In Chapter 2 we looked at the basics of using the IPython shell and Jupyter notebook. In this chapter, we explore some deeper functionality in the IPython system that can either be used from the console or within Jupyter.

B.1 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 mini- mal typing

• Persisting the command history between sessions

• Logging the input/output history to a file These features are more useful in the shell than in the notebook, since the notebook by design keeps a log of the input and output in each code cell.

Searching and Reusing the Command History

The IPython shell lets you search and execute previous code or other commands. This is useful, as you may often find yourself repeating the same commands, such as a %run command or some other code snippet. Suppose you had run:

```
In [1]: #  %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 and 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 the up arrow key 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 the down arrow key. 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 and then type a few characters contained in the input line you want to search for:

```
In [2]: #  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. An IPython session stores references to both the input commands and output Python objects in special variables. The previous two outputs are stored in the _ (one under- score) and __ (two underscores) variables, respectively:

```
In [3]: 2 ** 27
Out[3]: 134217728

In [4]: _
Out[4]: 134217728

In [5]: __
Out[5]: 134217728
```

Input variables are stored in variables named like _iX, where X is the input line num- ber. For each input variable 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 [6]: _3
Out[6]: 134217728

In [7]: _i3
Out[7]: '2 ** 27'

In [8]: foo = 'bar'

In [9]: foo
Out[9]: 'bar'

In [10]: _i8
Out[10]: "foo = 'bar'"

In [11]: _i9
Out[11]: 'foo'
```

```
In [12]: _9
```

```
Out[12]: 'bar'
```

Since the input variables are strings they can be executed again with the Python exec keyword:

```
In [13]: exec(_i9)
```

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 particu- lar object from the IPython machinery. See the documentation for both of these mag- ics for more details.

```
In [14]: %hist
```

```
#  %run first/second/third/data_script.py
#  a_command = foo(x, y, z)
2 ** 27
_
__
_3
_i3
foo = 'bar'
foo
_i8
_i9
_9
exec(_i9)
%hist
```

When working with very large datasets, keep in mind that IPy- thon'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.

B.2 Interacting with the Operating System

Another feature of IPython is that it allows you to seamlessly access the filesystem and 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, macOS) shell without having to exit IPython. This includes shell commands, chang- ing directories, and storing the results of a command in a Python object (list or string). There are also simple command aliasing and directory bookmarking features.

See Table B-1 for a summary of magic functions and syntax for calling shell com- mands. I'll briefly visit these features in the next few sections

Table B-1. IPython system-related commands

Command --> Description

!cmd --> Execute cmd in the system shell

output = !cmd args --> Run cmd and store the stdout in output

%alias --> alias_name cmd Define an alias for a system (shell) command

%bookmark --> Utilize IPython's directory bookmarking system

%cd --> directory Change system working directory to passed directory

%pwd --> Return the current system working directory

%pushd --> directory Place current directory on stack and change to target directory

%popd --> Change to directory popped off the top of the stack

%dirs --> Return a list containing the current directory stack

%dhist --> Print the history of visited directories

%env --> Return the system environment variables as a dict

%matplotlib --> Configure matplotlib integration options

Shell Commands and Aliases

Starting a line in IPython with an exclamation point !, or bang, tells IPython to exe- cute 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.

You can store the console output of a shell command in a variable by assigning the expression escaped with ! 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 [23]:
```
ip_info = !ipconfig
```

In [34]:
```
ip_info[12].strip()
```

Out[34]:
```
'Link-local IPv6 Address . . . . . : fe80::de68:b8cf:e26f:b43c%42'
```

In [35]:
```
# ip_info = !ifconfig wlan0 | grep "inet "
# In [2]: ip_info[0].strip()
```

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 [38]:
```
foo = 'test*'
!ls $foo
```

```
'ls' is not recognized as an internal or external command,
operable program or batch file.
```

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

In [39]:
```
%alias ll ls -l
ll /usr
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-39-2f1c784977b9> in <module>
      1 get_ipython().run_line_magic('alias', 'll ls -l')
----> 2 ll /usr

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

You can execute multiple commands just as on the command line by separating them with semicolons:

In [40]:
```
%alias test_alias (cd examples; ls; cd ..)
```

In [41]:
```
test_alias
```

```
The system cannot find the path specified.
```

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.

