ncoghlan$_{d}ev's Python Notes$

Release 1.0

Nick Coghlan

Jul 18, 2017

Contents

Contents:

CHAPTER 1

Python 3

At the instigation of Guido van Rossum (the Benevolent Dictator for Life for the Python language specification and the CPython reference implementation), the Python ecosystem is currently undertaking a transition from the Python 2 series to the infamously incompatible Python 3 series.

This page collects a series of essays I have written covering various aspects of the Python 3 transition (as well as Python 3 features in general).

Python 3 Q & A

Published 29th June, 2012

Last Updated 22nd April, 2017

Throughout the long transition to "Python 3 by default" in the Python ecosystem, the question was occasionally raised as to whether or not the core Python developers were acting as reasonable stewards of the Python language.

While it largely stopped being a concern after the release of Python 3.5 in September 2015, it was an entirely appropriate question prior to that, as Python 3 introduced backwards incompatible changes that more obviously helped future users of the language than they did current users, so existing users (especially library and framework developers) were being asked to devote time and effort to a transition that would cost them more in time and energy in the near term than it would save them for years to come.

Since I had seen variants of these questions several times over the years, I started this FAQ as an intermittently updated record of my thoughts on the topic, with updates generally being prompted by new iterations of the questions. You can see the full history of changes in the source repo.

The views expressed below are my own. While many of them are shared by other core developers, and I use "we" in several places where I believe that to be the case, I don't claim to be writing on the behalf of every core developer on every point. Several core developers (including Guido) *have* reviewed and offered comments on this document at various points in time, and aside from Guido noting that I was incorrect about his initial motivation in creating Python 3, none of them has raised any objections to specific points or the document in general.

I am also not writing on behalf of the Python Software Foundation (of which I am a nominated Fellow) nor on behalf of Red Hat (my current employer). However, I do use several Red Hat specific examples when discussing enterprise

perception and adoption of the Python platform - effectively bridging that gap between early adopters and the vast majority of prospective platform users is kinda what Red Hat specialises in, so I consider them an important measure of the inroads Python 3 is making into more conservative development communities.

There were several extensive discussions of the state of the Python 3 transition at PyCon US 2014 in Montreal, starting at the language summit, and continuing throughout the conference. These helped clarify many of the remaining points of contention, and resulted in a range of changes to Python 3.5, Python 2.7, and the available tools to support forward migration from Python 2 to Python 3. These discussions didn't stop, but have rather continued over the course of Python development, and can be expected to continue for as long as folks are developing software that either fits into the common subset of Python 2 & 3, or else are having to maintain software that continues to run solely under Python 2.

Note: If anyone is interested in writing about these issues in more formal media, please get in touch to check if particular answers are still accurate. Not only have the updates over the years been intermittent, they've also been less than completely comprehensive, so some answers may refer out to experiments that ultimately proved uninteresting or unsuccessful, or otherwise be out of date.

As with all essays on these pages, feedback is welcome via the issue tracker or Twitter.

TL;DR Version

- Yes, we know this migration was/is disruptive.
- Yes, we know that some sections of the community had never personally experienced the problems with the Python 2 Unicode model that this migration was designed to eliminate, or otherwise preferred the closer alignment between the Python 2 text model and the POSIX text model.
- Yes, we know that many of those problems had already been solved by some sections of the community to their own satisfaction.
- Yes, we know that by attempting to fix these problems in the core Unicode model we broke many of the workarounds that had been put in place to deal with the limitations of the old model
- Yes, we are trying to ensure there is a smooth migration path from Python 2 to Python 3 to minimise the inevitable disruption
- Yes, we know some members of the community would have liked the migration to move faster and found the "gently, gently, there's no rush" approach of the core development team frustrating
- No, we did not do this lightly
- No, we did not see any other way to ensure Python remained a viable development platform as developer communities grow in locations where English is not the primary spoken language. It should be at least possible for users to start learning the basics of Python without having to first learn English as a prerequisite (even if English remains a requirement for full participation in the global Python and open source ecosystems).

It is my perspective that the web and GUI developers have the right idea: dealing with Unicode text correctly is not optional in the modern world. In large part, the Python 3 redesign involved taking Unicode handling principles elaborated in those parts of the community and building them into the core design of the language.

Why was Python 3 made incompatible with Python 2?

According to Guido, he initiated the Python 3 project to clean up a variety of issues with Python 2 where he didn't feel comfortable with fixing them through the normal deprecation process. This included the removal of classic classes, changing integer division to automatically promote to a floating point result (retaining the separate floor division operation) and changing the core string type to be based on Unicode by default. With a compatibility break taking

place anyway, the case was made to just include some other changes in that process (like converting print to a function), rather than going through the full deprecation process within the Python 2 series.

If it had just been about minor cleanups, the transition would likely have been more straightforward, but also less beneficial. However, the changes to the text model in Python 3 are one of those ideas that has profoundly changed the way I think about software, and we receive similar feedback from many other users that never really understood how Unicode worked in Python 2, but were able to grasp it far more easily in Python 3. Redesigning the way the Python builtin types model binary and text data has the ultimate aim of helping *all* Python applications (including the standard library itself) to handle Unicode text in a more consistent and reliable fashion (I originally had "without needing to rely on third party libraries and frameworks" here, but those are still generally needed to handle system boundaries correctly, even in Python 3).

Note: For a more complete version of this answer that places it in the wider industry context of Unicode adoption, see this article of mine on the Red Hat Developer Blog: The Transition to Multilingual Programming with Python

I also gave a presentation on the topic at Python Australia 2015, which is available online here

---

The core Unicode support in the Python 2 series has the honour of being documented in PEP 100. It was created as Misc/unicode.txt in March 2000 (before the PEP process even existed) to integrate Unicode 3.0 support into Python 2.0. Once the PEP process was defined, it was deemed more appropriate to capture these details as an informational PEP.

Guido, along with the wider Python and software development communities, learned a lot about the best techniques for handling Unicode in the six years between the introduction of Unicode support in Python 2.0 and inauguration of the python-3000 mailing list in March 2006.

One of the most important guidelines for good Unicode handling is to ensure that all encoding and decoding occurs at system boundaries, with all internal text processing operating solely on Unicode data. The Python 2 Unicode model is essentially the POSIX text model with Unicode support bolted on to the side, so it doesn't follow that guideline: it allows implicit decoding at almost any point where an 8-bit string encounters a Unicode string, along with implicit encoding at almost any location where an 8-bit string is needed but a Unicode string is provided.

One reason this approach is problematic is that it means the traceback for an unexpected `UnicodeDecodeError` or `UnicodeEncodeError` in a large Python 2.x code base almost *never* points you to the code that is broken. Instead, you have to trace the origins of the *data* in the failing operation, and try to figure out where the unexpected 8-bit or Unicode code string was introduced. By contrast, Python 3 is designed to fail fast in most situations: when a `UnicodeError` of any kind occurs, it is more likely that the problem actually does lie somewhere close to the operation that failed. In those cases where Python 3 doesn't fail fast, it's because it is designed to "round trip" - so long as the output encoding matches the input encoding (even if it turns out the data isn't properly encoded according to that encoding), Python 3 will aim to faithfully reproduce the input byte sequence as the output byte sequence.

The implicit nature of the conversions in Python 2 also means that encoding operations may raise decoding errors and vice-versa, depending on the input types and the codecs involved.

A more pernicious problem arises when Python 2 *doesn't* throw an exception at all - this problem occurs when two 8-bit strings with data in different text encodings are concatenated or otherwise combined. The result is invalid data, but Python will happily pass it on to other applications in its corrupted form. Python 3 isn't completely immune to this problem, but it should arise in substantially fewer cases.

