**IPython**

1. Installed the ipython, By writing following code in cmd:

**Pip install ipython**

1. Installed the jupyter by writing following code in the cmd:

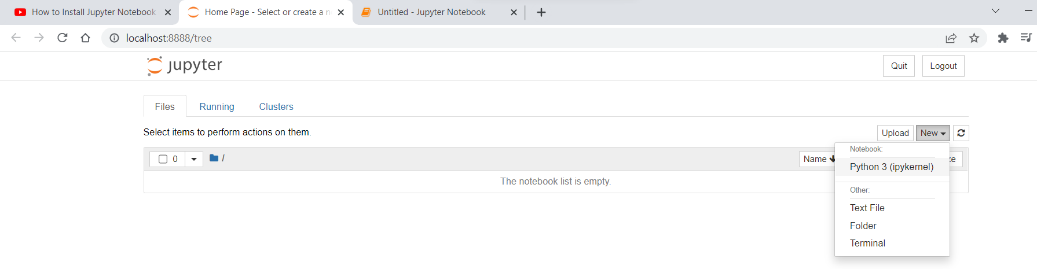
**Pip install jupyter**

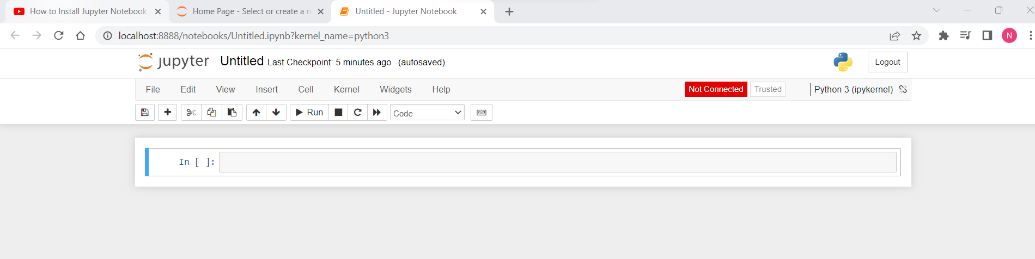
1. Than open the cmd in an folder where we want to use the jupyter notebook, And in it write the following code:

**jupyter notebook**

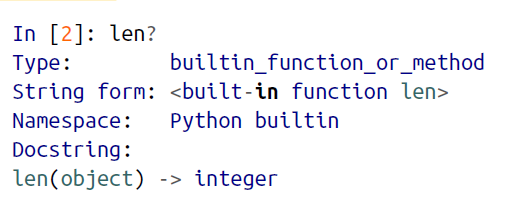
1. This command will launch a local web server that will be visible to your browser. It immediately spits out a log showing what it is doing and it will be working on the http://localhost:8888/

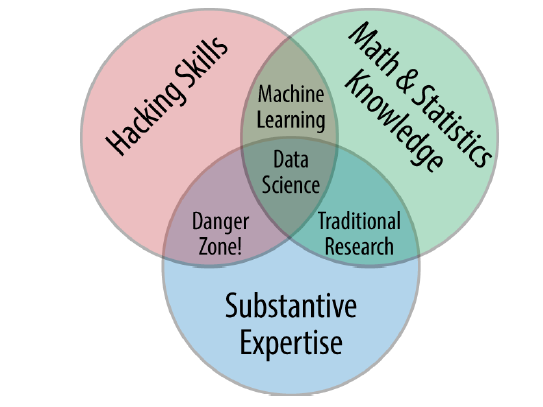
🡪Example:





1. In jupyter notebook, Here we’ll discuss IPython’s tools to quickly access this information, namely the **? character** to explore documentation, the **?? characters** to explore source code, and the **Tab key** for autocompletion.



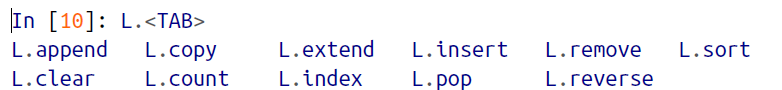
1. 
2. Python, IPython also provides a number of useful syntactic additions to the language
3. Exploring Modules with Tab Completion

🡪IPython’s other useful interface is the use of the Tab key for autocompletion and

exploration of the contents of objects, modules, and namespaces. In the examples that

follow, we’ll use <TAB> to indicate when the Tab key should be pressed.

