the interactive namespace using the `del` keyword. In such cases, careful usage of `%xdel` and `%reset` can help you avoid running into memory problems.

## Logging the Input and Output

IPython is capable of logging the entire console session including input and output. Logging is turned on by typing `%logstart`:

```
In [3]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename      : ipython_log.py
Mode          : rotate
Output logging : False
Raw input log  : False
```

```
Timestamping : False
State        : active
```

IPython logging can be enabled at any time and it will record your entire session (including previous commands). Thus, if you are working on something and you decide you want to save everything you did, you can simply enable logging. See the docstring of `%logstart` for more options (including changing the output file path), as well as the companion functions `%logoff`, `%logon`, `%logstate`, and `%logstop`.

## Interacting with the Operating System

Another important feature of IPython is that it provides very strong integration with the operating system shell. This means, among other things, that you can perform most standard command line actions as you would in the Windows or UNIX (Linux, OS X) shell without having to exit IPython. This includes executing shell commands, changing directories, and storing the results of a command in a Python object (list or string). There are also simple shell command aliasing and directory bookmarking features.

See [Table 3-3](#) for a summary of magic functions and syntax for calling shell commands. I'll briefly visit these features in the next few sections.

Table 3-3. IPython system-related commands

Command	Description
<code>!cmd</code>	Execute <code>cmd</code> in the system shell
<code>output = !cmd args</code>	Run <code>cmd</code> and store the stdout in <code>output</code>
<code>%alias alias_name cmd</code>	Define an alias for a system (shell) command
<code>%bookmark</code>	Utilize IPython's directory bookmarking system
<code>%cd directory</code>	Change system working directory to passed directory
<code>%pwd</code>	Return the current system working directory
<code>%pushd directory</code>	Place current directory on stack and change to target directory
<code>%popd</code>	Change to directory popped off the top of the stack
<code>%dirs</code>	Return a list containing the current directory stack
<code>%dhist</code>	Print the history of visited directories
<code>%env</code>	Return the system environment variables as a dict

## Shell Commands and Aliases

Starting a line in IPython with an exclamation point `!`, or bang, tells IPython to execute everything after the bang in the system shell. This means that you can delete files (using `rm` or `del`, depending on your OS), change directories, or execute any other process. It's even possible to start processes that take control away from IPython, even another Python interpreter:



```
In [2]: !python
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
Type "packages", "demo" or "enthought" for more information.
>>>
```

The console output of a shell command can be stored in a variable by assigning the `!`-escaped expression to a variable. For example, on my Linux-based machine connected to the Internet via ethernet, I can get my IP address as a Python variable:

```
In [1]: ip_info = !ifconfig eth0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:192.168.1.137 Bcast:192.168.1.255 Mask:255.255.255.0'
```

The returned Python object `ip_info` is actually a custom list type containing various versions of the console output.

IPython can also substitute in Python values defined in the current environment when using `!`. To do this, preface the variable name by the dollar sign `$`:

```
In [3]: foo = 'test*'

In [4]: !ls $foo
test4.py test.py test.xml
```

The `%alias` magic function can define custom shortcuts for shell commands. As a simple example:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x  2 root root  69632 2012-01-29 20:36 bin/
drwxr-xr-x  2 root root   4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root  20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x  44 root root  69632 2011-12-26 18:08 lib32/
lrwxrwxrwx  1 root root    3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x  15 root root   4096 2011-10-13 19:03 local/
drwxr-xr-x  2 root root  12288 2012-01-12 09:32/sbin/
drwxr-xr-x 387 root root  12288 2011-11-04 22:53 share/
drwxrwsr-x  24 root src   4096 2011-07-17 18:38 src/
```

Multiple commands can be executed just as on the command line by separating them with semicolons:

```
In [558]: %alias test_alias (cd ch08; ls; cd ..)

In [559]: test_alias
macrodata.csv spx.csv tips.csv
```

You'll notice that IPython "forgets" any aliases you define interactively as soon as the session is closed. To create permanent aliases, you will need to use the configuration system. See later in the chapter.

## Directory Bookmark System

IPython has a simple directory bookmarking system to enable you to save aliases for common directories so that you can jump around very easily. For example, I'm an avid user of Dropbox, so I can define a bookmark to make it easy to change directories to my Dropbox:

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

Once I've done this, when I use the `%cd` magic, I can use any bookmarks I've defined

```
In [7]: cd db
(bookmark:db) -> /home/wesm/Dropbox/
/home/wesm/Dropbox
```

If a bookmark name conflicts with a directory name in your current working directory, you can use the `-b` flag to override and use the bookmark location. Using the `-l` option with `%bookmark` lists all of your bookmarks:

```
In [8]: %bookmark -l
Current bookmarks:
db -> /home/wesm/Dropbox/
```

Bookmarks, unlike aliases, are automatically persisted between IPython sessions.

## Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython is well suited as a software development environment. In data analysis applications, it's important first to have *correct* code. Fortunately, IPython has closely integrated and enhanced the built-in Python `pdb` debugger. Secondly you want your code to be *fast*. For this IPython has easy-to-use code timing and profiling tools. I will give an overview of these tools in detail here.

## Interactive Debugger

IPython's debugger enhances `pdb` with tab completion, syntax highlighting, and context for each line in exception tracebacks. One of the best times to debug code is right after an error has occurred. The `%debug` command, when entered immediately after an exception, invokes the "post-mortem" debugger and drops you into the stack frame where the exception was raised:

```
In [2]: run ch03/ipython_bug.py
-----
AssertionError                                Traceback (most recent call last)
/home/wesm/book_scripts/ch03/ipython_bug.py in <module>()
    13     throws_an_exception()
    14
--> 15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in calling_things()
```

```

11 def calling_things():
12     works_fine()
----> 13     throws_an_exception()
14
15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in throws_an_exception()
7     a = 5
8     b = 6
----> 9     assert(a + b == 10)
10
11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
8     b = 6
----> 9     assert(a + b == 10)
10

ipdb>

```

Once inside the debugger, you can execute arbitrary Python code and explore all of the objects and data (which have been “kept alive” by the interpreter) inside each stack frame. By default you start in the lowest level, where the error occurred. By pressing **u** (up) and **d** (down), you can switch between the levels of the stack trace:

```

ipdb> u
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
12     works_fine()
---> 13     throws_an_exception()
14

```

Executing the `%pdb` command makes it so that IPython automatically invokes the debugger after any exception, a mode that many users will find especially useful.

It’s also easy to use the debugger to help develop code, especially when you wish to set breakpoints or step through the execution of a function or script to examine the state at each stage. There are several ways to accomplish this. The first is by using `%run` with the `-d` flag, which invokes the debugger before executing any code in the passed script. You must immediately press **s** (step) to enter the script:

```

In [5]: run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/book_scripts/ch03/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
2     a = 5
3     b = 6

```

After this point, it's up to you how you want to work your way through the file. For example, in the above exception, we could set a breakpoint right before calling the `works_fine` method and run the script until we reach the breakpoint by pressing `c` (continue):

```
ipdb> b 12
ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(12)calling_things()
11 def calling_things():
2--> 12     works_fine()
13     throws_an_exception()
```

At this point, you can step into `works_fine()` or execute `works_fine()` by pressing `n` (next) to advance to the next line:

```
ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
2 12     works_fine()
---> 13     throws_an_exception()
14
```

Then, we could step into `throws_an_exception` and advance to the line where the error occurs and look at the variables in the scope. Note that debugger commands take precedence over variable names; in such cases preface the variables with `!` to examine their contents.

```
ipdb> s
--Call--
> /home/wesm/book_scripts/ch03/ipython_bug.py(6)throws_an_exception()
5
----> 6 def throws_an_exception():
7     a = 5

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(7)throws_an_exception()
6 def throws_an_exception():
----> 7     a = 5
8     b = 6

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(8)throws_an_exception()
7     a = 5
----> 8     b = 6
9     assert(a + b == 10)

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
8     b = 6
----> 9     assert(a + b == 10)
10

ipdb> !a
5
ipdb> !b
6
```

Becoming proficient in the interactive debugger is largely a matter of practice and experience. See [Table 3-4](#) for a full catalogue of the debugger commands. If you are used to an IDE, you might find the terminal-driven debugger to be a bit bewildering at first, but that will improve in time. Most of the Python IDEs have excellent GUI debuggers, but it is usually a significant productivity gain to remain in IPython for your debugging.

Table 3-4. (I)Python debugger commands

Command	Action
<code>h(elp)</code>	Display command list
<code>help command</code>	Show documentation for <i>command</i>
<code>c(ontinue)</code>	Resume program execution
<code>q(uit)</code>	Exit debugger without executing any more code
<code>b(reak) number</code>	Set breakpoint at <i>number</i> in current file
<code>b path/to/file.py:number</code>	Set breakpoint at line <i>number</i> in specified file
<code>s(tep)</code>	Step <i>into</i> function call
<code>n(ext)</code>	Execute current line and advance to next line at current level
<code>u(p) / d(own)</code>	Move up/down in function call stack
<code>a(rgs)</code>	Show arguments for current function
<code>debug statement</code>	Invoke statement <i>statement</i> in new (recursive) debugger
<code>l(ist) statement</code>	Show current position and context at current level of stack
<code>w(here)</code>	Print full stack trace with context at current position

### Other ways to make use of the debugger

There are a couple of other useful ways to invoke the debugger. The first is by using a special `set_trace` function (named after `pdb.set_trace`), which is basically a “poor man’s breakpoint”. Here are two small recipes you might want to put somewhere for your general use (potentially adding them to your IPython profile as I do):

```
def set_trace():
    from IPython.core.debugger import Pdb
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
    from IPython.core.debugger import Pdb
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, `set_trace`, is very simple. Put `set_trace()` anywhere in your code that you want to stop and take a look around (for example, right before an exception occurs):

```
In [7]: run ch03/ipython_bug.py
> /home/wesm/book_scripts/ch03/ipython_bug.py(16)calling_things()
15     set_trace()
```

```

---> 16     throws_an_exception()
      17

```

Pressing `c` (continue) will cause the code to resume normally with no harm done.

The `debug` function above enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like

```

def f(x, y, z=1):
    tmp = x + y
    return tmp / z

```

and we wished to step through its logic. Ordinarily using `f` would look like `f(1, 2, z=3)`. To instead step into `f`, pass `f` as the first argument to `debug` followed by the positional and keyword arguments to be passed to `f`:

```

In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z

```

```
ipdb>
```

I find that these two simple recipes save me a lot of time on a day-to-day basis.

Lastly, the debugger can be used in conjunction with `%run`. By running a script with `%run -d`, you will be dropped directly into the debugger, ready to set any breakpoints and start the script:

```

In [1]: %run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

```

```
ipdb>
```

Adding `-b` with a line number starts the debugger with a breakpoint set already:

```

In [2]: %run -d -b2 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

```

```

ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(2)works_fine()
      1 def works_fine():
1---> 2     a = 5
      3     b = 6

```

```
ipdb>
```

## Timing Code: %time and %timeit

For larger-scale or longer-running data analysis applications, you may wish to measure the execution time of various components or of individual statements or function calls. You may want a report of which functions are taking up the most time in a complex process. Fortunately, IPython enables you to get this information very easily while you are developing and testing your code.

Timing code by hand using the built-in `time` module and its functions `time.clock` and `time.time` is often tedious and repetitive, as you must write the same uninteresting boilerplate code:

```
import time
start = time.time()
for i in range(iterations):
    # some code to run here
elapsed_per = (time.time() - start) / iterations
```

Since this is such a common operation, IPython has two magic functions `%time` and `%timeit` to automate this process for you. `%time` runs a statement once, reporting the total execution time. Suppose we had a large list of strings and we wanted to compare different methods of selecting all strings starting with a particular prefix. Here is a simple list of 700,000 strings and two identical methods of selecting only the ones that start with 'foo':

```
# a very large list of strings
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

It looks like they should be about the same performance-wise, right? We can check for sure using `%time`:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s
```

```
In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

The `Wall time` is the main number of interest. So, it looks like the first method takes more than twice as long, but it's not a very precise measurement. If you try `%time`-ing those statements multiple times yourself, you'll find that the results are somewhat variable. To get a more precise measurement, use the `%timeit` magic function. Given an arbitrary statement, it has a heuristic to run a statement multiple times to produce a fairly accurate average runtime.

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop
```

```
In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

This seemingly innocuous example illustrates that it is worth understanding the performance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milliseconds will start to add up!

`%timeit` is especially useful for analyzing statements and functions with very short execution times, even at the level of microseconds ( $1e-6$  seconds) or nanoseconds ( $1e-9$  seconds). These may seem like insignificant amounts of time, but of course a 20 microsecond function invoked 1 million times takes 15 seconds longer than a 5 microsecond function. In the above example, we could very directly compare the two string operations to understand their performance characteristics:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

## Basic Profiling: `%prun` and `%run -p`

Profiling code is closely related to timing code, except it is concerned with determining *where* time is spent. The main Python profiling tool is the `cProfile` module, which is not specific to IPython at all. `cProfile` executes a program or any arbitrary block of code while keeping track of how much time is spent in each function.

A common way to use `cProfile` is on the command line, running an entire program and outputting the aggregated time per function. Suppose we had a simple script which does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of  $100 \times 100$  matrices):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
    K = 100
    results = []
    for _ in xrange(niter):
        mat = np.random.randn(K, K)
        max_eigenvalue = np.abs(eigvals(mat)).max()
        results.append(max_eigenvalue)
    return results
some_results = run_experiment()
print 'Largest one we saw: %s' % np.max(some_results)
```



Don't worry if you are not familiar with NumPy. You can run this script through cProfile by running the following in the command line:

```
python -m cProfile cprof_example.py
```

If you try that, you'll find that the results are outputted sorted by function name. This makes it a bit hard to get an idea of where the most time is spent, so it's very common to specify a *sort order* using the `-s` flag:

```
$ python -m cProfile -s cumulative cprof_example.py
Largest one we saw: 11.923204422
15116 function calls (14927 primitive calls) in 0.720 seconds
```

Ordered by: cumulative time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.001	0.001	0.721	0.721	cprof_example.py:1(<module>)
100	0.003	0.000	0.586	0.006	linalg.py:702(eigvals)
200	0.572	0.003	0.572	0.003	{numpy.linalg.lapack_lite.dgeev}
1	0.002	0.002	0.075	0.075	__init__.py:106(<module>)
100	0.059	0.001	0.059	0.001	{method 'randn'}
1	0.000	0.000	0.044	0.044	add_newdocs.py:9(<module>)
2	0.001	0.001	0.037	0.019	__init__.py:1(<module>)
2	0.003	0.002	0.030	0.015	__init__.py:2(<module>)
1	0.000	0.000	0.030	0.030	type_check.py:3(<module>)
1	0.001	0.001	0.021	0.021	__init__.py:15(<module>)
1	0.013	0.013	0.013	0.013	numeric.py:1(<module>)
1	0.000	0.000	0.009	0.009	__init__.py:6(<module>)
1	0.001	0.001	0.008	0.008	__init__.py:45(<module>)
262	0.005	0.000	0.007	0.000	function_base.py:3178(add_newdoc)
100	0.003	0.000	0.005	0.000	linalg.py:162(_assertFinite)
...					

Only the first 15 rows of the output are shown. It's easiest to read by scanning down the *cumtime* column to see how much total time was spent *inside* each function. Note that if a function calls some other function, *the clock does not stop running*. cProfile records the start and end time of each function call and uses that to produce the timing.