The general guiding philosophy of the text model in Python 3 is essentially:

- try to do the right thing by default
- if we can't figure out the right thing to do, throw an exception
- as far as is practical, always require users to opt in to behaviours that pose a significant risk of silently corrupting data in non-ASCII compatible encodings

Ned Batchelder's wonderful Pragmatic Unicode talk/essay could just as well be titled "This is why Python 3 exists". There are a large number of Unicode handling bugs in the Python 2 standard library that have not been, and will not be, fixed, as fixing them within the constraints of the Python 2 text model is considered too hard to be worth the effort (to put that effort into context: if you judge the core development team by our *actions* it is clear that we consider that creating and promoting Python 3 was an *easier* and *more pleasant* alternative to attempting to fix those issues while abiding by Python 2's backwards compatibility requirements).

The revised text model in Python 3 also means that the *primary* string type is now fully Unicode capable. This brings Python closer to the model used in the JVM, Android, .NET CLR, and Unicode capable Windows APIs. One key consequence of this is that the interpreter core in Python 3 is far more tolerant of paths that contain Unicode characters on Windows (so, for example, having a non-ASCII character in your username should no longer cause any problems with running Python scripts from your home directory on Windows). The `surrogateescape` error handler added in **PEP 383** is designed to bridge the gap between the new text model in Python 3 and the possibility of receiving data through bytes oriented APIs on POSIX systems where the declared system encoding doesn't match the encoding of the data itself. That error handler is also useful in other cases where applications need to tolerate mismatches between declared encodings and actual data - while it does share some of the problems of the Python 2 Unicode model, it at least has the virtue of only causing problems in the case of errors either in the input data or the declared encoding, where Python 2 could get into trouble in the presence of multiple data sources with *different* encodings, even if all the input was correctly encoded in its declared encoding.

Python 3 also embeds Unicode support more deeply into the language itself. With the primary string type handling the full Unicode range, it became practical to make UTF-8 the default source encoding (instead of ASCII) and adjust many parts of the language that were previously restricted to ASCII text (such as identifiers) to now permit a much wider range of Unicode characters. This permits developers with a native language other than English to use names in their own language rather than being forced to use names that fit within the ASCII character set. Some areas of the interpreter that were previously fragile in the face of Unicode text (such as displaying exception tracebacks) are also far more robust in Python 3.

Removing the implicit type conversions entirely also made it more practical to implement the new internal Unicode data model for Python 3.3, where the internal representation of Unicode strings is automatically adjusted based on the highest value code point that needs to be stored (see PEP 393 for details).

What actually changed in the text model between Python 2 and Python 3?

The Python 2 core text model looks like this:

- `str`: 8-bit type containing binary data, or encoded text data in an unknown (hopefully ASCII compatible) encoding, represented as length 1 8-bit strings
- `unicode`: 16-bit or 32-bit type (depending on build options) containing Unicode code points, represented as length 1 Unicode strings

That first type is essentially the way POSIX systems model text data, so it is incredibly convenient for interfacing with POSIX environments, since it lets you just copy bits around without worrying about their encoding. It is also useful for dealing with the ASCII compatible segments that are part of many binary protocols.

The conceptual problem with this model is that it is an appropriate model for *boundary* code - the kind of code that handles the transformation between wire protocols and file formats (which are always a series of bytes), and the more structured data types actually manipulated by applications (which may include opaque binary blobs, but are more typically things like text, numbers and containers).

Actual *applications* shouldn't be manipulating values that "might be text, might be arbitrary binary data". In particular, manipulating text values as binary data in multiple different text encodings can easily cause a problem the Japanese named "mojibake": binary data that includes text in multiple encodings, but with no clear structure that defines which parts are in which encoding.

Unfortunately, Python 2 uses a type with exactly those semantics as its core string type, permits silent promotion from the "might be binary data" type to the "is definitely text" type and provides little support for accounting for encoding

differences.

So Python 3 changes the core text model to be one that is more appropriate for *application* code rather than boundary code:

- `str`: a sequence of Unicode code points, represented as length 1 strings (always contains text data)
- `bytes`: a sequence of integers between 0 and 255 inclusive (always contains arbitrary binary data). While it still has many operations that are designed to make it convenient to work on ASCII compatible segments in binary data formats, it *is not* implicitly interoperable with the `str` type.

The hybrid "might be encoded text, might be arbitrary binary data, can interoperate with both other instances of str and also with instances of unicode" type was *deliberately* removed from the core text model because using the same type for multiple distinct purposes makes it incredibly difficult to reason about correctly. The core model in Python 3 opts to handle the "arbitrary binary data" case and the "ASCII compatible segments in binary data formats" case, leaving the direct manipulation of encoded text to a (currently still hypothetical) third party type (due to the many issues that approach poses when dealing with multibyte and variable width text encodings).

The purpose of boundary code is then to hammer whatever comes in over the wire or is available on disk into a format suitable for passing on to application code.

Unfortunately, there have turned out to be some key challenges in making this model pervasive in Python 3:

- the same design changes that improve Python 3's Windows integration by changing several OS interfaces to operate on text rather than binary data also make it more sensitive to locale misconfiguration issues on POSIX operating systems other than Mac OS X. In Python 2, text is always sent and received from POSIX operating system interfaces as *binary* data, and the associated decoding and encoding operations are fully under the control of the application. In Python 3, the interpreter aims to handle these operations automatically, but in releases up to and including Python 3.6 it needs to rely on the default settings in the OS provided locale module to handle the conversion, making it potentially sensitive to configuration issues that many Python 2 applications could ignore. Most notably, if the OS erroneously claims that "ascii" is a suitable encoding to use for operating system interfaces (as happens by default in a number of cases, due to the formal definition of the ANSI C locale predating the invention of UTF-8 by a few years), the Python 3 interpreter will believe it, and will complain if asked to handle non-ASCII data. **PEP 538** and **PEP 540** offer some possible improvements in this area (by assuming UTF-8 as the preferred text encoding when running in the default `C` locale), but it isn't a trivial fix due to the phase of the interpreter startup sequence where the problem occurs. (Thanks go to Armin Ronacher for clearly articulating many of these details - see his write-up in the click documentation)
- when migrating libraries and frameworks from Python 2 to Python 3 that handle boundary API problems, the lack of the hybrid "might be text, might be arbitrary bytes" type can be keenly felt, as the implicitly interoperable type was essential to being able to cleanly share code between the two modes of operation. This usually isn't a major problem for *new* Python 3 code - such code is typically designed to operate in the binary domain (perhaps relying on the methods for working with ASCII compatible segments), the text domain, or to handle a transition between them. However, code being ported from Python 2 may need to continue to implement hybrid APIs in order to accommodate users that make different decisions regarding whether to operate in the binary domain or the text domain in Python 3 - because Python 2 blurred the distinction, different users will make different choices, and third party libraries and frameworks may need to account for that rather than forcing a particular answer for all users.
- in the initial Python 3 design, interpolation of variables into a format string was treated solely as a text domain operation. While this proved to be a reasonable design decision for the flexible Python-specific `str.format` operation, **PEP 461** restored printf-style interpolation for ASCII compatible segments in binary data in Python 3.5. Prior to that change, the lack of this feature could sometimes be an irritation when working extensively in Python 3 with wire protocols and file formats that include ASCII compatible segments.
- while the API design of the `str` type in Python 3 was based directly on the `unicode` type in Python 2, the `bytes` type doesn't have such a clean heritage. Instead, it evolved over the course of the initial Python 3 pre-release design period, starting from a model where the *only* type for binary data handling was the type now called `bytearray`. That type was modelled directly on the `array.array('B')` type, and hence

produced integers when iterating over it or indexing into it. During the pre-release design period, the lack of an immutable binary data type was identified as a problem, and the (then mutable) `bytes` type was renamed to `bytearray` and a new immutable `bytes` type added. The now familiar "bytes literal" syntax was introduced (prepending a "b" prefix to the string literal syntax) and the representations of the two types were also adjusted to be based on the new bytes literal syntax. With the benefit of hindsight, it has become clear another change should have been made at the same time: with so many affordances switched back to matching those of the Python 2 `str` type (including the use of the new bytes literal syntax to refer to that type in Python 2.6 and 2.7), `bytes` and `bytearray` should have been been switched away from behaving like a tuple of integers and list of integers (respectively) and instead modified to be containers of length 1 `bytes` objects, just as the `str` type is a container of length 1 `str` objects. Unfortunately, that change was not made at the time, and now backwards compatibility constraints within the Python 3 series itself makes it highly unlikely the behaviour will be changed in the future either. **PEP 467** covers a number of other still visible remnants of this convoluted design history that are more amenable to being addressed within the constraints of Python's normal Python deprecation processes.

