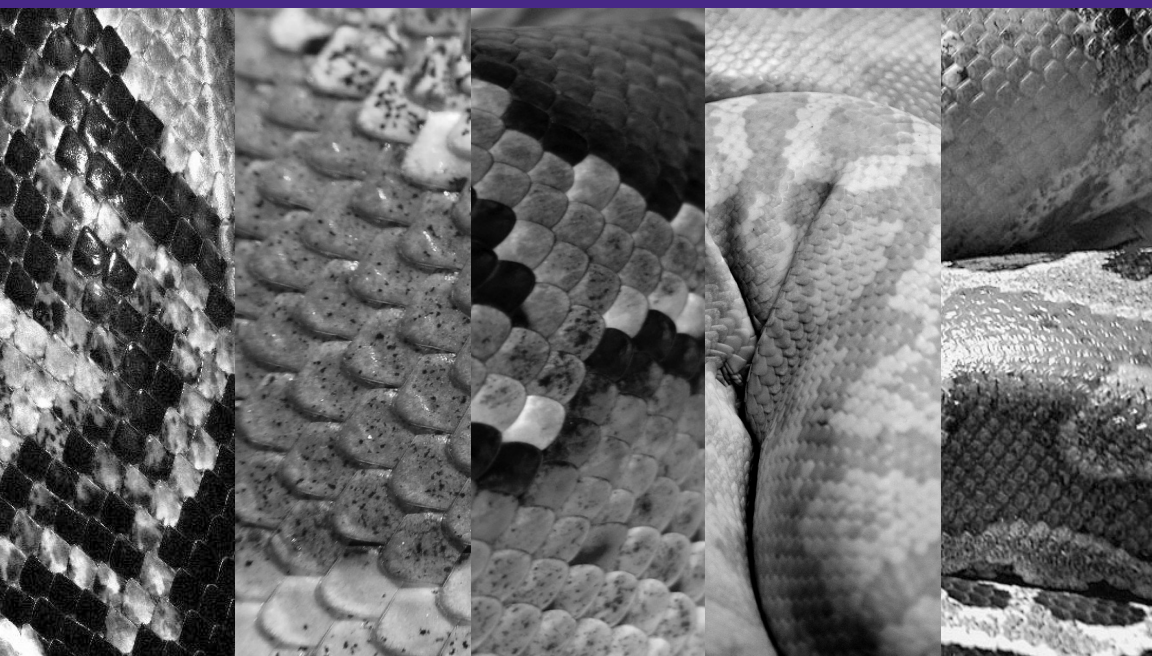# Picking a Python Version: A Manifesto

## From __future__ import Python

David Mertz

# Picking a Python Version: A Manifesto

*From __future__ import Python*

*David Mertz*

**Picking a Python Version: A Manifesto**

by David Mertz

Copyright © 2015 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

# Table of Contents

# Python Versions

There are two major versions of the Python programming language: the Python 2.x series, and the newer Python 3.x series. The Python 3.x series started with the release of Python 3.0 in December 2008; since that time, Python 2.7 was released and has continued to receive point version releases, with Python 2.7.9 being the latest version as of this writing. As a footnote, there *was*, naturally, a Python 1.x series that was highly successful in the 1990s, but that series has long been out-of-maintenance.

In his 2014 PyCon keynote (Figure 1-1), Guido van Rossum, Python's Benevolent Dictator for Life (sometimes affectionately called the BDFL or GvR) made it clear that the Python 2.x series will not continue past Python 2.7.x. The end-of-life for Python 2.7.x is 2020, and no new features will be added to the 2.x series of Python.

Within a series, Python places an especially strong philosophical emphasis on backward compatibility (in comparison to many other programming languages). It is extremely rare for a series to break this backward compatibility in later minor versions; in the few cases where it has occurred, it is only to address large previously undiscovered bugs or security issues, and even then great attention is paid to affecting as little running code as possible. That said, new Python versions inevitably add important features, either in the language itself, its built-in functions, or in standard library support modules.

*Figure 1-1. Guido van Rossum's PyCon 2014 keynote address (photo credit: David Lesieur 2014, CC-BY-SA)*

Python developers have numerous versions and implementations to choose from. Many of us still use Python 2.x and CPython—and that is still often a good choice—but there are many situations in which making a different choice makes sense.

Python 3.x, and in particular the Python 3.5 version coming out soon, offer numerous advantages over the Python 2.x series. New development should typically be done using Python 3.x, and although the 2020 end-of-life for Python 2.x isn't *all that* close, it is good for companies and projects to start think of porting plans for existing code.

# The (Small) Break

With the introduction of the so-called "Python 3000" version, which was released after long deliberation as Python 3.0 and subsequent Python 3.x versions, the designers deliberately allowed small language incompatibilities to be introduced in the new language series. A full discussion of this is contained in a *Python Enhancement Proposal:* PEP 3000 - Python 3000. Even during the Python 2.x to 3.x series transition, the philosophy of Python insisted on making

changes as incremental as possible. Unlike in the histories of some other programming language evolutions, Python 3.x was never meant as a *blue-sky* project, and avoids the pitfalls of a "second-system effect."[1] Python 3.x aims to improve a familiar Python 2.x language rather than to create a new or fundamentally different language.

While there is a significant subset of Python code that is able to be *version-neutral* "out of the box," most existing codebases require a small degree of rewriting and/or use of compatibility tools and libraries to move from Python 2.x to Python 3.x. Real-world code using the most natural idioms in Python 2.x usually does various things that are not quite compatible with Python 3.x, and the code that is version-neutral must be written in a careful way to achieve that. That is to say, if you wrote some module or script 10 years ago, it is quite unlikely is will "just run" in Python 3.x with no changes.

## Moving Forward

The bottom line is that Python 3.x is a better language than Python 2.x, albeit in a large collection of relatively small ways. But as with any transition—especially one that introduces genuine backward incompatibilities—it takes time to move users to the latest version. To a lesser extent, this is true even within a series: there were (are) certainly some few long-running and stable programs that ran (probably still run) on Python 2.1—or even on Python 1.5.2—where simply keeping the platform consistent was easier than testing a transition. "If it ain't broke, don't fix it!"

However, going forward, new projects should be written for Python 3.x, and actively maintained old projects should transition between versions or plan on doing so at the earliest opportunity. Still, even after the 2020 end-of-life, executables already running will not suddenly stop doing so (albeit, changes in underlying operatings systems or in supporting hardware *might* require new versions, but much the same upgrade issue exists for OSes as for programming languages, and letting old systems "just run" is the same option in both cases).