In addition to the above command-line usage, cProfile can also be used programmatically to profile arbitrary blocks of code without having to run a new process. IPython has a convenient interface to this capability using the `%prun` command and the `-p` option to `%run`. `%prun` takes the same "command line options" as cProfile but will profile an arbitrary Python statement instead of a whole .py file:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
4203 function calls in 0.643 seconds
```

Ordered by: cumulative time

List reduced from 32 to 7 due to restriction <7>

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.643	0.643	<string>:1(<module>)
1	0.001	0.001	0.643	0.643	cprof_example.py:4(run_experiment)
100	0.003	0.000	0.583	0.006	linalg.py:702(eigvals)

200	0.569	0.003	0.569	0.003	{numpy.linalg.lapack_lite.dgeev}
100	0.058	0.001	0.058	0.001	{method 'randn'}
100	0.003	0.000	0.005	0.000	linalg.py:162(_assertFinite)
200	0.002	0.000	0.002	0.000	{method 'all' of 'numpy.ndarray' objects}

Similarly, calling `%run -p -s cumulative cprof_example.py` has the same effect as the command-line approach above, except you never have to leave IPython.

## Profiling a Function Line-by-Line

In some cases the information you obtain from `%prun` (or another `cProfile`-based profile method) may not tell the whole story about a function's execution time, or it may be so complex that the results, aggregated by function name, are hard to interpret. For this case, there is a small library called `line_profiler` (obtainable via PyPI or one of the package management tools). It contains an IPython extension enabling a new magic function `%lprun` that computes a line-by-line-profiling of one or more functions. You can enable this extension by modifying your IPython configuration (see the IPython documentation or the section on configuration later in this chapter) to include the following line:

```
# A list of dotted module names of IPython extensions to load.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

`line_profiler` can be used programmatically (see the full documentation), but it is perhaps most powerful when used interactively in IPython. Suppose you had a module `prof_mod` with the following code doing some NumPy array operations:

```
from numpy.random import randn

def add_and_sum(x, y):
    added = x + y
    summed = added.sum(axis=1)
    return summed

def call_function():
    x = randn(1000, 1000)
    y = randn(1000, 1000)
    return add_and_sum(x, y)
```

If we wanted to understand the performance of the `add_and_sum` function, `%prun` gives us the following:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
4 function calls in 0.049 seconds
Ordered by: internal time
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.036    0.036    0.046    0.046  prof_mod.py:3(add_and_sum)
```

```

1    0.009    0.009    0.009    0.009 {method 'sum' of 'numpy.ndarray' objects}
1    0.003    0.003    0.049    0.049 <string>:1(<module>)
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

```

This is not especially enlightening. With the `line_profiler` IPython extension activated, a new command `%lprun` is available. The only difference in usage is that we must instruct `%lprun` which function or functions we wish to profile. The general syntax is:

```
%lprun -f func1 -f func2 statement_to_profile
```

In this case, we want to profile `add_and_sum`, so we run:

```

In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s

```

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def add_and_sum(x, y):
4	1	36510	36510.0	79.5	added = x + y
5	1	9425	9425.0	20.5	summed = added.sum(axis=1)
6	1	1	1.0	0.0	return summed

You'll probably agree this is much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the module code above, we could call `call_function` and profile that as well as `add_and_sum`, thus getting a full picture of the performance of the code:

```

In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s

```

Line #	Hits	Time	Per Hit	% Time	Line Contents
3					def add_and_sum(x, y):
4	1	4375	4375.0	79.2	added = x + y
5	1	1149	1149.0	20.8	summed = added.sum(axis=1)
6	1	2	2.0	0.0	return summed

```

File: book_scripts/prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s

```

Line #	Hits	Time	Per Hit	% Time	Line Contents
8					def call_function():
9	1	57169	57169.0	47.2	x = randn(1000, 1000)
10	1	58304	58304.0	48.2	y = randn(1000, 1000)
11	1	5543	5543.0	4.6	return add_and_sum(x, y)

As a general rule of thumb, I tend to prefer `%prun` (cProfile) for “macro” profiling and `%lprun` (line\_profiler) for “micro” profiling. It's worthwhile to have a good understanding of both tools.



The reason that you have to specify explicitly the names of the functions you want to profile with `%lprun` is that the overhead of “tracing” the execution time of each line is significant. Tracing functions that are not of interest would potentially significantly alter the profile results.

## IPython HTML Notebook

Starting in 2011, the IPython team, led by Brian Granger, built a web technology-based interactive computational document format that is commonly known as the IPython Notebook. It has grown into a wonderful tool for interactive computing and an ideal medium for reproducible research and teaching. I’ve used it while writing most of the examples in the book; I encourage you to make use of it, too.

It has a JSON-based `.ipynb` document format that enables easy sharing of code, output, and figures. Recently in Python conferences, a popular approach for demonstrations has been to use the notebook and post the `.ipynb` files online afterward for everyone to play with.

The notebook application runs as a lightweight server process on the command line. It can be started by running:

```
$ ipython notebook --pylab=inline
[NotebookApp] Using existing profile dir: u'/home/wesm/.config/ipython/profile_default'
[NotebookApp] Serving notebooks from /home/wesm/book_scripts
[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```

On most platforms, your primary web browser will automatically open up to the notebook dashboard. In some cases you may have to navigate to the listed URL. From there, you can create a new notebook and start exploring.

Since you use the notebook inside a web browser, the server process can run anywhere. You can even securely connect to notebooks running on cloud service providers like Amazon EC2. As of this writing, a new project NotebookCloud (<http://notebookcloud.appspot.com>) makes it easy to launch notebooks on EC2.

## Tips for Productive Code Development Using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately *use* interactively may be a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.

As such, most of this section is more of an art than a science and will require some experimentation on your part to determine a way to write your Python code that is effective and productive for you. Ultimately you want to structure your code in a way that makes it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible. I have found software designed with

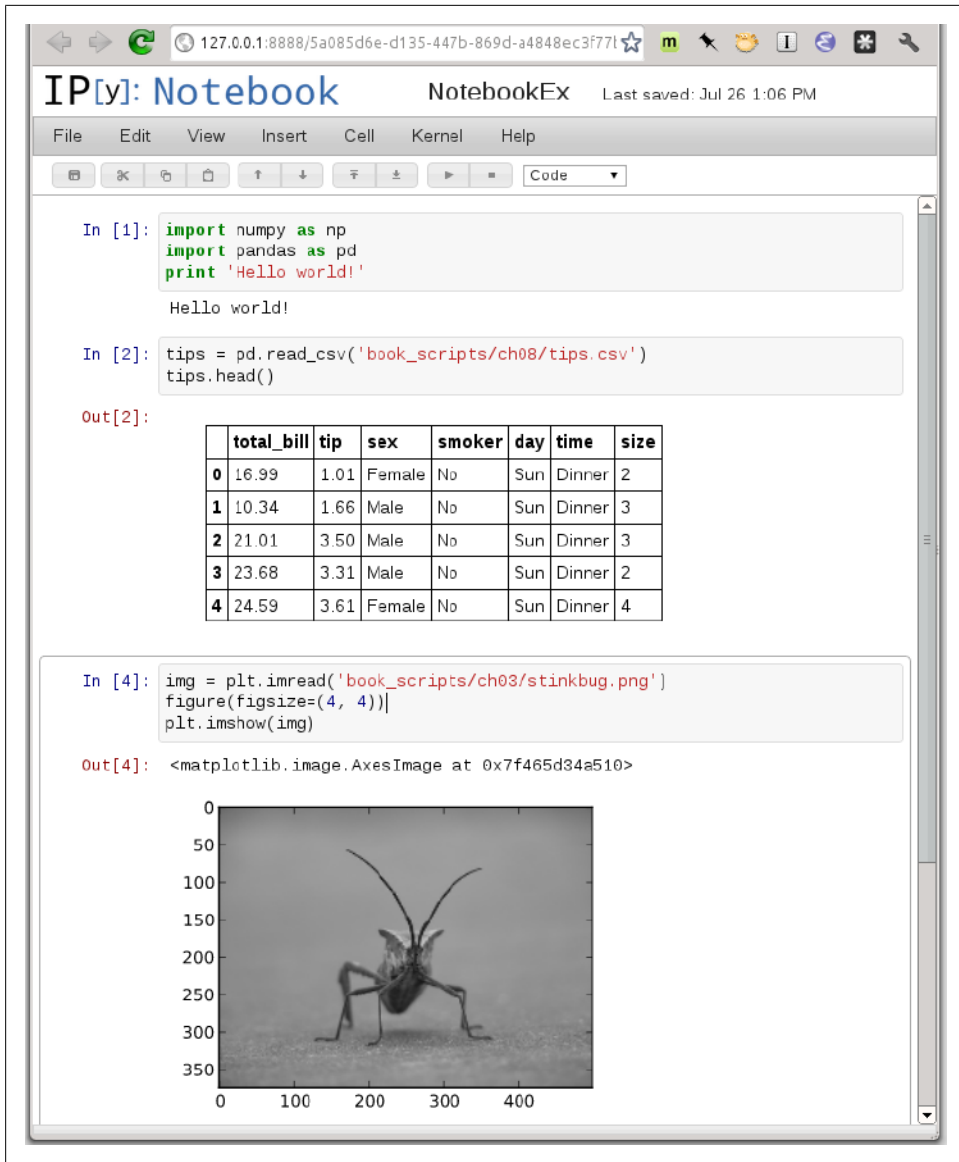


Figure 3-4. IPython Notebook

IPython in mind to be easier to work with than code intended only to be run as a standalone command-line application. This becomes especially important when something goes wrong and you have to diagnose an error in code that you or someone else might have written months or years beforehand.

## Reloading Module Dependencies

In Python, when you type `import some_lib`, the code in `some_lib` is executed and all the variables, functions, and imports defined within are stored in the newly created `some_lib` module namespace. The next time you type `import some_lib`, you will get a reference to the existing module namespace. The potential difficulty in interactive code development in IPython comes when you, say, `%run` a script that depends on some other module where you may have made changes. Suppose I had the following code in `test_script.py`:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

If you were to execute `%run test_script.py` then modify `some_lib.py`, the next time you execute `%run test_script.py` you will still get the *old version* of `some_lib` because of Python’s “load-once” module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes.<sup>1</sup> To cope with this, you have a couple of options. The first way is to use Python’s built-in `reload` function, altering `test_script.py` to look like the following:

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

This guarantees that you will get a fresh copy of `some_lib` every time you run `test_script.py`. Obviously, if the dependencies go deeper, it might be a bit tricky to be inserting usages of `reload` all over the place. For this problem, IPython has a special `dreload` function (*not* a magic function) for “deep” (recursive) reloading of modules. If I were to run `import some_lib` then type `dreload(some_lib)`, it will attempt to reload `some_lib` as well as all of its dependencies. This will not work in all cases, unfortunately, but when it does it beats having to restart IPython.

## Code Design Tips

There’s no simple recipe for this, but here are some high-level principles I have found effective in my own work.

1. Since a module or package may be imported in many different places in a particular program, Python caches a module’s code the first time it is imported rather than executing the code in the module every time. Otherwise, modularity and good code organization could potentially cause inefficiency in an application.

## Keep relevant objects and data alive

It's not unusual to see a program written for the command line with a structure somewhat like the following trivial example:

```
from my_functions import g

def f(x, y):
    return g(x + y)

def main():
    x = 6
    y = 7.5
    result = x + y

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

Do you see what might be wrong with this program if we were to run it in IPython? After it's done, none of the results or objects defined in the `main` function will be accessible in the IPython shell. A better way is to have whatever code is in `main` execute directly in the module's global namespace (or in the `if __name__ == '__main__':` block, if you want the module to also be importable). That way, when you `%run` the code, you'll be able to look at all of the variables defined in `main`. It's less meaningful in this simple example, but in this book we'll be looking at some complex data analysis problems involving large data sets that you will want to be able to play with in IPython.

## Flat is better than nested

Deeply nested code makes me think about the many layers of an onion. When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest? The idea that “flat is better than nested” is a part of the Zen of Python, and it applies generally to developing code for interactive use as well. Making functions and classes as decoupled and modular as possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

## Overcome a fear of longer files

If you come from a Java (or another such language) background, you may have been told to keep files short. In many languages, this is sound advice; long length is usually a bad “code smell”, indicating refactoring or reorganization may be necessary. However, while developing code using IPython, working with 10 small, but interconnected files (under, say, 100 lines each) is likely to cause you more headache in general than a single large file or two or three longer files. Fewer files means fewer modules to reload and less jumping between files while editing, too. I have found maintaining larger modules, each with high *internal* cohesion, to be much more useful and pythonic. After iterating toward a solution, it sometimes will make sense to refactor larger files into smaller ones.

Obviously, I don't support taking this argument to the extreme, which would be to put all of your code in a single monstrous file. Finding a sensible and intuitive module and package structure for a large codebase often takes a bit of work, but it is especially important to get right in teams. Each module should be internally cohesive, and it should be as obvious as possible where to find functions and classes responsible for each area of functionality.

## Advanced IPython Features

### Making Your Own Classes IPython-friendly

IPython makes every effort to display a console-friendly string representation of any object that you inspect. For many objects, like dicts, lists, and tuples, the built-in `pprint` module is used to do the nice formatting. In user-defined classes, however, you have to generate the desired string output yourself. Suppose we had the following simple class:

```
class Message:
    def __init__(self, msg):
        self.msg = msg
```

If you wrote this, you would be disappointed to discover that the default output for your class isn't very nice:

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython takes the string returned by the `__repr__` magic method (by doing `output = repr(obj)`) and prints that to the console. Thus, we can add a simple `__repr__` method to the above class to get a more helpful output:

```
class Message:
    def __init__(self, msg):
        self.msg = msg

    def __repr__(self):
        return 'Message: %s' % self.msg

In [579]: x = Message('I have a secret')

In [580]: x
Out[580]: Message: I have a secret
```



## Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython shell are configurable through an extensive configuration system. Here are some of the things you can do via configuration:

- Change the color scheme
- Change how the input and output prompts look, or remove the blank line after Out and before the next In prompt
- Change how the input and output prompts look
- Execute an arbitrary list of Python statements. These could be imports that you use all the time or anything else you want to happen each time you launch IPython
- Enable IPython extensions, like the `%lprun` magic in `line_profiler`
- Define your own magics or system aliases

All of these configuration options are specified in a special `ipython_config.py` file which will be found in the `~/.config/ipython/` directory on UNIX-like systems and `%HOME%/.ipython/` directory on Windows. Where your home directory is depends on your system. Configuration is performed based on a particular *profile*. When you start IPython normally, you load up, by default, the *default profile*, stored in the `profile_default` directory. Thus, on my Linux OS the full path to my default IPython configuration file is:

```
/home/wesm/.config/ipython/profile_default/ipython_config.py
```

I'll spare you the gory details of what's in this file. Fortunately it has comments describing what each configuration option is for, so I will leave it to the reader to tinker and customize. One additional useful feature is that it's possible to have *multiple profiles*. Suppose you wanted to have an alternate IPython configuration tailored for a particular application or project. Creating a new profile is as simple as typing something like

```
ipython profile create secret_project
```

Once you've done this, edit the config files in the newly-created `profile_secret_project` directory then launch IPython like so

```
$ ipython --profile=secret_project
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul  3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project
```

In [1]:

As always, the online IPython documentation is an excellent resource for more on profiles and configuration.

## Credits

Parts of this chapter were derived from the wonderful documentation put together by the IPython Development Team. I can't thank them enough for all of their work building this amazing set of tools.

---

# NumPy Basics: Arrays and Vectorized Computation

NumPy, short for Numerical Python, is the fundamental package required for high performance scientific computing and data analysis. It is the foundation on which nearly all of the higher-level tools in this book are built. Here are some of the things it provides:

- `ndarray`, a fast and space-efficient multidimensional array providing vectorized arithmetic operations and sophisticated *broadcasting* capabilities
- Standard mathematical functions for fast operations on entire arrays of data without having to write loops
- Tools for reading / writing array data to disk and working with memory-mapped files
- Linear algebra, random number generation, and Fourier transform capabilities
- Tools for integrating code written in C, C++, and Fortran

The last bullet point is also one of the most important ones from an ecosystem point of view. Because NumPy provides an easy-to-use C API, it is very easy to pass data to external libraries written in a low-level language and also for external libraries to return data to Python as NumPy arrays. This feature has made Python a language of choice for wrapping legacy C/C++/Fortran codebases and giving them a dynamic and easy-to-use interface.

While NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of NumPy arrays and array-oriented computing will help you use tools like pandas much more effectively. If you're new to Python and just looking to get your hands dirty working with data using pandas, feel free to give this chapter a skim. For more on advanced NumPy features like broadcasting, see [Chapter 12](#).

For most data analysis applications, the main areas of functionality I'll focus on are:

- Fast vectorized array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application). Much more on this in [Chapter 5](#)

While NumPy provides the computational foundation for these operations, you will likely want to use pandas as your basis for most kinds of data analysis (especially for structured or tabular data) as it provides a rich, high-level interface making most common data tasks very concise and simple. pandas also provides some more domain-specific functionality like time series manipulation, which is not present in NumPy.



In this chapter and throughout the book, I use the standard NumPy convention of always using `import numpy as np`. You are, of course, welcome to put `from numpy import *` in your code to avoid having to write `np.`, but I would caution you against making a habit of this.

## The NumPy ndarray: A Multidimensional Array Object

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data
Out[8]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])
```

```
In [9]: data * 10
Out[9]:
array([[ 9.5256, -2.4601, -8.8565],
       [ 5.6385,  2.3794,  9.104 ]])
```

```
In [10]: data + data
Out[10]:
array([[ 1.9051, -0.492 , -1.7713],
       [ 1.1277,  0.4759,  1.8208]])
```

An ndarray is a generic multidimensional container for homogeneous data; that is, all of the elements must be the same type. Every array has a `shape`, a tuple indicating the size of each dimension, and a `dtype`, an object describing the *data type* of the array:

```
In [11]: data.shape
Out[11]: (2, 3)
```

```
In [12]: data.dtype
Out[12]: dtype('float64')
```

This chapter will introduce you to the basics of using NumPy arrays, and should be sufficient for following along with the rest of the book. While it's not necessary to have a deep understanding of NumPy for many data analytical applications, becoming proficient in array-oriented programming and thinking is a key step along the way to becoming a scientific Python guru.