These changes are a key source of friction when it comes to Python 3 between the Python core developers and other experts that had fully mastered the Python 2 text model, especially those that focus on targeting POSIX platforms rather than Windows or the JVM, as well as those that focus on writing boundary code, such as networking libraries, web frameworks and file format parsers and generators. These developers bore a lot of the burden of adjusting to these changes on behalf of their users, often while gaining few or none of the benefits.

That said, while these issues certainly aren't ideal, they also won't impact many users that are relying on libraries and frameworks to deal with boundary issues, and can afford to ignore possible misbehaviour in misconfigured POSIX environments. As Python 3 has matured as a platform, most of those areas where it has regressed in suitability relative to Python 2 have been addressed. In particular, the ongoing migrations of Linux distribution utilities from Python 2 to Python 3 have seen many of the platform integration issues on POSIX systems dealt with in a cleaner fashion. The tuple-of-ints and list-of-ints behaviour of `bytes` and `bytearray` is unlikely to change, but proposals like **PEP 467** may bring better tools for dealing with them.

Why not just assume UTF-8 and avoid having to decode at system boundaries?

The design decision to go with a fixed width Unicode representation both externally and internally has a long history in Python, going all the way back to the addition of Python's original Unicode support in Python 2.0. Using a fixed width type at that point meant that many of the algorithms could be shared between the original 8-bit `str` type and the new 16-or-32-bit `unicode` type. (Note that adoption of this particular approach predates my own involvement in CPython core development - as with many other aspects of CPython's text handling support, it's something I've learned about while helping with the transition to pervasive Unicode support in the standard library and elsewhere for Python 3).

That design meant that, historically, CPython builds had to choose what size to use for the internal representation of Unicode text. We always chose to use "narrow" builds for the Windows binary installers published on python.org, as the UTF-16 internal representation was the best fit for the Windows text handling APIs.

Linux distributions, by contrast, almost all chose the memory hungry "wide" builds that allocated 32 bits per Unicode code point in Python 2 `unicode` objects and Python 3 `str` objects (up to & including Python 3.2), even for pure ASCII text. There's a reason they went for that option, though: it was better at handling Unicode code points outside the basic multilingual plane. In narrow builds the UTF-16 code points were exposed directly in both the C API and the Python API of the `unicode` type, and hence were prone to bugs related to incorrect handling of code points greater than 65,535 in code that assumed a one-to-one correspondence between Python code points and Unicode code points. This wasn't generally a big deal when code points in common use all tended to fit in the BMP, but started to become more problematic as things like mathematical and musical notation, ancient langages, emoticons and additional CJK ideographs were added. Given the choice between greater memory efficiency and correctness, the Linux distributions chose correctness, imposing a non-trivial memory usage penalty on Unicode heavy applications that couldn't rely entirely on `str` objects in Python 2 or `bytes` and `bytearray` objects in Python 3. Those larger strings also came

at a cost in speed, since they not only meant having more data to move around relative to narrow builds (or applications that only allowed 8-bit text), but the larger memory footprint also made CPU caches less effective.

When it came to the design of the C level text representation for Python 3, the existing Python 2 Unicode design wasn't up for reconsideration - the Python 2 `unicode` type was mapped directly to the Python 3 `str` type. This is most obvious in the Python 3 C API, which still uses the same `PyUnicode_*` prefix for text manipulation APIs, as that was the easiest way to preserve compatibility with C extensions that were originally written against Python 2.

However, removing the intertwining of the 8-bit str type and the unicode type that existed in Python 2 paved the way for eliminating the narrow vs wide build distinction in Python 3.3, and eliminating a significant portion of the memory cost associated with getting correct Unicode handling in earlier versions of Python. As a result of **PEP 393**, strings that consist solely of latin-1 or UCS2 code points in Python 3.3+ are able to use 8 or 16 bits per code point (as appropriate), while still being able to use string manipulation algorithms that rely on the assumption of consistent code point sizes within a given string. As with the original Python 3 implementation, there were also a large number of constraints imposed on this redesign of the internal representation based on the public C API, and that is reflected in some of the more complicated aspects of the PEP.

While it's theoretically possible to write string manipulation algorithms that work correctly with variable width encodings (potentially saving even more memory), it isn't *easy* to do so, and for cross-platform runtimes that interoperate closely with the underlying operating system the way CPython does, there isn't an obvious universally correct choice even today, let alone back in 2006 when Guido first started the Python 3 project. UTF-8 comes closest (hence the wording of this question), but it still poses risks of silent data corruption on Linux if you don't explicitly transcode data at system boundaries (particularly if the actual encoding of metadata provided by the system is ASCII incompatible, as can happen in East Asian countries using encodings like Shift-JIS and GB-18030) and still requires transcoding between UTF-16-LE and UTF-8 on Windows (the bytes-oriented APIs on Windows are generally restricted to the `mbcs` encoding, making them effectively useless for proper Unicode handling - it's necessary to switch to the Windows specific UTF-16 based APIs to make things work properly).

The Python 3 text model also trades additional memory usage for encoding and decoding speed in some cases, including caching the UTF-8 representation of a string when appropriate. In addition to UTF-8, other key codecs like ASCII, latin-1, UTF-16 and UTF-32 are closely integrated with the core text implementation in order to make them as efficient as is practical.

The current Python 3 text model certainly has its challenges, especially around Linux compatibility (see **PEP 383** for an example of the complexity associated with that problem), but those are considered the lesser evil when compared to the alternative of breaking C extension compatibility and having to rewrite all the string manipulation algorithms to handle a variable width internal encoding, while still facing significant integration challenges on both Windows and Linux. Instead of anyone pursuing such a drastic change, I expect the remaining Linux integration issues for the existing model to be resolved as we help Linux distributions like Ubuntu and Fedora migrate their system services to Python 3 (in the specific case of Fedora, that migration encompasses both the operating system installer *and* the package manager).

Still, for new runtimes invented today, particularly those aimed primarily at new server applications running on Linux that can afford to ignore the integration challenges that arise on Windows and older Linux systems using encodings other than UTF-8, using UTF-8 for their internal string representation makes a lot of sense. It's just best to avoid exposing the raw binary representation of text data for direct manipulation in user code: experience has shown that a Unicode code point based abstraction is much easier to work with, even if it means opting out of providing O(1) indexing for arbitrary code points in a string to avoid allocating additional memory per code point based on the largest code point in the string. For new languages that are specifically designed to accommodate a variable width internal encoding for text, a file-like opaque token based seek/tell style API is likely to be more appropriate for random access to strings than a Python style integer based indexing API. The kind of internal flexibility offered by the latter approach can be seen in Python's own `io.StringIO` implementation - in Python 3.4+, that aims to delay creation of a full string object for as long as possible, an optimisation that could be implemented transparently due to the file-like API that type exports.

Note: Python 3 does assume UTF-8 at system boundaries on Mac OS X, since that OS ensures that the assumption

will almost always be correct. Starting with Python 3.6, CPython on Windows also assumes that binary data passed to operating system interfaces is in UTF-8 and transcodes it to UTF-16-LE before passing it to the relevant Windows APIs.

For Python 3.7, **PEP 538** and **PEP 540** are likely to extend the UTF-8 assumption to the default `C` locale more generally (so other system encodings will still be supported through the locale system, but the problematic ASCII default will be largely ignored).