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, suppose you wanted to create a bookmark that points to the supplementary materials for this book:

In [42]:
```
%bookmark py4da /home/wesm/code/pydata-book
```

Once you've done this, when we use the %cd magic, we can use any bookmarks we've defined:

In [43]:
```
cd py4da
```

```
(bookmark:py4da) -> /home/wesm/code/pydata-book
[WinError 3] The system cannot find the path specified: '/home/wesm/code/pydata-boo
k'
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practic
e_Code\Python_Practice\Python_For_Data_Analysis
```

If a bookmark name conflicts with a directory name in your current working direc- tory, 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 [44]:
```
%bookmark -l
```

```
Current bookmarks:
py4da -> /home/wesm/code/pydata-book
```

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

B.3 Software Development Tools

In addition to being a comfortable environment for interactive computing and data exploration, IPython can also be a useful companion for general Python software development. 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 con- text 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 [45]:
```
run examples/ipython_bug.py
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in run(self, parameter_s, runner, file_finder)
    696             fpath = arg_lst[0]
--> 697             filename = file_finder(fpath)
    698         except IndexError:

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\utils\path.py in get_py_f
ilename(name, force_win32)
    108     else:
--> 109         raise IOError('File `%r` not found.' % name)
    110

OSError: File `'examples/ipython_bug.py'` not found.

During handling of the above exception, another exception occurred:

Exception                                 Traceback (most recent call last)
<ipython-input-45-ac2d87c3d5c0> in <module>
----> 1 get_ipython().run_line_magic('run', 'examples/ipython_bug.py')

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\interactiveshell.py
in run_line_magic(self, magic_name, line, _stack_depth)
   2324                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325             with self.builtin_trap:
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\decorator.py in fun(*args, **kw)
    230             if not kwsyntax:
    231                 args, kw = fix(args, kw, sig)
--> 232             return caller(func, *(extras + args), **kw)
    233     fun.__name__ = func.__name__
    234     fun.__doc__ = func.__doc__

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magic.py in <lambda>
(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in run(self, parameter_s, runner, file_finder)
    706             if os.name == 'nt' and re.match(r"^'.*'$",fpath):
    707                 warn('For Windows, use double quotes to wrap a filename: %ru
n "mypath\\myfile.py"')
```

```
--> 708             raise Exception(msg)
    709         except TypeError:
    710             if fpath in sys.meta_path:

Exception: File `'examples/ipython_bug.py'` not found.
```

