



Head First Python, 2nd Edition

PREV  
C. Top Ten Things we Didn't Cover: There's Always More

AA

NEXT  
E. Getting Involved: The Python Community

# Appendix D. Top Ten Projects not Covered: Even More Tools, Libraries, and Modules



We know what you're thinking as you read this appendix's title.

Why on Earth didn't they make the title of the last appendix: *The Top Twenty Things We Didn't Cover?* Why *another* 10? In the last appendix, we limited our discussion to stuff that comes baked in to Python (part of the language's "batteries included"). In this appendix, we cast the net much further afield, discussing a whole host of technologies that are available to you *because* Python exists. There's lots of good stuff here and—just like with the last appendix—a quick perusal won't hurt you *one single bit*.

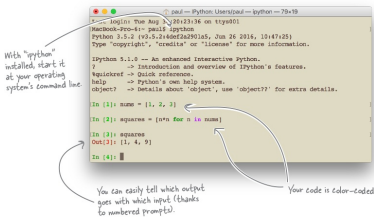
**1. Alternatives to >>>**

Throughout this book we've happily worked at Python's built-in >>> prompt, either from within a terminal window or from within IDLE. In doing so, we hope we've demonstrated just how effective using the >>> prompt can be when you're experimenting with ideas, exploring libraries, and trying out code.

There are lots of alternatives to the built-in >>> prompt, but the one that gets the most attention is called `ipython`, and if you find yourself wishing you could do more at the >>> prompt, `ipython` is worth a look. It is very popular with many Python programmers, but is *especially* popular within the scientific community.

To give you an idea of what `ipython` can do compared to the plain ol' >>> prompt, consider this short interactive `ipython` session:

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)



Find out more about `ipython` at <https://ipython.org>.

As with all third-party modules, you can use “pip” to install both “ipython” and “ptpython”.

There are other `>>>` alternatives, but the only other one that’s a match (in our view) for what `ipython` has to offer is `ptpython` (more information can be found here: <https://pypi.org/project/ptpython/>). If you like working within a text-based terminal window, but are looking for something a bit more “full screen” than `ipython`, take a look at `ptpython`. You won’t be disappointed.

NOTE

Pssst! Since discovering “ptpython”, Paul has used it every day.

2. Alternatives to IDLE

We’re not afraid to state this: we have a soft spot for *IDLE*. We really like the fact that Python not only comes with a capable `>>>` prompt, but also ships with a passable cross-platform GUI-based editor and debugger. There are few other mainstream programming languages that provide anything similar as part of their default install.

Regrettably, *IDLE* gets a fair amount of flack in the Python community, as it stacks up poorly against some of the more capable “professional” offerings. We think this is an *unfair* comparison, as *IDLE* was never designed to compete in that space. *IDLE*’s main goal is to get new users up and going as quickly as possible, and it does this *in spades*. Consequently, we feel *IDLE* should be celebrated more in the Python community.

*IDLE* aside, if you need a more professional IDE, you have choices. The most popular in the Python space include:

- *Eclipse*: <https://www.eclipse.org>
- *PyCharm*: <https://www.jetbrains.com/pycharm/>
- *WingWare*: <https://wingware.com>

*Eclipse* is a completely open source technology, so won’t cost you more than the download. If you’re already an *Eclipse* fan, its support for Python is very good. But, if you aren’t currently using *Eclipse*, we wouldn’t recommend its use to you, due to the existence of *PyCharm* and *WingWare*.



Both *PyCharm* and *WingWare* are commercial products, with “community versions” available for download at no cost (but with some restrictions). Unlike *Eclipse*, which targets many programming languages, both *PyCharm* and *WingWare* target Python programmers specifically and, like all IDEs, have great support for project work, links to source code management tools (like *git*), support for teams, links to the Python docs, and so on. We encourage you to try both, then make your choice.

If IDEs aren’t for you, fear not: all of the world’s major text editors offer excellent language support to Python programmers.

#### WHAT DOES PAUL USE?

Paul’s text editor of choice is *vim* (Paul uses *MacVim* on his development machines). When working on Python projects, Paul supplements his use of *vim* with *ptpython* (when experimenting with code snippets), and he’s also a fan of *IDLE*. Paul uses *git* for local version control.

For what it’s worth, Paul doesn’t use a full-featured IDE, but his students love *PyCharm*. Paul also uses (and recommends) *Jupyter Notebook*, which is discussed next.

