

01.05-IPython-And-Shell-Commands

November 10, 2024

1 IPython and Shell Commands

When working interactively with the standard Python interpreter, one of the frustrations 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. 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 discussion assumes you're on a Unix-like system, such as Linux or macOS. Some of the examples that follow will fail on Windows, which uses a different type of shell by default, though if you use the *Windows Subsystem for Linux* the examples here should run correctly. If you're unfamiliar with shell commands, I'd suggest reviewing the [Unix shell tutorial](#) put together by the always excellent Software Carpentry Foundation.

1.1 Quick Introduction to the Shell

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

Someone unfamiliar with the shell might ask why you would bother with this, when many of the same results can be accomplished by simply clicking on icons and menus. A shell user might reply with another question: why hunt for icons and menu items 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 be intimidating.

As an example, here is a sample of a Linux/macOS shell session where a user explores, creates, and modifies directories and files on their system (`osx:~` `$` is the prompt, and everything after the `$` 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 sitting 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_book  mpld3  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. With just a few commands (`pwd`, `ls`, `cd`, `mkdir`, and `cp`) you can do many of the most common file operations, but it's when you go beyond these basics that the shell approach becomes really powerful.

1.2 Shell Commands in IPython

Any standard shell command can be used directly 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

```

1.3 Passing Values to and from the Shell

Shell commands not only can 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, =:

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

```
In [5]: print(contents)
['myproject.txt']
```

```
In [6]: directory = !pwd
```

```
In [7]: print(directory)
['/Users/jakevdp/notebooks/tmp/myproject']
```

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 using 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 contents in the shell command.

1.4 Shell-Related Magic Commands

If you play with IPython’s shell commands for a while, you might notice that you cannot 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 subshell that does not maintain state from command to command. 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 the ability to execute such commands without an explicit % 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 session lets you more naturally combine Python and the shell in your workflows with fewer context switches.