A PyCon Notebook

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009) and *Python Cookbook* (3rd Edition, O'Reilly & Associates, 2013). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply/index.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

As I begin to write this, I'm returning on the plane from PyCon 2013, held March 13-17 in Santa Clara, California. When I started using Python some 16 years ago, the Python conference was an intimate affair involving around a hundred people. This year's conference featured more than 2,500 attendees and 167 sponsors—bigger than ever for an event that's still organized by the community (full disclaimer, I was also one of the sponsors). If you couldn't attend, video and slidedecks for virtually every talk and tutorial can be found online at http://pyvideo.org and https://speakerdeck.com/pyconslides.

There are any number of notable things I could discuss about the conference, such as the fact that everyone received a Raspberry Pi computer, there were programming labs for kids, or the record-setting conference attendance by women; however, in this article I'm primarily going to focus on the one project that seems to be taking over the Python universe—namely, the IPython Notebook project.

If you attend any Python conference these days, you'll quickly notice the widespread use of the IPython Notebook (http://ipython.org) for teaching, demonstrations, and day-to-day programming. What is the notebook and why are so many people using it, you ask? Let's dive in.

The IPython Shell

Before getting to the notebook, knowing about the more general IPython project that has evolved over the past ten years will help. In a nutshell, IPython is an alternative interactive shell for Python that provides a broad range of enhancements, such as better help features, tab completion of methods and file names, the ability to perform shell commands easily, better command history support, and more. Originally developed to support scientists and engineers, IPython is intended to provide a useful environment for exploring data and performing experiments. Think of it as a combination of the UNIX shell and interactive Python interpreter on steroids.

To provide a small taste of what IPython looks like, here is a sample session that mixes Python and shell commands together to determine how much disk space is used by different types of files in the current working directory:

```
bash-3.2$ ipython
Python 2.7.3 (default, Dec 10 2012, 06:24:09)
Type "copyright", "credits" or "license" for more information.

IPython 0.13.1 -- 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]: cd ~
/Users/beazley
```

```
In [2]: ls
Desktop/     Junk/      Music/     Public/     Tools/
Documents/   Library/   Pictures/  Sites/
Downloads/   Movies/    Projects/  Teaching/

In [3]: cd Pictures
/Users/beazley/Pictures

In [4]: import collections

In [5]: import os

In [6]: size_by_type = collections.Counter()

In [7]: for path, dirs, files in os.walk('.'):
   ...:     for filename in files:
   ...:         fullname = os.path.join(path, filename)
   ...:         if os.path.exists(fullname):
   ...:             _, ext = os.path.splitext(filename)
   ...:             sz = os.path.getsize(fullname)
   ...:             size_by_type[ext.upper()] += sz
   ...:

In [8]: for ext, sz in size_by_type.most_common(5):
   ...:     print ext, sz
   ...:
.JPG 50389086278
.MOV 38328837384
.AVI 9740373284
.APDB 733642752
.DATA 518045719

In [9]:
```

As you can see, a mix of UNIX shell commands and Python statements appear. The "In [*n*]:" prompt is the interpreter prompt at which you type commands. This prompt serves an important purpose in maintaining a history of your work. For example, if you wanted to redo a previous sequence of commands, you could use rerun to specify a range of previous operations like this:

```
In [9]: cd ../Music
/Users/beazley/Music

In [10]: rerun 6-8
=== Executing: ===
size_by_type = collections.Counter()
for path, dirs, files in os.walk('.'):
    for filename in files:
        fullname = os.path.join(path, filename)
        if os.path.exists(fullname):
            _, ext = os.path.splitext(filename)
            sz = os.path.getsize(fullname)
            size_by_type[ext.upper()] += sz

for ext, sz in size_by_type.most_common(5):
    print ext, sz

=== Output: ===
.M4A 9704243754
.MP3 2849783536
.M4P 2841844039
.M4V 744062510
.MP4 573729448

In [11]:
```

Or, if you wanted to save your commands to a file for later editing, you could use the save command like this:

```
In [11]: cd ~
/Users/beazley


In [12]: save usage.py 4-8
The following commands were written to file `usage.py`:
import collections
import os
size_by_type = collections.Counter()
for path, dirs, files in os.walk('.'):
    for filename in files:
        fullname = os.path.join(path, filename)
        if os.path.exists(fullname):
            _, ext = os.path.splitext(filename)
            sz = os.path.getsize(fullname)
            size_by_type[ext.upper()] += sz

for ext, sz in size_by_type.most_common(5):
    print ext, sz

In [13]:
```