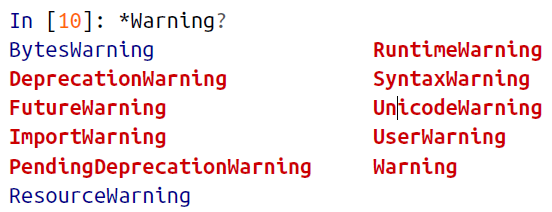
1. To see a list of all available attributes of an object, you can type the name of the object followed by a period (.) character and the Tab key:



1. Beyond tab completion: Wildcard matching

🡪Tab completion is useful if you know the first few characters of the object or attribute you’re looking for, but is little help if you’d like to match characters at the middle or end of the word. For this use case, IPython provides a means of wildcard matching for names using the \* character. For example, we can use this to list every object in the namespace that ends with

Warning:



🡪Notice that the \* character matches any string, including the empty string.

1. 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.

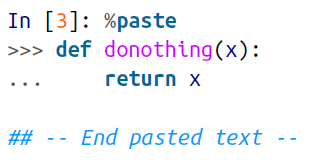
🡪The code is formatted as it would appear in the Python interpreter, and if you copy

and paste this directly into IPython you get an error:

🡪In 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:



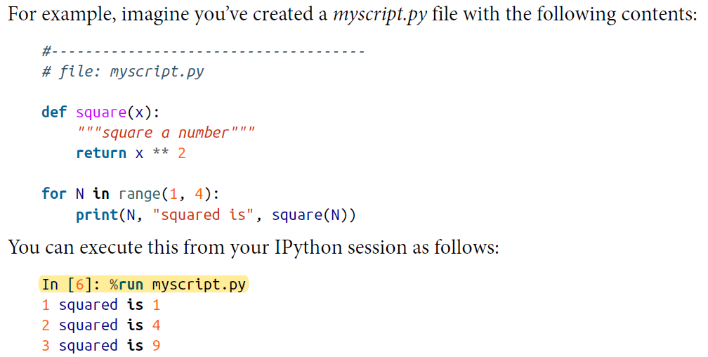
🡪A command with a similar intent is %cpaste, which opens up an interactive multiline

prompt in which you can paste one or more chunks of code to be executed in a batch

1. Running External Code: %run

🡪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:



1. 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:



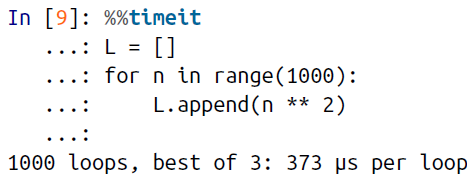
🡪The benefit of %timeit is that for short commands it will automatically perform multiple

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:



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

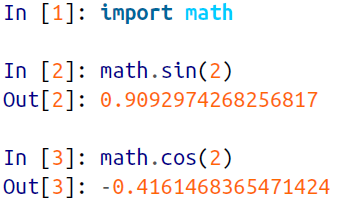
In [12]: %**lsmagic**

1. IPython’s In and Out Objects

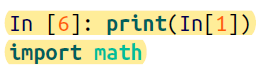
🡪The In[1]:/Out[1]: style prompts used by IPython.

🡪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. Imagine

you start a session that looks like this:



1. 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):



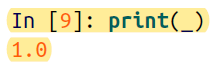
🡪Note that not all operations have outputs: for example, import statements and print

statements don’t affect the output.

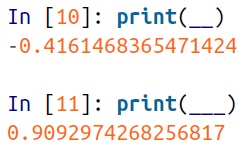
1. Underscore Shortcuts and Previous Outputs

🡪The standard Python shell contains just one simple shortcut for accessing previous

output; the variable \_ (i.e., a single underscore) is kept updated with the previous output; this works in IPython as well:



🡪But IPython takes this a bit further—you can use a double underscore to access the second-to-last output, and a triple underscore to access the third-to-last output (skipping any commands with no output):



1. Suppressing Output

🡪The easiest way to suppress the output of a command is to add a semicolon

to the end of the line:



1. Related Magic Commands

🡪Some magic commands are %rerun (which will re-execute some

portion of the command history) and %save (which saves some set of the command

history to a file)

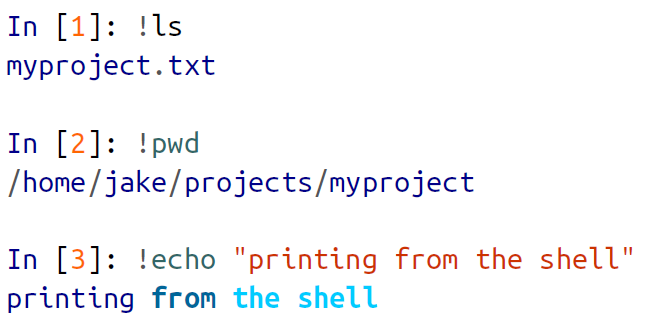
1. IPython and Shell Commands

🡪When working interactively with the standard Python interpreter, one of the frustrations 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 syntax for executing shell commands directly from within the IPython terminal.