OK, that explains Unicode, but what about all the other incompatible changes?

The other backwards incompatible changes in Python 3 largely fell into the following categories:

- dropping deprecated features that were frequent sources of bugs in Python 2, or had been replaced by superior alternatives and retained solely for backwards compatibility
- reducing the number of statements in the language
- replacing concrete list and dict objects with more memory efficient alternatives
- renaming modules to be more PEP 8 compliant and to automatically use C accelerators when available

The first of those were aimed at making the language easier to learn, and easier to maintain. Keeping deprecated features around isn't free: in order to maintain code that uses those features, everyone needs to remember them and new developers need to be taught them. Python 2 had acquired a lot of quirks over the years, and the 3.x series allowed such design mistakes to be corrected.

While there were advantages to having `print` and `exec` as statements, they introduced a sharp discontinuity when switching from the statement forms to any other alternative approach (such as changing `print` to `logging.debug` or `exec` to `execfile`), and also required the use of awkward hacks to cope with the fact that they couldn't accept keyword arguments. For Python 3, they were demoted to builtin functions in order to remove that discontinuity and to exploit the benefits of keyword only parameters.

The increased use of iterators and views was motivated by the fact that many of Python's core APIs were designed *before* the introduction of the iterator protocol. That meant a lot unnecessary lists were being created when more memory efficient alternatives were now possible. We didn't get them all (you'll still find APIs that unnecessarily return concrete lists and dictionaries in various parts of the standard library), but the core APIs are all now significantly more memory efficient by default.

As with the removal of deprecated features, the various renaming operations were designed to make the language smaller and easier to learn. Names that don't follow standard conventions need to be remembered as special cases, while those that follow a pattern can be derived just be remembering the pattern. Using the API compatible C accelerators automatically also means that end users no longer need to know about and explicitly request the accelerated variant, and alternative implementations don't need to provide the modules under two different names.

No backwards incompatible changes were made just for the sake of making them. Each one was justified (at least at the time) on the basis of making the language either easier to learn or easier to use.

With the benefit of hindsight, a number of these other changes would probably have been better avoided (especially some of the renaming ones), but even those cases at least seemed like a good idea at the time. At this point, internal backwards compatibility requirements within the Python 3.x series mean it isn't worth the hassle of changing them back, especially given the existence of the six compatibility project and other third party modules that support both Python 2 and Python 3 (for example, the `requests` package is an excellent alternative to using the low level `urllib` interfaces directly, even though `six` does provide appropriate cross-version compatible access through the `six.moves.urllib` namespace).

What other notable changes in Python 3 depend on the text model change?

One of the consequences of the intertwined implementations of the `str` and `unicode` types in Python 2 is that it made it difficult to update them to correctly interoperate with anything *else*. The dual type text model also made it quite difficult to add Unicode support to various APIs that previously didn't support it.

This isn't an exhaustive list, but here are several of the enhancements in Python 3 that would likely be prohibitively difficult to backport to Python 2 (even when they're technically backwards compatible):

- **PEP 393** (more efficient text storage in memory)
- Unicode identifier support
- full Unicode module name support
- improvements in Unicode path handling on Windows
- multiple other improvements in Unicode handling when interfacing with Windows APIs
- more robust and user friendly handling of Unicode characters in object representations and when displaying exceptions
- increased consistency in Unicode handling in files and at the interactive prompt (although the C locale on POSIX systems still triggers undesirable behaviour in Python 3)
- greater functional separation between text encodings and other codecs, including tailored exceptions nudging users towards the more generic APIs when needed (this change in Python 3.4 also eliminates certain classes of remote DOS attack targeted at the compression codecs in the codec machinery when using the convenience methods on the core types rather than the unrestricted interfaces in the codecs module)
- using the new IO model (with automatic encoding and decoding support) by default

What are (or were) some of the key dates in the Python 3 transition?

Note: This list is rather incomplete and I'm unlikely to find the time to complete it - if anyone is curious enough to put together a more comprehensive timeline, feel free to use this answer as a starting point, or else just send a PR to add more entries to this list.

At least the following events should be included in a more complete list:

- NumpPy 1.5.0 and SciPy 0.9.0 (these added Python 3 support)
- matplotlib Python 3 support
- IPython Python 3 support
- Cython Python 3 support
- SWIG Python 3 support
- links for the Ubuntu, Fedora and openSUSE "Python 3 as default" migration plans
- SQL Alchemy Python 3 support
- pytz Python 3 support
- PyOpenSSL support
- mod_wsgi Python 3 support (first 3.x WSGI implementation)
- Tornado Python 3 support (first 3.x async web server)
- Twisted Python 3 support (most comprehensive network protocol support)

- Pyramid Python 3 support (first major 3.x compatible web framework)
- Django 1.5 and 1.6 (experimental and stable Python 3 support)
- Werkzeug and Flask Python 3 support
- requests Python 3 support
- pyside Python 3 support (first Python 3.x Qt bindings)
- pygtk and/or pygobject Python support
- wxPython phoenix project
- VTK Python 3 support in August 2015 (blocked Mayavi, which blocked Canopy)
- cx-Freeze Python 3 support
- setuptools and pip Python 3 support
- Pillow (PIL fork) Python 3 support
- greenlet Python 3 support
- pylint Python 3 support
- nose2 Python 3 support
- pytest Python 3 support
- Editor/IDE support for Python 3 in: PyDev, Spyder, Python Tools for Visual Studio, PyCharm, WingIDE, Komodo (others?)
- Embedded Python 3 support in: Blender, Kate, vim, gdb, gcc, LibreOffice (others?)
- version availability in services like Google DataLab and Azure Notebooks
- Python 3 availability in Heroku
- availability in the major Chinese public cloud platforms (Alibaba/Aliyun, Tencent Qcloud, Huawei Enterprise Cloud, etc)
- the day any bar on https://python3wos.appspot.com/ or wedge on http://py3readiness.org/ turned green was potentially a significant step for some subsection of the community :)

2006

March 2006: Guido van Rossum (the original creator of Python and hence Python's Benevolent Dictator for Life), with financial support from Google, took the previously hypothetical "Python 3000" project and turned it into an active development project, aiming to create an updated Python language definition and reference interpreter implementation that addressed some fundamental limitations in the ability of the Python 2 reference interpreter to correctly handle non-ASCII text. (The project actually started earlier than this - March 2006 was when the python-3000 list was created to separate out the longer term Python 3 discussions from the active preparation for the Python 2.5 final release)

April 2006: Guido published **PEP 3000**, laying the ground rules for Python 3 development, and detailing the proposed migration strategy for Python 2 projects (the recommended porting approach has changed substantially since then, see *What other changes have occurred that simplify migration?* for more details). **PEP 3100** describes several of the overall goals of the project, and lists many smaller changes that weren't covered by their own PEPs. **PEP 3099** covers a number of proposed changes that were explicitly declared out of scope of the Python 3000 project.

At this point in time, Python 2 and Python 3 started being developed in parallel by the core development team for the reference interpreter.

2007

August 2007: The first alpha release of Python 3.0 was published.

2008

February 2008: The first alpha release of Python 2.6 was published alongside the third alpha of Python 3.0. The release schedules for both Python 2.6 and 3.0 are covered in **PEP 361**.

October 2008: Python 2.6 was published, including the backwards compatible features defined for Python 3.0, along with a number of `__future__` imports and the `-3` switch to help make it practical to add Python 3 support to existing Python 2 software (or to migrate entirely from Python 2 to Python 3). While Python 2.6 received its final upstream security update in October 2013, maintenance & support remains available through some commercial redistributors.