Should you be inclined to carry out more sophisticated shell operations, you can usually execute arbitrary commands by prefixing them with the exclamation point and refer to Python variables using $ variable substitutions. For example:

```
In [13]: pid = os.getpid()

In [14]: !lsof -p$pid
COMMAND PID   USER    FD   TYPE DEVICE  SIZE/OFF       NODE NAME
Python  8686 beazley cwd  DIR 14,2      238 2805734    /Users/...
Python  8686 beazley txt  REG 14,2      12396 2514070  /Library/...
...

In [15]:
```

You can capture the output of a shell command by simply assigning it to a variable:

```
In [15]: out = !lsof -p$pid -F n

In [16]: out
Out[16]:
['p8686',
 'n/Users/beazley/Desktop/UsenixLogin/beazley_jun_13',
```

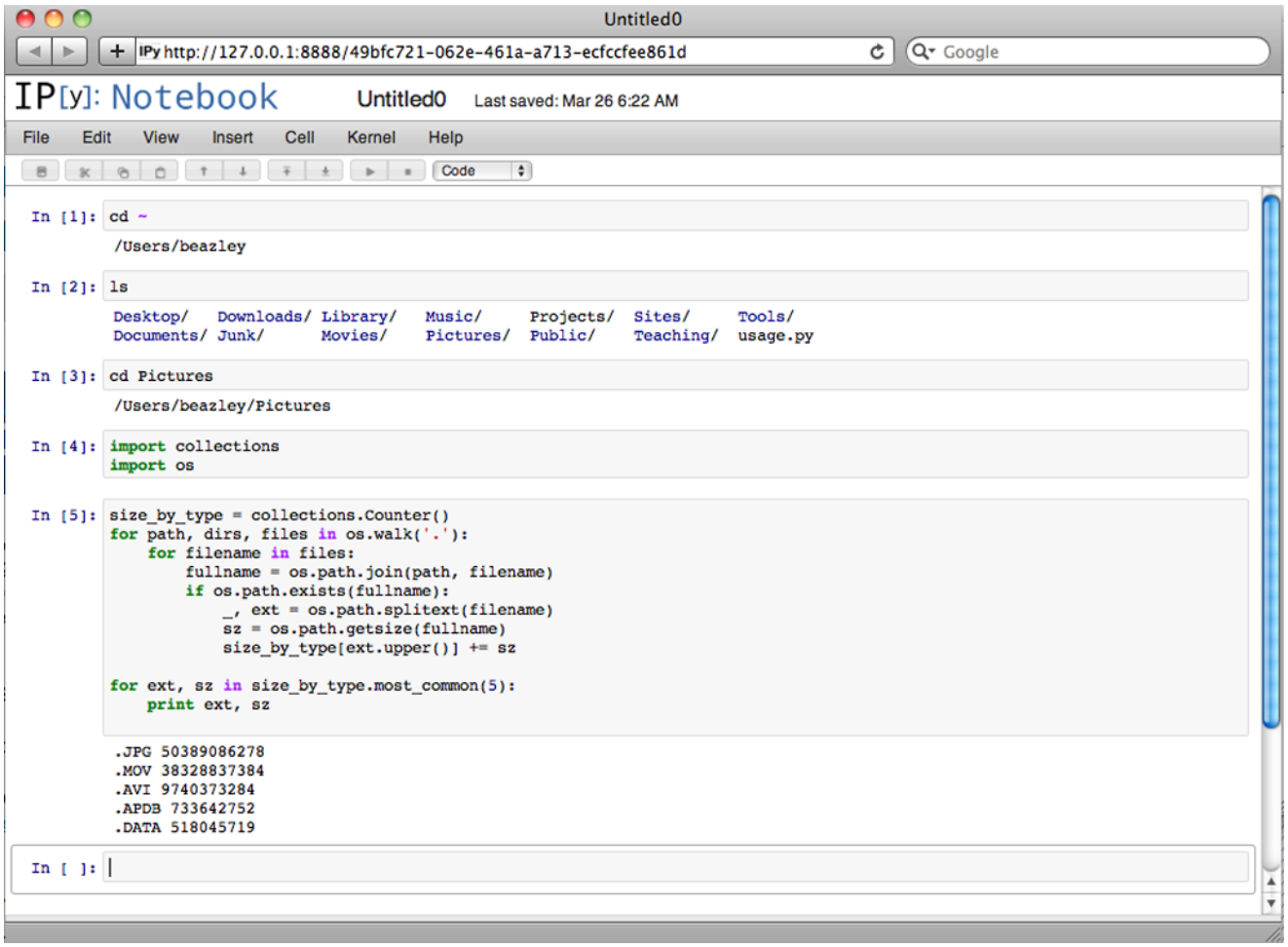

Figure 1: Notebook works with IPython, and at first appears not that different from using IPython alone

```
 'n/Library/Frameworks/Python.framework/Versions/7.3/Python',
 ...
]

In [17]:
```

This session gives you a small glimpse of what IPython is about and why you might use it; however, this is not the IPython Notebook.

From the Shell to the Notebook

Imagine, if you will, the idea of taking the above shell session and turning it into a kind of interactive document featuring executable code cells, documentation, inline plots, and arbitrary Web content (images, maps, videos, etc.). Think of the document as the kind of content you might see written down in a scientist's lab notebook. Well, that is basically the idea of the IPython Notebook project. Conveying the spectacle it provides in print is a little hard, so a good place to start might be some of the videos at http://pyvideo.org.

To get started with the IPython notebook yourself, you'll need to spend a fair bit of time fiddling with your Python installation. There are a number of required dependencies, including pyzmq (https://pypi.python.org/pypi/pyzmq/) and Tornado (https://pypi.python.org/pypi/tornado). Additionally, to realize all of the IPython notebook benefits, you'll need to install a variety of scientific packages, including NumPy (http://numpy.org) and matplotlib (http://matplotlib.org). Frankly, working with a Python distribution in which it's already included, such as EPDFree (http://www.enthought.com/products/epd_free.php) or Anaconda CE (http://continuum.io/anacondace.html), is probably easier. If you're on Linux, you might be able to install the required packages using the system package manager, although your mileage might vary.