1. Shell Commands in IPython

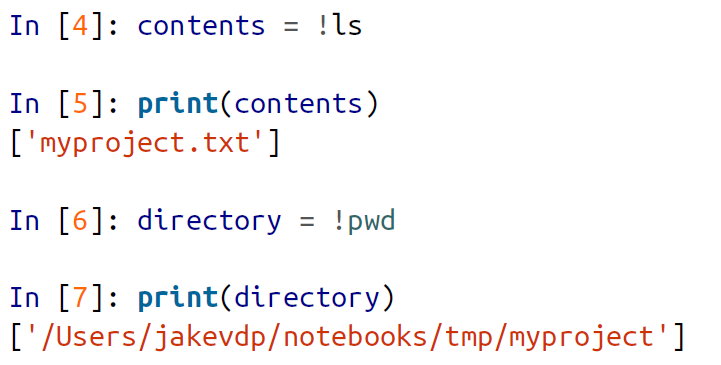
🡪You can use any command that works at the command line in IPython by prefixing it with the ! character. For example, the ls, pwd, and echo commands can be run as

follows:



1. Passing Values to and from the Shell

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



1. Errors and Debugging

🡪Code development and data analysis always require a bit of trial and error, and

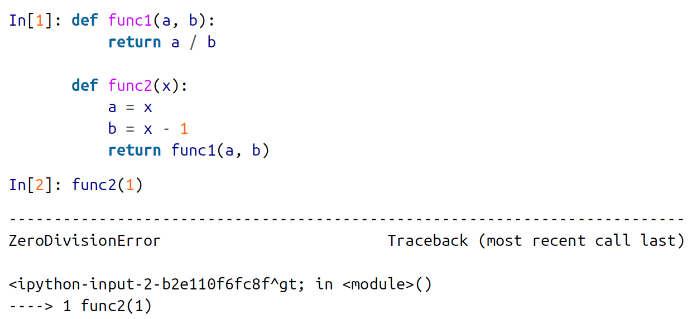
IPython contains tools to streamline this process. This section will briefly cover some

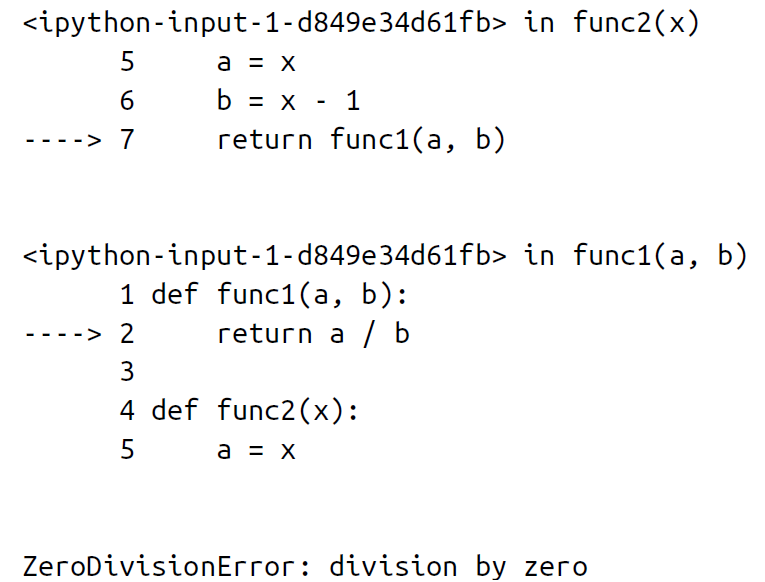
options for controlling Python’s exception reporting, followed by exploring tools for

debugging errors in code.