### 3. Jupyter Notebook: The Web-Based IDE

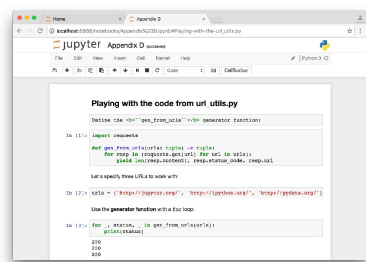
In item #1, we drew your attention to *ipython* (which is an excellent >>> alternative). From the same project team comes *Jupyter Notebook* (previously known as *iPython Notebook*).

#### NOTE

The next generation of Jupyter Notebook is called Jupyter Lab, and it was in “alpha” as work on this book was concluding. Keep an eye out for the Jupyter Lab project: it’s going to be something rather special.

*Jupyter Notebook* can be described as the power of *ipython* in an interactive web page (which goes by the generic name of “notebook”). What’s amazing about *Jupyter Notebook* is that your code is editable and runnable from within the notebook, and—if you feel the need—you can add text and graphics, too.

Here’s some code from [Chapter 13](#) running within a *Jupyter Notebook*. Note how we’ve added textual descriptions to the notebook to indicate what’s going on:



Learn more about *Jupyter Notebook* from its website (<http://jupyter.org>), and use `pip` to install it onto your computer, then start exploring. You will be glad you did. *Jupyter Notebook* is a killer Python application.

4. Doing Data Science

When it comes to Python adoption and usage, there's one domain that continues to experience explosive growth: the world of **data science**.

This is not an accident. The tools available to data scientists using Python are world class (and the envy of many other programming communities). What's great for non-data scientists is that the tools favored by the data folks have wide applicability outside the *Big Data* landscape.

Entire books have been (and continue to be) written about using Python within the *data science* space. Although you may think this advice biased, the books on this subject from *O'Reilly Media* are excellent (and plentiful). *O'Reilly Media* has made a business out of spotting where the technology industry is heading, then ensuring there's plenty of great, high-quality learning material available to those wanting to learn more.



Here's just a selection of some of the libraries and modules available to you if you do data science (or any other science calculations, for that matter). If *data science* isn't your thing, check out this stuff anyway—there's lots to like here:

- **bokeh**: A set of technologies for publishing interactive graphics on web pages.
- **matplotlib/seaborn**: A comprehensive set of graphing modules (which integrates with *ipython* and *Jupyter Notebook*).
- **numpy**: Among other things, allows you to efficiently store and manipulate multidimensional data. If you're a fan of matrices, you'll love *numpy*.
- **scipy**: A set of scientific modules optimized for numerical data analysis, which complements and expands upon what's provided by *numpy*.
- **pandas**: If you are coming to Python from the *R* language, then you'll feel right at home with *pandas*, which provides optimized analysis data structures and tools (and is built on top of *numpy* and *matplotlib*). The need to use *pandas* is what brings a lot of data folk to the community (and long may this continue). *pandas* is another *killer* Python application.
- **scikit-learn**: A set of machine learning algorithms and technologies implemented in Python.

Note: most of these libraries and modules are *pip*-installable.

The best place to start learning about the intersection of Python and *data science* is the *PyData* website: <http://pydata.org> (<http://pydata.org>). Click on *Downloads*, then marvel at what's available (all as open source). Have fun!

### 5. Web Development Technologies

Python is very strong in the web space, but *Flask* (with *Jinja2*) isn't the only game in town when it comes to building server-side webapps (even though *Flask* is a very popular choice, especially if your needs are modest).

The best-known technology for building webapps with Python is *Django*. It wasn't used in this book due to the fact that (unlike *Flask*) you have to learn and understand quite a bit before you create your first *Django* webapp (so, for a book like this, which concentrates on teaching the basics of Python *well*, *Django* is a poor fit). That said, there's a reason *Django* is so popular among Python programmers: it's really, really good.

If you class yourself as a "web developer," you should take the time to (at the very least) work through *Django*'s tutorial. In doing so, you'll be better informed as to whether you'll stick with *Flask* or move to *Django*.