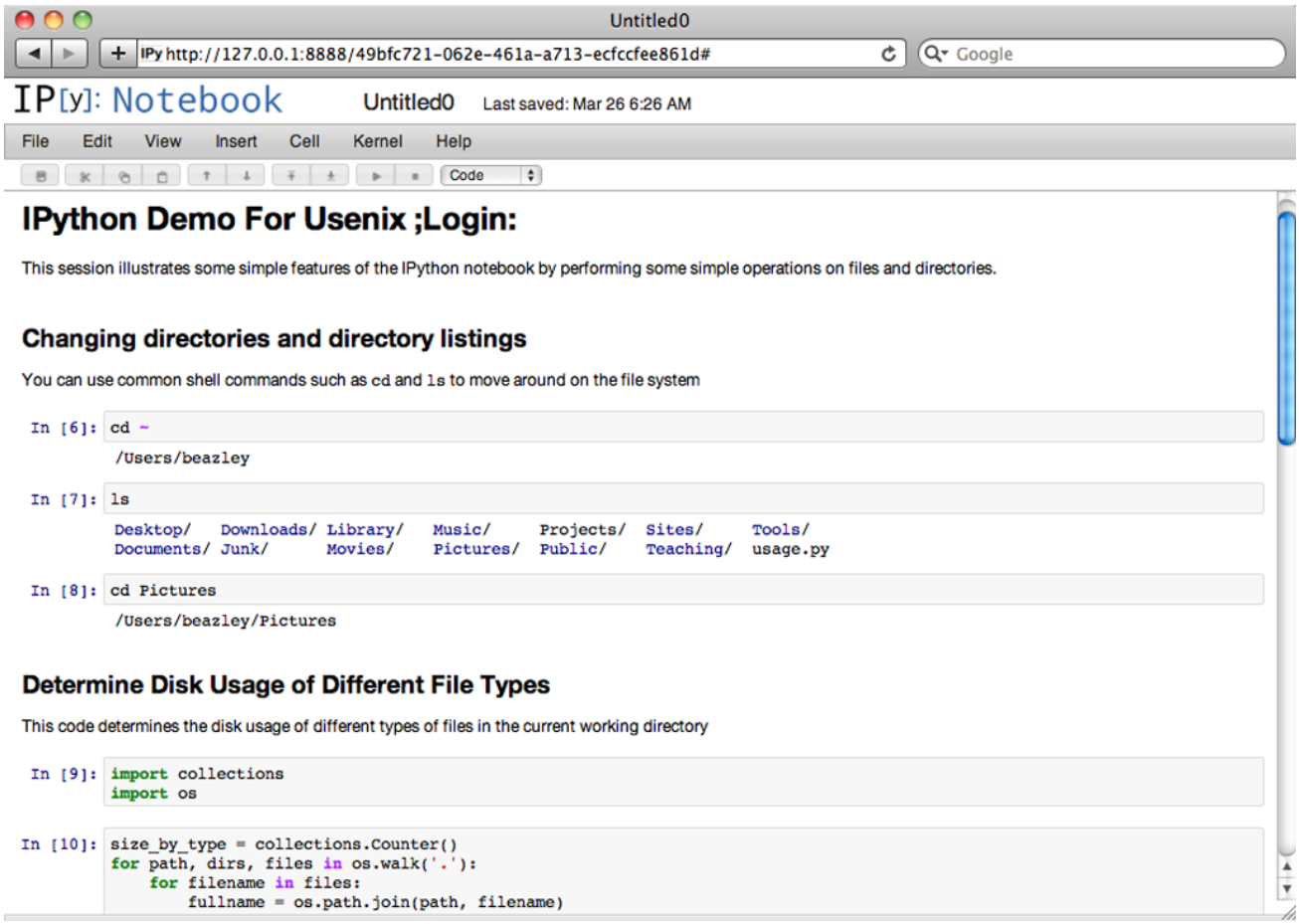

Figure 2: Notebook includes the ability to document what appears in a notebook, using Markdown (https://pypi.python.org/pypi/Markdown)

Assuming you have everything installed, you can launch the notebook from the shell. Go to the directory in which you want to do your work and type "ipython notebook". For example:

```
bash $ ipython notebook
[NotebookApp] Using existing profile dir: u'/Users/beazley/.
   ipython/profile_default'
[NotebookApp] Serving notebooks from /Users/beazley/Work
[NotebookApp] The IPython Notebook is running at:
   http://127.0.0.1:8888/
[NotebookApp] Use Control-C to stop this server and shut down
   all kernels.
```

Unlike a normal session, the Notebook runs entirely as a server that needs to be accessed through a browser. As soon as you launch it, a browser window like the one in Figure 1 should appear.

If you click on the link to create a new notebook, you'll be taken to a page on which you can start typing the usual IPython commands, as in Figure 1.

At this point, the notebook doesn't seem much different from the shell; however, the benefits start to appear once you start editing the document. For example, unlike the shell, you can move around and edit any of the previous cells (e.g., change the code, re-execute, delete, copy, and move around within the document). You can also start to insert documentation at any point in the form of Markdown. Figure 2 shows the above session annotated with some documentation.

Assuming you've installed matplotlib and NumPy, you can also start making inline plots and charts. For example, Figure 3 shows what it looks like to take the file-usage data and make a pie chart.

Needless to say, the idea of having your work captured inside a kind of executable document opens up a wide range of possibilities limited only by your imagination. Once you realize that these notebooks can be saved, modified, and shared with others, why the notebook project is quickly taking over the Python universe starts to become clear. In that vein, I've shared the above

