UNIT - I

Launching the IPython Shell

This chapter, like most of this book, is not designed to be absorbed passively. I recom- mend that as you read through it, you follow along and experiment with the tools and syntax we cover: the muscle-memory you build through doing this will be far more useful than the simple act of reading about it. Start by launching the IPython inter- preter by typing **ipython** on the command line; alternatively, if you've installed a dis- tribution like Anaconda or EPD, there may be a launcher specific to your system.

Once you do this, you should see a prompt like the following:

```
IPython 4.0.1 -- 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]:
```

With that, you're ready to follow along.

Launching the Jupyter Notebook

The Jupyter notebook is a browser-based graphical interface to the IPython shell, and builds on it a rich set of dynamic display capabilities. As well as executing Python/ IPython statements, the notebook allows the user to include formatted text, static and dynamic visualizations, mathematical equations, JavaScript widgets, and much more. Furthermore, these documents can be saved in a way that lets other people open them and execute the code on their own systems.

Though the IPython notebook is viewed and edited through your web browser win- dow, it must connect to a running Python process in order to execute code. To start this process (known as a “kernel”), run the following command in your system shell:

```
$ jupyter notebook
```

This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing; that log will look something like this:

Keyboard Shortcuts in the IPython Shell

If you spend any amount of time on the computer, you’ve probably found a use for keyboard shortcuts in your workflow. Most familiar perhaps are Cmd-C and Cmd-V (or Ctrl-C and Ctrl-V) for copying and pasting in a wide variety of programs and sys- tems. Power users tend to go even further: popular text editors like Emacs, Vim, and others provide users an incredible range of operations through intricate combina- tions of keystrokes.

The IPython shell doesn’t go this far, but does provide a number of keyboard short- cuts for fast navigation while you’re typing commands. These shortcuts are not in fact provided by IPython itself, but through its dependency on the GNU Readline library: thus, some of the following shortcuts may differ depending on your system configu- ration. Also, while some of these shortcuts do work in the browser-based notebook, this section is primarily about shortcuts in the IPython shell.

Once you get accustomed to these, they can be very useful for quickly performing certain commands without moving your hands from the “home” keyboard position. If you’re an Emacs user or if you have experience with Linux-style shells, the follow- ing will be very familiar. We’ll group these shortcuts into a few categories: *navigation shortcuts*, *text entry shortcuts*, *command history shortcuts*, and *miscellaneous shortcuts*.

Navigation Shortcuts

While the use of the left and right arrow keys to move backward and forward in the line is quite obvious, there are other options that don't require moving your hands from the "home" keyboard position:

Keystroke	Action
Ctrl-a	Move cursor to the beginning of the line
Ctrl-e	Move cursor to the end of the line
Ctrl-b (or the left arrow key)	Move cursor back one character
Ctrl-f (or the right arrow key)	Move cursor forward one character

Text Entry Shortcuts

While everyone is familiar with using the Backspace key to delete the previous char- acter, reaching for the key often requires some minor finger gymnastics, and it only deletes a single character at a time. In IPython there are several shortcuts for remov- ing some portion of the text you're typing. The most immediately useful of these are the commands to delete entire lines of text. You'll know these have become second nature if you find yourself using a combination of Ctrl-b and Ctrl-d instead of reach- ing for the Backspace key to delete the previous character!

Keystroke	Action
Backspace key	Delete previous character in line
Ctrl-d	Delete next character in line
Ctrl-k	Cut text from cursor to end of line
Ctrl-u	Cut text from beginning fo line to cursor

Ctrl-y	Yank (i.e., paste) text that was previously cut
Ctrl-t	Transpose (i.e., switch) previous two characters

Miscellaneous Shortcuts

Finally, there are a few miscellaneous shortcuts that don’t fit into any of the preceding categories, but are nevertheless useful to know:

Ctrl-l	Clear terminal screen
Ctrl-c	Interrupt current Python command
Ctrl-d	Exit IPython session

The Ctrl-c shortcut in particular can be useful when you inadvertently start a very long-running job.