1. Controlling Exceptions: %xmode

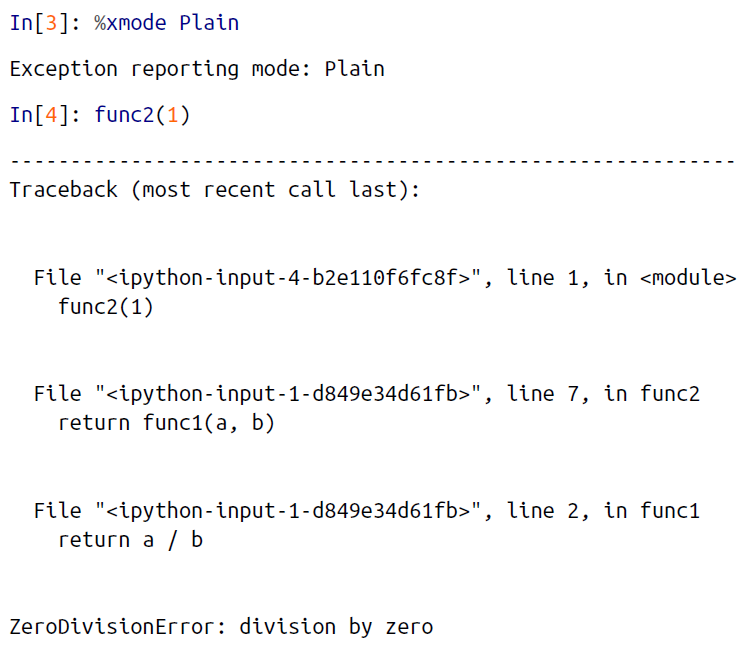
🡪Most of the time when a Python script fails, it will raise an exception. When the interpreter hits one of these exceptions, information about the cause of the error can be found in the *traceback*, which can be accessed from within Python. With the %xmode magic function, IPython allows you to control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



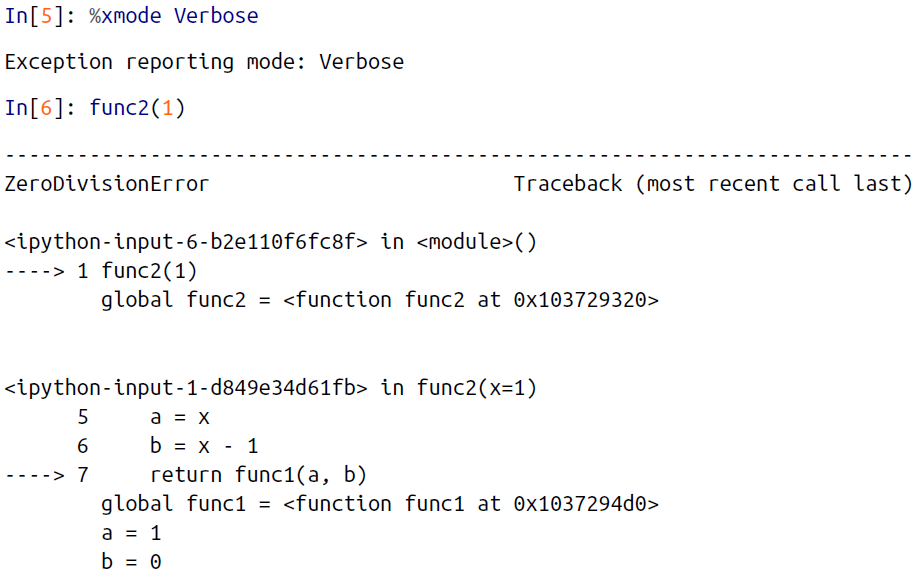


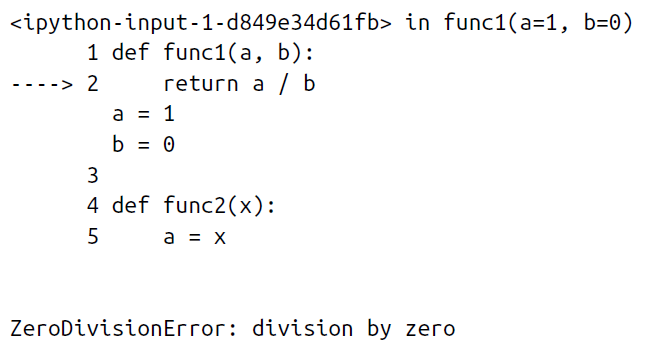
🡪Calling func2 results in an error, and reading the printed trace lets us see exactly what happened. By default, this trace includes several lines showing the context of each step that led to the error. Using the %xmode magic function (short for *exception mode*), we can change what information is printed.