December 2008: In a fit of misguided optimism, Python 3.0 was published with an unusably slow pure Python IO implementation - it worked tolerably well for small data sets, but was entirely impractical for handling realistic workloads on the CPython reference interpreter. (Python 3.0 received a single maintenance release, but was otherwise entirely superceded by the release of Python 3.1)

ActiveState became the first company I am aware of to start offering commercial Python 3 support by shipping ActivePython 3.0 almost immediately after the upstream release was published. They have subsequently continued this trend of closely following upstream Python 3 releases.

2009

March 2009: The first alpha release of Python 3.1, with an updated C accelerated IO stack, was published. **PEP 375** covers the details of the Python 3.1 release cycle.

June 2009: Python 3.1 final was published, providing the first version of the Python 3 runtime that was genuinely usable for realistic workloads. Python 3.1 received its final security update in April 2012, and even commercial support for this version is no longer available.

October 2009: **PEP 3003** was published, declaring a moraratorium on language level changes in Python 2.7 and Python 3.2. This was done to deliberately slow down the pace of core development for a couple of years, with additional effort focused on standard library improvements (as well as some improvements to the builtin types).

December 2009: The first alpha of Python 2.7 was published. **PEP 373** covers the details of the Python 2.7 release cycle.

2010

July 2010: Python 2.7 final was published, providing many of the backwards compatible features added in the Python 3.1 and 3.2 releases. Python 2.7 is currently still fully supported by the core development team and will continue receiving maintenance & security updates until at least January 2020.

Once the Python 2.7 maintenance branch was created, the py3k development branch was retired: for the first time, the default branch in the main CPython repo was the upcoming version of Python 3.

August 2010: The first alpha of Python 3.2 was published. **PEP 392** covers the details of the Python 3.2 release cycle. Python 3.2 restored preliminary support for the binary and text transform codecs that had been removed in Python 3.0.

October 2010: **PEP 3333** was published to define WSGI 1.1, a Python 3 compatible version of the Python Web Server Gateway Interface.

2011

February 2011: Python 3.2 final was published, providing the first version of Python 3 with support for the Web Server Gateway Interface. Python 3.2 received its final security update in February 2016, and even commercial support for this version is no longer available.

March 2011: After Arch Linux updated their Python symlink to refer to Python 3 (breaking many scripts that expected it to refer to Python 2), **PEP 394** was published to provide guidance to Linux distributions on more gracefully handling the transition from Python 2 to Python 3.

Also in March, CPython migrated from Subversion to Mercurial (see **PEP 385**), with the first message from Mercurial to the python-checkins list being this commit from Senthil Kumaran. This ended more than two years of managing parallel updates of four active branches using `svnmerge` rather than a modern DVCS.

November 2011: **PEP 404** (the Python 2.8 Un-release Schedule) was published to make it crystal clear that the core development team had no plans to make a third parallel release in the Python 2.x series.

2012

March 2012: The first alpha of Python 3.3 was published. **PEP 398** covers the details of the Python 3.3 release cycle. Notably, Python 3.3 restored support for Python 2 style Unicode literals after Armin Ronacher and other web framework developers pointed out that this was one change that the web frameworks couldn't handle on behalf of their users. **PEP 414** covers the detailed rationale for that change.

April 2012: Canonical published Ubuntu 12.04 LTS, including commercial support for both Python 2.7 and Python 3.2.

September 2012: Six and half years after the inauguration of the Python 3000 project, Python 3.3 final was published as the first Python 3 release without a corresponding Python 2 feature release. This release introduced the **PEP 380** `yield from` syntax that was used heavily in the `asyncio` coroutine framework provisionally introduced to the standard library in Python 3.4, and subsequently declared stable in Python 3.6.

October 2012: **PEP 430** was published, and the online Python documentation updated to present the Python 3 documentation by default. In order to preserve existing links, deep links continue to be interpreted as referring to the Python 2.7 documentation.

2013

March 2013: **PEP 434** redefined IDLE as an application shipped with Python rather than part of the standard library, allowing the addition of new features in maintenance releases. Significantly, this allowed the Python 2.7 IDLE to be brought more into line with the features of the Python 3.x version.

Continuum Analytics started offering commercial support for cross-platform Python 3.3+ environments through their "Anaconda" Python distributions.

August 2013: The first alpha of Python 3.4 was published. **PEP 429** covers the details of the Python 3.4 release cycle. Amongst other changes, Python 3.4 restored full support for the binary and text transform codecs that were reinstated in Python 3.2, while maintaining the "text encodings only" restriction for the convenience methods on the builtin types.

September 2013: Red Hat published "Red Hat Software Collections 1.0", providing commercial support for both Python 2.7 and Python 3.3 on Red Hat Enterprise Linux systems, with later editions adding support for additional 3.x releases.

December 2013: The initial development of MicroPython, a variant of Python 3 specifically for microcontrollers, was successfully crowdfunded on Kickstarter.

2014

March 2014: Python 3.4 final was published as the second Python 3 release without a corresponding Python 2 release. It included several features designed to provide a better starting experience for newcomers to Python, such as bundling the "pip" installer by default, and including a rich asynchronous IO library.

April 2014: Ubuntu 14.04 LTS, initial target release for the "Only Python 3 on the install media" Ubuntu migration plan. (They didn't quite make it - a few test packages short on Ubuntu Touch, further away on the server and desktop images)

Red Hat also announced the creation of softwarecollections.org as the upstream project powering the Red Hat Software Collections product. The whole idea of both the project and the product is to make it easy to run applications using newer (or older!) language, database and web server runtimes, without interfering with the versions of those runtimes integrated directly into the operating system.

Note: With the original "5 years for migration to Python 3" target date approaching, April 2014 is also when Guido van Rossum amended the **Python 2.7 release PEP** to move the expected end-of-life date for Python 2.7 out to 2020.

May 2014: Python 2.7.7 was published, the first Python 2.7 maintenance release to incorporate additional security enhancement features as described in **PEP 466**. Also the first release where Microsoft contributed developer time to the creation of the Windows installers.

June 2014: The first stable release of PyPy3, providing a version of the PyPy runtime that is compatible with Python 3.2.5 (together with **PEP 414**'s restoration of the `u''` string literal prefix that first appeared in Python 3.3 for CPython).

Red Hat published Red Hat Enterprise Linux 7, with Python 2.7 as the system Python. This release ensures that Python 2.7 will remain a commercially supported platform until *at least* 2024 (based on Red Hat's 10 year support lifecycle).

Note: June 2014 also marked 5 years after the first production capable Python 3.x release (Python 3.1), and the original target date for completion of the Python 3 migration.

July 2014: CentOS 7 was released, providing a community distro based on Red Hat Enterprise Linux 7, and marking the beginning of the end of the Python 2.7 rollout (the CentOS system Python is a key dependency for many Python users).

boto v2.32.0 released with Python 3 support for most modules.

nltk 3.0b1 released with Python 3 support and the NLTK book switched over to covering Python 3 by default.

2015

February 2015: The first alpha of Python 3.5 was published. **PEP 478** covers the details of the Python 3.5 release cycle. Amongst other changes, **PEP 461** restored support for printf-style interpolation of binary data, addressing a significant usability regression in Python 3 relative to Python 2.

October 2014: SUSE Linux Enterprise Server 12 was released, containing supported Python 3.4 RPMs, adding SUSE to the list of commercial Python 3 redistributors.

March 2015: Microsoft Azure App Service launched with both Python 2.7 and Python 3.4 support, adding Microsoft to the list of commercial Python redistributors for the first time.

August 2015: At the Fedora community's annual Flock conference, Denise Dumas (Red Hat's VP of Platform Engineering), explicitly states that it is an engineering goal to include only Python 3 in the base operating system for the next major version of Red Hat Enterprise Linux (previously this had been implied by Red Hat's work on migrating Fedora and its infrastructure to Python 3, but not explicitly stated in a public venue)

September 2015: Python 3.5 final was released, bringing native syntactic support for asynchronous coroutines and a matrix multiplication operator, as well as the typing module for static type hints. Applications, libraries and frameworks wishing to take advantage of the new syntactic features need to reconsider whether or not to continue supporting Python 2.7.