While some of the shortcuts discussed here may seem a bit tedious at first, they quickly become automatic with practice. Once you develop that muscle memory, I suspect you will even find yourself wishing they were available in other contexts.

IPython Magic Commands

The previous two sections showed how IPython lets you use and explore Python effi- ciently and interactively. Here we’ll begin discussing some of the enhancements that
IPython adds on top of the normal Python syntax. These are known in IPython as *magic commands*, and are prefixed by the % character. These magic commands are designed to

succinctly solve various common problems in standard data analysis. Magic commands come in two flavors: *line magics*, which are denoted by a single % prefix and operate on a single line of input, and *cell magics*, which are denoted by a double %% prefix and operate on multiple lines of input. We'll demonstrate and dis- cuss a few brief examples here, and come back to more focused discussion of several useful magic commands later in the chapter.

Pasting Code Blocks: %paste and %cpaste

When you're working in the IPython interpreter, one common gotcha is that pasting multiline code blocks can lead to unexpected errors, especially when indentation and interpreter markers are involved. A common case is that you find some example code on a website and want to paste it into your interpreter. Consider the following simple function:

```
>>>        def donothing(x):

...return x
```

The code is formatted as it would appear in the Python interpreter, and if you copy and paste this directly into IPython you get an error:

```
In [2]: >>> def donothing(x):

   ...:      ...      return x

   ...:

  File "<ipython-input-20-5a66c8964687>", line 2

    ...      return x

                   ^

SyntaxError: invalid syntax
```

In the direct paste, the interpreter is confused by the additional prompt characters. But never fear—IPython's %paste magic function is designed to handle this exact type of multiline, marked-up input:

```
In [3]: %paste

>>>         def donothing(x):

...return x

## -- End pasted text --
```

The %paste command both enters and executes the code, so now the function is ready to be used:

```
In [4]: donothing(10)

Out[4]: 10
```

A command with a similar intent is %cpaste, which opens up an interactive multiline prompt in which you can paste one or more chunks of code to be executed in a batch:

```
In [5]: %cpaste

Pasting code; enter '--' alone on the line to stop or use Ctrl-D.

:>>> def donothing(x):

:...      return x

:--
```

These magic commands, like others we'll see, make available functionality that would be difficult or impossible in a standard Python interpreter.

Running External Code: %run

As you begin developing more extensive code, you will likely find yourself working in both IPython for interactive exploration, as well as a text editor to store code that you want to reuse. Rather than running this code in a new window, it can be convenient to run it within your IPython session. This can be done with the %run magic.

For example, imagine you’ve created a *myscript.py* file with the following contents:

```
#-------------------------------------

# file: myscript.py

def square(x):

    """square a number"""

    return x ** 2

for N in range(1, 4):

    print(N, "squared is", square(N))
```

You can execute this from your IPython session as follows:

```
In [6]: %run myscript.py

1 squared is 1

2 squared is 4

3 squared is 9
```

Note also that after you've run this script, any functions defined within it are available for use in your IPython session:

```
In [7]: square(5)

Out[7]: 25
```

There are several options to fine-tune how your code is run; you can see the docu- mentation in the normal way, by typing **%run?** in the IPython interpreter.

Timing Code Execution: %timeit

Another example of a useful magic function is %timeit, which will automatically determine the execution time of the single-line Python statement that follows it. For example, we may want to check the performance of a list comprehension:

```
In [8]: %timeit L = [n ** 2 for n in range(1000)]

1000 loops, best of 3: 325 µs per loop
```

The benefit of %timeit is that for short commands it will automatically perform mul- tiple runs in order to attain more robust results. For multiline statements, adding a second % sign will turn this into a cell magic that can handle multiple lines of input. For example, here's the equivalent construction with a for loop:

```
In [9]: %%timeit

   ...: L = []

   ...: for n in range(1000):

   ...:         L.append(n ** 2)

   ...:
```

```
1000 loops, best of 3: 373 µs per loop
```

We can immediately see that list comprehensions are about 10% faster than the equivalent for loop construction in this case. We'll explore %timeit and other approaches to timing and profiling code in "Profiling and Timing Code" on page 25.