In [46]:
```
%debug
```

```
> c:\users\ankit19.gupta\onedrive - reliance corporate it park limited\desktop\practice_code\python_practice\python_for_data_analysis\myenv\lib\site-packages\ipython\core\magics\execution.py(708)run()
    706             if os.name == 'nt' and re.match(r"^'.*'$",fpath):
    707                 warn('For Windows, use double quotes to wrap a filename: %run "mypath\\myfile.py"')
--> 708             raise Exception(msg)
    709         except TypeError:
    710             if fpath in sys.meta_path:

ipdb> 1
1
ipdb> 2
2
ipdb> 3
3
ipdb> 4
4
ipdb> 5
5
ipdb> 6
6
ipdb> 7
7
ipdb> 8
8
ipdb>
8
--KeyboardInterrupt--
--KeyboardInterrupt--
ipdb> u
> c:\users\ankit19.gupta\onedrive - reliance corporate it park limited\desktop\practice_code\python_practice\python_for_data_analysis\myenv\lib\site-packages\ipython\core\magic.py(187)<lambda>()
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

ipdb> u
> c:\users\ankit19.gupta\onedrive - reliance corporate it park limited\desktop\practice_code\python_practice\python_for_data_analysis\myenv\lib\site-packages\decorator.py(232)fun()
    230             if not kwsyntax:
    231                 args, kw = fix(args, kw, sig)
--> 232             return caller(func, *(extras + args), **kw)
    233     fun.__name__ = func.__name__
    234     fun.__doc__ = func.__doc__

ipdb> u
> c:\users\ankit19.gupta\onedrive - reliance corporate it park limited\desktop\practice_code\python_practice\python_for_data_analysis\myenv\lib\site-packages\ipython\core\interactiveshell.py(2326)run_line_magic()
   2324                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325             with self.builtin_trap:
```

```
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328

ipdb> u
> <ipython-input-45-ac2d87c3d5c0>(1)<module>()
----> 1 get_ipython().run_line_magic('run', 'examples/ipython_bug.py')

ipdb> u
*** Oldest frame
ipdb> u
*** Oldest frame
ipdb> u
*** Oldest frame
ipdb> d
> c:\users\ankit19.gupta\onedrive - reliance corporate it park limited\desktop\pract
ice_code\python_practice\python_for_data_analysis\myenv\lib\site-packages\ipython\co
re\interactiveshell.py(2326)run_line_magic()
   2324                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325             with self.builtin_trap:
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328

ipdb> run -d examples/ipython_bug.py
```

```
---------------------------------------------------------------------------
Restart                                   Traceback (most recent call last)
<ipython-input-46-bc99e4ec804d> in <module>
----> 1 get_ipython().run_line_magic('debug', '')

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\interactiveshell.py
in run_line_magic(self, magic_name, line, _stack_depth)
   2324                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325             with self.builtin_trap:
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\decorator.py in fun(*args, **kw)
    230             if not kwsyntax:
    231                 args, kw = fix(args, kw, sig)
--> 232             return caller(func, *(extras + args), **kw)
    233     fun.__name__ = func.__name__
    234     fun.__doc__ = func.__doc__

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magic.py in <lambda>
(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in debug(self, line, cell)
    467
    468         if not (args.breakpoint or args.statement or cell):
--> 469             self._debug_post_mortem()
    470         else:
    471             code = "\n".join(args.statement)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in _debug_post_mortem(self)
    475
    476     def _debug_post_mortem(self):
--> 477         self.shell.debugger(force=True)
    478
    479     def _debug_exec(self, code, breakpoint):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\interactiveshell.py
in debugger(self, force)
   1189             return
   1190
-> 1191         self.InteractiveTB.debugger(force=True)
   1192
```

```
   1193     #--------------------------------------------------------------------------
--

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\ultratb.py in debugg
er(self, force)
   1264                     etb = etb.tb_next
   1265                 self.pdb.botframe = etb.tb_frame
-> 1266                 self.pdb.interaction(None, etb)
   1267
   1268         if hasattr(self, 'tb'):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\debugger.py in inter
action(self, frame, traceback)
    303     def interaction(self, frame, traceback):
    304         try:
--> 305             OldPdb.interaction(self, frame, traceback)
    306         except KeyboardInterrupt:
    307             self.stdout.write("\n" + self.shell.get_exception_only())

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\pdb.py in interaction(self, frame, traceback)
    350             return
    351         self.print_stack_entry(self.stack[self.curindex])
--> 352         self._cmdloop()
    353         self.forget()
    354

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\pdb.py in _cmdloop(self)
    319                 # the current command, so allow them during interactive inpu
t
    320                 self.allow_kbdint = True
--> 321                 self.cmdloop()
    322                 self.allow_kbdint = False
    323                 break

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\cmd.py in cmdloop(self, intro)
    136                             line = line.rstrip('\r\n')
    137                 line = self.precmd(line)
--> 138                 stop = self.onecmd(line)
    139                 stop = self.postcmd(stop, line)
    140             self.postloop()

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\pdb.py in onecmd(self, line)
    416         """
    417         if not self.commands_defining:
--> 418             return cmd.Cmd.onecmd(self, line)
    419         else:
    420             return self.handle_command_def(line)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\cmd.py in onecmd(self, line)
```

```
    215             except AttributeError:
    216                 return self.default(line)
--> 217             return func(arg)
    218
    219     def emptyline(self):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\pdb.py in do_run(self, arg)
   1026             sys.argv[:0] = argv0
   1027         # this is caught in the main debugger loop
-> 1028         raise Restart
   1029
   1030     do_restart = do_run

Restart:
```