Whenever you see “array”, “NumPy array”, or “ndarray” in the text, with few exceptions they all refer to the same thing: the ndarray object.

## Creating ndarrays

The easiest way to create an array is to use the `array` function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data. For example, a list is a good candidate for conversion:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]

In [14]: arr1 = np.array(data1)

In [15]: arr1
Out[15]: array([ 6. ,  7.5,  8. ,  0. ,  1. ])
```

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

In [19]: arr2.ndim
Out[19]: 2

In [20]: arr2.shape
Out[20]: (2, 4)
```

Unless explicitly specified (more on this later), `np.array` tries to infer a good data type for the array that it creates. The data type is stored in a special `dtype` object; for example, in the above two examples we have:

```
In [21]: arr1.dtype
Out[21]: dtype('float64')
```

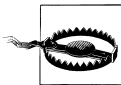
```
In [22]: arr2.dtype
Out[22]: dtype('int64')
```

In addition to `np.array`, there are a number of other functions for creating new arrays. As examples, `zeros` and `ones` create arrays of 0's or 1's, respectively, with a given length or shape. `empty` creates an array without initializing its values to any particular value. To create a higher dimensional array with these methods, pass a tuple for the shape:

```
In [23]: np.zeros(10)
Out[23]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
```

```
In [24]: np.zeros((3, 6))
Out[24]:
array([[ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [25]: np.empty((2, 3, 2))
Out[25]:
array([[[ 4.94065646e-324,  4.94065646e-324],
        [ 3.87491056e-297,  2.46845796e-130],
        [ 4.94065646e-324,  4.94065646e-324]],
       [[ 1.90723115e+083,  5.73293533e-053],
        [-2.33568637e+124, -6.70608105e-012],
        [ 4.42786966e+160,  1.27100354e+025]]])
```



It's not safe to assume that `np.empty` will return an array of all zeros. In many cases, as previously shown, it will return uninitialized garbage values.

`arange` is an array-valued version of the built-in Python `range` function:

```
In [26]: np.arange(15)
Out[26]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

See [Table 4-1](#) for a short list of standard array creation functions. Since NumPy is focused on numerical computing, the data type, if not specified, will in many cases be `float64` (floating point).

*Table 4-1. Array creation functions*

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list.
<code>ones</code> , <code>ones_like</code>	Produce an array of all 1's with the given shape and dtype. <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead

Function	Description
<code>empty</code> , <code>empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
<code>eye</code> , <code>identity</code>	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

## Data Types for ndarrays

The *data type* or `dtype` is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)
```

```
In [29]: arr1.dtype      In [30]: arr2.dtype
Out[29]: dtype('float64') Out[30]: dtype('int32')
```

Dtypes are part of what make NumPy so powerful and flexible. In most cases they map directly onto an underlying machine representation, which makes it easy to read and write binary streams of data to disk and also to connect to code written in a low-level language like C or Fortran. The numerical dtypes are named the same way: a type name, like `float` or `int`, followed by a number indicating the number of bits per element. A standard double-precision floating point value (what's used under the hood in Python's `float` object) takes up 8 bytes or 64 bits. Thus, this type is known in NumPy as `float64`. See [Table 4-2](#) for a full listing of NumPy's supported data types.



Don't worry about memorizing the NumPy dtypes, especially if you're a new user. It's often only necessary to care about the general *kind* of data you're dealing with, whether floating point, complex, integer, boolean, string, or general Python object. When you need more control over how data are stored in memory and on disk, especially large data sets, it is good to know that you have control over the storage type.

Table 4-2. NumPy data types

Type	Type Code	Description
<code>int8</code> , <code>uint8</code>	<code>i1</code> , <code>u1</code>	Signed and unsigned 8-bit (1 byte) integer types
<code>int16</code> , <code>uint16</code>	<code>i2</code> , <code>u2</code>	Signed and unsigned 16-bit integer types
<code>int32</code> , <code>uint32</code>	<code>i4</code> , <code>u4</code>	Signed and unsigned 32-bit integer types
<code>int64</code> , <code>uint64</code>	<code>i8</code> , <code>u8</code>	Signed and unsigned 32-bit integer types
<code>float16</code>	<code>f2</code>	Half-precision floating point
<code>float32</code>	<code>f4</code> or <code>f</code>	Standard single-precision floating point. Compatible with C float
<code>float64</code>	<code>f8</code> or <code>d</code>	Standard double-precision floating point. Compatible with C double and Python <code>float</code> object

Type	Type Code	Description
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

You can explicitly convert or *cast* an array from one dtype to another using ndarray's `astype` method:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])

In [32]: arr.dtype
Out[32]: dtype('int64')

In [33]: float_arr = arr.astype(np.float64)

In [34]: float_arr.dtype
Out[34]: dtype('float64')
```

In this example, integers were cast to floating point. If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [36]: arr
Out[36]: array([ 3.7, -1.2, -2.6,  0.5, 12.9, 10.1])

In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2,  0, 12, 10], dtype=int32)
```

Should you have an array of strings representing numbers, you can use `astype` to convert them to numeric form:

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [39]: numeric_strings.astype(float)
Out[39]: array([ 1.25, -9.6, 42.])
```

If casting were to fail for some reason (like a string that cannot be converted to `float64`), a `TypeError` will be raised. See that I was a bit lazy and wrote `float` instead of `np.float64`; NumPy is smart enough to alias the Python types to the equivalent dtypes.

You can also use another array's dtype attribute:

```
In [40]: int_array = np.arange(10)
```



```
In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.]
```

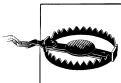
There are shorthand type code strings you can also use to refer to a dtype:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')

In [44]: empty_uint32
Out[44]:
array([ 0, 0, 65904672, 0, 64856792, 0,
       39438163, 0], dtype=uint32)
```



Calling `astype` *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.



It's worth keeping in mind that floating point numbers, such as those in `float64` and `float32` arrays, are only capable of approximating fractional quantities. In complex computations, you may accrue some *floating point error*, making comparisons only valid up to a certain number of decimal places.

## Operations between Arrays and Scalars

Arrays are important because they enable you to express batch operations on data without writing any `for` loops. This is usually called *vectorization*. Any arithmetic operations between equal-size arrays applies the operation elementwise:

```
In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [46]: arr
Out[46]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [47]: arr * arr
Out[47]:
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

In [48]: arr - arr
Out[48]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Arithmetic operations with scalars are as you would expect, propagating the value to each element:

```
In [49]: 1 / arr
Out[49]:
array([[ 1.    ,  0.5    ,  0.3333],
       [ 0.25   ,  0.2    ,  0.1667]])

In [50]: arr ** 0.5
Out[50]:
array([[ 1.    ,  1.4142,  1.7321],
       [ 2.    ,  2.2361,  2.4495]])
```

Operations between differently sized arrays is called *broadcasting* and will be discussed in more detail in [Chapter 12](#). Having a deep understanding of broadcasting is not necessary for most of this book.

## Basic Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements. One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
Out[56]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or *broadcasted* henceforth) to the entire selection. An important first distinction from lists is that array slices are *views* on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array:

```
In [57]: arr_slice = arr[5:8]

In [58]: arr_slice[1] = 12345

In [59]: arr
Out[59]: array([ 0,  1,  2,  3,  4, 12, 12345, 12,  8,  9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

If you are new to NumPy, you might be surprised by this, especially if you have used other array programming languages which copy data more zealously. As NumPy has been designed with large data use cases in mind, you could imagine performance and memory problems if NumPy insisted on copying data left and right.



If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example `arr[5:8].copy()`.

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
In [63]: arr2d[2]
Out[63]: array([7, 8, 9])
```

Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements. So these are equivalent:

```
In [64]: arr2d[0][2]
Out[64]: 3
```

```
In [65]: arr2d[0, 2]
Out[65]: 3
```

See [Figure 4-1](#) for an illustration of indexing on a 2D array.

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2

Figure 4-1. Indexing elements in a NumPy array

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the  $2 \times 2 \times 3$  array `arr3d`

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

```
In [67]: arr3d
Out[67]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
[ 4,  5,  6]],
[[ 7,  8,  9],
 [10, 11, 12]])
```

`arr3d[0]` is a  $2 \times 3$  array:

```
In [68]: arr3d[0]
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Both scalar values and arrays can be assigned to `arr3d[0]`:

```
In [69]: old_values = arr3d[0].copy()
```

```
In [70]: arr3d[0] = 42
```

```
In [71]: arr3d
Out[71]:
array([[42, 42, 42],
       [42, 42, 42]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

```
In [72]: arr3d[0] = old_values
```

```
In [73]: arr3d
Out[73]:
array([[ 1,  2,  3],
       [ 4,  5,  6]],
      [[ 7,  8,  9],
       [10, 11, 12]])
```

Similarly, `arr3d[1, 0]` gives you all of the values whose indices start with `(1, 0)`, forming a 1-dimensional array:

```
In [74]: arr3d[1, 0]
Out[74]: array([7, 8, 9])
```

Note that in all of these cases where subsections of the array have been selected, the returned arrays are views.

## Indexing with slices

Like one-dimensional objects such as Python lists, `ndarrays` can be sliced using the familiar syntax:

```
In [75]: arr[1:6]
Out[75]: array([ 1,  2,  3,  4, 64])
```

Higher dimensional objects give you more options as you can slice one or more axes and also mix integers. Consider the 2D array above, `arr2d`. Slicing this array is a bit different:

```
In [76]: arr2d
Out[76]:
In [77]: arr2d[:2]
Out[77]:
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
array([[1, 2, 3],
       [4, 5, 6]])
```

As you can see, it has sliced along axis 0, the first axis. A slice, therefore, selects a range of elements along an axis. You can pass multiple slices just like you can pass multiple indexes:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
       [5, 6]])
```

When slicing like this, you always obtain array views of the same number of dimensions. By mixing integer indexes and slices, you get lower dimensional slices:

```
In [79]: arr2d[1, :2]
Out[79]: array([4, 5])
In [80]: arr2d[2, :1]
Out[80]: array([7])
```

See [Figure 4-2](#) for an illustration. Note that a colon by itself means to take the entire axis, so you can slice only higher dimensional axes by doing:

```
In [81]: arr2d[:, :1]
Out[81]:
array([[1],
       [4],
       [7]])
```

Of course, assigning to a slice expression assigns to the whole selection:

```
In [82]: arr2d[:2, 1:] = 0
```

## Boolean Indexing

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the `randn` function in `numpy.random` to generate some random normally distributed data:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
```

```
In [84]: data = randn(7, 4)
```

```
In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
      dtype='<S4')
```

```
In [86]: data
Out[86]:
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289],
```

```
[ 0.1913,  0.4544,  0.4519,  0.5535],
[ 0.5994,  0.8174, -0.9297, -1.2564]])
```

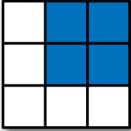
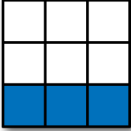
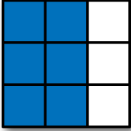
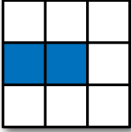
	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code>	<code>(3,)</code>
	<code>arr[2, :]</code>	<code>(3,)</code>
	<code>arr[2:, :]</code>	<code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code>	<code>(2,)</code>
	<code>arr[1:2, :2]</code>	<code>(1, 2)</code>

Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the `data` array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing `names` with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False,  True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[ -0.2349,  1.2792],
```

```
[-0.0523, 0.0672]])
```

```
In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792, 0.0672])
```

To select everything but 'Bob', you can either use `!=` or negate the condition using `~`:

```
In [91]: names != 'Bob'
Out[91]: array([False,  True,  True, False,  True,  True,  True], dtype=bool)
```

```
In [92]: data[~(names == 'Bob')]
Out[92]:
array([[ -0.268 ,  0.5465,  0.0939, -2.0445],
       [ -0.047 , -2.026 ,  0.7719,  0.3103],
       [ -1.0023, -0.1698,  1.1503,  1.7289],
       [  0.1913,  0.4544,  0.4519,  0.5535],
       [  0.5994,  0.8174, -0.9297, -1.2564]])
```

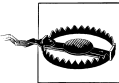
Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like `&` (and) and `|` (or):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([ True, False,  True,  True,  True, False, False], dtype=bool)
```

```
In [95]: data[mask]
Out[95]:
array([[ -0.048 ,  0.5433, -0.2349,  1.2792],
       [ -0.047 , -2.026 ,  0.7719,  0.3103],
       [  2.1452,  0.8799, -0.0523,  0.0672],
       [ -1.0023, -0.1698,  1.1503,  1.7289]])
```

Selecting data from an array by boolean indexing *always* creates a copy of the data, even if the returned array is unchanged.



The Python keywords `and` and `or` do not work with boolean arrays.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in `data` to 0 we need only do:

```
In [96]: data[data < 0] = 0

In [97]: data
Out[97]:
array([[ 0.    ,  0.5433,  0.    ,  1.2792],
       [ 0.    ,  0.5465,  0.0939,  0.    ],
       [ 0.    ,  0.    ,  0.7719,  0.3103],
       [ 2.1452,  0.8799,  0.    ,  0.0672],
       [ 0.    ,  0.    ,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.    ,  0.    ]])
```

Setting whole rows or columns using a 1D boolean array is also easy:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
array([[ 7.,    7.,    7.,    7. ],
       [ 0.,    0.5465,  0.0939,  0. ],
       [ 7.,    7.,    7.,    7. ],
       [ 7.,    7.,    7.,    7. ],
       [ 7.,    7.,    7.,    7. ],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174,  0.,    0. ]])
```

## Fancy Indexing

*Fancy indexing* is a term adopted by NumPy to describe indexing using integer arrays. Suppose we had a  $8 \times 4$  array:

```
In [100]: arr = np.empty((8, 4))

In [101]: for i in range(8):
.....:     arr[i] = i

In [102]: arr
Out[102]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.],
       [ 5.,  5.,  5.,  5.],
       [ 6.,  6.,  6.,  6.],
       [ 7.,  7.,  7.,  7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4.,  4.,  4.,  4.],
       [ 3.,  3.,  3.,  3.],
       [ 0.,  0.,  0.,  0.],
       [ 6.,  6.,  6.,  6.]])
```

Hopefully this code did what you expected! Using negative indices select rows from the end:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5.,  5.,  5.,  5.],
       [ 3.,  3.,  3.,  3.],
       [ 1.,  1.,  1.,  1.]])
```



Passing multiple index arrays does something slightly different; it selects a 1D array of elements corresponding to each tuple of indices:

```
# more on reshape in Chapter 12
In [105]: arr = np.arange(32).reshape((8, 4))
```

```
In [106]: arr
Out[106]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
```

```
In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Take a moment to understand what just happened: the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. The behavior of fancy indexing in this case is a bit different from what some users might have expected (myself included), which is the rectangular region formed by selecting a subset of the matrix's rows and columns. Here is one way to get that:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Another way is to use the `np.ix_` function, which converts two 1D integer arrays to an indexer that selects the square region:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Keep in mind that fancy indexing, unlike slicing, always copies the data into a new array.