One issue to consider when moving to the latest version of Python is simply what version of the language comes preinstalled with your

1 Fred Brooks, *The Mythical Man-Month*. Addison-Wesley, 1975.

operating system, if any. Microsoft Windows® does not ship with any real built-in developer tools or languages. Of course, Pythons—in various versions and distributions—are free to download from python.org and other sites (as are many other programming languages and developer tools). Apple OS X® ships with Python 2.x pre-installed (in all recent versions; their future plans are confidential and presumably strategic), so while freely available, installing Python 3.x takes at least some extra effort. Linux® distributions have traditionally shipped with Python 2.x installed, but increasingly the latest versions of these "distros" use Python 3.x for all of their internal tooling, ship with Python 3.x on their default media (either exclusively or with both Python 3.x and Python 2.x included), and generally encourage the use of Python 3.x. Of course, for many reasons similar to those discussed above, upgrading existing hardware systems from older operating systems is itself an effort, and poses risks of breaking existing programs and workflows.

## Unix-Like Systems

PEP 394, entitled *The "python" Command on Unix-Like Systems*, specifies the recommended configuration of Python on Unix-like systems. This includes Linux distros, of course. It equally applies to BSD-family Unix systems, such as FreeBSD®, OpenBSD™, NetBSD™, and even to Apple OS X. Of course, while the Python Software Foundation can make a recommendation via a PEP, no other entity is bound to follow such recommendations (some do, some don't). The basic purpose of this recommendation is to govern the behavior of files that use the "shebang convention"—that is, files that look at the first few bytes of a file to see if they are an executable script by seeing if they look like:

```
#!/path/to/executable
```

Or often indirectly as:

```
#!/usr/bin/env executable
```

In quick summary, PEP 394 recommends that within an installed operating system environment:

- `python2` will refer to some version of Python 2.x.
- `python3` will refer to some version of Python 3.x.
- Currently, `python` will refer to the same target as `python2`.

- Eventually, `python` will refer to the same target as `python3`.
- Python 2.x-only scripts should either be updated to be source compatible with Python 3.x or use `python2` in the shebang line.

While some currently shipping systems like Apple OS X only ship with Python 2.x, others like Arch Linux™ ship with `python` aliased to `python3` already. In (almost) all cases, explicitly specifying `python2` or `python3` in the shebang line will resolve any ambiguity.

## Python 3 on Fedora and Red Hat

Major Linux distributions generally follow the recommendation of PEP 394, and furthermore, are moving at a consistent pace towards general internal use of Python 3.x. For example, Fedora's wiki documents this effort:

> The main goal is switching to Python 3 as a default, in which state:
>
> - DNF is the default package manager instead of Yum, which only works with Python 2
> - Python 3 is the only Python implementation in the minimal buildroot
> - Python 3 is the only Python implementation on the LiveCD
> - Anaconda and all of its dependencies run on Python 3
> - cloud-init and all of its dependencies run on Python 3

## Python 3 on Ubuntu

Ubuntu is following a similar path vis-à-vis Python versioning as is Fedora (and Red Hat). Ubuntu's wiki describes their goals:

> For both Ubuntu and Debian, we have ongoing project goals to make Python 3 the default, preferred Python version in the distros. This means:
>
> - Python 3 will be the only Python version installed by default. Python 3 will be the only Python version in any installation media (i.e. image ISOs)
> - Only Python 3 will be allowed on the Ubuntu touch images.
> - All upstream libraries that support Python 3 will have their Python 3 version available in the archive.

- All applications that run under Python 3 will use Python 3 by default.

- All system scripts in the archive will use Python 3.

Ubuntu 14.04 LTS has recently been released. We made great progress toward these goals, but we must acknowledge that it is a daunting, multi-cycle process. A top goal for 14.04 was to remove Python 2 from the touch images, and sadly we almost but didn't quite make it. There were still a few autopilot tests for which the Python 3 ports did not land in time, thus keeping Python 2 autopilot support on the base touch image. This work is being completed for Utopic and we expect to remove Python 2 from the touch images early in the 14.10 cycle (actually, any day now).

# Python 3 Uptake

It is difficult to know with any confidence just how widely used Python 3.x is compared to Python 2.x. Indeed, it is not easy to know how widely used Python is in general, either in absolute terms or compared to other programming languages. For that matter, there are many meanings one could give to "how widely used" to begin with: how many local applications? How many servers? Serving how many clients? How much CPU time used in the process? How important are the various applications? How many lines of code in the version? And so on.

Certainly Python in general is near the top of popular programming languages if one looks at indices like the TIOBE Programming Community Index, the Transparent Language Popularity Index, the PYPL PopularitY of Programming Language, IEEE Spectrum's 2014 Ranking, or The RedMonk Programming Language Rankings.

Being active in the Python community, and also being a director of the Python Software Foundation, this writer has some access to some rough indicators of Python version usage that while not confidential, also haven't been widely published (mostly because of their lack of statistical rigor). But a few points are suggestive.

## Downloads from python.org

On the python.org website itself, downloads of 3.x versions started outnumbering downloads of 2.x versions beginning in early 2013. As discussed above, many operating system distributions come with Python versions installed, so those probably do not need to be

downloaded from python.org. Moreover, one download from python.org might result in anywhere from zero to tens of thousands of installs on individual machines at companies. And furthermore, other sites are free to, and many do, mirror Python archives, so not all downloads are via python.org, even initially. Adding to that, and discussed later in this paper, various third parties have created their own Python distributions that include various other sets of "batteries included" beyond what the distributions provided by the PSF do. So the indicator mentioned is very rough, but a positive suggestion at least. (Figure 1-2 shows the python.org download menu.)
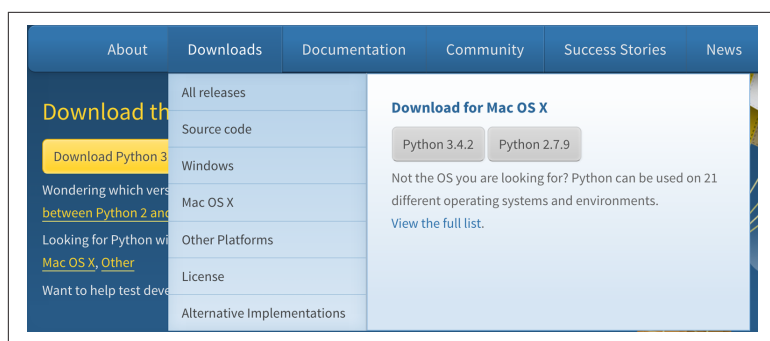


*Figure 1-2. The download navigation menu of python.org*

Downloads of third-party software and libraries from the Python Package Index (PyPI) have the same caveats as those about the language itself. Many downloads from PyPI are automated ones done by `pip`, `easy_install`, or other automated installers—including many that are not Python-specific, such as `apt-get`, `yum`, `port`, and `brew` (albeit, probably none of the non-Python installers listed directly access PyPI, but use their own archives instead). Many downloads are also done as part of automated testing of configuration as well, which may create inertia in repeatedly downloading older packages within such automated scripts.