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:

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 [ ]:
```
# In [5]: run -d examples/ipython_bug.py
# Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
# NOTE: Enter 'c' at the ipdb> prompt to start your script.
# > <string>(1)<module>()
# ipdb> s
# --Call--
# > /home/wesm/code/pydata-book/examples/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 preceding 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):

In [47]:
```
# ipdb> b 12
# ipdb> c
# > /home/wesm/code/pydata-book/examples/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:

In [48]:
```
# ipdb> n
# > /home/wesm/code/pydata-book/examples/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:

In [50]:
```
# ipdb> s
# --Call--
# > /home/wesm/code/pydata-book/examples/ipython_bug.py(6)throws_an_exception()
#  5
# 6 def throws_an_exception():
#  7 a = 5
# ipdb> n
# > /home/wesm/code/pydata-book/examples/ipython_bug.py(7)throws_an_exception()
#  6 def throws_an_exception():
# ----> 7 a = 5
#  8 b = 6
# ipdb> n
# > /home/wesm/code/pydata-book/examples/ipython_bug.py(8)throws_an_exception()
#  7 a = 5
# ----> 8 b = 6
#  9 assert(a + b == 10)
# ipdb> n
# > /home/wesm/code/pydata-book/examples/ipython_bug.py(9)throws_an_exception()
#  8 b = 6
# ----> 9 assert(a + b == 10)
#  10
# ipdb> !a
# 5
# ipdb> !b
# 6
```

Developing proficiency with the interactive debugger is largely a matter of practice and experience. See Table B-2 for a full catalog of the debugger commands. If you are accustomed to using an IDE, you might find the terminal-driven debugger to be a bit unforgiving at first, but that will improve in time. Some of the Python IDEs have excellent GUI debuggers, so most users can find something that works for them.

Table B-2. (I)Python debugger commands

Command --> Action

h(elp) --> Display command list

help --> command Show documentation for command

c(ontinue) --> Resume program execution

q(uit) --> Exit debugger without executing any more code

b(reak) --> number Set breakpoint at number in current file

b --> path/to/file.py:number Set breakpoint at line number in specified file

s(tep) --> Step into function call

n(ext) --> Execute current line and advance to next line at current level

u(p)/d(own) --> Move up/down in function call stack

a(rgs) --> Show arguments for current function

debug --> statement Invoke statement statement in new (recursive) debugger

l(ist) --> statement Show current position and context at current level of stack

w(here) --> 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):

In [51]:
```
from IPython.core.debugger import Pdb
def set_trace():
    Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)
def debug(f, *args, **kwargs):
    pdb = Pdb(color_scheme='Linux')
    return pdb.runcall(f, *args, **kwargs)
```

The first function, set_trace, is very simple. You can use a set_trace in any part of your code that you want to temporarily stop in order to more closely examine it (e.g., right before an exception occurs):

In [52]:
```
# run examples/ipython_bug.py
# > /home/wesm/code/pydata-book/examples/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 we just looked at enables you to invoke the interactive debugger easily on an arbitrary function call. Suppose we had written a function like the fol- lowing and we wished to step through its logic:

In [53]:
```
def f(x, y, z=1):
    tmp = x + y
    return tmp / z
```

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 [54]:
```
debug(f, 1, 2, z=3)
```

```
C:\Users\ankit19.gupta\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practic
e_Code\Python_Practice\Python_For_Data_Analysis\myenv\lib\site-packages\ipykernel_la
uncher.py:5: DeprecationWarning: The `color_scheme` argument is deprecated since ver
sion 5.1
  """
> <ipython-input-53-359ec13d6433>(2)f()
      1 def f(x, y, z=1):
----> 2     tmp = x + y
      3     return tmp / z

ipdb> c
```