## Transposing Arrays and Swapping Axes

Transposing is a special form of reshaping which similarly returns a view on the underlying data without copying anything. Arrays have the `transpose` method and also the special `T` attribute:

```
In [110]: arr = np.arange(15).reshape((3, 5))
```

```
In [111]: arr
```

```
In [112]: arr.T
```

```

Out[111]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

Out[112]:
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])

```

When doing matrix computations, you will do this very often, like for example computing the inner matrix product  $X^T X$  using `np.dot`:

```

In [113]: arr = np.random.randn(6, 3)

In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 ,  1.8753,  0.8888],
       [ 1.8753,  6.6636,  0.3884],
       [ 0.8888,  0.3884,  3.9781]])

```

For higher dimensional arrays, `transpose` will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```

In [115]: arr = np.arange(16).reshape((2, 2, 4))

In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])

```

Simple transposing with `.T` is just a special case of swapping axes. `ndarray` has the method `swapaxes` which takes a pair of axis numbers:

```

In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [119]: arr.swapaxes(1, 2)
Out[119]:
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])

```

`swapaxes` similarly returns a view on the data without making a copy.

# Universal Functions: Fast Element-wise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

Many ufuncs are simple elementwise transformations, like `sqrt` or `exp`:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.    ,  1.    ,  1.4142,  1.7321,  2.    ,  2.2361,  2.4495,
        2.6458,  2.8284,  3.    ])

In [122]: np.exp(arr)
Out[122]:
array([  1.    ,   2.7183,   7.3891,  20.0855,  54.5982,
 148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

These are referred to as *unary* ufuncs. Others, such as `add` or `maximum`, take 2 arrays (thus, *binary* ufuncs) and return a single array as the result:

```
In [123]: x = randn(8)

In [124]: y = randn(8)

In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655,  0.9222,  0.446 ,
        -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323,  0.4788,  0.4315,
        -0.7147])

In [127]: np.maximum(x, y) # element-wise maximum
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655,  0.9222,  0.446 ,
        -0.7147])
```

While not common, a ufunc can return multiple arrays. `modf` is one example, a vectorized version of the built-in Python `divmod`: it returns the fractional and integral parts of a floating point array:

```
In [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808,  0.0636, -0.386 ,  0.1393, -0.8806,  0.9363, -0.883 ]),
 array([-2.,  4., -3.,  5., -3.,  3., -6.]))
```

See [Table 4-3](#) and [Table 4-4](#) for a listing of available ufuncs.

*Table 4-3. Unary ufuncs*

Function	Description
<code>abs</code> , <code>fabs</code>	Compute the absolute value element-wise for integer, floating point, or complex values. Use <code>fabs</code> as a faster alternative for non-complex-valued data
<code>sqrt</code>	Compute the square root of each element. Equivalent to <code>arr ** 0.5</code>
<code>square</code>	Compute the square of each element. Equivalent to <code>arr ** 2</code>
<code>exp</code>	Compute the exponent $e^x$ of each element
<code>log</code> , <code>log10</code> , <code>log2</code> , <code>log1p</code>	Natural logarithm (base $e$ ), log base 10, log base 2, and $\log(1 + x)$ , respectively
<code>sign</code>	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
<code>ceil</code>	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
<code>floor</code>	Compute the floor of each element, i.e. the largest integer less than or equal to each element
<code>rint</code>	Round elements to the nearest integer, preserving the dtype
<code>modf</code>	Return fractional and integral parts of array as separate array
<code>isnan</code>	Return boolean array indicating whether each value is NaN (Not a Number)
<code>isfinite</code> , <code>isinf</code>	Return boolean array indicating whether each element is finite (non- <code>inf</code> , non-NaN) or infinite, respectively
<code>cos</code> , <code>cosh</code> , <code>sin</code> , <code>sinh</code> , <code>tan</code> , <code>tanh</code>	Regular and hyperbolic trigonometric functions
<code>arccos</code> , <code>arccosh</code> , <code>arcsin</code> , <code>arcsinh</code> , <code>arctan</code> , <code>arctanh</code>	Inverse trigonometric functions
<code>logical_not</code>	Compute truth value of <code>not x</code> element-wise. Equivalent to <code>-arr</code> .

*Table 4-4. Binary universal functions*

Function	Description
<code>add</code>	Add corresponding elements in arrays
<code>subtract</code>	Subtract elements in second array from first array
<code>multiply</code>	Multiply array elements
<code>divide</code> , <code>floor_divide</code>	Divide or floor divide (truncating the remainder)
<code>power</code>	Raise elements in first array to powers indicated in second array
<code>maximum</code> , <code>fmax</code>	Element-wise maximum. <code>fmax</code> ignores NaN
<code>minimum</code> , <code>fmin</code>	Element-wise minimum. <code>fmin</code> ignores NaN
<code>mod</code>	Element-wise modulus (remainder of division)
<code>copysign</code>	Copy sign of values in second argument to values in first argument

Function	Description
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=
logical_and, logical_or, logical_xor	Compute element-wise truth value of logical operation. Equivalent to infix operators & , ^

## Data Processing Using Arrays

Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as *vectorization*. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in [Chapter 12](#), I will explain *broadcasting*, a powerful method for vectorizing computations.

As a simple example, suppose we wished to evaluate the function  $\sqrt{x^2 + y^2}$  across a regular grid of values. The `np.meshgrid` function takes two 1D arrays and produces two 2D matrices corresponding to all pairs of  $(x, y)$  in the two arrays:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
```

```
In [131]: xs, ys = np.meshgrid(points, points)
```

```
In [132]: ys
```

```
Out[132]:
array([[ -5.   , -5.   , -5.   , ..., -5.   , -5.   , -5.   ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Now, evaluating the function is a simple matter of writing the same expression you would write with two points:

```
In [134]: import matplotlib.pyplot as plt
```

```
In [135]: z = np.sqrt(xs ** 2 + ys ** 2)
```

```
In [136]: z
```

```
Out[136]:
array([[ 7.0711,  7.064 ,  7.0569, ...,  7.0499,  7.0569,  7.064 ],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       ...,
       [ 7.0499,  7.0428,  7.0357, ...,  7.0286,  7.0357,  7.0428],
       [ 7.0569,  7.0499,  7.0428, ...,  7.0357,  7.0428,  7.0499],
       [ 7.064 ,  7.0569,  7.0499, ...,  7.0428,  7.0499,  7.0569]])
```

```
In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[137]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [138]: plt.title("Image plot of  $\sqrt{x^2 + y^2}$  for a grid of values")
Out[138]: <matplotlib.text.Text at 0x4565790>
```

See [Figure 4-3](#). Here I used the matplotlib function `imshow` to create an image plot from a 2D array of function values.

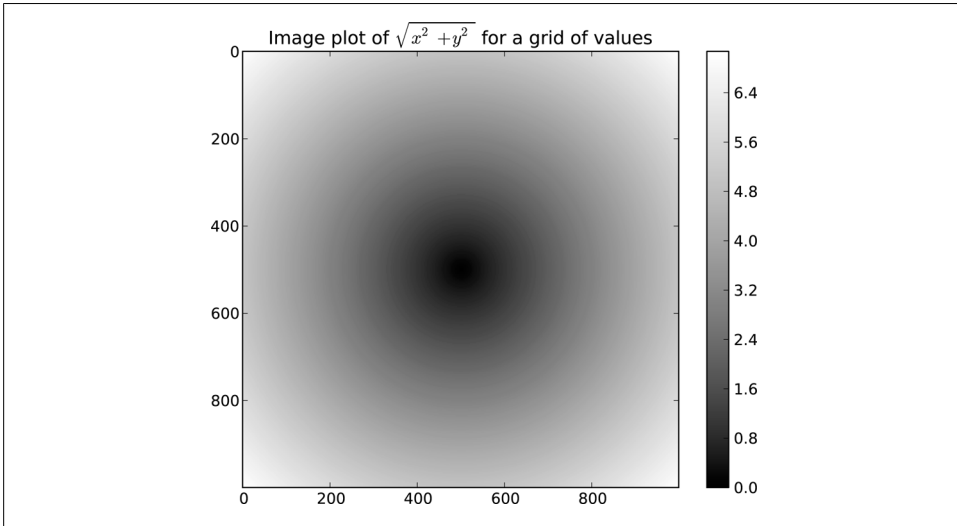


Figure 4-3. Plot of function evaluated on grid

## Expressing Conditional Logic as Array Operations

The `numpy.where` function is a vectorized version of the ternary expression `x if condition else y`. Suppose we had a boolean array and two arrays of values:

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
In [142]: cond = np.array([True, False, True, True, False])
```

Suppose we wanted to take a value from `xarr` whenever the corresponding value in `cond` is `True` otherwise take the value from `yarr`. A list comprehension doing this might look like:

```
In [143]: result = [(x if c else y)
.....:                 for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

This has multiple problems. First, it will not be very fast for large arrays (because all the work is being done in pure Python). Secondly, it will not work with multidimensional arrays. With `np.where` you can write this very concisely:

```
In [145]: result = np.where(cond, xarr, yarr)

In [146]: result
Out[146]: array([ 1.1,  2.2,  1.3,  1.4,  2.5])
```

The second and third arguments to `np.where` don't need to be arrays; one or both of them can be scalars. A typical use of `where` in data analysis is to produce a new array of values based on another array. Suppose you had a matrix of randomly generated data and you wanted to replace all positive values with 2 and all negative values with -2. This is very easy to do with `np.where`:

```
In [147]: arr = randn(4, 4)

In [148]: arr
Out[148]:
array([[ 0.6372,  2.2043,  1.7904,  0.0752],
       [-1.5926, -1.1536,  0.4413,  0.3483],
       [-0.1798,  0.3299,  0.7827, -0.7585],
       [ 0.5857,  0.1619,  1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)
Out[149]:
array([[ 2,  2,  2,  2],
       [-2, -2,  2,  2],
       [-2,  2,  2, -2],
       [ 2,  2,  2, -2]])

In [150]: np.where(arr > 0, 2, arr) # set only positive values to 2
Out[150]:
array([[ 2.      ,  2.      ,  2.      ,  2.      ],
       [-1.5926, -1.1536,  2.      ,  2.      ],
       [-0.1798,  2.      ,  2.      , -0.7585],
       [ 2.      ,  2.      ,  2.      , -1.3865]])
```

The arrays passed to `where` can be more than just equal sized array or scalars.

With some cleverness you can use `where` to express more complicated logic; consider this example where I have two boolean arrays, `cond1` and `cond2`, and wish to assign a different value for each of the 4 possible pairs of boolean values:

```
result = []
for i in range(n):
    if cond1[i] and cond2[i]:
        result.append(0)
    elif cond1[i]:
        result.append(1)
    elif cond2[i]:
        result.append(2)
    else:
        result.append(3)
```

While perhaps not immediately obvious, this `for` loop can be converted into a nested `where` expression:

```
np.where(cond1 & cond2, 0,
        np.where(cond1, 1,
                 np.where(cond2, 2, 3)))
```

In this particular example, we can also take advantage of the fact that boolean values are treated as 0 or 1 in calculations, so this could alternatively be expressed (though a bit more cryptically) as an arithmetic operation:

```
result = 1 * (cond1 & ~cond2) + 2 * (cond2 & ~cond1) + 3 * ~(cond1 | cond2)
```

## Mathematical and Statistical Methods

A set of mathematical functions which compute statistics about an entire array or about the data along an axis are accessible as array methods. Aggregations (often called *reductions*) like `sum`, `mean`, and standard deviation `std` can either be used by calling the array instance method or using the top level NumPy function:

```
In [151]: arr = np.random.randn(5, 4) # normally-distributed data
```

```
In [152]: arr.mean()
Out[152]: 0.062814911084854597
```

```
In [153]: np.mean(arr)
Out[153]: 0.062814911084854597
```

```
In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Functions like `mean` and `sum` take an optional `axis` argument which computes the statistic over the given axis, resulting in an array with one fewer dimension:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833,  0.2844,  0.6574,  0.6743, -0.0187])
```

```
In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189,  1.4044,  4.5712])
```

Other methods like `cumsum` and `cumprod` do not aggregate, instead producing an array of the intermediate results:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
```

```
In [158]: arr.cumsum(0)
Out[158]: array([[ 0,  1,  2],
 [ 3,  5,  7],
 [ 9, 12, 15]])

In [159]: arr.cumprod(1)
Out[159]: array([[ 0,  0,  0],
 [ 3, 12, 60],
 [ 6, 42, 336]])
```

See [Table 4-5](#) for a full listing. We'll see many examples of these methods in action in later chapters.



Table 4-5. Basic array statistical methods

Method	Description
<code>sum</code>	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
<code>mean</code>	Arithmetic mean. Zero-length arrays have NaN mean.
<code>std, var</code>	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator <i>n</i> ).
<code>min, max</code>	Minimum and maximum.
<code>argmin, argmax</code>	Indices of minimum and maximum elements, respectively.
<code>cumsum</code>	Cumulative sum of elements starting from 0
<code>cumprod</code>	Cumulative product of elements starting from 1

## Methods for Boolean Arrays

Boolean values are coerced to 1 (`True`) and 0 (`False`) in the above methods. Thus, `sum` is often used as a means of counting `True` values in a boolean array:

```
In [160]: arr = randn(100)
```

```
In [161]: (arr > 0).sum() # Number of positive values
Out[161]: 44
```

There are two additional methods, `any` and `all`, useful especially for boolean arrays. `any` tests whether one or more values in an array is `True`, while `all` checks if every value is `True`:

```
In [162]: bools = np.array([False, False, True, False])
```

```
In [163]: bools.any()
Out[163]: True
```

```
In [164]: bools.all()
Out[164]: False
```

These methods also work with non-boolean arrays, where non-zero elements evaluate to `True`.

## Sorting

Like Python's built-in list type, NumPy arrays can be sorted in-place using the `sort` method:

```
In [165]: arr = randn(8)
```

```
In [166]: arr
Out[166]:
array([ 0.6903,  0.4678,  0.0968, -0.1349,  0.9879,  0.0185, -1.3147,
        -0.5425])
```

```
In [167]: arr.sort()
```

```
In [168]: arr
Out[168]:
array([-1.3147, -0.5425, -0.1349,  0.0185,  0.0968,  0.4678,  0.6903,
        0.9879])
```

Multidimensional arrays can have each 1D section of values sorted in-place along an axis by passing the axis number to `sort`:

```
In [169]: arr = randn(5, 3)

In [170]: arr
Out[170]:
array([[ -0.7139, -1.6331, -0.4959],
       [  0.8236, -1.3132, -0.1935],
       [-1.6748,  3.0336, -0.863 ],
       [-0.3161,  0.5362, -2.468 ],
       [  0.9058,  1.1184, -1.0516]])

In [171]: arr.sort(1)

In [172]: arr
Out[172]:
array([[ -1.6331, -0.7139, -0.4959],
       [-1.3132, -0.1935,  0.8236],
       [-1.6748, -0.863 ,  3.0336],
       [-2.468 , -0.3161,  0.5362],
       [-1.0516,  0.9058,  1.1184]])
```

The top level method `np.sort` returns a sorted copy of an array instead of modifying the array in place. A quick-and-dirty way to compute the quantiles of an array is to sort it and select the value at a particular rank:

```
In [173]: large_arr = randn(1000)

In [174]: large_arr.sort()

In [175]: large_arr[int(0.05 * len(large_arr))] # 5% quantile
Out[175]: -1.5791023260896004
```

For more details on using NumPy's sorting methods, and more advanced techniques like indirect sorts, see [Chapter 12](#). Several other kinds of data manipulations related to sorting (for example, sorting a table of data by one or more columns) are also to be found in pandas.

## Unique and Other Set Logic

NumPy has some basic set operations for one-dimensional ndarrays. Probably the most commonly used one is `np.unique`, which returns the sorted unique values in an array:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
```

```
array(['Bob', 'Joe', 'Will'],
      dtype='<S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

Contrast `np.unique` with the pure Python alternative:

```
In [180]: sorted(set(names))
Out[180]: ['Bob', 'Joe', 'Will']
```

Another function, `np.in1d`, tests membership of the values in one array in another, returning a boolean array:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

See [Table 4-6](#) for a listing of set functions in NumPy.