Help on Magic Functions: ?, %magic, and %lsmagic

Like normal Python functions, IPython magic functions have docstrings, and this useful documentation can be accessed in the standard manner. So, for example, to read the documentation of the %timeit magic, simply type this:

```
In [10]: %timeit?
```

Documentation for other functions can be accessed similarly. To access a general description of available magic functions, including some examples, you can type this:

```
In [11]: %magic
```

For a quick and simple list of all available magic functions, type this:

```
In [12]: %lsmagic
```

Finally, I'll mention that it is quite straightforward to define your own magic func- tions if you wish. We won't discuss it here, but if you are interested, see the references listed in "More IPython Resources" on page 30.

Input and Output History

Previously we saw that the IPython shell allows you to access previous commands with the up and down arrow keys, or equivalently the Ctrl-p/Ctrl-n shortcuts. Addi- tionally, in both the shell and the notebook, IPython exposes several ways to obtain the output of previous commands, as well as string versions of the commands them- selves. We'll explore those here.

IPython’s In and Out Objects

By now I imagine you’re quite familiar with the In[1]:/Out[1]: style prompts used

by IPython. But it turns out that these are not just pretty decoration: they give a clue as to how you can access previous inputs and outputs in your current session. Imag- ine you start a session that looks like this:

```
In [1]: import math

In [2]: math.sin(2)

Out[2]: 0.9092974268256817

In [3]: math.cos(2)

Out[3]: -0.4161468365471424
```

We’ve imported the built-in math package, then computed the sine and the cosine of the number 2. These inputs and outputs are displayed in the shell with In/Out labels, but there’s more—IPython actually creates some Python variables called In and Out that are automatically updated to reflect this history:

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

['', 'import math', 'math.sin(2)', 'math.cos(2)', 'print(In)']

In [5]: Out

Out[5]: {2: 0.9092974268256817, 3: -0.4161468365471424}
```

The In object is a list, which keeps track of the commands in order (the first item in the list is a placeholder so that In[1] can refer to the first command):

```
In [6]: print(In[1])

import math
```

The Out object is not a list but a dictionary mapping input numbers to their outputs (if any):

```
In [7]: print(Out[2])

0.9092974268256817
```

Note that not all operations have outputs: for example, import statements and print statements don’t affect the output. The latter may be surprising, but makes sense if you consider that print is a function that returns None; for brevity, any command that returns None is not added to Out.

Where this can be useful is if you want to interact with past results. For example, let’s check the sum of sin(2) ** 2 and cos(2) ** 2 using the previously computed results:

```
In [8]: Out[2] ** 2 + Out[3] ** 2

Out[8]: 1.0
```

The result is 1.0 as we’d expect from the well-known trigonometric identity. In this case, using these previous results probably is not necessary, but it can become very handy if you execute a very expensive computation and want to reuse the result!

IPython and Shell Commands

When working interactively with the standard Python interpreter, one of the frustra- tions you’ll face is the need to switch between multiple windows to access Python tools and system command-line tools. IPython bridges this gap, and gives you a syn- tax for executing shell commands directly from within the IPython terminal. The magic happens with the exclamation point: anything appearing after ! on a line will be executed not by the Python kernel, but by the system command line.

The following assumes you’re on a Unix-like system, such as Linux or Mac OS X. Some of the examples that follow will fail on Windows, which uses a different type of shell by default (though with the 2016 announcement of native Bash shells on Win- dows, soon this may no longer be an issue!). If you’re unfamiliar with shell com- mands, I’d suggest reviewing the Shell Tutorial put together by the always excellent Software Carpentry Foundation.

Quick Introduction to the Shell

A full intro to using the shell/terminal/command line is well beyond the scope of this chapter, but for the uninitiated we will offer a quick introduction here. The shell is a way to interact textually with your computer. Ever since the mid-1980s, when Micro- soft and Apple introduced the first versions of their now ubiquitous graphical operat- ing systems, most computer users have interacted with their operating system through familiar clicking of menus and drag-and-drop movements. But operating systems existed long before these graphical user interfaces, and were primarily con- trolled through sequences of text input: at the prompt, the user would type a com- mand, and the computer would do what the user told it to. Those early prompt
systems are the precursors of the shells and terminals that most active data scientists still use today.

Someone unfamiliar with the shell might ask why you would bother with this, when you can accomplish many results by simply clicking on icons and menus. A shell user might reply with another question: why hunt icons and click menus when you can accomplish things much more easily by typing? While it might sound like a typical tech preference impasse, when moving beyond basic tasks it quickly becomes clear that the shell offers much more control of advanced tasks, though admittedly the learning curve can intimidate the average computer user.

As an example, here is a sample of a Linux/OS X shell session where a user explores, creates, and modifies directories and files on their system (osx:~ $ is the prompt, and everything after the $ sign is the typed command; text that is preceded by a # is meant just as description, rather than something you would actually type in):

```
osx:~ $ echo "hello world"          # echo is like Python's print function
```

```
hello world
osx:~        $
pwd                              # pwd = print working directory
/home/jake                       # this is the "path" that we're in
osx:~ $ ls                       # ls = list working directory contents
notebooks  projects
osx:~ $ cd projects/             # cd = change directory
osx:projects $ pwd
/home/jake/projects
osx:projects $ ls
datasci_boo    mpld
k              3        myproject.txt

osx:projects $ mkdir myproject   # mkdir = make new directory

osx:projects $ cd myproject/

osx:myproject $ mv ../myproject.txt ./     # mv = move file. Here we're moving the

                                 # file myproject.txt from one directory

                                 # up (../) to the current directory (./)

osx:myproject $ ls

myproject.txt
```

Notice that all of this is just a compact way to do familiar operations (navigating a directory structure, creating a directory, moving a file, etc.) by typing commands rather than clicking icons and menus. Note that with just a few commands (pwd, ls, cd, mkdir, and cp) you can do many of the most common file operations. It’s when you go beyond these basics that the shell approach becomes really powerful.

Shell Commands in IPython

You can use any command that works at the command line in IPython by prefixing it with the ! character. For example, the ls, pwd, and echo commands can be run as follows:

```
In [1]: !ls

myproject.txt

In [2]: !pwd

/home/jake/projects/myproject

In [3]: !echo "printing from the shell"

printing from the shell
```

Passing Values to and from the Shell

Shell commands can not only be called from IPython, but can also be made to inter- act with the IPython namespace. For example, you can save the output of any shell command to a Python list using the assignment operator:

```
In [4]: contents = !ls

In [5]: print(contents)

['myproject.txt']

In [6]: directory = !pwd

In [7]: print(directory)

['/Users/jakevdp/notebooks/tmp/myproject']
```

Note that these results are not returned as lists, but as a special shell return type defined in IPython:

```
In [8]: type(directory)

IPython.utils.text.SList
```

This looks and acts a lot like a Python list, but has additional functionality, such as the grep and fields methods and the s, n, and p properties that allow you to search, filter, and display the results in convenient ways. For more information on these, you can use IPython's built-in help features.

Communication in the other direction—passing Python variables into the shell—is possible through the {varname} syntax:

```
In [9]: message = "hello from Python"

In [10]: !echo {message}

hello from Python
```

The curly braces contain the variable name, which is replaced by the variable's con- tents in the shell command.

Shell-Related Magic Commands

If you play with IPython's shell commands for a while, you might notice that you can- not use !cd to navigate the filesystem:

```
In [11]: !pwd

/home/jake/projects/myproject

In [12]: !cd ..
In [13]: !pwd

/home/jake/projects/myproject
```

The reason is that shell commands in the notebook are executed in a temporary sub- shell. If you'd like to change the working directory in a more enduring way, you can use the %cd magic command:

```
In [14]: %cd ..