Out[54]:
```
1.0
```

In [55]:
```
# 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 [56]:
```
# In [1]: %run -d examples/ipython_bug.py
# Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:1
# NOTE: Enter 'c' at the ipdb> prompt to start your script.
# > <string>(1)<module>()
# ipdb>
```

In [57]:
```
# Adding -b with a line number starts the debugger with a breakpoint set already:
```

In [58]:
```
# In [2]: %run -d -b2 examples/ipython_bug.py
# Breakpoint 1 at /home/wesm/code/pydata-book/examples/ipython_bug.py:2
# NOTE: Enter 'c' at the ipdb> prompt to start your script.
# > <string>(1)<module>()
# ipdb> c
# > /home/wesm/code/pydata-book/examples/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 meas- ure 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 com- plex 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 uninterest- ing boilerplate code:

In [63]:
```
import time
start = time.time()
for i in range(5):
    x=1
    # 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 600,000 strings and two identical methods of selecting only the ones that start with 'foo':

In [64]:
```
# 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 [65]:
```
%time method1 = [x for x in strings if x.startswith('foo')]
```

```
Wall time: 129 ms
```

In [66]:
```
%time method2 = [x for x in strings if x[:3] == 'foo']
```

```
Wall time: 72 ms
```

The Wall time (short for “wall-clock 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 more accurate average runtime:

In [67]:
```
%timeit [x for x in strings if x.startswith('foo')]
```

```
121 ms ± 25.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [68]:
```
%timeit [x for x in strings if x[:3] == 'foo']
```

```
37 ms ± 613 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

This seemingly innocuous example illustrates that it is worth understanding the per- formance characteristics of the Python standard library, NumPy, pandas, and other libraries used in this book. In larger-scale data analysis applications, those milli- seconds 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 (millionths of a second) or nano- seconds (billionths of a second). 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 preceding example, we could very directly compare the two string operations to understand their performance characteristics:

In [69]:
```
x = 'foobar'
y = 'foo'
```

In [70]:
```
%timeit x.startswith(y)
```

```
283 ns ± 45.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [71]:
```
%timeit x[:3] == y
```

```
130 ns ± 29.4 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Basic Profiling: %prun and %run -p

Profiling code is closely related to timing code, except it is concerned with determin- ing 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 that does some linear algebra in a loop (computing the maximum absolute eigenvalues of a series of 100 × 100 matrices):

You can run this script through cProfile using the following in the command line:

In [ ]:
```
# python -m cProfile cprof_example.py
```

If you try that, you'll find that the output is 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:

In [72]:
```
# $ python -m cProfile -s cumulativcprof_example.pypy
```

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 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 [74]:
```
# %prun -l 7 -s cumulative run_experiment()
```

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

In the Jupyter notebook, you can use the %%prun magic (two % signs) to profile an entire code block. This pops up a separate window with the profile output. This can be useful in getting possibly quick answers to questions like, "Why did that code block take so long to run?"

In [77]:
```
# %run -p -s cumulative cprof_example.py
```

There are other tools available that help make profiles easier to understand when you are using IPython or Jupyter. One of these is SnakeViz, which produces an interactive visualization of the profile results using d3.js.

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 inter- pret. 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:

In [79]:
```
# # A list of dotted module names of IPython extensions to load.
# c.TerminalIPythonApp.extensions = ['line_profiler']
```

You can also run the command:

In [81]:
```
! pip install line_profiler
```

```
Collecting line_profiler
  Downloading line_profiler-4.1.2-cp36-cp36m-win_amd64.whl (129 kB)
Installing collected packages: line-profiler
Successfully installed line-profiler-4.1.2
```