🡪%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:



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



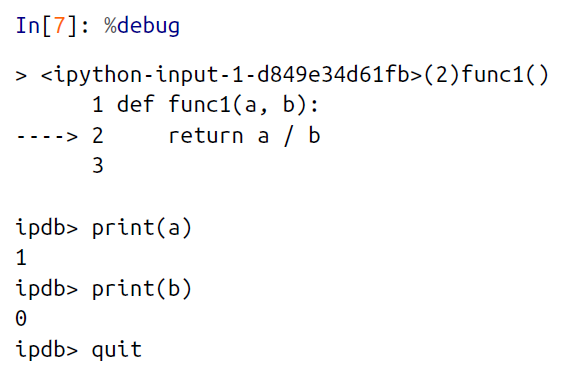


🡪This extra information can help you narrow in on why the exception is being raised. So why not use the Verbose mode all the time? As code gets complicated, this kind of traceback can get extremely long. Depending on the context, sometimes the brevity of Default mode is easier to work with.

\*Here, brevity means the state of being sort

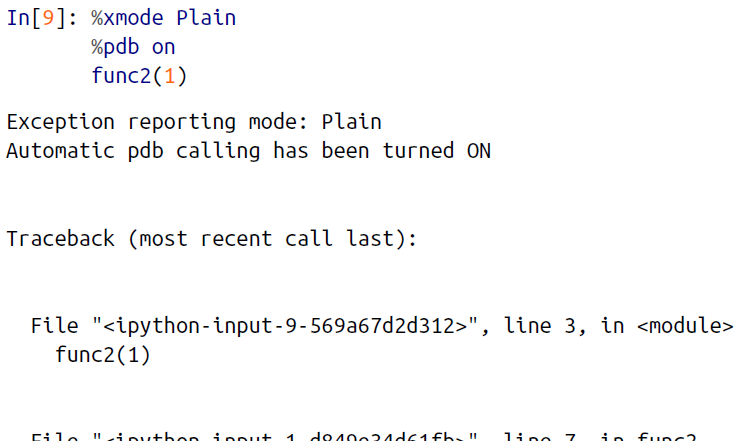
1. Debugging: When Reading Tracebacks Is Not Enough

🡪The standard Python tool for interactive debugging is pdb, the Python debugger. This debugger lets the user step through the code line by line in order to see what might be causing a more difficult error. The IPython-enhanced version of this is ipdb, the IPython debugger.



🡪The interactive debugger allows much more than this, though—we can even step up and down through the stack and explore the values of variables there:

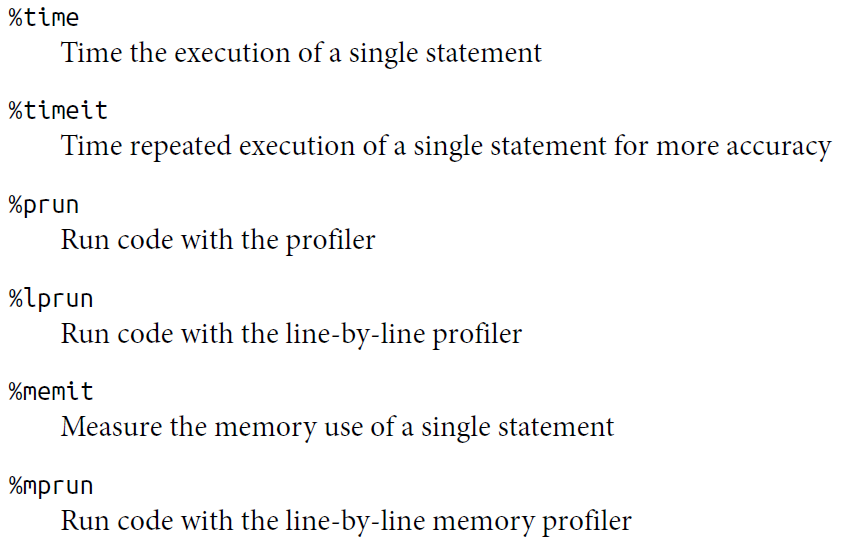
🡪If you’d like the debugger to launch automatically whenever an exception is raised, you can use the %pdb magic function to turn on this automatic behavior:



1. 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 commands; 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:

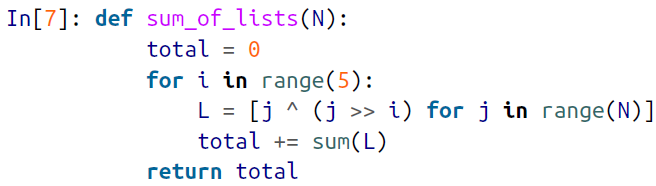


🡪The last four commands are not bundled with IPython—you’ll need to install the line\_profiler and memory\_profiler extensions, which we will discuss in the following sections.