## Downloads from the Python Package Index

In any case, the internal logs for PyPI suggest that downloads of Python 3.x-specific packages remain below 25% of the downloads from the site. It's hard to know the exact reasons—a positive thought is that this might be partially because Python 3.x is even more "batteries included" in the basic distribution than 2.x was, and hence there is less need for third-party tools. However, likely that explana-

tion is overly Panglossian, and the new modules in Python 3.x have only a small effect on the demand and use of third-party libraries and tools. (Figure 1-3 shows PyPI's navigation screen.)



*Figure 1-3. The navigation screen of the Python Package Index*

## A 2013/2014 Survey

A survey was completed at the beginning of 2014 to gauge relative usage, based on responses from postings on comp.lang.python, python-dev, and hacker news. While still unscientific, the 2.x-vs-3.x-survey might be of interest to readers:

| Question | Yes | No |
| --- | --- | --- |
| Have you ever written code in Python 2.x? | 97.29% | 2.48% |
| Have you ever written code in Python 3.x? | 59.75% | 39.83% |
| Do you currently write more code in Python 2.x than 3.x? | 77.08% | 21.63% |
| Do you think Python 3.x was a mistake? | 22.36% | 75.22% |
| Do you have dependencies keeping you on Python 2.x? | 60.19% | 37.75% |
| Have you ever ported code from Python 2.x to Python 3.x? | 32.44% | 66.37% |
| Have you ever written/ported code using 2to3? | 16.03% | 82.53% |
| Have you ever written/ported code using 3to2? | 1.90% | 96.60% |
| [. . .] code to run on Python 2.x and Python 3.x unmodified? | 30.75% | 68.08% |

# Is It Enough?

There seems to be an ongoing perception in at least parts of the Python community that projects are "stuck" on Python 2.x, whether either widely used libraries or in-house codebases. This perception is mostly false when it comes to widely used third-party libraries: the large majority of the most important FLOSS support libraries have Python 3.x-compatible versions today. Because some of those library versions have been created relatively recently, perceptions of the "ecosystem" not having moved to Python 3.x may simply reflect infrequent review by developers of the overall snapshot of porting statuses (it is a not inconsiderable project to conduct such a review as it applies to one's own large codebase).

Adoption of Python 3.x was also slowed somewhat by missteps in Python 3.0 that were not fixed until Python 3.1. So there was not a really *good* and *stable* Python 3.x version until mid-2009, in truth. For applications that work intensively with text processing, the benchmark version is probably even Python 3.3, because of the improvements in PEP 393 - Flexible String Representation. That release happened in September 2012. The variable-width storage of unicode strings made for a big win in memory allocation and usage:

> The memory usage of Python 3.3 is two to three times smaller than Python 3.2, and a little bit better than Python 2.7, on a Django benchmark (see the PEP for details).

More than the details of what improvements arrived on what dates in the history of Python 3.x, what probably feeds many developers' sense of being "stuck" is not any concrete large conceptual or infrastructure problem in porting, but simply the fact that it takes more work to change versions—on a short-term basis—than it does to leave things as they are within a large codebase.

Even adding some minor functionality, bug fix, kludge, or work-around to a large, in-house, Python 2.x codebase is less work *today* than is porting (and more importantly, testing and validating that port) to Python 3.x. The *next* problem would often have been solved by the port, or at least much easier to address after it is made. But today's work is done now, and the next problem not addressed until later. In corporations, profits and expenses are accounted for quarterly; and even open source projects are also often constrained by

what is possible—or rewarding to volunteer developers—*immediately* rather than what makes things better in the long term.

This writer's take on migration to Python 3.x is that:

- Overall, the migration is inevitable.
- Some projects or long-running processes will make a decision (often a reasonable one) to stick with what they know is stable and works for their specific purpose until the code itself fades from relevance.
- Migration is moving at a reasonable and steady pace, even if slightly more slowly than I'd like to have seen.
- The release schedules and end-of-life dates are well timed for a gradual and successful transition.
- In some ways, Python's conservative philosophy of compatibility and stability pushes against migration. Python 2.x has a long maintenance period, and while the improvements in Python 3.x really are *great*, intentionally none are *revolutionary* or *fundamental*. Python got most things right from the start, and those right choices haven't been changed in Python 3.x. There is no new *paradigm* here, just new *capabilities*.
- The sky isn't falling, and there *still* isn't ever going to be a Python 2.8 (and neither are users going to abandon Python in droves for some other language because of the minor transition difficulties from Python 2.x to Python 3.x).

## Python 3.5 Release Schedule

The current plan of record, documented in PEP 478, entitled *Python 3.5 Release Schedule*, sets the release date of Python 3.5.0 final as September 13, 2015. As with other minor versions, Python 3.5 will add several interesting and useful features. In general, this paper takes a modestly forward-looking perspective. Where advantages are discussed later, they extend through Python 3.5 but will not generally differentiate exactly when in the Python 3.x series a feature was introduced.

# Porting

Several tools are available to somewhat automate porting of code from Python 2.x to Python 3.x. Using any of these will generally require some manual developer attention during the process, but in many cases a fairly small degree of such attention.

There are two approaches to consider when porting code:

- Is this a one time transition? For in-house code, it is almost certainly a better idea to make a clean break and commit to the Python 3.x series (after sufficient testing, code review, etc.). Even FLOSS libraries may want to make the jump as well, perhaps maintaining parallel versions for a certain period of time (e.g., a `Foolib` and `Foolib3` might both get updated for a while).

- Do you want to instead create a *version-neutral* codebase that will run on both Python 2.x and Python 3.x? It is possible to use compatibility shims to write such code while still fairly elegantly falling back or falling forward with differences in features. However, using these shims will probably require writing some code in a manner less than the most idiomatic style for a Python series, at least at times.