In [84]:
```
%load_ext 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 mod- ule prof_mod with the following code doing some NumPy array operations:

In [85]:
```
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 [86]:
```
%run prof_mod
x = randn(3000, 3000)
y = randn(3000, 3000)
%prun add_and_sum(x, y)
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in run(self, parameter_s, runner, file_finder)
    696             fpath = arg_lst[0]
--> 697             filename = file_finder(fpath)
    698         except IndexError:

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\utils\path.py in get_py_f
ilename(name, force_win32)
    108     else:
--> 109         raise IOError('File `%r` not found.' % name)
    110

OSError: File `'prof_mod.py'` not found.

During handling of the above exception, another exception occurred:

Exception                                 Traceback (most recent call last)
<ipython-input-86-1fc711dc3fdf> in <module>
----> 1 get_ipython().run_line_magic('run', 'prof_mod')
      2 x = randn(3000, 3000)
      3 y = randn(3000, 3000)
      4 get_ipython().run_line_magic('prun', 'add_and_sum(x, y)')

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\interactiveshell.py
in run_line_magic(self, magic_name, line, _stack_depth)
   2324                 kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325             with self.builtin_trap:
-> 2326                 result = fn(*args, **kwargs)
   2327             return result
   2328

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\decorator.py in fun(*args, **kw)
    230             if not kwsyntax:
    231                 args, kw = fix(args, kw, sig)
--> 232             return caller(func, *(extras + args), **kw)
    233     fun.__name__ = func.__name__
    234     fun.__doc__ = func.__doc__

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magic.py in <lambda>
(f, *a, **k)
    185     # but it's overkill for just that one bit of state.
    186     def magic_deco(arg):
--> 187         call = lambda f, *a, **k: f(*a, **k)
    188
    189         if callable(arg):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magics\execution.py
in run(self, parameter_s, runner, file_finder)
```

```
    706             if os.name == 'nt' and re.match(r"^'.*'$",fpath):
    707                 warn('For Windows, use double quotes to wrap a filename: %ru
n "mypath\\myfile.py"')
--> 708             raise Exception(msg)
    709         except TypeError:
    710             if fpath in sys.meta_path:

Exception: File `'prof_mod.py'` not found.
```

This is not especially enlightening. With the line_profiler IPython extension acti- vated, 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:

In [88]:
```
# %lprun -f func1 -f func2 statement_to_profile
```

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

In [89]:
```
%lprun -f add_and_sum add_and_sum(x, y)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-89-c740a2825365> in <module>
----> 1 get_ipython().run_line_magic('lprun', '-f add_and_sum add_and_sum(x, y)')

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\interactiveshell.py
in run_line_magic(self, magic_name, line, _stack_depth)
   2324                kwargs['local_ns'] = sys._getframe(stack_depth).f_locals
   2325            with self.builtin_trap:
-> 2326                result = fn(*args, **kwargs)
   2327            return result
   2328

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\decorator.py in fun(*args, **kw)
    230            if not kwsyntax:
    231                args, kw = fix(args, kw, sig)
--> 232            return caller(func, *(extras + args), **kw)
    233    fun.__name__ = func.__name__
    234    fun.__doc__ = func.__doc__

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\IPython\core\magic.py in <lambda>
(f, *a, **k)
    185    # but it's overkill for just that one bit of state.
    186    def magic_deco(arg):
--> 187        call = lambda f, *a, **k: f(*a, **k)
    188
    189        if callable(arg):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\line_profiler\ipython_extension.p
y in lprun(self, parameter_s)
    128        try:
    129            try:
--> 130                profile.runctx(arg_str, global_ns, local_ns)
    131                message = ""
    132            except SystemExit:

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\line_profiler\line_profiler.py in
runctx(self, cmd, globals, locals)
    183        self.enable_by_count()
    184        try:
--> 185            exec(cmd, globals, locals)
    186        finally:
    187            self.disable_by_count()

<string> in <module>

<ipython-input-85-0355268a1452> in add_and_sum(x, y)
      2 def add_and_sum(x, y):
      3     added = x + y
----> 4     summed = added.sum(axis=1)
      5     return summed
```

```
      6 def call_function():

AttributeError: 'str' object has no attribute 'sum'
```

This can be much easier to interpret. In this case we profiled the same function we used in the statement. Looking at the preceding module code, 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 [90]:
```
# %lprun -f add_and_sum -f call_function call_function()
```

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 must explicitly specify the names of the func- tions you want to profile with %lprun is that the overhead of “trac- ing” the execution time of each line is substantial. Tracing functions that are not of interest has the potential to significantly alter the profile results.