*Table 4-6. Array set operations*

Method	Description
<code>unique(x)</code>	Compute the sorted, unique elements in <code>x</code>
<code>intersect1d(x, y)</code>	Compute the sorted, common elements in <code>x</code> and <code>y</code>
<code>union1d(x, y)</code>	Compute the sorted union of elements
<code>in1d(x, y)</code>	Compute a boolean array indicating whether each element of <code>x</code> is contained in <code>y</code>
<code>setdiff1d(x, y)</code>	Set difference, elements in <code>x</code> that are not in <code>y</code>
<code>setxor1d(x, y)</code>	Set symmetric differences; elements that are in either of the arrays, but not both

## File Input and Output with Arrays

NumPy is able to save and load data to and from disk either in text or binary format. In later chapters you will learn about tools in pandas for reading tabular data into memory.

### Storing Arrays on Disk in Binary Format

`np.save` and `np.load` are the two workhorse functions for efficiently saving and loading array data on disk. Arrays are saved by default in an uncompressed raw binary format with file extension `.npy`.

```
In [183]: arr = np.arange(10)

In [184]: np.save('some_array', arr)
```

If the file path does not already end in `.npz`, the extension will be appended. The array on disk can then be loaded using `np.load`:

```
In [185]: np.load('some_array.npy')
Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

You save multiple arrays in a zip archive using `np.savez` and passing the arrays as keyword arguments:

```
In [186]: np.savez('array_archive.npz', a=arr, b=arr)
```

When loading an `.npz` file, you get back a dict-like object which loads the individual arrays lazily:

```
In [187]: arch = np.load('array_archive.npz')

In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

## Saving and Loading Text Files

Loading text from files is a fairly standard task. The landscape of file reading and writing functions in Python can be a bit confusing for a newcomer, so I will focus mainly on the `read_csv` and `read_table` functions in pandas. It will at times be useful to load data into vanilla NumPy arrays using `np.loadtxt` or the more specialized `np.genfromtxt`.

These functions have many options allowing you to specify different delimiters, converter functions for certain columns, skipping rows, and other things. Take a simple case of a comma-separated file (CSV) like this:

```
In [191]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

This can be loaded into a 2D array like so:

```
In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')

In [193]: arr
Out[193]:
array([[ 0.5801,  0.1867,  1.0407,  1.1344],
       [ 0.1942, -0.6369, -0.9387,  0.1241],
       [-0.1264,  0.2686, -0.6957,  0.0474],
       [-1.4844,  0.0042, -0.7442,  0.0055],
       [ 2.3029,  0.2001,  1.6702, -1.8811],
       [-0.1932,  1.0472,  0.4828,  0.9603]])
```

`np.savetxt` performs the inverse operation: writing an array to a delimited text file. `genfromtxt` is similar to `loadtxt` but is geared for structured arrays and missing data handling; see [Chapter 12](#) for more on structured arrays.



For more on file reading and writing, especially tabular or spreadsheet-like data, see the later chapters involving pandas and DataFrame objects.

## Linear Algebra

Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Unlike some languages like MATLAB, multiplying two two-dimensional arrays with `*` is an element-wise product instead of a matrix dot product. As such, there is a function `dot`, both an array method, and a function in the `numpy` namespace, for matrix multiplication:

```
In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])
```

```
In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])
```

```
In [196]: x
Out[196]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [197]: y
Out[197]:
array([[ 6., 23.],
       [-1.,  7.],
       [ 8.,  9.]])
```

```
In [198]: x.dot(y) # equivalently np.dot(x, y)
Out[198]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

A matrix product between a 2D array and a suitably sized 1D array results in a 1D array:

```
In [199]: np.dot(x, np.ones(3))
Out[199]: array([ 6., 15.] )
```

`numpy.linalg` has a standard set of matrix decompositions and things like inverse and determinant. These are implemented under the hood using the same industry-standard Fortran libraries used in other languages like MATLAB and R, such as like BLAS, LAPACK, or possibly (depending on your NumPy build) the Intel MKL:

```
In [201]: from numpy.linalg import inv, qr
```

```
In [202]: X = randn(5, 5)
```

```
In [203]: mat = X.T.dot(X)
```

```
In [204]: inv(mat)
Out[204]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])
```

```
In [205]: mat.dot(inv(mat))
```

```

Out[205]:
array([[ 1.,  0.,  0.,  0., -0.],
       [ 0.,  1., -0.,  0.,  0.],
       [ 0., -0.,  1.,  0.,  0.],
       [ 0., -0., -0.,  1., -0.],
       [ 0.,  0.,  0.,  0.,  1.]])

In [206]: q, r = qr(mat)

In [207]: r
Out[207]:
array([[ -6.9271,  7.389 ,  6.1227, -7.1163, -4.9215],
       [  0.    , -3.9735, -0.8671,  2.9747, -5.7402],
       [  0.    ,  0.    , -10.2681,  1.8909,  1.6079],
       [  0.    ,  0.    ,  0.    , -1.2996,  3.3577],
       [  0.    ,  0.    ,  0.    ,  0.    ,  0.5571]])

```

See [Table 4-7](#) for a list of some of the most commonly-used linear algebra functions.



The scientific Python community is hopeful that there may be a matrix multiplication infix operator implemented someday, providing syntactically nicer alternative to using `np.dot`. But for now this is the way.

Table 4-7. Commonly-used *numpy.linalg* functions

Function	Description
<code>diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
<code>dot</code>	Matrix multiplication
<code>trace</code>	Compute the sum of the diagonal elements
<code>det</code>	Compute the matrix determinant
<code>eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>inv</code>	Compute the inverse of a square matrix
<code>pinv</code>	Compute the Moore-Penrose pseudo-inverse inverse of a matrix
<code>qr</code>	Compute the QR decomposition
<code>svd</code>	Compute the singular value decomposition (SVD)
<code>solve</code>	Solve the linear system $Ax = b$ for $x$ , where $A$ is a square matrix
<code>lstsq</code>	Compute the least-squares solution to $Ax = b$

## Random Number Generation

The `numpy.random` module supplements the built-in Python `random` with functions for efficiently generating whole arrays of sample values from many kinds of probability

distributions. For example, you can get a 4 by 4 array of samples from the standard normal distribution using `normal`:

```
In [208]: samples = np.random.normal(size=(4, 4))
```

```
In [209]: samples
```

```
Out[209]:  
array([[ 0.1241,  0.3026,  0.5238,  0.0009],  
       [ 1.3438, -0.7135, -0.8312, -2.3702],  
       [-1.8608, -0.8608,  0.5601, -1.2659],  
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

Python's built-in `random` module, by contrast, only samples one value at a time. As you can see from this benchmark, `numpy.random` is well over an order of magnitude faster for generating very large samples:

```
In [210]: from random import normalvariate
```

```
In [211]: N = 1000000
```

```
In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]  
1 loops, best of 3: 1.33 s per loop
```

```
In [213]: %timeit np.random.normal(size=N)  
10 loops, best of 3: 57.7 ms per loop
```

See [Table 4-8](#) for a partial list of functions available in `numpy.random`. I'll give some examples of leveraging these functions' ability to generate large arrays of samples all at once in the next section.

*Table 4-8. Partial list of `numpy.random` functions*

Function	Description
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples from a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

## Example: Random Walks

An illustrative application of utilizing array operations is in the simulation of random walks. Let's first consider a simple random walk starting at 0 with steps of 1 and -1 occurring with equal probability. A pure Python way to implement a single random walk with 1,000 steps using the built-in `random` module:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)
```

See [Figure 4-4](#) for an example plot of the first 100 values on one of these random walks.

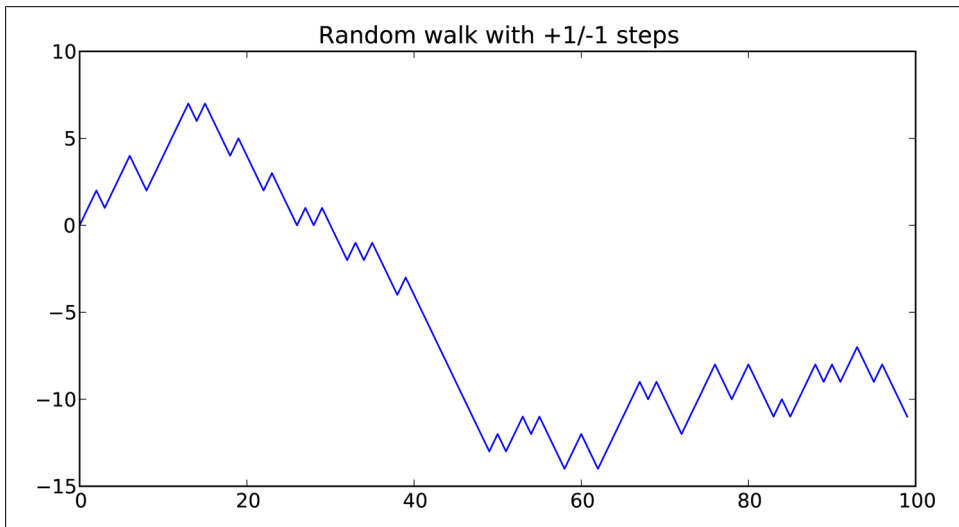


Figure 4-4. A simple random walk

You might make the observation that `walk` is simply the cumulative sum of the random steps and could be evaluated as an array expression. Thus, I use the `np.random` module to draw 1,000 coin flips at once, set these to 1 and -1, and compute the cumulative sum:

```
In [215]: nsteps = 1000

In [216]: draws = np.random.randint(0, 2, size=nsteps)

In [217]: steps = np.where(draws > 0, 1, -1)

In [218]: walk = steps.cumsum()
```



From this we can begin to extract statistics like the minimum and maximum value along the walk's trajectory:

```
In [219]: walk.min()      In [220]: walk.max()
Out[219]: -3             Out[220]: 31
```

A more complicated statistic is the *first crossing time*, the step at which the random walk reaches a particular value. Here we might want to know how long it took the random walk to get at least 10 steps away from the origin 0 in either direction. `np.abs(walk) >= 10` gives us a boolean array indicating where the walk has reached or exceeded 10, but we want the index of the *first* 10 or -10. Turns out this can be computed using `argmax`, which returns the first index of the maximum value in the boolean array (True is the maximum value):

```
In [221]: (np.abs(walk) >= 10).argmax()
Out[221]: 37
```

Note that using `argmax` here is not always efficient because it always makes a full scan of the array. In this special case once a True is observed we know it to be the maximum value.

## Simulating Many Random Walks at Once

If your goal was to simulate many random walks, say 5,000 of them, you can generate all of the random walks with minor modifications to the above code. The `numpy.random` functions if passed a 2-tuple will generate a 2D array of draws, and we can compute the cumulative sum across the rows to compute all 5,000 random walks in one shot:

```
In [222]: nwalks = 5000

In [223]: nsteps = 1000

In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [225]: steps = np.where(draws > 0, 1, -1)

In [226]: walks = steps.cumsum(1)

In [227]: walks
Out[227]:
array([[ 1,  0,  1, ...,  8,  7,  8],
       [ 1,  0, -1, ..., 34, 33, 32],
       [ 1,  0, -1, ...,  4,  5,  4],
       ...,
       [ 1,  2,  1, ..., 24, 25, 26],
       [ 1,  2,  3, ..., 14, 13, 14],
       [-1, -2, -3, ..., -24, -23, -22]])
```

Now, we can compute the maximum and minimum values obtained over all of the walks:

```
In [228]: walks.max()      In [229]: walks.min()
Out[228]: 138             Out[229]: -133
```

Out of these walks, let's compute the minimum crossing time to 30 or -30. This is slightly tricky because not all 5,000 of them reach 30. We can check this using the any method:

```
In [230]: hits30 = (np.abs(walks) >= 30).any(1)

In [231]: hits30
Out[231]: array([False,  True,  False, ...,  False,  True,  False], dtype=bool)

In [232]: hits30.sum() # Number that hit 30 or -30
Out[232]: 3410
```

We can use this boolean array to select out the rows of `walks` that actually cross the absolute 30 level and call `argmax` across axis 1 to get the crossing times:

```
In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [234]: crossing_times.mean()
Out[234]: 498.88973607038122
```

Feel free to experiment with other distributions for the steps other than equal sized coin flips. You need only use a different random number generation function, like `normal` to generate normally distributed steps with some mean and standard deviation:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,
.....:                               size=(nwalks, nsteps))
```

# Getting Started with pandas

pandas will be the primary library of interest throughout much of the rest of the book. It contains high-level data structures and manipulation tools designed to make data analysis fast and easy in Python. pandas is built on top of NumPy and makes it easy to use in NumPy-centric applications.

As a bit of background, I started building pandas in early 2008 during my tenure at AQR, a quantitative investment management firm. At the time, I had a distinct set of requirements that were not well-addressed by any single tool at my disposal:

- Data structures with labeled axes supporting automatic or explicit data alignment. This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQL-based, for example).

I wanted to be able to do all of these things in one place, preferably in a language well-suited to general purpose software development. Python was a good candidate language for this, but at that time there was not an integrated set of data structures and tools providing this functionality.

Over the last four years, pandas has matured into a quite large library capable of solving a much broader set of data handling problems than I ever anticipated, but it has expanded in its scope without compromising the simplicity and ease-of-use that I desired from the very beginning. I hope that after reading this book, you will find it to be just as much of an indispensable tool as I do.

Throughout the rest of the book, I use the following import conventions for pandas:

```
In [1]: from pandas import Series, DataFrame
```

```
In [2]: import pandas as pd
```

Thus, whenever you see `pd.` in code, it's referring to `pandas`. `Series` and `DataFrame` are used so much that I find it easier to import them into the local namespace.

## Introduction to pandas Data Structures

To get started with `pandas`, you will need to get comfortable with its two workhorse data structures: *Series* and *DataFrame*. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

### Series

A `Series` is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest `Series` is formed from only an array of data:

```
In [4]: obj = Series([4, 7, -5, 3])
```

```
In [5]: obj
```

```
Out[5]:
```

```
0    4
1    7
2   -5
3    3
```

The string representation of a `Series` displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through  $N - 1$  (where  $N$  is the length of the data) is created. You can get the array representation and index object of the `Series` via its `values` and `index` attributes, respectively:

```
In [6]: obj.values
```

```
Out[6]: array([ 4,  7, -5,  3])
```

```
In [7]: obj.index
```

```
Out[7]: Int64Index([0, 1, 2, 3])
```

Often it will be desirable to create a `Series` with an index identifying each data point:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
```

```
In [9]: obj2
```

```
Out[9]:
```

```
d    4
b    7
a   -5
c    3
```

```
In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)
```

Compared with a regular NumPy array, you can use values in the index when selecting single values or a set of values:

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c    3
a   -5
d    6
```

NumPy array operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3

In [15]: obj2[obj2 > 0]
Out[15]:
d    6
b    7
c    3

In [16]: obj2 * 2
Out[16]:
d   12
b   14
a  -10
c    6

In [17]: np.exp(obj2)
Out[17]:
d   403.428793
b  1096.633158
a    0.006738
c   20.085537
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be substituted into many functions that expect a dict:

```
In [18]: 'b' in obj2
Out[18]: True

In [19]: 'e' in obj2
Out[19]: False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3
Out[22]:
Ohio    35000
Oregon   16000
```

```
Texas    71000
Utah      5000
```

When only passing a dict, the index in the resulting Series will have the dict's keys in sorted order.

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
In [24]: obj4 = Series(sdata, index=states)
```

```
In [25]: obj4
Out[25]:
California    NaN
Ohio          35000
Oregon        16000
Texas         71000
```

In this case, 3 values found in `sdata` were placed in the appropriate locations, but since no value for 'California' was found, it appears as `NaN` (not a number) which is considered in pandas to mark missing or `NA` values. I will use the terms “missing” or “NA” to refer to missing data. The `isnull` and `notnull` functions in pandas should be used to detect missing data:

```
In [26]: pd.isnull(obj4)      In [27]: pd.notnull(obj4)
Out[26]:
California    True
Ohio          False
Oregon        False
Texas         False
Out[27]:
California    False
Ohio          True
Oregon        True
Texas         True
```

Series also has these as instance methods:

```
In [28]: obj4.isnull()
Out[28]:
California    True
Ohio          False
Oregon        False
Texas         False
```

I discuss working with missing data in more detail later in this chapter.