/home/jake/projects
```

In fact, by default you can even use this without the % sign:

```
In [15]: cd myproject

/home/jake/projects/myproject
```

This is known as an automagic function, and this behavior can be toggled with the %automagic magic function.

Besides %cd, other available shell-like magic functions are %cat, %cp, %env, %ls, %man, %mkdir, %more, %mv, %pwd, %rm, and %rmdir, any of which can be used without the % sign if automagic is on. This makes it so that you can almost treat the IPython prompt as if it's a normal shell:

```
In [16]: mkdir tmp

In [17]: ls

myproject.txt     tmp/

In [18]: cp myproject.txt tmp/
In [19]: ls tmp

myproject.txt

In [20]: rm -r tmp
```

This access to the shell from within the same terminal window as your Python ses- sion means that there is a lot less switching back and forth between interpreter and shell as you write your Python code.

Errors and Debugging

Code development and data analysis always require a bit of trial and error, and IPython contains tools to streamline this process. This section will briefly cover some options for controlling Python's exception reporting, followed by exploring tools for debugging errors in code.

Controlling Exceptions: %xmode

Most of the time when a Python script fails, it will raise an exception. When the inter- preter hits one of these exceptions, information about the cause of the error can be found in the *traceback*, which can be accessed from within Python. With the %xmode magic function, IPython allows you to control the amount of information printed when the exception is raised. Consider the following code:

```
In[1]: def func1(a, b):

return a / b

def func2(x):

        a = x

        b = x - 1

        return func1(a, b)
```

```
In[2]: func2(1)

---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-2-b2e110f6fc8f^gt; in <module>()

----> 1 func2(1)



<ipython-input-1-d849e34d61fb> in func2(x)

      5     a = x

      6     b = x - 1

----> 7     return func1(a, b)



<ipython-input-1-d849e34d61fb> in func1(a, b)

1       def func1(a, b):
----> 2     return a / b
3
4       def func2(x):
5           a = x



ZeroDivisionError: division by zero
```

Calling func2 results in an error, and reading the printed trace lets us see exactly what happened. By default, this trace includes several lines showing the context of each step that led to the error. Using the %xmode magic function (short for *exception mode*), we can change what information is printed.

%xmode takes a single argument, the mode, and there are three possibilities: Plain, Context, and Verbose. The default is Context, and gives output like that just shown. Plain is more compact and gives less information:

```
In[3]: %xmode Plain

Exception reporting mode: Plain

In[4]: func2(1)

---------------------------------------------------------

Traceback (most recent call last):


  File "<ipython-input-4-b2e110f6fc8f>", line 1, in <module> func2(1)


  File "<ipython-input-1-d849e34d61fb>", line 7, in func2 return
    func1(a, b)


  File "<ipython-input-1-d849e34d61fb>", line 2, in func1 return a / b


ZeroDivisionError: division by zero
```

The Verbose mode adds some extra information, including the arguments to any functions that are called:

```
In[5]: %xmode Verbose
```

Exception reporting mode: Verbose

```
In[6]: func2(1)

---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-6-b2e110f6fc8f> in <module>()

----> 1 func2(1)

        global func2 = <function func2 at 0x103729320>



<ipython-input-1-d849e34d61fb> in func2(x=1)

      5     a = x

      6     b = x - 1

----> 7     return func1(a, b)

        global func1 = <function func1 at 0x1037294d0>

        a = 1

        b = 0
<ipython-input-1-d849e34d61fb> in func1(a=1, b=0)

1       def func1(a, b):
----> 2     return a / b
```

```
          a = 1
          b = 0
3
4         def func2(x):
5             a = x