Basically, if you want to make a clean break, use the tool `2to3` which comes bundled with recent Python versions, in both series. If you want to create version-neutral code, use the support libraries `six` or `Python-Futures`. There was, for a while, a short-lived project called `3to2` to backport code already written for Python 3.x; however, it has not been well maintained, and using it is not recommended.

# 2to3

The documentation for *2to3 - Automated Python 2 to 3 code translation* describes it as follows:

> 2to3 is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.x code. The standard library contains a rich set of fixers that will handle almost all code. [The] 2to3 supporting library lib2to3 is, however, a flexible and generic library, so it is possible to write your own fixers for 2to3. lib2to3 could also be adapted to custom applications in which Python code needs to be edited automatically.

As a little experiment for this paper, I decided to take a look at my long out-of-maintenance library Gnosis_Utils. I do not particularly recommend using any part of this library, since most of the useful things it did have been superceded by newer libraries. For the most part, the library was always a teaching tool, and a way of publishing examples and concepts that I discussed in my long-running column for IBM developerWorks, *Charming Python* (albeit, I was pleased to hear from many correspondents who used parts of it in real production systems, and to know it was included with some Linux and BSD distributions). In particular, this library stopped being maintained in 2006, and was probably last run by me using Python 2.5.

However, because of its age, it might make a good experiment in porting. In its simplest mode, running 2to3 simply proposes a diff to use in updating files. There are switches to perform the actions automatically, and some others to limit which transformations are performed. Let us look at what the tool might do in default mode:

```
% 2to3 Gnosis_Utils-1.2.2/gnosis/trigramlib.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: set_literal
RefactoringTool: Skipping implicit fixer: ws_comma
RefactoringTool: Refactored  Gnosis_Utils-1.2.2/gnosis/trigram-
lib.py
--- Gnosis_Utils-1.2.2/gnosis/trigramlib.py (original)
+++ Gnosis_Utils-1.2.2/gnosis/trigramlib.py (refactored)
@@ -1,6 +1,6 @@
"Support functions for trigram analysis"
-from __future__ import generators
-import string, cPickle
+
+import string, pickle
```

```
def simplify(text):
    ident = [chr(x) for x in range(256)]
```
```
    return " ".join(ts.text_splitter(text, casesensitive=1))

def simplify_null(text):
-    ident = ''.join(map(chr, range(256)))
+    ident = ''.join(map(chr, list(range(256))))
    return text.translate(ident, '\n\r')

def generate_trigrams(text, simplify=simplify):
    "Iterator on trigrams in (simplified) text"
    text = simplify(text)
-    for i in xrange(len(text)-3):
+    for i in range(len(text)-3):
        yield text[i:i+3]

def read_trigrams(fname):
```
```
        trigrams = {}
        for line in open(fname):
            trigram = line[:3]
-                        spam,good = map(lambda s: int(s,16),
line[3:].split(':'))
+                        spam,good = [int(s,16) for s in
line[3:].split(':')]
            trigrams[trigram] = [spam,good]
        return trigrams
    except IOError:
```
```
def write_trigrams(trigrams, fname):
    fh = open(fname,'w')
-    for trigram,(spam,good) in trigrams.items():
-        print >> fh, '%s%x:%x' %(trigram,spam,good)
+    for trigram,(spam,good) in list(trigrams.items()):
+        print('%s%x:%x' %(trigram,spam,good), file=fh)
    fh.close()

def interesting(rebuild=0):
    "Identify the interesting trigrams"
    if not rebuild:
        try:
-                        return cPickle.load(open('interesting-
trigrams','rb'))
+                        return pickle.load(open('interesting-
trigrams','rb'))
        except IOError:
            pass
    trigrams = read_trigrams('trigrams')
    interesting = {}
```

```
-       for trigram,(spam,good) in trigrams.items():
+       for trigram,(spam,good) in list(trigrams.items()):
            ratio = float(spam)/(spam+good)
            if spam+good >= 10:
                if ratio < 0.05 or ratio > 0.95:
                    interesting[trigram] = ratio
-                   cPickle.dump(interesting,    open('interesting-
trigrams','wb'), 1)
+                   pickle.dump(interesting,    open('interesting-
trigrams','wb'), 1)
        return interesting

RefactoringTool: Files that need to be modified:
RefactoringTool: Gnosis_Utils-1.2.2/gnosis/trigramlib.py
```

Everything suggested here will produce valid Python 3.x code, and applying the diff is completely sufficient to do so. But a few of the suggestions are neither idiomatic nor necessary. For example:

```
- ident = ''.join(map(chr, range(256)))
+ ident = ''.join(map(chr, list(range(256))))
```

Since the built-in function `range()` has become lazy (like `xrange()` was in Python 2.x), `2to3` conservatively shows us how to create a concrete list. In almost all contexts, lazy is not only acceptable, but even superior in performance (it doesn't matter for `range(256)`, but it might for `range(1000000000)`).

On the other hand, some of the suggestions `2to3` makes are actually to change constructs that are compatible with Python 3.x but simply not idiomatic, nor as elegant. For example:

```
- spam,good = map(lambda s: int(s,16), line[3:].split(':'))
+ spam,good = [int(s,16) for s in line[3:].split(':')]
```

The existing line will still work correctly in Python 3.x, but using a list comprehension is more idiomatic and more readable than using `map(lambda s: ..., ...)`.

Past merely *working* after taking or ignoring such a list of proposed changes by an informed developer, the next step in a high-quality port is a more systematic code review. Not just "What is the syntactic way to do this?", but also "Are there standard library modules or new constructs that do this faster? Or more generally? Or more correctly? Or using less code and/or more readable code?" The answer to these questions is often yes—but then, that answer is often yes within a version series, or simply when new eyes look at old code.

Getting code simply to the level of working correctly is usually as simple as running `2to3` and applying its recommendations. Further improvements can be done incrementally, and as time and requirements permit.

# six.py

The documentation for *Six: Python 2 and 3 Compatibility Library* describes it as follows:

> Six provides simple utilities for wrapping over differences between Python 2 and Python 3. It is intended to support codebases that work on both Python 2 and 3 without modification. [S]ix consists of only one Python file, so it is painless to copy into a project.

There are a variety of constructs that differ between Python 2.x and 3.x, but it is often possible to wrap the basic functionality in a function, class, or method that abstracts away the difference (perhaps by explicitly detecting interpreter version and branching). The effect is that in version-neutral code using `six.py`, one frequently utilizes calls of the `six.do_something(...)` sort as the way of doing the close-enough-to-equivalent thing under whichever Python version the program runs in.