Twisted 15.4 was released, the first version to include a Python 3 compatible version of the "Twisted Trial" test runner. This allowed the Twisted project to start running its test suite under Python 3, leading to steadily increasing Python 3 compatibility in subsequent Twisted releases.

October 2015: Fedora 23 ships with only Python 3 in the LiveCD and all default images other than the Server edition.

MicroPython support for the BBC micro:bit project is publicly announced, ensuring first class Python 3 support in a significant educational initiative.

PyInstaller 3.0 was released, supporting Python 2.7, and 3.3+.

2016

March 2016: gevent 1.1 was released, supporting Python 2.6, 2.7, and 3.3+.

May 2016: Several key projects in the Scientific Python community publish the Python 3 Statement, explicitly declaring their intent to end Python 2 support in line with the reference interpreter's anticipated 2020 date for the end of free community support.

August 2016: Google App Engine added official Python 3.4(!) support to their Flexible Environments (Python 3.5 support followed not long after, but the original announcement was for Python 3.4).

As part of rolling out Python 3.5 support, Microsoft Azure published instructions on how to select a particular Python version using App Service Site Extensions.

Initial release of Enthought Deployment Manager, with support for Python 2.7 and 3.5.

Mozilla provided the PyPy project with a development grant to bring their PyPy3 variant up to full compatibility with Python 3.5.

December 2016: Python 3.6 final was released, bringing further syntactic enhancements for asynchronous coroutines and static type hints, as well as a new compiler assisted string formatting syntax that manages to be both more readable (due to the use of inline interpolation expressions) and faster (due to the compiler assisted format parsing) than previous string formatting options. Through **PEP 528** and **PEP 529**, this release also featured significant improvements to the Windows compatibility of bytes-centric POSIX applications, and the Windows-specific *py* launcher started using Python 3 by default when both Python 2.x and 3.x are available on the system.

2017

March 2017: The first beta release of PyPy3 largely compatible with Python 3.5 was published (including support for the Python 3.6 f-string syntax).

Enthought Canopy 2.0.0 available, supporting Python 2.7 and 3.5 (official binary release date TBD - as of April 2017, the download page still offers Canopy 1.7.4)

April 2017: AWS Lambda added official Python 3.6 support, making Python 3 available by default through the 3 largest public cloud providers (Amazon, Microsoft, Google).

IPython 6.0 was released, the first feature release to require Python 3. The IPython 5.x series remains in maintenance mode as the last version supporting Python 2.7 (and Python 3 based variants of IPython retain full support for running and interacting with Python 2 language kernels using Project Jupyter's language independent notebook protocol).

Future

Note: At time of writing, the events below are in the future, and hence speculative as to their exact nature and timing. However, they reflect currently available information based on the stated intentions of developers and distributors.

December 2017: Anticipated date for the Django 2.0 release, which is expected to be the first version of Django to drop support for Python 2.7.

April 2018: Revised anticipated date for Ubuntu and Fedora to have finished migrating default components of their respective server editions to Python 3 (some common Linux components, most notably the Samba protocol server, proved challenging to migrate, so the stateful server variants of these distributions ended up taking longer to migrate to Python 3 than other variants that omitted those components from their default package set)

January? 2020: Anticipated date for Python 2.7 to switch to security fix only mode, ending roughly thirteen years of parallel maintenance of Python 2 and 3 by the core development team for the reference interpreter.

April 2021: Anticipated date for Ubuntu LTS 16.04 to go end of life, the first potential end date for commercial Python 2 support from Canonical (if Python 2.7 is successfully migrated to the community supported repositories for the Ubuntu 18.04 LTS release)

April 2024: Anticipated date for Ubuntu LTS 18.04 to go end of life, the second potential end date for commercial Python 2 support from Canonical (if it proves necessary to keep Python 2.7 in the commercially supported repositories as a dependency for the Ubuntu 18.04 LTS release)

June 2024: Anticipated date for Red Hat Enterprise Linux 7 to go end of life, also anticipated to be the last commercially supported redistribution of the Python 2 series.

When did Python 3 become the obvious choice for new projects?

I put the date for this as the release of Python 3.5, in September 2015. This release brought with it two major syntactic enhancemens (one giving Python's coroutine support its own dedicated syntax, distinct from generators, and another providing a binary operator for matrix multiplication), and restored a key feature that had been missing relative to Python 2 (printf-style binary interpolation support). It also incorporated a couple of key reliability and maintainability enhancements, in the form of automated handling of EINTR signals, and the inclusion of a gradual typing framework in the standard library.

Others may place the boundary at the release of Python 3.6, in December 2016, as the new "f-string" syntax provides a form of compiler-assisted string interpolation that is both faster and more readable than its predecessors:

```
print("Hello %s!" % name)         # All versions
print("Hello {0}!".format(name)) # Since Python 2.6 & 3.0
print("Hello {}!".format(name))  # Since Python 2.7 & 3.2
print(f"Hello {name}!")          # Since Python 3.6
```

Python 3.6 also provides further enhancements to the native coroutine syntax, as well as full syntactic support for annotating variables with static type hints.

Going in to this transition process, my personal estimate was that it would take roughly 5 years to get from the first production ready release of Python 3 to the point where its ecosystem would be sufficiently mature for it to be recommended unreservedly for all *new* Python projects.

Since 3.0 turned out to be a false start due to its IO stack being unusably slow, I start that counter from the release of 3.1: June 27, 2009. With Python 3.5 being released a little over 6 years after 3.1 and 3.6 a little more than a year after that, that means we clearly missed that original goal - the text model changes in particular proved to be a larger barrier to migration than expected, which slowed adoption by existing library and framework developers.

However, despite those challenges, key parts of the ecosystem were able to successfully add Python 3 support well before the 3.5 release. NumPy and the rest of the scientific Python stack supported both versions by 2015, as did several GUI frameworks (including PyGame).

The Pyramid, Django and Flask web frameworks supported both versions, as did the mod_wsgi Python application server, and the py2exe, py2app and cx-Freeze binary creators. The upgrade of Pillow from a repackaging project to a full development fork also brought PIL support to Python 3.

nltk supported Python 3 as of nltk 3.0, and the NLTK bookswitched to be based on Python 3 at the same time.

For AWS users, most `boto` modules became available on Python 3 as of http://boto.readthedocs.org/en/latest/releasenotes/v2.32.0.html.

PyInstaller is a popular option for creating native system installers for Python applications, and it has supported Python 3 since the 3.0 release in October 2015.

gevent is a popular alternative to writing natively asynchronous code, and it became generally available for Python 3 with the 1.1 release in March 2016.

As of April 2017, porting the full Twisted networking framework to Python 3 is still a work in progress, but many parts of it are already fully operational, and for new projects, native asyncio-based alternatives are often going to be available in Python 3 (especially for common protocols like HTTPS).

I think Python 3.5 is a superior language to 2.7 in almost every way (with the error reporting improvements being the ones I missed most when my day job involved working on a Python 2.6 application).

For educational purposes, there are a few concepts like functions, iterables and Unicode that need to be introduced earlier than was needed in Python 2, and there are still a few rough edges in adapting between the POSIX text model and the Python 3 one, but these are more than compensated for through improved default behaviours and more helpful error messages.

While students in enterprise environments may still need to learn Python 2 for a few more years, there are some significant benefits in learning Python 3 *first*, as that means students will already know which concepts survived the transition, and be more naturally inclined to write code that fits into the common subset of Python 2 and Python 3. This approach will also encourage new Python users that need to use Python 2 for professional reasons to take advantage of the backports and other support modules on PyPI to bring their Python 2.x usage as close to writing Python 3 code as is practical.