A critical Series feature for many applications is that it automatically aligns differently-indexed data in arithmetic operations:

```
In [29]: obj3      In [30]: obj4
Out[29]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
Out[30]:
California    NaN
Ohio          35000
Oregon        16000
Texas         71000
```

```
In [31]: obj3 + obj4
Out[31]:
California    NaN
Ohio          70000
Oregon        32000
```

```
Texas      142000
Utah        NaN
```

Data alignment features are addressed as a separate topic.

Both the Series object itself and its index have a `name` attribute, which integrates with other key areas of pandas functionality:

```
In [32]: obj4.name = 'population'

In [33]: obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California      NaN
Ohio            35000
Oregon          16000
Texas           71000
Name: population
```

A Series's index can be altered in place by assignment:

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob      4
Steve    7
Jeff    -5
Ryan     3
```

## DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index). Compared with other such DataFrame-like structures you may have used before (like R's `data.frame`), row-oriented and column-oriented operations in DataFrame are treated roughly symmetrically. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are far outside the scope of this book.



While DataFrame stores the data internally in a two-dimensional format, you can easily represent much higher-dimensional data in a tabular format using hierarchical indexing, a subject of a later section and a key ingredient in many of the more advanced data-handling features in pandas.

There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order:

```
In [38]: frame
Out[38]:
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

If you specify a sequence of columns, the DataFrame's columns will be exactly what you pass:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9

As with Series, if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
.....:                        index=['one', 'two', 'three', 'four', 'five'])
```

```
In [41]: frame2
Out[41]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN

```
In [42]: frame2.columns
Out[42]: Index([year, state, pop, debt], dtype=object)
```

A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [43]: frame2['state']
Out[43]:
```

one
Ohio

```
In [44]: frame2.year
Out[44]:
```

one
2000



two	Ohio	two	2001
three	Ohio	three	2002
four	Nevada	four	2001
five	Nevada	five	2002
Name: state		Name: year	

Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

Rows can also be retrieved by position or name by a couple of methods, such as the `ix` indexing field (much more on this later):

```
In [45]: frame2.ix['three']
Out[45]:
year      2002
state     Ohio
pop        3.6
debt      NaN
Name: three
```

Columns can be modified by assignment. For example, the empty `'debt'` column could be assigned a scalar value or an array of values:

```
In [46]: frame2['debt'] = 16.5
```

```
In [47]: frame2
Out[47]:
   year  state  pop  debt
one  2000   Ohio  1.5  16.5
two   2001   Ohio  1.7  16.5
three 2002   Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
```

```
In [48]: frame2['debt'] = np.arange(5.)
```

```
In [49]: frame2
Out[49]:
   year  state  pop  debt
one  2000   Ohio  1.5    0
two   2001   Ohio  1.7    1
three 2002   Ohio  3.6    2
four   2001  Nevada  2.4    3
five   2002  Nevada  2.9    4
```

When assigning lists or arrays to a column, the value's length must match the length of the DataFrame. If you assign a Series, it will be instead conformed exactly to the DataFrame's index, inserting missing values in any holes:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

```
In [51]: frame2['debt'] = val
```

```
In [52]: frame2
Out[52]:
   year  state  pop  debt
one  2000   Ohio  1.5    0
two   2001   Ohio  1.7    1
three 2002   Ohio  3.6    2
four   2001  Nevada  2.4    3
five   2002  Nevada  2.9    4
```

one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
In [54]: frame2
```

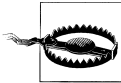
```
Out[54]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

```
In [55]: del frame2['eastern']
```

```
In [56]: frame2.columns
```

```
Out[56]: Index([year, state, pop, debt], dtype=object)
```



The column returned when indexing a DataFrame is a *view* on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied using the Series's `copy` method.

Another common form of data is a nested dict of dicts format:

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
.....:         'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [58]: frame3 = DataFrame(pop)
```

```
In [59]: frame3
```

```
Out[59]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Of course you can always transpose the result:

```
In [60]: frame3.T
```

```
Out[60]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

The keys in the inner dicts are unioned and sorted to form the index in the result. This isn't true if an explicit index is specified:

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])
Out[61]:
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

Dicts of Series are treated much in the same way:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][::-1],
.....:           'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)
Out[63]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7

For a complete list of things you can pass the DataFrame constructor, see [Table 5-1](#).

If a DataFrame's index and columns have their name attributes set, these will also be displayed:

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
```

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Like Series, the `values` attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [66]: frame3.values
Out[66]:
```

```
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

If the DataFrame's columns are different dtypes, the dtype of the values array will be chosen to accomodate all of the columns:

```
In [67]: frame2.values
Out[67]:
```

```
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, Nevada, 2.9, -1.7]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

## Index Objects

pandas’s Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels used when constructing a Series or DataFrame is internally converted to an Index:

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])
```

```
In [69]: index = obj.index
```

```
In [70]: index
Out[70]: Index([a, b, c], dtype=object)
```

```
In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Index objects are immutable and thus can’t be modified by the user:

```
In [72]: index[1] = 'd'
-----
Exception                                 Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302     def __setitem__(self, key, value):
    303         """Disable the setting of values."""
--> 304         raise Exception(str(self.__class__) + ' object is immutable')
    305
    306     def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Immutability is important so that Index objects can be safely shared among data structures:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True
```

[Table 5-2](#) has a list of built-in Index classes in the library. With some development effort, Index can even be subclassed to implement specialized axis indexing functionality.



Many users will not need to know much about Index objects, but they're nonetheless an important part of pandas's data model.

*Table 5-2. Main Index objects in pandas*

Class	Description
Index	The most general Index object, representing axis labels in a NumPy array of Python objects.
Int64Index	Specialized Index for integer values.
MultiIndex	“Hierarchical” index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
DatetimeIndex	Stores nanosecond timestamps (represented using NumPy's <code>datetime64</code> dtype).
PeriodIndex	Specialized Index for Period data (timespans).

In addition to being array-like, an Index also functions as a fixed-size set:

```
In [76]: frame3
Out[76]:
state Nevada Ohio
year
2000      NaN  1.5
2001      2.4  1.7
2002      2.9  3.6

In [77]: 'Ohio' in frame3.columns
Out[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

Each Index has a number of methods and properties for set logic and answering other common questions about the data it contains. These are summarized in [Table 5-3](#).

Table 5-3. Index methods and properties

Method	Description
<code>append</code>	Concatenate with additional Index objects, producing a new Index
<code>diff</code>	Compute set difference as an Index
<code>intersection</code>	Compute set intersection
<code>union</code>	Compute set union
<code>isin</code>	Compute boolean array indicating whether each value is contained in the passed collection
<code>delete</code>	Compute new Index with element at index <code>i</code> deleted
<code>drop</code>	Compute new index by deleting passed values
<code>insert</code>	Compute new Index by inserting element at index <code>i</code>
<code>is_monotonic</code>	Returns <code>True</code> if each element is greater than or equal to the previous element
<code>is_unique</code>	Returns <code>True</code> if the Index has no duplicate values
<code>unique</code>	Compute the array of unique values in the Index

## Essential Functionality

In this section, I'll walk you through the fundamental mechanics of interacting with the data contained in a `Series` or `DataFrame`. Upcoming chapters will delve more deeply into data analysis and manipulation topics using `pandas`. This book is not intended to serve as exhaustive documentation for the `pandas` library; I instead focus on the most important features, leaving the less common (that is, more esoteric) things for you to explore on your own.

## Reindexing

A critical method on `pandas` objects is `reindex`, which means to create a new object with the data *conformed* to a new index. Consider a simple example from above:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
```

```
In [80]: obj
```

```
Out[80]:
```

```
d    4.5
```

```
b    7.2
```

```
a   -5.3
```

```
c    3.6
```

Calling `reindex` on this `Series` rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
In [82]: obj2
```

```
Out[82]:
```

```
a   -5.3
```

```

b    7.2
c    3.6
d    4.5
e    NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a    -5.3
b     7.2
c     3.6
d     4.5
e     0.0

```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The `method` option allows us to do this, using a method such as `ffill` which forward fills the values:

```

In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0    blue
1    blue
2   purple
3   purple
4   yellow
5   yellow

```

Table 5-4 lists available `method` options. At this time, interpolation more sophisticated than forward- and backfilling would need to be applied after the fact.

Table 5-4. *reindex method (interpolation) options*

Argument	Description
<code>ffill</code> or <code>pad</code>	Fill (or carry) values forward
<code>bfill</code> or <code>backfill</code>	Fill (or carry) values backward

With `DataFrame`, `reindex` can alter either the (row) index, columns, or both. When passed just a sequence, the rows are reindexed in the result:

```

In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
.....:                    columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
   Ohio  Texas  California
a      0      1           2
c      3      4           5
d      6      7           8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])

In [89]: frame2
Out[89]:

```

	Ohio	Texas	California
a	0	1	2
b	NaN	NaN	NaN
c	3	4	5
d	6	7	8

The columns can be reindexed using the `columns` keyword:

```
In [90]: states = ['Texas', 'Utah', 'California']
```

```
In [91]: frame.reindex(columns=states)
```

```
Out[91]:
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

Both can be reindexed in one shot, though interpolation will only apply row-wise (axis 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
.....:                  columns=states)
```

```
Out[92]:
```

	Texas	Utah	California
a	1	NaN	2
b	1	NaN	2
c	4	NaN	5
d	7	NaN	8

As you'll see soon, reindexing can be done more succinctly by label-indexing with `ix`:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
```

```
Out[93]:
```

	Texas	Utah	California
a	1	NaN	2
b	NaN	NaN	NaN
c	4	NaN	5
d	7	NaN	8

Table 5-5. *reindex* function arguments

Argument	Description
<code>index</code>	New sequence to use as index. Can be <code>Index</code> instance or any other sequence-like Python data structure. An <code>Index</code> will be used exactly as is without any copying
<code>method</code>	Interpolation (fill) method, see <a href="#">Table 5-4</a> for options.
<code>fill_value</code>	Substitute value to use when introducing missing data by reindexing
<code>limit</code>	When forward- or backfilling, maximum size gap to fill
<code>level</code>	Match simple <code>Index</code> on level of <code>Multindex</code> , otherwise select subset of
<code>copy</code>	If <code>True</code> , always copy underlying data even if new index is equivalent to old index. Otherwise, do not copy the data when the indexes are equivalent.



## Dropping entries from an axis

Dropping one or more entries from an axis is easy if you have an index array or list without those entries. As that can require a bit of munging and set logic, the `drop` method will return a new object with the indicated value or values deleted from an axis:

```
In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [95]: new_obj = obj.drop('c')
```

```
In [96]: new_obj
```

```
Out[96]:
```

```
a    0
b    1
d    3
e    4
```

```
In [97]: obj.drop(['d', 'c'])
```

```
Out[97]:
```

```
a    0
b    1
e    4
```

With `DataFrame`, index values can be deleted from either axis:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),
.....:                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                    columns=['one', 'two', 'three', 'four'])
```

```
In [99]: data.drop(['Colorado', 'Ohio'])
```

```
Out[99]:
```

```
      one  two  three  four
Utah      8   9    10    11
New York  12  13    14    15
```

```
In [100]: data.drop('two', axis=1)
```

```
Out[100]:
```

```
      one  three  four
Ohio      0     2     3
Colorado  4     6     7
Utah      8    10    11
New York  12    14    15
```

```
In [101]: data.drop(['two', 'four'], axis=1)
```

```
Out[101]:
```

```
      one  three
Ohio      0     2
Colorado  4     6
Utah      8    10
New York  12    14
```

## Indexing, selection, and filtering

Series indexing (`obj[...]`) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

```
In [103]: obj['b']
```

```
Out[103]: 1.0
```

```
In [104]: obj[1]
```

```
Out[104]: 1.0
```

```
In [105]: obj[2:4]
```

```
Out[105]:
```

```
In [106]: obj[['b', 'a', 'd']]
```

```
Out[106]:
```

```

c    2
d    3

b    1
a    0
d    3

In [107]: obj[[1, 3]]
Out[107]:
b    1
d    3

In [108]: obj[obj < 2]
Out[108]:
a    0
b    1

```

Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive:

```

In [109]: obj['b':'c']
Out[109]:
b    1
c    2

```

*Setting* using these methods works just as you would expect:

```

In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
a    0
b    5
c    5
d    3

```

As you've seen above, indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```

In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
.....:                    index=['Ohio', 'Colorado', 'Utah', 'New York'],
.....:                    columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
   one  two  three  four
Ohio   0   1     2     3
Colorado 4   5     6     7
Utah   8   9    10    11
New York 12  13    14    15

In [114]: data['two']
Out[114]:
Ohio      1
Colorado   5
Utah      9
New York  13
Name: two

In [115]: data[['three', 'one']]
Out[115]:
   three  one
Ohio     2   0
Colorado  6   4
Utah    10   8
New York 14  12

```

Indexing like this has a few special cases. First selecting rows by slicing or a boolean array:

```

In [116]: data[:2]
Out[116]:
   one  two  three  four
Ohio   0   1     2     3
Colorado 4   5     6     7

In [117]: data[data['three'] > 5]
Out[117]:
   one  two  three  four
Utah  10   9    10    11
New York 14  13    14    15

```

Ohio	0	1	2	3	Colorado	4	5	6	7
Colorado	4	5	6	7	Utah	8	9	10	11
					New York	12	13	14	15

This might seem inconsistent to some readers, but this syntax arose out of practicality and nothing more. Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

```
In [118]: data < 5
Out[118]:
```

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
In [119]: data[data < 5] = 0
```

```
In [120]: data
Out[120]:
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

This is intended to make DataFrame syntactically more like an ndarray in this case.

For DataFrame label-indexing on the rows, I introduce the special indexing field `ix`. It enables you to select a subset of the rows and columns from a DataFrame with NumPy-like notation plus axis labels. As I mentioned earlier, this is also a less verbose way to do reindexing:

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
```

two	5
three	6

Name: Colorado

```
In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
```

	four	one	two
Colorado	7	0	5
Utah	11	8	9

In [123]: data.ix[2]	In [124]: data.ix[:'Utah', 'two']		
Out[123]:	Out[124]:		
one	8	Ohio	0
two	9	Colorado	5
three	10	Utah	9
four	11		
Name: Utah		Name: two	

```
In [125]: data.ix[data.three > 5, :3]
Out[125]:
```

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

So there are many ways to select and rearrange the data contained in a pandas object. For DataFrame, there is a short summary of many of them in [Table 5-6](#). You have a number of additional options when working with hierarchical indexes as you’ll later see.



When designing pandas, I felt that having to type `frame[:, col]` to select a column was too verbose (and error-prone), since column selection is one of the most common operations. Thus I made the design trade-off to push all of the rich label-indexing into `ix`.

*Table 5-6. Indexing options with DataFrame*

Type	Notes
<code>obj[val]</code>	Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
<code>obj.ix[val]</code>	Selects single row or subset of rows from the DataFrame.
<code>obj.ix[:, val]</code>	Selects single column or subset of columns.
<code>obj.ix[val1, val2]</code>	Select both rows and columns.
<code>reindex</code> method	Conform one or more axes to new indexes.
<code>xs</code> method	Select single row or column as a Series by label.
<code>icol</code> , <code>irow</code> methods	Select single column or row, respectively, as a Series by integer location.
<code>get_value</code> , <code>set_value</code> methods	Select single value by row and column label.

## Arithmetic and data alignment

One of the most important pandas features is the behavior of arithmetic between objects with different indexes. When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. Let’s look at a simple example:

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [128]: s1      In [129]: s2
Out[128]:      Out[129]:
a      7.3         a    -2.1
c     -2.5         c     3.6
d      3.4         e    -1.5
```

```
e    1.5          f    4.0
          g    3.1
```

Adding these together yields:

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

The internal data alignment introduces NA values in the indices that don't overlap. Missing values propagate in arithmetic computations.