For example, almost certainly the most obvious difference between Python 2.x and Python 3.x—especially for beginners—is that the `print` statement has been replaced by the `print()` function. What `six` gives users is a function `six.print_()` to use everywhere. In Python 3.x, it is simply an alias for the built-in `print()` while in Python 2.x it reimplements the behavior of that function (with the various keyword arguments). So, for example, where `2to3` would suggest this change:

```
- print >>fh, '%s%x:%x' % (trigram, spam, good)
+ print('%s%x:%x' % (trigram, spam, good), file=fh)
```

The version-neutral change would use:

```
six.print_('%s%x:%x' % (trigram, spam, good), file=fh)
```

Python 2.7.x has itself already backported many things—where it is possible to do so without breaking compatibility. So in some cases `six` is useful instead for supporting even earlier versions of Python 2.x. For example, this is probably what you'd actually do if you only cared about supporting Python 2.7.x and Python 3.x:

```
from __future__ import print_function
import sys
print("Hello error world", file=sys.stderr, sep=" ")
```

Use of `six.print_()`, however, lets your program run even on Python 2.4. In a similar vein:

```
six.get_function_globals(func)
    Get the globals of func. This is equivalent to func.__glob
als__
    on Python 2.6+ and func.func_globals on Python 2.5.
```

Likewise, in an arguably more obscure case, the way metaclasses are declared changed between Python 2.x and 3.x; `six` abstracts that also:

```
import six
@six.add_metaclass(Meta)
class MyClass(object):
    pass
```

Which is equivalent to Python 3.x's:

```
class MyClass(object, metaclass=Meta):
    pass
```

Or on Python 2.6+ to:

```
class MyClass(object):
    __metaclass__ = Meta
```

But to support Python 2.5 and earlier, you would have to use instead:

```
class MyClass(object):
    pass
MyClass = six.add_metaclass(Meta)(MyClass)
```

The effect, however, of writing version-neutral code using `six.py` is that you wind up writing code that is not particularly idiomatic for either Python 2.x or Python 3.x, but instead winds up utilizing many functions in the `six` module rather than native syntax or built-ins.

## Python-Future

The documentation for Python-Future describes it as follows:

> python-future is the missing compatibility layer between Python 2 and Python 3. It allows you to use a single, clean Python 3.x-compatible codebase to support both Python 2 and Python 3 with minimal overhead.

It provides future and past packages with backports and forward ports of features from Python 3 and 2. It also comes with futurize and pasteurize, customized 2to3-based scripts that helps you to convert either Py2 or Py3 code easily to support both Python 2 and 3 in a single clean Py3-style codebase, module by module.

Python-Future is cleaner than is `six.py`, but it does so in part by not attempting to support early versions within the Python 2.x series—and to some degree also ignoring early versions in the Python 3.x series. The core developers of Python have added a number of convenience in Python 2.7.x, and in Python 3.3+ to bring the two closer to compatibility. For example, Python 2.7.x allows importing from `__future__` to change the behavior of the interpreter to be more like Python 3.x. For example:

```python
from __future__ import (absolute_import,
                        division,
                        print_function)
```

Mind you, adding this line can—and probably will—break existing code that runs in a module that previously lacked that line; importing from the future is at least a step towards an actual `2to3` conversion (in fact, using those two techniques together is often a good idea; i.e., you can modernize your Python 2.7.x code but not yet actually move to Python 3.x).

In the other direction, PEP 414, entitled *Explicit Unicode Literal for Python 3.3*, added so-called unicode literals to Python 3.x. Notice that this is purely a compatibility convenience, in Python 3.3+, there is absolutely no difference in meaning between `"Foobar"` and `u"Foobar"` because all strings are unicode already. But it lets Python 2.x code that uses the unicode literals run on Python 3.3+ (obviously, assuming other features are converted or shimmed, as needed).

Using Python-Future does not preclude you from also using `six.py`. In fact, the Python-Future documentation recommends using the following lines at the top of modules:

```python
import future      # pip install future
import builtins    # pip install future
import past        # pip install future
import six         # pip install six
```

The module `builtins` is especially interesting: it provides Python 2.x implementations of Python 3.x built-ins that either behave differently or simply do not exist in Python 2.7.x. So often, along with

the `from __future__ import ...` line, a "futurized" application will contain a line like this:

```python
from builtins import (bytes, str, open, super, range,
                       zip, round, input, int, pow, object)
```

Under Python 3.x, this will have no effect, but under Python 2.x, familiar functions will have enhanced behaviors. Along a similar line, the `futures.standard_library` submodule also makes Python 2.x more like Python 3.x:

```python
>>> from future import standard_library
>>> help(standard_library.install_aliases)
Help on function install_aliases in module
future.standard_library:

install_aliases()
    Monkey-patches the standard library in Py2.6/7 to provide
    aliases for better Py3 compatibility.
```

Moreover, Python-Future combines the approaches of `2to3` with `six.py` in some ways, in particular, the tool `futurize` that comes with it does a conversion of code like `2to3` does, but the result is version neutral, and yet still generally idiomatic for Python 3.x (somewhat unlike with `six.py`).

In some cases, it is also possible to automatically utilize modules written for Python 2.x within Python 3.x programs without explicitly saving `futurize` or `2to3` (or manually) converted files first. For example:

```python
from past import autotranslate
autotranslate(['useful_2x_only'])
import useful_2x_only
```

The `autotranslate()` function is still in alpha at the time of this writing, so (as with all changes) be sure to test after utilizing it.

## Library Support

Most major libraries have been ported to Python 3.x as of the time of this writing. This paper cannot be exhaustive in listing popular libraries that already support Python 3.x, nor in discussing ones that do not have that support—there are hundreds or thousands in each category. Some "big name" libraries that are available are listed below. Of course, there remain—and indeed, always *will* remain, some libraries that do not get ported; if non-ported libraries are

essential to the task you need to do, that is an obstacle. Of course, you might address that obstacle by:

1. Sticking with Python 2.x.

2. Porting the needed support library yourself.

3. Finding a suitable substitute library covering the same general domain.

Still, among libraries already well supported in Python 3.x are the following:

- Web frameworks
  — django
  — flask
  — bottle
  — pyramid
  — Jinja2
  — tornado
- Numeric/scientific
  — NumPy
  — SciPy
  — pandas
  — SimPy
  — matplotlib
- Cryptography
  — pycrypto
  — ssl
  — rsa
- Network
  — requests
  — httplib2
  — gunicorn
  — pyzmq
  — pycurl

- Database
  - psycopg2
  - redis
  - pymongo
  - SQLAlchemy
- Environments
  - idle
  - IPython
  - IPython Notebook
  - virtualenv
- Data formats
  - lxml
  - simplejson
  - anyjson
  - PyYaml
  - Sphinx
  - pyparsing
  - ply
- Testing
  - nose
  - coverage
  - mock
  - pyflakes
  - pylint
  - pytest
  - WebTest
- Concurrency
  - greenlet

Some likely obstacles include:

- Twisted. Many, but definitely not all, of the capabilities in this framework have been ported. See the wiki entry for Plan/ Python3 – Twisted.
- Mechanize.
- Scrapy.