B.4 Tips for Productive Code Development Using IPython

Writing code in a way that makes it easy to develop, debug, and ultimately use inter- actively 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.

Therefore, implementing most of the strategies described in 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 for you. Ultimately you want to struc- ture 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 IPython in mind to be easier to work with than code intended only to be run as as 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 IPython code development 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

In [92]:
```
# 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.py because of Python's "load-once" module system. This behavior differs from some other data analysis environments, like MATLAB, which automatically propagate code changes.1 To cope with this, you have a couple of options. The first way is to use the reload function in the importlib module in the standard library:

In [93]:
```
# import some_lib
# import importlib
# importlib.reload(some_lib)
```

This guarantees that you will get a fresh copy of some_lib.py 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 some_lib.py 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, unfortu- nately, 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.

Keep relevant objects and data alive

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

In [95]:
```
# 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 go wrong if we were to run this program 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. This is equivalent to defining toplevel variables in cells in the Jupyter notebook.

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 eas- ier 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. How- ever, while developing code using IPython, working with 10 small but interconnected files (under, say, 100 lines each) is likely to cause you more headaches in general than 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 to be to put all of your code in a single monstrous file. Finding a sensible and intuitive mod- ule and package structure for a large codebase often takes a bit of work, but it is espe- cially 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

B.5 Advanced IPython Features

Making full use of the IPython system may lead you to write your code in a slightly different way, or to dig into the configuration.

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:

In [96]:
```
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 [97]:
```
x = Message('I have a secret')
x
```

Out[97]:
```
<__main__.Message at 0x26421f7e6a0>
```

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 preceding class to get a more helpful output:

In [98]:
```
class Message:
    def __init__(self, msg):
        self.msg = msg
    def __repr__(self):
        return 'Message: %s' % self.msg
```

In [99]:
```
x = Message('I have a secret')
x
```

Out[99]:
```
Message: I have a secret
```

Profiles and Configuration

Most aspects of the appearance (colors, prompt, spacing between lines, etc.) and behavior of the IPython and Jupyter environments are configurable through an extensive configuration system. Here are some 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

• Execute an arbitrary list of Python statements (e.g., imports that you use all the time or anything else you want to happen each time you launch IPython)

• Enable always-on IPython extensions, like the %lprun magic in line_profiler

• Enabling Jupyter extensions

• Define your own magics or system aliases

Configurations for the IPython shell are specified in special ipython_config.py files, which are usually found in the .ipython/ directory in your user home directory. Con- figuration is performed based on a particular profile. When you start IPython nor- mally, 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:

In [100…
```
# /home/wesm/.ipython/profile_default/ipython_config.py
```

To initialize this file on your system, run in the terminal:

In [101…
```
# ipython profile create
```

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 multi- ple profiles. Suppose you wanted to have an alternative IPython configuration tailored for a particular application or project. Creating a new profile is as simple as typing something like the following:

In [102…
```
# ipython profile create secret_project
```

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

In [103…
```
# $ ipython --profile=secret_project
# Python 3.5.1 | packaged by conda-forge | (default, May 20 2016, 05:22:56)
# Type "copyright", "credits" or "license" for more information.
# IPython 5.1.0 -- 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
```

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

Configuration for Jupyter works a little differently because you can use its notebooks with languages other than Python. To create an analogous Jupyter config file, run:

In [104…
```
# jupyter notebook --generate-config
```

This writes a default config file to the .jupyter/jupyter_notebook_config.py directory in your home directory. After editing this to suit your needs, you may rename it to a different file, like:

In [105…
```
# $ mv ~/.jupyter/jupyter_notebook_config.py ~/.jupyter/my_custom_config.py
```

When launching Jupyter, you can then add the --config argument:

In [106…
```
# jupyter notebook --config=~/.jupyter/my_custom_config.py
```

In [107…
```
# https://nbviewer.org/
```

In [ ]:
```
```