Support in enterprise Linux distributions is also a key point for uptake of Python 3. Canonical have already shipped long term support for three versions of Python 3 (Python 3.2 in Ubuntu 12.04 LTS, 3.4 in 14.04 LTS, and 3.5 in 14.04 LTS) and are continuing with the process of eliminating Python 2 from the installation images.

A Python 3 stack has existed in Fedora since Fedora 13 and has been growing over time, with Python 2 successfully removed from the live install CDs in late 2015 (Fedora 23). Red Hat also now ship fully supported Python 3.x runtimes as part of the Red Hat Software Collections product and the OpenShift Enterprise self-hosted Platform-as-a-Service offering (with new 3.x versions typically becoming commercially available within 6-12 months of the upstream release, and then remaining supported for 3 years from that point).

At Fedora's annual Flock conference in August 2015, Denise Dumas (VP of Platform Engineering) also indicated that Red Hat aimed to have the next major version of Red Hat Enterprise Linux ship only Python 3 in the base operating system, with Python 2 available solely through the Software Collections model (inverting the current situation, where Python 2 is available in both Software Collections and the base operating system, while Python 3 is only commercially available through Software Collections and the Software Collections based OpenShift environments).

The Arch Linux team have gone even further, making Python 3 the default Python on Arch installations. I am dubious as to the wisdom of their specific migration strategy, but I certainly can't complain about the vote of confidence!

The OpenStack project, likely the largest open source Python project short of the Linux distro aggregations, is also in the process of migrating from Python 2 to Python 3, and maintains a detailed status tracking page for the migration.

Outside the Linux ecosystem, other Python redistributors like ActiveState, Enthought, and Continuum Analytics provide both Python 2 and Python 3 releases, and Python 3 environments are also available through the major public cloud platforms.

When can we expect Python 2 to be a purely historical relic?

Python 2 is still a good language. While I think Python 3 is a *better* language (especially when it comes to the text model, error reporting, the native coroutine syntax in Python 3.5, and the string formatting syntax in Python 3.6), we've deliberately designed the migration plan so users can update on *their* timetable rather than ours (at least within a window of several years), and we expect commercial redistributors to extend that timeline even further.

The PyPy project have also stated their intention to continue providing a Python 2.7 compatible runtime indefinitely, since the RPython language used to implement PyPy is a subset of Python 2 rather than of Python 3.

I personally expect CPython 2.7 to remain a reasonably common deployment platform until mid 2024. Red Hat Enterprise Linux 7 (released in June 2014) uses CPython 2.7 as the system Python, and many library, framework and application developers base their minimum supported version of Python on the system Python in RHEL (especially since that also becomes the system Python in downstream rebuilds like CentOS and Scientific Linux). While Red Hat's actively trying to change that slow update cycle by encouraging application developers to target the Software Collections runtimes rather than the system Python, that change in itself is a significant cultural shift for the RHEL/CentOS user base.

Aside from Blender, it appears most publishing and animation tools with Python support (specifically Scribus, InkScape and AutoDesk tools like Maya and MotionBuilder) are happy enough with Python 2.7. GIS tools similarly currently still use Python 2.7. This actually makes a fair bit of sense, especially for the commercial tools, since the Python support in these tools is there primarily to manipulate the application data model and there arguably aren't any major improvements in Python 3 for that kind of use case as yet, but still some risk of breaking existing scripts if the application updates to Python 3.

For the open source applications when Python 2 is currently seen as a "good enough" scripting engine, the likely main driver for Python 3 scripting support is likely to be commercial distribution vendors looking to drop commercial Python 2 runtime support - the up front investment in application level Python 3 support would be intended to pay off in the form of reduced long term sustaining engineering costs at the language runtime level.

That said, the Python 3 reference interpreter also offers quite a few new low level configuration options that let embedding applications control the memory allocators used, monitor and control all bytecode execution, and various other improvements to the runtime embedding functionality, so the natural incentives for application developers to migrate are starting to accumulate, which means we may see more activity on that front as the 2020 date for the end of community support of the Python 2 series gets closer.

But uptake is so slow, doesn't this mean Python 3 is failing as a platform?

While the frequency with which this question is asked has declined markedly since 2015 or so, a common thread I saw running through such declarations of "failure" was people not quite understanding the key questions where the transition plan was aiming to change the answers. These are the three key questions:

- "I am interested in learning Python. Should I learn Python 2 or Python 3?"
- "I am teaching a Python class. Should I teach Python 2 or Python 3?"
- "I am an experienced Python developer starting a new project. Should I use Python 2 or Python 3?"

At the start of the migration, the answer to all of those questions was *obviously* "Python 2". By August 2015, I considered the answer to be "Python 3.4, unless you have a compelling reason to choose Python 2 instead". Possible compelling reasons included "I am using existing course material that was written for Python 2", "I am teaching the course to maintainers of an existing Python 2 code base", "We have a large in-house collection of existing Python 2 only support libraries we want to reuse" and "I only use the version of Python provided by my Linux distro vendor and

they currently only support Python 2" (in regards to that last point, we realised early that the correct place to tackle it was on the *vendor* side, and by late 2014, all of Canonical, Red Hat, and SUSE had commercial Python 3 offerings available).

Note the question that *isn't* on the list: "I have a large Python 2 application which is working well for me. Should I migrate it to Python 3?".

While OpenStack and some key Linux distributions have answered "Yes", for most organisations the answer to *that* question remained "No" for several years while companies like Canonical, Red Hat, Facebook, Google, Dropbox, and others worked to migrate their own systems, and published the related migration tools (such as the `pylint --py3k` option, and the work that has gone into the `mypy` and `typeshed` projects to allow Python 3 static type analysis to be applied to Python 2 programs prior to attempting to migrate them).

While platform effects are starting to shift even the answer to that question towards "Maybe" for the majority of users (and Python 3 gives Python 2 a much nicer exit strategy to a newer language than COBOL ever did), the time frame for *that* change is a lot longer than the five years that was projected for changing the default choice of Python version for green field projects.

That said, reducing or eliminating any major remaining barriers to migration is an ongoing design goal for Python 3.x releases, at least in those cases where the change is also judged to be an internal improvement within Python 3 (for example, the restoration of binary interpolation support in Python 3.5 was motivated not just by making it easier to migrate from Python 2, but also to make certain kinds of network programming and other stream processing code easier to write in Python 3).

In the earlier days of the Python 3 series, several of the actions taken by the core development team were actually deliberately designed to keep conservative users *away* from Python 3 as a way of providing time for the ecosystem to mature.

Now, if Python 3 had failed to offer a desirable platform, nobody would have cared about this in the slightest. Instead, what we saw was the following:

- people coming up with great migration guides and utilities *independently* of the core development team. While six was created by a core developer (Benjamin Peterson), and `lib2to3` and the main porting guides are published by the core development team, python-modernize was created by Armin Ronacher (creator of Jinja2 and Flask), while python-future was created by Ed Schofield based on that earlier work. Lennart Regebro has also done stellar work in creating an in-depth guide to porting to Python 3
- Linux distributions aiming to make Python 2 an optional download and have only Python 3 installed by default
- commercial Python redistributors and public cloud providers ensuring that Python 3 was included as one of their supported offerings
- customers approaching operating system vendors and asking for assistance in migrating large proprietary code bases from Python 2 to Python 3
- more constrained plugin ecosystems that use an embedded Python interpreter (like Blender, gcc, and gdb) either adding Python 3 support, or else migrating entirely from Python 2 to 3
- developers lamenting the fact that they *wanted* to use Python 3, but were being blocked by various dependencies being missing, or because they previously used Python 2, and needed to justify the cost of migration to their employer
- library and framework developers that hadn't already added Python 3 support for their own reasons being strongly encouraged by their users to offer it (sometimes in the form of code contributions, other times in the form of tracker issues, mailing list posts and blog entries)
- interesting new implementations/variants like MyPy and MicroPython taking advantage of the removal of legacy behaviour to target the leaner Python 3 language design rather than trying to handle the full backwards compatibility implications of implementing Python 2

- developers complaining that the core development team wasn't being aggressive enough in forcing the community to migrate promptly rather than allowing the migration to proceed at its own pace (!)