The list here is, of course, somewhat subjective and impressionistic. The tools or libraries this writer is most familiar with are not necessarily the ones that matter most to *you*.

# Advantages of Python 3.x

A pretty thorough overview of improvements to Python 3.x versions is contained in the series of "What's new in Python *N.m*" documents. Each document focuses on the improvements that were made since the prior minor version, rather than globally what's new in the history of Python.

- What's New in Python 3.0
- What's New In Python 3.1
- What's New In Python 3.2
- What's New In Python 3.3
- What's New In Python 3.4
- What's New In Python 3.5

Here are some highlights of new capabilities, somewhat subjectively listed because this writer finds them interesting or important. Lots of worthwhile features and modules did not make the cut for discussion in this paper.

## Unicode Handling

In Python 3.x, the `str` type is always Unicode (with efficient encoding strategies for strings in recent Python 3.x versions, based on the specific code points occuring within strings). There are far too many intricacies to the correct handling of all the characters in all the world (thoughout the history of human language, moreover) within the same program, with many semantic nuances of directionality, character composition, the Basic Multilingual Plane versus Supple-

mentary Planes, code surrogates, canonicalization, and on and on, to discuss in this paper.

Let it suffice for the quick summary to say that recent versions of Python 3.x versions get all these details correct and make them usable. That is not to say that multilingual programming is *easy*, but Python 3.x gets closer to making it *possible* than does Python 2.x— and in fact does better than essentially any other existing programming languages (most of which, even when they use "native Unicode" still handle various edge cases in approximate or broken ways).

The reality is that the world is no longer just the places that use ASCII for text (nor was it actually ever so), and creating user interfaces and processing data from all the world's languages is of growing importance, and of a significance that will only grow further. Python 3.x makes the easy things easy and the difficult things possible in this domain.

# The Greatly Improved Email and Mailbox Modules

Getting everything in the `email` and `mailbox` modules working correctly required a lot of concrete work, but also access to the full and robust handling of Unicode strings. There are too many specific improvements to document in this paper, but for a far better version of it, use Python 3.x.

# The concurrent.futures Module

The module `concurrent` abstracts features common to threads, processes, and remote procedure calls. It supports status checks (running or done), timeouts, cancellations, adding callbacks, and access to results or exceptions. For example:

```python
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

# Coroutine Support and yield from

PEP 380, entitled *Syntax for Delegating to a Subgenerator*, creates a syntax enhancement that allows multiple generator functions to be composed and refactored with the same ease that functions can be refactored. The regular `yield` statement is perfectly adequate for simply looping over values. However, refactoring it is difficult and less composable when using `send()` or `throw()` to direct data or exceptions to a subgenerator. The `yield from` construction effectively treats a subgenerator as if it were part of the parent generator code.

An effect of this new syntactic mechanism is that it is much easier to write coroutines (which still need a trampoline mechanism, such as that in `asyncio`) or cooperative micro-threads that want to communicate data among themselves.

# The asyncio Module

Building partially on `yield from`, `asyncio` provides infrastructure for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives. Quoting further from its official documentation, `asyncio` provides:

- A pluggable event loop with various system-specific implementations;

- Transport and protocol abstractions (similar to those in Twisted);

- Concrete support for TCP, UDP, SSL, subprocess pipes, delayed calls, and others (some may be system-dependent);

- A `Future` class that mimics the one in the concurrent.futures module, but adapted for use with the event loop;

- Coroutines and tasks based on yield from (PEP 380), to help write concurrent code in a sequential fashion;

- Cancellation support for Futures and coroutines;

- Synchronization primitives for use between coroutines in a single thread, mimicking those in the threading module;

- An interface for passing work off to a threadpool, for times when you absolutely, positively have to use a library that makes blocking I/O calls.

The event loop in `asyncio` is, at least in spirit, similar to what the *Twisted* framework provides, but it provides it with a standardized API and in the standard library.

# Views and Iterators Everywhere

Most everything that returns something sequence-like that can be an iterable rather than a concrete list has become so in Python 3.x. This often saves memory and improves performance of programs. For example:

- dict methods `dict.keys()`, `dict.items()`, and `dict.values()` return "views" instead of lists.
- `map()` and `filter()` return iterators. If you really need a list, a quick fix is `list(map(...))`, but a better fix is often to use a list comprehension `[...]`, or rewriting the code so it doesn't need a list at all. Particularly tricky is `map()` invoked for the side effects of the function; the correct transformation is to use a regular for loop.
- `range()` now behaves like `xrange()` used to behave, except it works with values of arbitrary size. The latter no longer exists.
- `zip()` now returns an iterator.

# Function Annotations

Snuck into Python 3.0, in 2006, was new syntax for adding annotations to function signatures. This allowed users to attach values to the arguments and return value of function definitions, but provided no specific guidance as to *what* values might be so attached syntactically. In some cases, these annotations were used for documentation, in some cases for typing information, in some cases for other purposes, but mostly they have just been ignored if not forgotten.

However, with PEP 484, entitled *Type Hints*, a canonical use for function annotations will be blessed. In Python 3.5+, when function

annotations are used, they will indicate information about the types of arguments and return values that *static type checkers* will be able to validate before programs even run. Moreover, the type system that is specified is a rich system that allows not just basic types, but nested and structured typing information similar to that found in research and pure functional languages. A bare-bones example could be a function like:

```python
def radius(c: complex) -> float:
    return sqrt(c.real**2 + c.imag**2)
```

Without explaining the full semantics here, a richer example could look like the following (based on a custom user class `Employee`):