In the case of DataFrame, alignment is performed on both the rows and the columns:

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
.....:                    index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [133]: df1
Out[133]:
   b  c  d
Ohio  0  1  2
Texas  3  4  5
Colorado 6  7  8

In [134]: df2
Out[134]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11
```

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

```
In [135]: df1 + df2
Out[135]:
   b  c  d  e
Colorado NaN NaN NaN NaN
Ohio      3 NaN  6 NaN
Oregon   NaN NaN NaN NaN
Texas     9 NaN 12 NaN
Utah     NaN NaN NaN NaN
```

## Arithmetic methods with fill values

In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

```
In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))

In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))

In [138]: df1
Out[138]:
   a  b  c  d
a  b  c  d

In [139]: df2
Out[139]:
   a  b  c  d  e
```

0	0	1	2	3	0	0	1	2	3	4
1	4	5	6	7	1	5	6	7	8	9
2	8	9	10	11	2	10	11	12	13	14
					3	15	16	17	18	19

Adding these together results in NA values in the locations that don't overlap:

```
In [140]: df1 + df2
Out[140]:
```

	a	b	c	d	e
0	0	2	4	6	NaN
1	9	11	13	15	NaN
2	18	20	22	24	NaN
3	NaN	NaN	NaN	NaN	NaN

Using the add method on df1, I pass df2 and an argument to fill\_value:

```
In [141]: df1.add(df2, fill_value=0)
Out[141]:
```

	a	b	c	d	e
0	0	2	4	6	4
1	9	11	13	15	9
2	18	20	22	24	14
3	15	16	17	18	19

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)
Out[142]:
```

	a	b	c	d	e
0	0	1	2	3	0
1	4	5	6	7	0
2	8	9	10	11	0

Table 5-7. Flexible arithmetic methods

Method	Description
add	Method for addition (+)
sub	Method for subtraction (-)
div	Method for division (/)
mul	Method for multiplication (*)

## Operations between DataFrame and Series

As with NumPy arrays, arithmetic between DataFrame and Series is well-defined. First, as a motivating example, consider the difference between a 2D array and one of its rows:

```
In [143]: arr = np.arange(12.).reshape((3, 4))

In [144]: arr
Out[144]:
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]])
```

```

[ 8.,  9., 10., 11.])

In [145]: arr[0]
Out[145]: array([ 0.,  1.,  2.,  3.])

In [146]: arr - arr[0]
Out[146]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])

```

This is referred to as *broadcasting* and is explained in more detail in [Chapter 12](#). Operations between a DataFrame and a Series are similar:

```

In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
.....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [148]: series = frame.ix[0]

In [149]: frame
Out[149]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11

In [150]: series
Out[150]:
b    0
d    1
e    2
Name: Utah

```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

```

In [151]: frame - series
Out[151]:
   b  d  e
Utah  0  0  0
Ohio  3  3  3
Texas  6  6  6
Oregon 9  9  9

```

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

```

In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
   b  d  e  f
Utah  0 NaN  3 NaN
Ohio  3 NaN  6 NaN
Texas  6 NaN  9 NaN
Oregon 9 NaN 12 NaN

```

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

```

In [154]: series3 = frame['d']

In [155]: frame
Out[155]:
   b  d  e
Utah  0  1  2
Ohio  3  4  5
Texas  6  7  8
Oregon 9 10 11

In [156]: series3
Out[156]:
d
Utah    1
Ohio    4
Texas    7
Oregon 10

```

```

Out[155]:
      b  d  e
Utah   0  1  2
Ohio   3  4  5
Texas  6  7  8
Oregon 9 10 11
      Name: d

In [157]: frame.sub(series3, axis=0)
Out[157]:
      b  d  e
Utah  -1  0  1
Ohio  -1  0  1
Texas -1  0  1
Oregon -1  0  1

```

The axis number that you pass is the *axis to match on*. In this case we mean to match on the DataFrame's row index and broadcast across.

## Function application and mapping

NumPy ufuncs (element-wise array methods) work fine with pandas objects:

```

In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
.....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: frame
Out[159]:
      b         d         e
Utah -0.204708  0.478943 -0.519439
Ohio -0.555730  1.965781  1.393406
Texas  0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [160]: np.abs(frame)
Out[160]:
      b         d         e
Utah  0.204708  0.478943  0.519439
Ohio  0.555730  1.965781  1.393406
Texas  0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221

```

Another frequent operation is applying a function on 1D arrays to each column or row. DataFrame's `apply` method does exactly this:

```

In [161]: f = lambda x: x.max() - x.min()

In [162]: frame.apply(f)
Out[162]:
b    1.802165
d    1.684034
e    2.689627

In [163]: frame.apply(f, axis=1)
Out[163]:
Utah    0.998382
Ohio    2.521511
Texas    0.676115
Oregon    2.542656

```

Many of the most common array statistics (like `sum` and `mean`) are DataFrame methods, so using `apply` is not necessary.

The function passed to `apply` need not return a scalar value, it can also return a Series with multiple values:

```

In [164]: def f(x):
.....:     return Series([x.min(), x.max()], index=['min', 'max'])

In [165]: frame.apply(f)

```



```
Out[165]:
           b           d           e
min -0.555730  0.281746 -1.296221
max  1.246435  1.965781  1.393406
```

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating point value in `frame`. You can do this with `applymap`:

```
In [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
           b           d           e
Utah    -0.20  0.48  -0.52
Ohio    -0.56  1.97  1.39
Texas    0.09  0.28  0.77
Oregon   1.25  1.01 -1.30
```

The reason for the name `applymap` is that Series has a `map` method for applying an element-wise function:

```
In [168]: frame['e'].map(format)
Out[168]:
Utah    -0.52
Ohio     1.39
Texas     0.77
Oregon  -1.30
Name: e
```

## Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
a    1
b    2
c    3
d    0
```

With a `DataFrame`, you can sort by index on either axis:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
.....:                      columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index()
Out[172]:
           d  a  b  c
one        4  5  6  7
three     0  1  2  3

In [173]: frame.sort_index(axis=1)
Out[173]:
           a  b  c  d
three     1  2  3  0
one       5  6  7  4
```

The data is sorted in ascending order by default, but can be sorted in descending order, too:

```
In [174]: frame.sort_index(axis=1, ascending=False)
Out[174]:
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

To sort a Series by its values, use its `order` method:

```
In [175]: obj = Series([4, 7, -3, 2])

In [176]: obj.order()
Out[176]:
```

2	-3
3	2
0	4
1	7

Any missing values are sorted to the end of the Series by default:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [178]: obj.order()
Out[178]:
```

4	-3
5	2
0	4
2	7
1	NaN
3	NaN

On DataFrame, you may want to sort by the values in one or more columns. To do so, pass one or more column names to the `by` option:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [180]: frame
Out[180]:
```

	a	b
0	0	4
1	1	7
2	0	-3
3	1	2

```
In [181]: frame.sort_index(by='b')
Out[181]:
```

	a	b
2	0	-3
3	1	2
0	0	4
1	1	7

To sort by multiple columns, pass a list of names:

```
In [182]: frame.sort_index(by=['a', 'b'])
Out[182]:
```

	a	b
2	0	-3
0	0	4
3	1	2
1	1	7

*Ranking* is closely related to sorting, assigning ranks from one through the number of valid data points in an array. It is similar to the indirect sort indices produced by `numpy.argsort`, except that ties are broken according to a rule. The `rank` methods for `Series` and `DataFrame` are the place to look; by default `rank` breaks ties by assigning each group the mean rank:

```
In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])
```

```
In [184]: obj.rank()
```

```
Out[184]:
```

```
0    6.5
1     1.0
2     6.5
3     4.5
4     3.0
5     2.0
6     4.5
```

Ranks can also be assigned according to the order they're observed in the data:

```
In [185]: obj.rank(method='first')
```

```
Out[185]:
```

```
0    6
1    1
2    7
3    4
4    3
5    2
6    5
```

Naturally, you can rank in descending order, too:

```
In [186]: obj.rank(ascending=False, method='max')
```

```
Out[186]:
```

```
0    2
1    7
2    2
3    4
4    5
5    6
6    4
```

See [Table 5-8](#) for a list of tie-breaking methods available. `DataFrame` can compute ranks over the rows or the columns:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
.....:                      'c': [-2, 5, 8, -2.5]})
```

```
In [188]: frame
```

```
Out[188]:
```

```
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5
```

```
In [189]: frame.rank(axis=1)
```

```
Out[189]:
```

```
   a  b  c
0  2  3  1
1  1  3  2
2  2  1  3
3  2  3  1
```

Table 5-8. Tie-breaking methods with rank

Method	Description
'average'	Default: assign the average rank to each entry in the equal group.
'min'	Use the minimum rank for the whole group.
'max'	Use the maximum rank for the whole group.
'first'	Assign ranks in the order the values appear in the data.

## Axis indexes with duplicate values

Up until now all of the examples I've showed you have had unique axis labels (index values). While many pandas functions (like `reindex`) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

```
In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [191]: obj
Out[191]:
a    0
a    1
b    2
b    3
c    4
```

The index's `is_unique` property can tell you whether its values are unique or not:

```
In [192]: obj.index.is_unique
Out[192]: False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a value with multiple entries returns a Series while single entries return a scalar value:

```
In [193]: obj['a']
Out[193]:
a    0
a    1

In [194]: obj['c']
Out[194]: 4
```

The same logic extends to indexing rows in a DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [196]: df
Out[196]:
      0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [197]: df.ix['b']
Out[197]:
      0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

```
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

## Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of *reductions* or *summary statistics*, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the equivalent methods of vanilla NumPy arrays, they are all built from the ground up to exclude missing data. Consider a small DataFrame:

```
In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],
.....:                  [np.nan, np.nan], [0.75, -1.3]],
.....:                  index=['a', 'b', 'c', 'd'],
.....:                  columns=['one', 'two'])

In [199]: df
Out[199]:
   one  two
a  1.40 NaN
b  7.10 -4.5
c   NaN NaN
d  0.75 -1.3
```

Calling DataFrame's `sum` method returns a Series containing column sums:

```
In [200]: df.sum()
Out[200]:
one    9.25
two   -5.80
```

Passing `axis=1` sums over the rows instead:

```
In [201]: df.sum(axis=1)
Out[201]:
a    1.40
b    2.60
c     NaN
d   -0.55
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the `skipna` option:

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
a     NaN
b    1.300
c     NaN
d   -0.275
```

See [Table 5-9](#) for a list of common options for each reduction method options.

Table 5-9. Options for reduction methods

Method	Description
axis	Axis to reduce over. 0 for DataFrame's rows and 1 for columns.
skipna	Exclude missing values, True by default.
level	Reduce grouped by level if the axis is hierarchically-indexed (MultiIndex).

Some methods, like `idxmin` and `idxmax`, return indirect statistics like the index value where the minimum or maximum values are attained:

```
In [203]: df.idxmax()
Out[203]:
one      b
two      d
```

Other methods are *accumulations*:

```
In [204]: df.cumsum()
Out[204]:
   one  two
a  1.40 NaN
b  8.50 -4.5
c   NaN NaN
d  9.25 -5.8
```

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [205]: df.describe()
Out[205]:
           one          two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

On non-numeric data, `describe` produces alternate summary statistics:

```
In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [207]: obj.describe()
Out[207]:
count      16
unique       3
top         a
freq        8
```

See [Table 5-10](#) for a full list of summary statistics and related methods.

Table 5-10. Descriptive and summary statistics

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

## Correlation and Covariance

Some summary statistics, like correlation and covariance, are computed from pairs of arguments. Let's consider some DataFrames of stock prices and volumes obtained from Yahoo! Finance:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
    all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
                   for tic, data in all_data.iteritems()})
volume = DataFrame({tic: data['Volume']
                    for tic, data in all_data.iteritems()})
```

I now compute percent changes of the prices:

```
In [209]: returns = price.pct_change()

In [210]: returns.tail()
```

```
Out[210]:
```

	AAPL	GOOG	IBM	MSFT
Date				
2009-12-24	0.034339	0.011117	0.004420	0.002747
2009-12-28	0.012294	0.007098	0.013282	0.005479
2009-12-29	-0.011861	-0.005571	-0.003474	0.006812
2009-12-30	0.012147	0.005376	0.005468	-0.013532
2009-12-31	-0.004300	-0.004416	-0.012609	-0.015432

The `corr` method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, `cov` computes the covariance:

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838
```

```
In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

DataFrame's `corr` and `cov` methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

```
In [213]: returns.corr()
Out[213]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.000000	0.470660	0.410648	0.424550
GOOG	0.470660	1.000000	0.390692	0.443334
IBM	0.410648	0.390692	1.000000	0.496093
MSFT	0.424550	0.443334	0.496093	1.000000

```
In [214]: returns.cov()
Out[214]:
```

	AAPL	GOOG	IBM	MSFT
AAPL	0.001028	0.000303	0.000252	0.000309
GOOG	0.000303	0.000580	0.000142	0.000205
IBM	0.000252	0.000142	0.000367	0.000216
MSFT	0.000309	0.000205	0.000216	0.000516

Using DataFrame's `corrwith` method, you can compute pairwise correlations between a DataFrame's columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

```
In [215]: returns.corrwith(returns.IBM)
Out[215]:
```

AAPL	0.410648
GOOG	0.390692
IBM	1.000000
MSFT	0.496093

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

```
In [216]: returns.corrwith(volume)
Out[216]:
```

AAPL	-0.057461
GOOG	0.062644



```
IBM      -0.007900
MSFT     -0.014175
```

Passing `axis=1` does things row-wise instead. In all cases, the data points are aligned by label before computing the correlation.

## Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [218]: uniques = obj.unique()

In [219]: uniques
Out[219]: array([c, a, d, b], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [220]: obj.value_counts()
Out[220]:
c      3
a      3
b      2
d      1
```

The Series is sorted by value in descending order as a convenience. `value_counts` is also available as a top-level pandas method that can be used with any array or sequence:

```
In [221]: pd.value_counts(obj.values, sort=False)
Out[221]:
a      3
b      2
c      3
d      1
```

Lastly, `isin` is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])

In [223]: mask
Out[223]:
0      True
1     False
2     False
3     False
4     False
5      True
6      True

In [224]: obj[mask]
Out[224]:
0      c
5      b
6      b
7      c
8      c
```

```
7    True
8    True
```

See [Table 5-11](#) for a reference on these methods.

*Table 5-11. Unique, value counts, and binning methods*

Method	Description
<code>isin</code>	Compute boolean array indicating whether each Series value is contained in the passed sequence of values.
<code>unique</code>	Compute array of unique values in a Series, returned in the order observed.
<code>value_counts</code>	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order.