If you do move to *Django*, you'll be in very good company: *Django* is such a large community within the wider Python community that it's able to sustain its own conference: *DjangoCon*. To date, *DjangoCon* has occurred in the US, Europe, and Australia. Here are some links to learn more:

- Django's landing page (which has a link to the tutorial):

<https://www.djangoproject.com>

- DjangoCon US:

<https://djangocon.us>

- DjangoCon Europe:

<https://djangocon.eu>

- DjangoCon Australia:

<http://djangocon.com.au> (<http://djangocon.com.au>)

#### BUT WAIT, THERE'S MORE

As well as *Flask* and *Django*, there are other web frameworks (and we know we'll neglect to mention somebody's favorite). Those we hear the most about include: *Pyramid*, *TurboGears*, *web2py*, *CherryPy*, and *Bottle*. Find a more complete list on the Python wiki:

<https://wiki.python.org/moin/WebFrameworks>

## 6. Working with Web Data

In [Chapter 13](#), we briefly used the `requests` library to demonstrate just how cool our generator was (compared to its equivalent comprehension). Our decision to use `requests` was no accident. If you ask most Python developers working with the Web what their favorite PyPI module is, the majority responds with one word: "requests."

PyPI: The Python Package Index lives at <https://pypi.org/>.

The `requests` module lets you work with HTTP and web services via a simple, yet powerful, Python API. Even if your day job doesn't involve working directly with the Web, you'll learn a lot just from looking at the code for `requests` (the entire `requests` project is regarded as a master class in how to do things the Python way).

Find out more about `requests` here:

<http://docs.python-requests.org/en/master/> (<http://docs.python-requests.org/en/master/>)

#### SCRAPE THAT WEB DATA!

As the Web is primarily a text-based platform, Python has always worked well in that space, and the *standard library* has modules for working with JSON, HTML, XML, and the other similar text-based formats, as well as all the relevant Internet protocols. See the following sections of the Python docs for a list of modules that come with the *standard library* and are of most interest to web/Internet programmers:

- Internet Data Handling:

<https://docs.python.org/3/library/netdata.html>

- Structured Markup Processing Tools:

<https://docs.python.org/3/library/markup.html>

- Internet Protocols and Support:

<https://docs.python.org/3/library/internet.html>



If you find yourself having to work with data that's only available to you via a static web page, you'll likely want to *scrape* that data (for a quick scraping primer, see [https://en.wikipedia.org/wiki/Web\\_scraping](https://en.wikipedia.org/wiki/Web_scraping)). Python has two third-party modules that will save you lots of time:

- *Beautiful Soup*:

<https://www.crummy.com/software/BeautifulSoup/>

- *Scrapy*:

<http://scrapy.org> (<http://scrapy.org>)

Try both, see which one solves your problem best, and then get on with whatever else needs doing.

## 7. More Data Sources

To keep things as real as possible (while trying to keep it simple), we used *MySQL* as our database backend in this book. If you spend a lot of time working with SQL (regardless of the database vendor you favor), then stop whatever you're doing and take two minutes to use `pip` to install `sqlalchemy`—it may be your best two-minute installation *ever*.

The `sqlalchemy` module is to SQL geeks what `requests` is to web geeks: indispensable. The *SQLAlchemy* project provides a high-level, Python-inspired set of technologies for working with tabular data (as stored in the likes of *MySQL*, *PostgreSQL*, *Oracle*, *SQL Server*, and so on). If you liked what we did with the `DBcm` module, you're going to love *SQLAlchemy*, which bills itself as *the* database toolkit for Python.

Find out more about the project at:

<http://www.sqlalchemy.org> (<http://www.sqlalchemy.org>)

## THERE'S MORE TO QUERYING DATA THAN SQL

Not all the data you'll ever need is in an SQL database, so there will be times when *SQLAlchemy* won't do. NoSQL database backends are now accepted as a valid addition to any data center, with *MongoDB* serving as the classic example as well as the most popular choice (even though there are many).



If you end up working with data that's being presented to you as JSON, or in a nontabular (yet structured) format, *MongoDB* (or something similar) may be just what you're looking for. Find out more about *MongoDB* here:

<https://www.mongodb.com>

And check out the Python support for programming *MongoDB* using the pymongo database driver from the *PyMongo* documentation page:

<https://api.mongodb.com/python/current/>

## 8. Programming Tools

No matter how good you think your code is, bugs happen.

When they do, Python has lots to help you: the `>>>` prompt, the `pdb` debugger, `IDLE`, `print` statements, `unittest`, and `doctest`. When these options aren't enough, there are some third-party modules that might help.

Sometimes, you'll make a classic mistake that everyone else has made before you. Or perhaps you've forgotten to import some required module, and the problem doesn't crop up until you're showing off how great your code is to a room full of strangers (whoops).