ZeroDivisionError: division by zero
```

This extra information can help you narrow in on why the exception is being raised. So why not use the Verbose mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of Default mode is easier to work with.

Partial list of debugging commands

There are many more available commands for interactive debugging than we've listed here; the following table contains a description of some of the more common and useful ones:

Command	Description
list	Show the current location in the file
h(elp)	Show a list of commands, or find help on a specific command
q(uit)	Quit the debugger and the program
c(ontinue)	Quit the debugger; continue in the program
n(ext)	Go to the next step of the program
<enter>	Repeat the previous command
p(rint)	Print variables
s(tep)	Step into a subroutine
r(eturn)	Return out of a subroutine

Profiling and Timing Code

In the process of developing code and creating data processing pipelines, there are often trade-offs you can make between various implementations. Early in developing your algorithm, it can be counterproductive to worry about such things. As Donald Knuth famously quipped, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

But once you have your code working, it can be useful to dig into its efficiency a bit. Sometimes it's useful to check the execution time of a given command or set of com- mands; other times it's useful to dig into a multiline process and determine where the bottleneck lies in some complicated series of operations. IPython provides access to a wide array of functionality for this kind of timing and profiling of code. Here we'll discuss the following IPython magic commands:

%time

Time the execution of a single statement

%timeit

Time repeated execution of a single statement for more accuracy

%prun

Run code with the profiler

%lprun

Run code with the line-by-line profiler

%memit

Measure the memory use of a single statement

%mprun

Run code with the line-by-line memory profiler

The last four commands are not bundled with IPython—you'll need to install the line_profiler and memory_profiler extensions, which we will discuss in the fol- lowing sections.

Timing Code Snippets: %timeit and %time

We saw the %timeit line magic and %%timeit cell magic in the introduction to magic functions in "IPython Magic Commands" on page 10; %%timeit can be used to time the repeated execution of snippets of code:

```
In[1]: %timeit sum(range(100))

100000 loops, best of 3: 1.54 µs per loop
```

Note that because this operation is so fast, %timeit automatically does a large number of repetitions. For slower commands, %timeit will automatically adjust and perform fewer repetitions:

```
In[2]: %%timeit

    total = 0

    for i in range(1000):

        for j in range(1000):

            total += i * (-1) ** j

1 loops, best of 3: 407 ms per loop
```

Sometimes repeating an operation is not the best option. For example, if we have a list that we'd like to sort, we might be misled by a repeated operation. Sorting a pre-sorted list is much faster than sorting an unsorted list, so the repetition will skew the result:

```
In[3]: import random

       L = [random.random() for i in range(100000)] %timeit
       L.sort()

100 loops, best of 3: 1.9 ms per loop
```

For this, the %time magic function may be a better choice. It also is a good choice for longer-running commands, when short, system-related delays are unlikely to affect the result. Let’s time the sorting of an unsorted and a presorted list:

```
In[4]: import random

       L = [random.random() for i in range(100000)]
       print("sorting an unsorted list:")
       %time L.sort()

sorting an unsorted list:

CPU times: user 40.6 ms, sys: 896 µs, total: 41.5 ms

Wall time: 41.5 ms

In[5]: print("sorting an already sorted list:")

       %time L.sort()

sorting an already sorted list:

CPU times: user 8.18 ms, sys: 10 µs, total: 8.19 ms

Wall time: 8.24 ms
```

Notice how much faster the presorted list is to sort, but notice also how much longer the timing takes with %time versus %timeit, even for the presorted list! This is a result of the fact that %timeit does some clever things under the hood to prevent sys- tem calls from interfering with the timing. For example, it prevents cleanup of unused Python objects (known as *garbage collection*) that might otherwise affect the timing. For this reason, %timeit results are usually noticeably faster than %time results.

For %time as with %timeit, using the double-percent-sign cell-magic syntax allows timing of multiline scripts:

```
In[6]: %%time

    total = 0

    for i in range(1000):

        for j in range(1000):

            total += i * (-1) ** j

CPU times: user 504 ms, sys: 979 µs, total: 505 ms

Wall time: 505 ms
```

For more information on %time and %timeit, as well as their available options, use the IPython help functionality (i.e., type **%time?** at the IPython prompt).

Profiling Full Scripts: %prun

A program is made of many single statements, and sometimes timing these state- ments in context is more important than timing them on their own. Python contains a built-in code profiler (which you can read about in the Python documentation), but IPython offers a much more convenient way to use this profiler, in the form of the magic function %prun.

By way of example, we'll define a simple function that does some calculations:

```
In[7]: def sum_of_lists(N):
           total = 0
           for i in range(5):
               L = [j ^ (j >> i) for j in range(N)]
               total += sum(L)
           return total
```

Now we can call %prun with a function call to see the profiled results:

```
In[8]: %prun sum_of_lists(1000000)
```

In the notebook, the output is printed to the pager, and looks something like this:

```
14 function calls in 0.714 seconds

   Ordered by: internal time

                           cumtim
   ncalls  tottime  percall  e      percall  filename:lineno(function)
                                             <ipython-input-
   5       0.599    0.120    0.599  0.120    19>:4(<listcomp>)
   5       0.064    0.013    0.064  0.013    {built-in method sum}
   1       0.036    0.036    0.699  0.699    <ipython-input-19>:1(sum_of_lists)
   1       0.014    0.014    0.714  0.714    <string>:1(<module>)
   1       0.000    0.000    0.714  0.714    {built-in method exec}
```

The result is a table that indicates, in order of total time on each function call, where the execution is spending the most time. In this case, the bulk of execution time is in the list

comprehension inside sum_of_lists. From here, we could start thinking about what changes we might make to improve the performance in the algorithm.
For more information on %prun, as well as its available options, use the IPython help functionality (i.e., type **%prun?** at the IPython prompt).

Line-by-Line Profiling with %lprun

The function-by-function profiling of %prun is useful, but sometimes it's more conve- nient to have a line-by-line profile report. This is not built into Python or IPython, but there is a line_profiler package available for installation that can do this. Start by using Python's packaging tool, pip, to install the line_profiler package:

```
$ pip install line_profiler
```

Next, you can use IPython to load the line_profiler IPython extension, offered as part of this package:

```
In[9]: %load_ext line_profiler
```

Now the %lprun command will do a line-by-line profiling of any function—in this case, we need to tell it explicitly which functions we're interested in profiling:

```
In[10]: %lprun -f sum_of_lists sum_of_lists(5000)
```

As before, the notebook sends the result to the pager, but it looks something like this:

```
Timer unit: 1e-06 s

Total time: 0.009382 s

File: <ipython-input-19-fa2be176cc3e>

Function: sum_of_lists at line 1
```

```
                                    %
Line #   Hits      Time    Per Hit  Time    Line Contents
==============================================
===============
1                                           def sum_of_lists(N):
2        1         2       2.0      0.0    total      = 0
3        6         8       1.3      0.1    for i      in range(5):
4        5         9001    1800.2   95.9   L          = [j ^   (j >> i) ...
5        5         371     74.2     4.0    total +=          sum(L)
6        1         0       0.0      0.0    return total
```

The information at the top gives us the key to reading the results: the time is reported in microseconds and we can see where the program is spending the most time. At this point, we may be able to use this information to modify aspects of the script and make it perform better for our desired use case.

For more information on %lprun, as well as its available options, use the IPython help functionality (i.e., type **%lprun?** at the IPython prompt).

Profiling Memory Use: %memit and %mprun

Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the memory_profiler. As with the line_profiler, we start by pip-installing the extension:

```
$ pip install memory_profiler
```

Then we can use IPython to load the extension:

```
In[12]: %load_ext memory_profiler
```

The memory profiler extension contains two useful magic functions: the %memit magic (which offers a memory-measuring equivalent of %timeit) and the %mprun function (which offers a memory-measuring equivalent of %lprun). The %memit func- tion can be used rather simply:

```
In[13]: %memit sum_of_lists(1000000)

peak memory: 100.08 MiB, increment: 61.36 MiB
```

We see that this function uses about 100 MB of memory.

For a line-by-line description of memory use, we can use the %mprun magic. Unfortu- nately, this magic works only for functions defined in separate modules rather than the notebook itself, so we'll start by using the %%file magic to create a simple module called mprun_demo.py, which contains our sum_of_lists function, with one addition that will make our memory profiling results more clear:

```
In[14]: %%file mprun_demo.py

def sum_of_lists(N):

    total = 0

    for i in range(5):

        L = [j ^ (j >> i) for j in range(N)]

        total += sum(L)

        del L # remove reference to L

    return total

Overwriting mprun_demo.py
```

We can now import the new version of this function and run the memory line profiler:

```
In[15]: from mprun_demo import sum_of_lists
```

```
%mprun -f sum_of_lists sum_of_lists(1000000)
```

The result, printed to the pager, gives us a summary of the memory use of the func- tion, and looks something like this:

```
Filename:
./mprun_demo.py
                Mem
Line #        usage  Increment  Line Contents
=====================================
                                       ===========
                   Mi            Mi
      4     71.9    B       0.0    B             L = [j ^ (j >> i) for j in range(N)]
Filename:
./mprun_demo.py
                Mem
Line #        usage  Increment  Line Contents
=====================================
                                       ===========
                   Mi            Mi
      1     39.0    B       0.0    B   def sum_of_lists(N):
                   Mi            Mi
      2     39.0    B       0.0    B        total = 0
                   Mi            Mi
      3     46.5    B       7.5    B        for i in range(5):
                   Mi            Mi
      4     71.9    B     25.4    B             L = [j ^ (j >> i) for j in range(N)]
                   Mi            Mi
      5     71.9    B       0.0    B             total += sum(L)
                   Mi
      6     46.5    B  -25.4 MiB            del L # remove reference to L
                   Mi
      7     39.1    B    -7.4 MiB        return total
```

Here the Increment column tells us how much each line affects the total memory budget: observe that when we create and delete the list L, we are adding about 25 MB of memory usage. This is on top of the background memory usage from the Python interpreter itself.

For more information on %memit and %mprun, as well as their available options, use the IPython help functionality (i.e., type **%memit?** at the IPython prompt).

More IPython Resources

In this chapter, we've just scratched the surface of using IPython to enable data sci- ence tasks. Much more information is available both in print and on the Web, and here we'll list some other resources that you may find helpful.

Web Resources

The IPython website

The IPython website links to documentation, examples, tutorials, and a variety of other resources.

The nbviewer website

This site shows static renderings of any IPython notebook available on the Inter- net. The front page features some example notebooks that you can browse to see what other folks are using IPython for!

A Gallery of Interesting IPython Notebooks

This ever-growing list of notebooks, powered by nbviewer, shows the depth and breadth of numerical analysis you can do with IPython. It includes everything from short examples and tutorials to full-blown courses and books composed in the notebook format!

Video tutorials

Searching the Internet, you will find many video-recorded tutorials on IPython. I'd especially recommend seeking tutorials from the PyCon, SciPy, and PyData conferences by Fernando Perez and Brian Granger, two of the primary creators and maintainers of IPython and Jupyter.

Books

Python for Data Analysis

Wes McKinney's book includes a chapter that covers using IPython as a data sci- entist. Although much of the material overlaps what we've discussed here, another perspective is always helpful.

Learning IPython for Interactive Computing and Data Visualization

This short book by Cyrille Rossant offers a good introduction to using IPython for data analysis.

IPython Interactive Computing and Visualization Cookbook

Also by Cyrille Rossant, this book is a longer and more advanced treatment of using IPython for data science. Despite its name, it's not just about IPython—it also goes into some depth on a broad range of data science topics.

Finally, a reminder that you can find help on your own: IPython's ?-based help func- tionality (discussed in "Help and Documentation in IPython" on page 3) can be very useful if you use it well and use it often. As you go through the examples here and elsewhere, you can use it to familiarize yourself with all the tools that IPython has to offer.