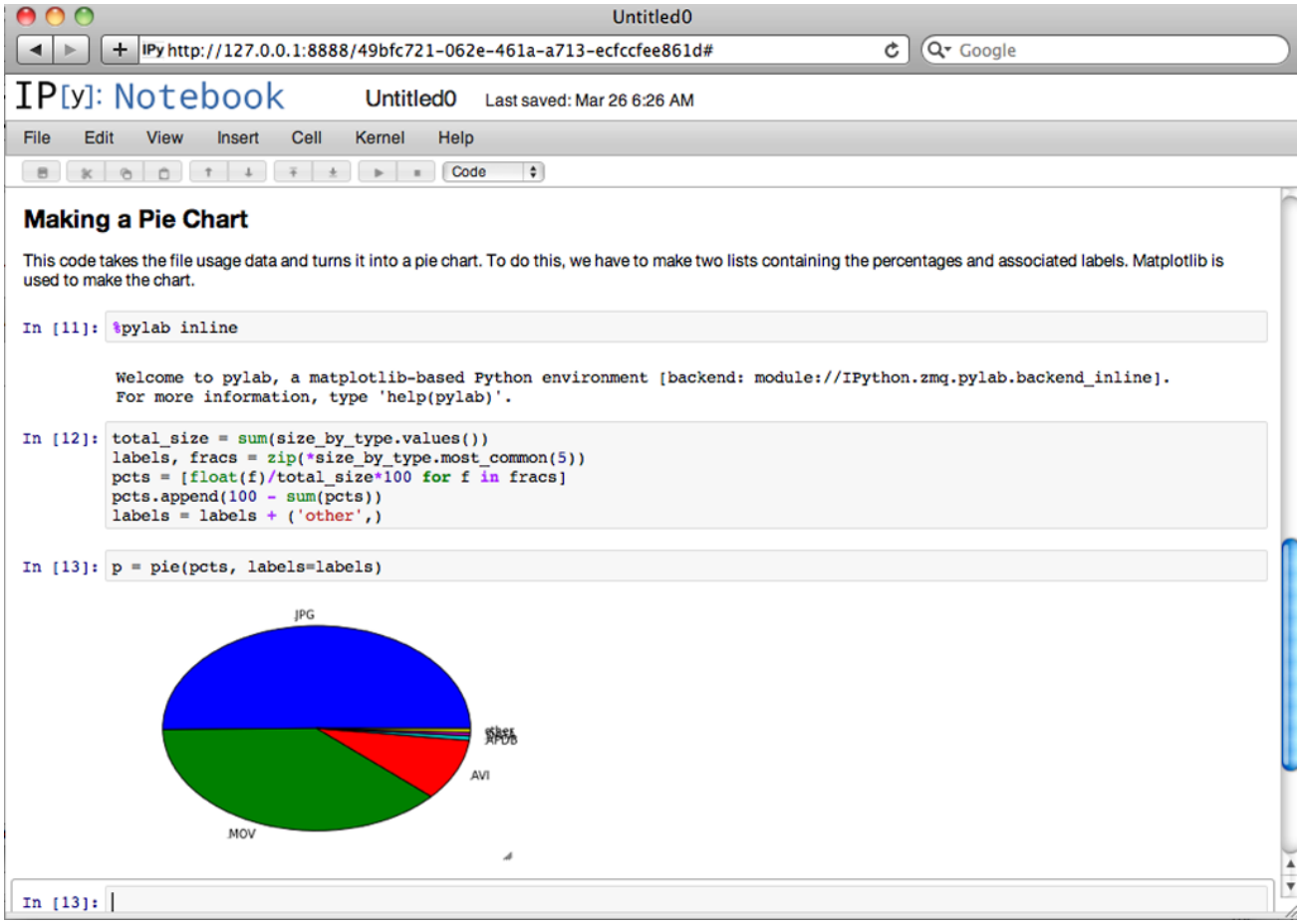

Figure 3: Notebook works with matplotlib and NumPy so you can include inline plots and charts

notebook at http://nbviewer.ipython.org/5244469. You can go there to view it in more detail.

Other Notable PyCon Developments

Although this article has primarily focused on IPython, a few other notable developments were featured at the recent conference. First, there seems to be a general consensus that the mechanism currently used to install third-party packages (the procedure of typing `python setup.py install`) should probably die. How that actually happens is not so clear, but the topic of packaging is definitely on a lot of people's minds. Somewhat recently, a new binary packaging format known as a "wheel file" appeared and is described in PEP-427 (http://www.python.org/dev/peps/pep-0427/). Although I have yet to encounter wheels in the wild, it's something that you might encounter down the road, especially if you're the one maintaining a Python Installation.

Also worthy of note is the fact that Python seems to be gaining a standard event loop. Over the past several years, there has been growing interest in asynchronous and event-driven I/O libraries (e.g., Twisted, Tornado, GEvent, Eventlet, etc.) for network programming. One of the benefits of such libraries is that they are able to handle a large number of client connections, without relying on threads or separate processes. Although the standard library has long included the asyncore library for asynchronous I/O, nobody has ever been all that satisfied with it; in fact, most people seem to avoid it.

Guido van Rossum's keynote talk at PyCon went into some depth about PEP 3156 (http://www.python.org/dev/peps/pep-3156/), which is a new effort to put a standard event loop into the standard library. Although one wouldn't think that an event loop would be that exciting, it's interesting in that it aims to standardize a feature that is currently being implemented separately by many different libraries that don't currently interoperate with each other so well. This effort is also notable in that the PEP involves the use of co-routines and requires Python 3.3 or newer. Could asynchronous I/O be the killer feature that brings everyone to Python 3? Only time will tell.