```python
from typing import Union, Sequence
from mystuff import Employee
def handle_employees(e: Union[Employee, Sequence[Employee]]) -
> int:
    if isinstance(e, Employee):
        e = [e]
    ...
```

This capability only exists, at the time of this writing, in the third-party research project mypy, but including it in standard Python could substantially enhance the way developers use the language.

# Other Things

Several small specific changes to functions and methods improve usability.

## super() with No Arguments

Using `super()` in class definitions was confusing to get right in Python 2.x; PEP 3135 - New Super made this much easier:

> Replacing the old usage of super, calls to the next class in the MRO (method resolution order) can be made without explicitly passing the class object (although doing so will still be supported).

## The str.format() Mini-Language

Formatting of strings became more versatile with PEP 3101 — Advanced String Formatting. Technically this is also backported to Python 2.7.x, so it is not strictly a Python 3.x feature. The general idea is that you can use multiple curly-brace expresss inside a string, and apply the `str.format()` method to them with values to

interpolate. The formatting options are much more diverse and powerful than string interpolation with `%`. Here's a simple example: `"{0!r:20} world!".format("Hello")`. The string "Hello" will be inserted, with quotes, in a field at least 20 characters wide.

## The argparse Module

As with many other capabilities, the `argparse` module has been backported to Python 2.7.x, so is not technically Python 3.x only at this point. But it is certainly a nice improvement:

> A new module for command-line parsing, `argparse`, was introduced to overcome the limitations of `optparse` which did not provide support for positional arguments (not just options), subcommands, required options, and other common patterns of specifying and validating options.

## The itertools Module Keeps Getting Better

There are many powerful functions in `itertools` (which existed already in several Python 2.x minor versions), albeit with a philosophy of providing a strong, minimal set of functions, and leaving functions that can be composed of expressions involving several of these building blocks in recipes (sometimes in the official documentation). Some of the capabilities in the module have been backported to Python 2.7.x, but others remain Python 3.x only.

## pip Everywhere

The package management tool `pip` has been a popular third-party way of installing additional modules and packages for many Python versions. With Python 3.4+, it is included in every base Python distribution, hence making "more batteries" no more than one command away on any installation.

# Other Implementations

Several implementations of the Python interpreter exist other than the *CPython* implementation written and maintained by the Python Software Foundation (that is, actually written by volunteer core contributors, but with the intellectual property maintained by the PSF). The version of Python you can download directly from the python.org website—and the version included in most operating system distributions that have any version—is CPython, so named because it is written in the C programming language (although a lot of bootstrapping is done in Python itself, even for CPython).

## PyPy

A fascinating, arcane, and remarkable project is *PyPy*, an implementation of the Python interpreter written in … Python. Well, OK, technically it is written in a constrained subset of Python called *RPython*, but every RPython program is a Python program, in any case.

The PyPy website describes the project this way:

> PyPy is a fast, compliant alternative implementation of the Python language (2.7.8 and 3.2.5). It has several advantages and distinct features:
>
> - Speed: thanks to its Just-in-Time compiler, Python programs often run faster on PyPy.
>
> - Memory usage: memory-hungry Python programs (several hundreds of MBs or more) might end up taking less space than they do in CPython.

- Compatibility: PyPy is highly compatible with existing python code. It supports cffi and can run popular Python libraries like twisted and django.

- Sandboxing: PyPy provides the ability to run untrusted code in a fully secure way.

- Stackless: PyPy comes by default with support for stackless mode, providing micro-threads for massive concurrency.

PyPy is often *vastly faster* than CPython, especially in computational/numeric code (think 5 to 100 times faster, sometimes rivaling C or Fortran). It is stable and well researched. Some sticking points with library support remain, especially support libraries written in C.

There is active interaction between CPython core developers and PyPy developers, and often ideas from one source wind up influencing or being borrowed by the other. As well, the Python Software Foundation has, from time to time, funded specific efforts within PyPy development.

# PyPy-STM

It is a quirky bit of the culture of Python to speak of complex programming concepts or libraries as "melting your brain"—and no project has this said of it quite as often as does PyPy (for people who try to actually understand the internals; installing and running it is no harder than any other implementation). That is, nothing prior to *PyPy-STM* ever earned this honor.

PyPy-STM is just past experimental, but it promises possibilities of huge additional speedups over regular PyPy for high concurrency on multi-core systems. PyPy-STM is a version of PyPy that implements *Software Transactional Memory* (STM). According to Wikipedia:

> [STM is a] concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. It is an alternative to lock-based synchronization. […]

> STM is very optimistic: a thread completes modifications to shared memory without regard for what other threads might be doing, recording every read and write that it is performing in a log. […] the reader, […] after completing an entire transaction, verifies that other threads have not concurrently made changes to memory that it accessed in the past.

You can think of STM as speculative threading—i.e., perhaps among multiple cores, avoiding the global interpreter lock (GIL). According to the PyPy-STM docs:

> Threads execute their code speculatively, and at known points (e.g. between bytecodes) they coordinate with each other to agree on which order their respective actions should be "committed", i.e. become globally visible. Each duration of time between two commit-points is called a transaction.

The bottom line is that PyPy can be *a lot* faster than CPython, but cannot easily gain further from multiple cores on a CPU. After a roughly constant overhead hit, PyPy-STM concurrent performance starts to scale roughly linearly with the number of cores in performance:

| # of threads | PyPy (head) | PyPy-STM (2.3r2) |
| --- | --- | --- |
| N = 1 | real 0.92s | real 1.34s |
| N = 2 | real 1.77s | real 1.39s |
| N = 3 | real 2.57s | real 1.58s |
| N = 4 | real 3.38s | real 1.64s |

# Jython

*Jython* is an implementation of the Python programming language that is designed to run on the Java™ platform. It contains a compiler that transforms Python source code to Java bytecodes, which can then run on a JVM. It also includes Python standard library modules which are used by the compiled Java bytecodes. But of greatest relevance for users, Jython lets users import and use Java packages as if they were Python modules.

Jython is highly compatible with CPython when both rely on pure Python modules or standard library modules. The difference arises in that Jython can access all of the packages developers have written in Java or other JVM-compatible languages, whereas CPython can access C extension modules written for use with CPython.

Although JVMs use just-in-time compilation and other optimization techniques, Jython winds up with similar performance as

CPython overall, so unlike with PyPy there is no performance gain (nor any loss on average; obviously specific micro-benchmarks will vary, perhaps widely). Unfortunately, Jython so far only supports the Python 2.x series.

Here is a quick example of use, taken from the Jython documentation:

```
>>> from java.util import Random
>>> r = Random()
>>> r.nextInt()
501203849
```

Obviously, Python itself—including Jython—has its own `random` module with similar functionality. But the code sample illustrates the seamless use of Java packages in a simple case.

# IronPython

*IronPython* is a lot like Jython overall, merely substituting the .NET Framework and the CLR in place of Java packages and JVMs. Like Jython, IronPython is highly compatible with CPython (assuming pure Python or standard library modules are used), has similar overall performance, and is incompatible with C extension modules written for use with CPython.

IronPython is useful for Python programmers who want to access the various libraries written for the .NET Framework, whatever the supported language they were written in. Like Jython, however, it so far only supports the Python 2.x series.

Here is a similar quick example to the one given for Jython, taken from the IronPython documentation:

```
>>> from System.Collections.Generic import List, Dictionary
>>> int_list = List[int]()
>>> str_float_dict = Dictionary[str, float]()
```

In this case, Python has `list` and `dict`, but they are not generics in that they cannot be typed in manner shown in the example. Third parties may well have written analogous collections in Python, or as C extension modules, but they are not part of the standard library of CPython.

# Cython

*Cython* is an enhanced version of Python chiefly used for writing extension modules. It allows for optional type annotations to make large speed gains in numeric code (and in some other cases). The Cython website describes it like so:

> Cython is an optimising static compiler for both the Python programming language and the extended Cython programming language. […]

> The Cython language is a superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes. This allows the compiler to generate very efficient C code from Cython code. The C code is generated once and then compiles with all major C/C++ compilers in CPython 2.6, 2.7 (2.4+ with Cython 0.20.x) as well as 3.2 and all later versions.

Let us borrow a simple example from the Cython documentation:

> Consider the following pure Python code:

```python
def f(x):
    return x**2-x
def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

> Simply compiling this in Cython merely gives a 35% speedup. This is better than nothing, but adding some static types can make a much larger difference.

> With additional type declarations, this might look like:

```python
def f(double x):
    return x**2-x
def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

> Since the iterator variable i is typed with C semantics, the for-loop will be compiled to pure C code. Typing a, s and dx is important as they are involved in arithmetic within the for-loop; typing b and N

makes less of a difference, but in this case it is not much extra work to be consistent and type the entire function.

This results in a 4 times speedup over the pure Python version.

While you *could* make Cython your general Python interpreter, in practice it is used as an adjunct to CPython, with particular modules that are performance criticial being compiled as C extension modules. Cython modules use the extension `.pyx`, but produce an intermediate `.c` file that is compiled to an actual `.so` or `.pyd` module. Hence the result genuinely *is* a C extension module, albeit one whose code was auto-generated from a Python superset that is likely to be far more readable than writing in C to start with.

# Numba

*Numba* is not technically a Python implementation, but rather "merely" a C extension module for CPython. However, what it does is akin to a mixture of PyPy and Cython. Without any semantic changes, importing and using the decorators supplied by `numba` causes just-in-time compilation and optimization of code on a per-function basis, substituting a fast machine code path for CPython interpretation of the decorated function. Like PyPy, this gives you dynamic compilation, type inference, and just-in-time optimization; like Cython, you can also annotate types explicitly where they are known in advance. Moreover, Numba "plays well" with NumPy, so using both together can produce extremely fast execution.

Quoting from the Numba documentation:

Numba gives you the power to speed up your applications with high performance functions written directly in Python.

Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba's main features are:

- on-the-fly code generation (at import time or runtime, at the user's preference)
- native code generation for the CPU (default) and GPU hardware integration with the Python scientific software stack (thanks to Numpy)

Here is how a Numba-optimized function, taking a Numpy array as argument, might look like:

```python
@numba.jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result
```

[…] You can also tell Numba the function signature you are expecting. The function f() would now look like:

```python
from numba import jit, int32
@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

# Python Distributions

Several companies provide Python distributions with "even more batteries included."

## Continuum Analytics' Anaconda

Continuum Analytics produces a Python distribution (for Python 2.7 and Python 3.4 currently) called Anaconda that comes bundled with "195+ of the most popular Python packages for science, math, engineering, and data analysis." It also uses its own package manager called `conda` that has some advantages over `pip` and the PyPI ecosystem. In particular, this distribution comes bundled with NumPy, SciPy, Pandas, IPython, Matplotlib, Numba, Blaze, and Bokeh.

While the basic distribution is free to download, and free even to redistribute, Continuum Analytics offers several different enhanced versions of the distribution for a cost. One of the most interesting of these commercial enhancements is well integrated with CUDA, allowing extremely fast operations on numeric arrays by utilizing GPUs. For an announcement about this technology, bringing Python even further into the world of high-performance scientific computing, see "NVIDIA and Continuum Analytics Announce NumbaPro, A Python CUDA Compiler".

## Enthought's Canopy

Enthought produces a Python distribution with similar goals to those of Continuum Analytics—indeed, the latter company is largely an offshoot of Enthought, and both share a goal of making Python

the default language for scientific computing. Enthought describes their distribution as:

> Enthought Canopy is a comprehensive Python analysis environment that provides easy installation of the core scientific analytic and scientific Python packages, creating a robust platform you can explore, develop, and visualize on. In addition to its pre-built, tested Python distribution, Enthought Canopy has valuable tools for iterative data analysis, visualization and application development.

As with Anaconda, Canopy has a basic free version and several enhanced versions for additional cost. Enthought's focus in its enhanced distributions is more on integration with Python training than with performance enhancements per se.

## ActiveState's ActivePython

ActiveState produces a Python distribution that bundles a wide range of Python versions and a good collection of external libraries —more business-oriented than scientific in contrast to the other distributions mentioned—and that also comes with its own enhanced package manager.

As with other companies producing distributions, ActiveState has several options for enhanced distributions that are available at additional cost. In the case of ActivePython, the extra cost pays for support guarantees (email, phone, etc.).

# About the Author

**David Mertz** is a director of the PSF, and chair of its Trademarks Committee and Outreach & Education Committee. He wrote the columns *Charming Python* and *XML Matters* for IBM developer-Works and the Addison-Wesley book *Text Processing in Python*, has spoken at multiple OSCONs and PyCons, and was invited to be a keynote speaker at PyCon India, PyCon UK, PyCon ZA, and PyCon Belarus.

In the distant past, David spent some time as a university professor, teaching in areas far removed from computer programming, but gained some familiarity with the vicissitudes of pedagogy.

Since 2008, David has worked with folks who have built the world's fastest supercomputer for performing molecular dynamics. He is pleased to find Python becoming the default high-level language for most scientific computing projects.