1. Profiling Full Scripts: %prun

🡪A program is made of many single statements, and sometimes timing these statements 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:



🡪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

🡪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.

1. Profiling Memory Use: %memit and %mprun

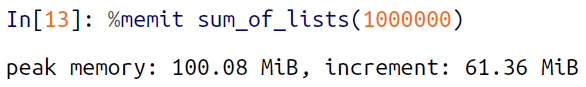
🡪Another aspect of profiling is the amount of memory an operation uses. This can be evaluated with another IPython extension, the memory\_profiler. As with the line\_profiler, we start by pip-installing the extension:

**$ pip install memory\_profiler**

🡪Then we can use IPython to load the extension:

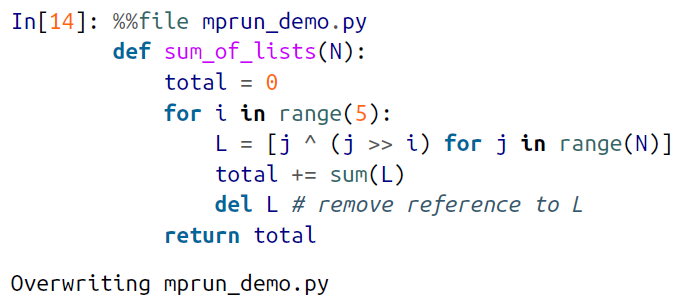
**In[12]: %load\_ext memory\_profiler**

🡪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 function can be used rather simply:



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

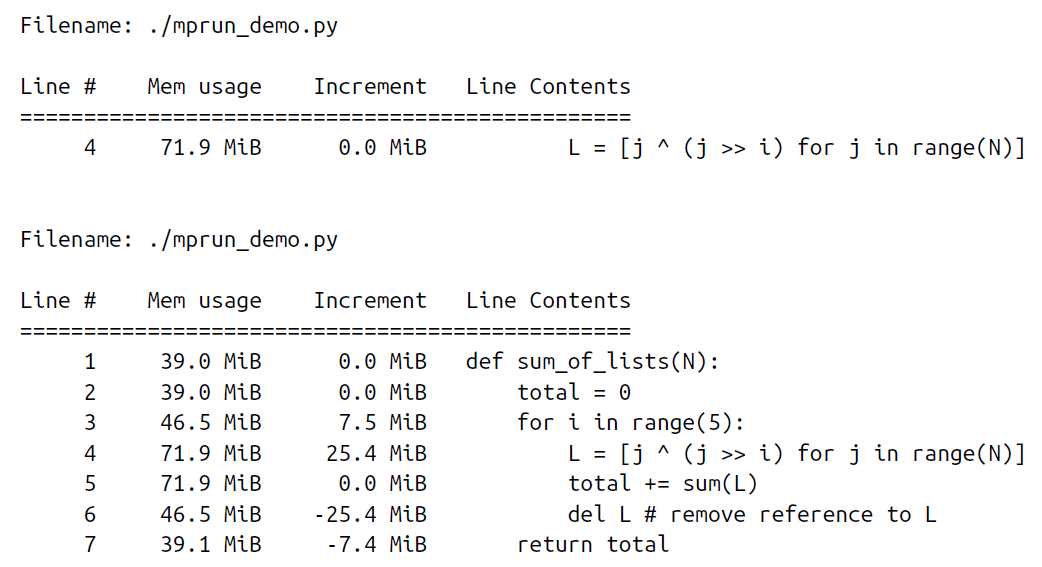
🡪For a line-by-line description of memory use, we can use the %mprun magic. 🡪Unfortunately, this magic works only for functions defined in separate modules rather than the notebook itself, so we’ll start by using the %%file magic to create a simple module called mprun\_demo.py, which contains our sum\_of\_lists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



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



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



🡪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).

🡪this chapter, we’ve just scratched the surface of using IPython to enable data science 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.