To help avoid this type of thing, get *PyLint*, Python's code analysis tool:

<https://www.pylint.org>

*PyLint* takes your code and tells you what might be wrong with it *before* you run it for the first time.

If you use *PyLint* on your code before you run it in front of a room full of strangers, it may very well prevent blushing. *PyLint* might also hurt your feelings, as no one likes to be told their code is not up to scratch. But the pain is worth the gain (or maybe that should be: *the pain is better than the public embarrassment*).





#### MORE HELP WITH TESTING, TOO

In *Appendix C*, #9, we discussed the built-in support Python provides for automated testing. There are other such tools, too, and you already know that `py.test` is one of them (as we used it earlier in this book to check our code for *PEP 8* compliance).

Testing frameworks are like web frameworks: everyone has their favorite. That said, more Python programmers than not favor `py.test`, so we'd encourage you to take a closer look:

<http://doc.pytest.org/en/latest/> (<http://doc.pytest.org/en/latest/>)

#### 9. Kivy: Our Pick for “Coolest Project Ever”

One area where Python is not as strong as it could be is in the world of mobile touch devices. There are a lot of reasons why this is (which we aren't going to get into here). Suffice it to say, at the time of publication, it is still a challenge to create an *Android* or *iOS* app with Python alone.

One project is attempting to make progress in this area: *Kivy*.

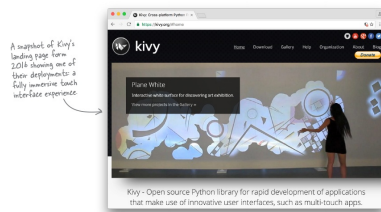
*Kivy* is a Python library that allows for the development of applications that use multitouch interfaces. Pop on over to the *Kivy* landing page to see what's on offer:

<https://kivy.org>

Once there, click on the *Gallery* link and sit back for a moment while the page loads. If a project grabs your eye, click on the graphic for more information and a demo. While you view the demo, keep the following in mind: *everything you are looking at was coded with Python*. The *Blog* link has some excellent material, too.

What's really cool is that your *Kivy* user interface code is written once, then deployed on any supported platform *unchanged*.

If you are looking around for a Python project to contribute to, consider donating your time to *Kivy*: it's a great project, has a great team working on it, and is technically challenging. If nothing else, you won't be bored.



## 10. Alternative Implementations

You already know from item #1 in *Appendix C* that there's more than one Python language release (Python 2 and Python 3). This means that there's *at least* two Python interpreters: one that runs Python 2 code, and one that runs Python 3 code (which is the one we've used throughout this book). When you download and install one of the Python interpreters from the Python website (like you did in *Appendix A*), the interpreter is referred to as the *CPython reference implementation*. *CPython* is the version of Python distributed by the *Python core developers*, and takes its name from the fact that it's written in portable C code: it's designed to be easily ported to other computing platforms. As you saw in *Appendix A*, you can download installers for *Windows* and *Mac OS X*, as well as find the interpreter preinstalled within your favorite Linux distribution. All of these interpreters are based on *CPython*.

Python is open source, so anyone is free to take *CPython* and change it in any way they wish. Developers can also take the Python language and implement their own interpreter for it in whichever programming language they wish, using whichever compiler techniques they like, running on whatever platform they're using. Although doing all of this is not for the faint of heart, plenty of developers do this (some of them describe it as "fun"). Here are short descriptions and links to some of the more active projects:

- *PyPy* (pronounced "pie-pie") is an experimental compiler testbed for Python 2 (with Python 3 support on the way). *PyPy* takes your Python code and runs it through a just-in-time compilation process, producing a final product that runs faster than *CPython* in many instances. Find out more here:

<http://pypy.org> (<http://pypy.org>)

- *IronPython* is a version of Python 2 for the .NET platform:

<http://ironpython.net> (<http://ironpython.net>)

- *Jython* is a version of Python 2 that runs on *Java's JVM*:

<http://www.jython.org> (<http://www.jython.org>)

- *MicroPython* is a port of Python 3 for use on the *pyboard* microcontroller, which is no bigger than your two thumbs side by side, and may well be the coolest little thing you've ever seen. Take a look:

<http://micropython.org> (<http://micropython.org>)



Despite all these alternative Python interpreters, the majority of Python programmers remain happy with *CPython*. Increasingly, more developers are choosing Python 3.



PREV

[C. Top Ten Things we Didn't Cover: There's Always More to Learn](#)

NEXT

[E. Getting Involved: The Python Community](#)