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
.....:                    'Qu2': [2, 3, 1, 2, 3],
.....:                    'Qu3': [1, 5, 2, 4, 4]})
```

```
In [226]: data
Out[226]:
   Qu1  Qu2  Qu3
0     1     2     1
1     3     3     5
2     4     1     2
3     3     2     4
4     4     3     4
```

Passing `pandas.value_counts` to this DataFrame's `apply` function gives:

```
In [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
   Qu1  Qu2  Qu3
1     1     1     1
2     0     2     1
3     2     2     0
4     2     0     2
5     0     0     1
```

## Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data as you've seen earlier in the chapter.

pandas uses the floating point value **NaN** (Not a Number) to represent missing data in both floating as well as in non-floating point arrays. It is just used as a *sentinel* that can be easily detected:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

```
In [230]: string_data
Out[230]:
0    aardvark
1    artichoke
2         NaN
3     avocado

In [231]: string_data.isnull()
Out[231]:
0    False
1    False
2     True
3    False
```

The built-in Python **None** value is also treated as NA in object arrays:

```
In [232]: string_data[0] = None
```

```
In [233]: string_data.isnull()
Out[233]:
0     True
1    False
2     True
3    False
```

I do not claim that pandas's NA representation is optimal, but it is simple and reasonably consistent. It's the best solution, with good all-around performance characteristics and a simple API, that I could concoct in the absence of a true NA data type or bit pattern in NumPy's data types. Ongoing development work in NumPy may change this in the future.

*Table 5-12. NA handling methods*

Argument	Description
<code>dropna</code>	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
<code>fillna</code>	Fill in missing data with some value or using an interpolation method such as <code>'ffill'</code> or <code>'bfill'</code> .
<code>isnull</code>	Return like-type object containing boolean values indicating which values are missing / NA.
<code>notnull</code>	Negation of <code>isnull</code> .

## Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, `dropna` can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
```

```
0    1.0
2    3.5
4    7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]
Out[237]:
0    1.0
2    3.5
4    7.0
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. `dropna` by default drops any row containing a missing value:

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
.....:                      [NA, NA, NA], [NA, 6.5, 3.]])

In [239]: cleaned = data.dropna()

In [240]: data
Out[240]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
2 NaN  NaN  NaN
3 NaN  6.5    3

In [241]: cleaned
Out[241]:
   0    1    2
0  1  6.5    3
```

Passing `how='all'` will only drop rows that are all NA:

```
In [242]: data.dropna(how='all')
Out[242]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
3 NaN  6.5    3
```

Dropping columns in the same way is only a matter of passing `axis=1`:

```
In [243]: data[4] = NA

In [244]: data
Out[244]:
   0    1    2    4
0  1  6.5    3  NaN
1  1  NaN  NaN  NaN
2 NaN  NaN  NaN  NaN
3 NaN  6.5    3  NaN

In [245]: data.dropna(axis=1, how='all')
Out[245]:
   0    1    2
0  1  6.5    3
1  1  NaN  NaN
2 NaN  NaN  NaN
3 NaN  6.5    3
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the `thresh` argument:

```
In [246]: df = DataFrame(np.random.randn(7, 3))

In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

```

In [248]: df
Out[248]:
   0         1         2
0 -0.577087    NaN    NaN
1  0.523772    NaN    NaN
2 -0.713544    NaN    NaN
3 -1.860761    NaN  0.560145
4 -1.265934    NaN -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

In [249]: df.dropna(thresh=3)
Out[249]:
   0         1         2
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

## Filling in Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways. For most purposes, the `fillna` method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:

```

In [250]: df.fillna(0)
Out[250]:
   0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

Calling `fillna` with a dict you can use a different fill value for each column:

```

In [251]: df.fillna({'1': 0.5, '3': -1})
Out[251]:
   0         1         2
0 -0.577087  0.500000    NaN
1  0.523772  0.500000    NaN
2 -0.713544  0.500000    NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

`fillna` returns a new object, but you can modify the existing object in place:

```

# always returns a reference to the filled object
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
Out[253]:
   0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145

```

```

4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030

```

The same interpolation methods available for reindexing can be used with `fillna`:

```
In [254]: df = DataFrame(np.random.randn(6, 3))
```

```
In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA
```

```
In [256]: df
```

```

Out[256]:
   0      1      2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674      NaN  1.004812
3  1.327195      NaN -1.549106
4  0.022185      NaN      NaN
5  0.862580      NaN      NaN

```

```
In [257]: df.fillna(method='ffill')
```

```

Out[257]:
   0      1      2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185  1.349742 -1.549106
5  0.862580  1.349742 -1.549106

```

```
In [258]: df.fillna(method='ffill', limit=2)
```

```

Out[258]:
   0      1      2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185      NaN -1.549106
5  0.862580      NaN -1.549106

```

With `fillna` you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])
```

```
In [260]: data.fillna(data.mean())
```

```

Out[260]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000

```

See [Table 5-13](#) for a reference on `fillna`.

Table 5-13. *fillna* function arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation, by default 'ffill' if function called with no other arguments
axis	Axis to fill on, default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

# Hierarchical Indexing

*Hierarchical indexing* is an important feature of pandas enabling you to have multiple (two or more) index *levels* on an axis. Somewhat abstractly, it provides a way for you to work with higher dimensional data in a lower dimensional form. Let's start with a simple example; create a Series with a list of lists or arrays as the index:

```
In [261]: data = Series(np.random.randn(10),
.....:                  index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
.....:                  [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])

In [262]: data
Out[262]:
a 1    0.670216
  2    0.852965
  3   -0.955869
b 1   -0.023493
  2   -2.304234
  3   -0.652469
c 1   -1.218302
  2   -1.332610
d 2    1.074623
  3    0.723642
```

What you're seeing is a prettified view of a Series with a `MultiIndex` as its index. The “gaps” in the index display mean “use the label directly above”:

```
In [263]: data.index
Out[263]:
MultiIndex
[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1)
 ('c', 2) ('d', 2) ('d', 3)]
```

With a hierarchically-indexed object, so-called *partial* indexing is possible, enabling you to concisely select subsets of the data:

```
In [264]: data['b']
Out[264]:
1   -0.023493
2   -2.304234
3   -0.652469

In [265]: data['b':'c']
Out[265]:
b 1   -0.023493
  2   -2.304234
  3   -0.652469
c 1   -1.218302
  2   -1.332610

In [266]: data.ix[['b', 'd']]
Out[266]:
b 1   -0.023493
  2   -2.304234
  3   -0.652469
d 2    1.074623
  3    0.723642
```

Selection is even possible in some cases from an “inner” level:

```
In [267]: data[:, 2]
Out[267]:
a    0.852965
```

```

b    -2.304234
c    -1.332610
d     1.074623

```

Hierarchical indexing plays a critical role in reshaping data and group-based operations like forming a pivot table. For example, this data could be rearranged into a DataFrame using its `unstack` method:

```

In [268]: data.unstack()
Out[268]:
           1         2         3
a  0.670216  0.852965 -0.955869
b -0.023493 -2.304234 -0.652469
c -1.218302 -1.332610      NaN
d      NaN   1.074623  0.723642

```

The inverse operation of `unstack` is `stack`:

```

In [269]: data.unstack().stack()
Out[269]:
a 1    0.670216
  2    0.852965
  3   -0.955869
b 1   -0.023493
  2   -2.304234
  3   -0.652469
c 1   -1.218302
  2   -1.332610
d 2    1.074623
  3    0.723642

```

`stack` and `unstack` will be explored in more detail in [Chapter 7](#).

With a DataFrame, either axis can have a hierarchical index:

```

In [270]: frame = DataFrame(np.arange(12).reshape((4, 3)),
.....:                      index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
.....:                      columns=[['Ohio', 'Ohio', 'Colorado'],
.....:                               ['Green', 'Red', 'Green']])

In [271]: frame
Out[271]:
           Ohio      Colorado
        Green Red      Green
a 1         0    1          2
  2         3    4          5
b 1         6    7          8
  2         9   10         11

```

The hierarchical levels can have names (as strings or any Python objects). If so, these will show up in the console output (don't confuse the index names with the axis labels!):

```

In [272]: frame.index.names = ['key1', 'key2']

In [273]: frame.columns.names = ['state', 'color']

In [274]: frame

```



```
Out[274]:
state      Ohio      Colorado
color      Green  Red      Green
key1 key2
a    1         0    1         2
     2         3    4         5
b    1         6    7         8
     2         9   10        11
```

With partial column indexing you can similarly select groups of columns:

```
In [275]: frame['Ohio']
Out[275]:
color      Green  Red
key1 key2
a    1         0    1
     2         3    4
b    1         6    7
     2         9   10
```

A `MultiIndex` can be created by itself and then reused; the columns in the above Data-Frame with level names could be created like this:

```
MultiIndex.from_arrays([[ 'Ohio', 'Ohio', 'Colorado'], [ 'Green', 'Red', 'Green']],
                      names=[ 'state', 'color'])
```

## Reordering and Sorting Levels

At times you will need to rearrange the order of the levels on an axis or sort the data by the values in one specific level. The `swaplevel` takes two level numbers or names and returns a new object with the levels interchanged (but the data is otherwise unaltered):

```
In [276]: frame.swaplevel('key1', 'key2')
Out[276]:
state      Ohio      Colorado
color      Green  Red      Green
key2 key1
1    a         0    1         2
2    a         3    4         5
1    b         6    7         8
2    b         9   10        11
```

`sortlevel`, on the other hand, sorts the data (stably) using only the values in a single level. When swapping levels, it's not uncommon to also use `sortlevel` so that the result is lexicographically sorted:

<pre>In [277]: frame.sortlevel(1) Out[277]: state      Ohio      Colorado color      Green  Red      Green key1 key2 a    1         0    1         2 b    1         6    7         8 a    2         3    4         5 b    2         9   10        11</pre>	<pre>In [278]: frame.swaplevel(0, 1).sortlevel(0) Out[278]: state      Ohio      Colorado color      Green  Red      Green key2 key1 1    a         0    1         2      b         6    7         8 2    a         3    4         5      b         9   10        11</pre>
--	--



Data selection performance is much better on hierarchically indexed objects if the index is lexicographically sorted starting with the outermost level, that is, the result of calling `sortlevel(0)` or `sort_index()`.

## Summary Statistics by Level

Many descriptive and summary statistics on `DataFrame` and `Series` have a `level` option in which you can specify the level you want to sum by on a particular axis. Consider the above `DataFrame`; we can sum by level on either the rows or columns like so:

```
In [279]: frame.sum(level='key2')
Out[279]:
state  Ohio      Colorado
color  Green  Red      Green
key2
1         6    8         10
2        12   14         16

In [280]: frame.sum(level='color', axis=1)
Out[280]:
color      Green  Red
key1 key2
a     1         2    1
      2         8    4
b     1        14    7
      2        20   10
```

Under the hood, this utilizes pandas's `groupby` machinery which will be discussed in more detail later in the book.

## Using a `DataFrame`'s Columns

It's not unusual to want to use one or more columns from a `DataFrame` as the row index; alternatively, you may wish to move the row index into the `DataFrame`'s columns. Here's an example `DataFrame`:

```
In [281]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
.....:                      'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
.....:                      'd': [0, 1, 2, 0, 1, 2, 3]})

In [282]: frame
Out[282]:
   a  b  c  d
0  0  7 one  0
1  1  6 one  1
2  2  5 one  2
3  3  4 two  0
4  4  3 two  1
5  5  2 two  2
6  6  1 two  3
```

DataFrame's `set_index` function will create a new DataFrame using one or more of its columns as the index:

```
In [283]: frame2 = frame.set_index(['c', 'd'])
```

```
In [284]: frame2
```

```
Out[284]:
```

	a	b
one		
two		
one	0	7
1	1	6
2	2	5
two	0	3
1	4	3
2	5	2
3	6	1

By default the columns are removed from the DataFrame, though you can leave them in:

```
In [285]: frame.set_index(['c', 'd'], drop=False)
```

```
Out[285]:
```

	a	b	c	d
one				
two				
one	0	7	one	0
1	1	6	one	1
2	2	5	one	2
two	0	3	two	0
1	4	3	two	1
2	5	2	two	2
3	6	1	two	3

`reset_index`, on the other hand, does the opposite of `set_index`; the hierarchical index levels are moved into the columns:

```
In [286]: frame2.reset_index()
```

```
Out[286]:
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

## Other pandas Topics

Here are some additional topics that may be of use to you in your data travels.

### Integer Indexing

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data

structures like lists and tuples. For example, you would not expect the following code to generate an error:

```
ser = Series(np.arange(3.))
ser[-1]
```

In this case, pandas could “fall back” on integer indexing, but there’s not a safe and general way (that I know of) to do this without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult::

```
In [288]: ser
Out[288]:
0    0
1    1
2    2
```

On the other hand, with a non-integer index, there is no potential for ambiguity:

```
In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])

In [290]: ser2[-1]
Out[290]: 2.0
```

To keep things consistent, if you have an axis index containing indexers, data selection with integers will always be label-oriented. This includes slicing with `ix`, too:

```
In [291]: ser.ix[:1]
Out[291]:
0    0
1    1
```

In cases where you need reliable position-based indexing regardless of the index type, you can use the `iget_value` method from Series and `irow` and `icol` methods from DataFrame:

```
In [292]: ser3 = Series(range(3), index=[-5, 1, 3])

In [293]: ser3.iget_value(2)
Out[293]: 2

In [294]: frame = DataFrame(np.arange(6).reshape(3, 2), index=[2, 0, 1])

In [295]: frame.irow(0)
Out[295]:
0    0
1    1
Name: 2
```

## Panel Data

While not a major topic of this book, pandas has a Panel data structure, which you can think of as a three-dimensional analogue of DataFrame. Much of the development focus of pandas has been in tabular data manipulations as these are easier to reason about,

and hierarchical indexing makes using truly N-dimensional arrays unnecessary in a lot of cases.

To create a Panel, you can use a dict of DataFrame objects or a three-dimensional ndarray:

```
import pandas.io.data as web

pdata = pd.Panel(dict((stk, web.get_data_yahoo(stk, '1/1/2009', '6/1/2012'))
                      for stk in ['AAPL', 'GOOG', 'MSFT', 'DELL'])))
```

Each item (the analogue of columns in a DataFrame) in the Panel is a DataFrame:

```
In [297]: pdata
Out[297]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 861 (major) x 6 (minor)
Items: AAPL to MSFT
Major axis: 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Minor axis: Open to Adj Close

In [298]: pdata = pdata.swapaxes('items', 'minor')

In [299]: pdata['Adj Close']
Out[299]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 861 entries, 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Data columns:
AAPL      861  non-null values
DELL      861  non-null values
GOOG      861  non-null values
MSFT      861  non-null values
dtypes: float64(4)
```

ix-based label indexing generalizes to three dimensions, so we can select all data at a particular date or a range of dates like so:

```
In [300]: pdata.ix[:, '6/1/2012', :]
Out[300]:
```

	Open	High	Low	Close	Volume	Adj Close
AAPL	569.16	572.65	560.52	560.99	18606700	560.99
DELL	12.15	12.30	12.05	12.07	19396700	12.07
GOOG	571.79	572.65	568.35	570.98	3057900	570.98
MSFT	28.76	28.96	28.44	28.45	56634300	28.45

```
In [301]: pdata.ix['Adj Close', '5/22/2012':, :]
Out[301]:
```

	AAPL	DELL	GOOG	MSFT
Date				
2012-05-22	556.97	15.08	600.80	29.76
2012-05-23	570.56	12.49	609.46	29.11
2012-05-24	565.32	12.45	603.66	29.07
2012-05-25	562.29	12.46	591.53	29.06
2012-05-29	572.27	12.66	594.34	29.56
2012-05-30	579.17	12.56	588.23	29.34

```

2012-05-31  577.73  12.33  580.86  29.19
2012-06-01  560.99  12.07  570.98  28.45

```

An alternate way to represent panel data, especially for fitting statistical models, is in “stacked” DataFrame form:

```
In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()
```

```
In [303]: stacked
```

```
Out[303]:
```

		Open	High	Low	Close	Volume	Adj Close
2012-05-30	major						
	minor						
2012-05-30	AAPL	569.20	579.99	566.56	579.17	18908200	579.17
	DELL	12.59	12.70	12.46	12.56	19787800	12.56
	GOOG	588.16	591.90	583.53	588.23	1906700	588.23
	MSFT	29.35	29.48	29.12	29.34	41585500	29.34
2012-05-31	AAPL	580.74	581.50	571.46	577.73	17559800	577.73
	DELL	12.53	12.54	12.33	12.33	19955500	12.33
	GOOG	588.72	590.00	579.00	580.86	2968300	580.86
	MSFT	29.30	29.42	28.94	29.19	39134000	29.19
2012-06-01	AAPL	569.16	572.65	560.52	560.99	18606700	560.99
	DELL	12.15	12.30	12.05	12.07	19396700	12.07
	GOOG	571.79	572.65	568.35	570.98	3057900	570.98
	MSFT	28.76	28.96	28.44	28.45	56634300	28.45

DataFrame has a related `to_panel` method, the inverse of `to_frame`:

```
In [304]: stacked.to_panel()
```

```
Out[304]:
```

```
<class 'pandas.core.panel.Panel'>
```

```
Dimensions: 6 (items) x 3 (major) x 4 (minor)
```

```
Items: Open to Adj Close
```

```
Major axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00
```

```
Minor axis: AAPL to MSFT
```

# Data Loading, Storage, and File Formats

The tools in this book are of little use if you can't easily import and export data in Python. I'm going to be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process. NumPy, for example, features low-level but extremely fast binary data loading and storage, including support for memory-mapped array. See [Chapter 12](#) for more on those.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with network sources like web APIs.

## Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. [Table 6-1](#) has a summary of all of them, though `read_csv` and `read_table` are likely the ones you'll use the most.

*Table 6-1. Parsing functions in pandas*

Function	Description
<code>read_csv</code>	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
<code>read_table</code>	Load delimited data from a file, URL, or file-like object. Use tab ('\\t') as default delimiter
<code>read_fwf</code>	Read data in fixed-width column format (that is, no delimiters)
<code>read_clipboard</code>	Version of <code>read_table</code> that reads data from the clipboard. Useful for converting tables from web pages