That last case only appeared around 2014 (~5 years into the migration), and the difference in perspective appears to be an instance of the classic early adopter/early majority divide in platform adoption. The deliberately gentle migration plan was (and is) for the benefit of the late adopters that drive Python's overall popularity, not the early adopters that make up both the open source development community and the (slightly) broader software development blogging community.

It's important to keep in mind that Python 2.6 (released October 2008) has long stood as one of the most widely deployed versions of Python, purely through being the system Python in Red Hat Enterprise Linux 6 and its derivatives, and usage of Python 2.4 (released November 2004) remained non-trivial through to at least March 2017 for the same reason with respect to Red Hat Enterprise Linux 5.

I expect there is a similar effect from stable versions of Debian, Ubuntu LTS releases and SUSE Linux Enterprise releases, but (by some strange coincidence) I'm not as familiar with the Python versions and end-of-support dates for those as I am with those for the products sold by my employer ;)

If we weren't getting complaints from the early adopter crowd about the pace of the migration, *then* I would have been worried (as it would have indicated they had abandoned Python entirely and moved on to something else).

The final key point to keep in mind is that the available metrics on Python 3 adoption are quite limited, and that remains true regardless of whether we think the migration is going well or going poorly. The three main quantitative options are to analyse user agents on the Python Package Index, declarations of Python 3 support on PyPI and binary installer downloads for Mac OS X and Windows from python.org.

The first of those remains heavily dominated by *existing* Python 2 users, but the trend in Python 3 usage is still upwards. These metrics are stored as a public data set in Google Big Query, and this post goes over some of the queries that are possible with the available data. The records are incomplete prior to June 2016, but running the query in April 2017 shows downloads from Python 3 clients increasing from around 7% of approximately 430 million downloads in June 2016 to around 12% of approximately 720 million downloads in March 2017.

The second is based on publisher provided package metadata rather than automated version compatibility checking.

Of the top 360 most downloaded packages, ~94% offer Python 3 support, with several of those that are Python 2 only (such as graphite-web and supervisord) typically being run as standalone services rather than as imported modules that necessarily need to be using the same version of Python. Again, the trend is upwards (the number in 2014 was closer to 70%), and I'm not aware of anyone *adding* Python 3 support, and then removing it as imposing too much maintenance overhead.

The last metric reached the point where Python 3 downloads outnumbered Python 2 downloads (54% vs 46%) back in 2013. Those stats needs to be collected manully from the `www.python.org` server access logs, so I don't have anything more recent than that.

The Python 3 ecosystem is definitely still the smaller of the two as of April 2017 (by a non-trivial margin), but users that start with Python 3 are able to move parts of their applications and services to Python 2 readily enough if the need arises, and hopefully with a clear idea of which parts of Python 2 are the modern recommended parts that survived the transition to Python 3, and which parts are the legacy cruft that only survives in the latest Python 2.x releases due to backwards compatibility concerns.

For the inverse question relating to the concern that the existing migration plan is too *aggressive*, see *Aren't you abandoning Python 2 users?*.

Is the ultimate success of Python 3 as a platform assured?

Yes, its place as the natural successor to the already dominant Python 2 platform is now assured. Commercial support has long been available from multiple independent vendors, the vast majority of the core components from the Python 2 ecosystem are already available, and the combination of the Python 3.5+ releases and Python's uptake in the education and data analysis sectors provide assurance of a steady supply of both Python developers, and work for those

developers (in the 2016 edition of IEEE's survey of programming languages, Python was 3rd, trailing only Java and C, overtaking C++ relative to its 2015 position, and both C++ and C# relative to the initial 2014 survey).

For me, with my Linux-and-infrastructure-software bias, the tipping point has been Ubuntu and Fedora successfully making the transition to only having Python 3 in their default install. That change means that a lot of key Linux infrastructure software is now Python 3 compatible, as well as representing not only a significant statement of trust in the Python 3 platform by a couple of well respected organisations (Canonical and Red Hat), but also a non-trivial investment of developer time and energy in performing the migration. This change will also mean that Python 3 will be more readily available than Python 2 on those platforms in the future, and hence more likely to be used as the chosen language variant for Python utility scripts, and hence increase the attractiveness of supporting Python 3 for library and framework developers.

A significant milestone only attained over 2016 and 2017 has been the three largest public cloud providers (Amazon Web Services, Microsoft Azure, and Google Cloud Platform) ensuring that Python 3 is a fully supported development option on their respective platforms, adding to the support already previously available in platforms like Heroku and OpenShift Online.

Specifically in the context of infrastructure, I also see the ongoing migration of OpenStack components from being Python 2 only applications to being Python 3 compatible as highly significant, as OpenStack is arguably one of the most notable Python projects currently in existence in terms of spreading awareness outside the traditional open source and academic environs. In particular, as OpenStack becomes a Python 3 application, then the plethora of regional cloud provider developers and hardware vendor plugin developers employed to work on it will all be learning Python 3 rather than Python 2.

A notable early contribution to adoption has been the education community's staunch advocacy for the wider Python community to catch up with them in embracing Python 3, rather than confusing their students with occasional recommendations to learn Python 2 directly, rather than learning Python 3 first.

As far as the scientific community goes, they were amongst the earliest adopters of Python 3 - I assume the reduced barriers to learnability were something they appreciated, and the Unicode changes were not a problem that caused them significant trouble.

I think the web development community has certainly had the roughest time of it. Not only were the WSGI update discussions long and drawn out (and as draining as any standards setting exercise), resulting in a compromise solution that at least works but isn't simple to deal with, but they're also the most directly affected by the additional challenges faced when working directly with binary data in Python 3. However, even in the face of these issues, the major modern Python web frameworks, libraries and database interfaces *do* support Python 3, and the return of binary interpolation support in Python 3.5 addressed some of the key concerns raised by the developers of the Twisted networking library.

The adoption of `asyncio` as *the* standard framework for asynchronous IO and the subsequent incorporation of first class syntactic support for coroutines have also helped the web development community resolve a long standing issue with a lack of a standard way for web servers and web frameworks to communicate regarding long lived client connections (such as those needed for WebSockets support), providing a clear incentive for migration to Python 3.3+ that didn't exist with earlier Python 3 versions.

Python 3 is meant to make Unicode easier, so why is <X> harder?

As of 2015, the Python community as a whole had had more than 15 years to get used to the Python 2 way of handling Unicode. By contrast, for Python 3, we'd only had a production ready release available for just over 5 years, and since some of the heaviest users of Unicode are the web framework developers, and they'd only had a stable WSGI target since the release of 3.2, you could drop that down to just under 5 years of intensive use by a wide range of developers with extensive practical experiencing in handling Unicode (we have some *excellent* Unicode developers in the core team, but feedback from a variety of sources is invaluable for a change of this magnitude).

That feedback has already resulted in major improvements in the Unicode support for the Python 3.2, 3.3, 3.4, 3.5, and 3.6 releases. With the `codecs` and `email` modules being brought into line, the Python 3.4 release was the first one where the transition felt close to being "done" to me in terms of coping with the full implications of a strictly enforced

distinction between binary and text data in the standard library, while Python 3.5 revisited some of the earlier design decisions of the Python 3 series and changed some of them based on several years of additional experience. Python 3.6 brought some major changes to the way binary system APIs are handled on Windows, and changes of similar scope are anticipated for 3.7 on non-Windows systems.

While I'm optimistic that the system boundary handling changes proposed for Python 3.7 will resolve the last of the major issues, I nevertheless expect that feedback process will continue throughout the 3.x series, since "mostly done" and "done" aren't quite the same thing, and attempting to closely integrate with POSIX systems that may be using ASCII incompatible encodings while using a text model with strict binary/text separation hasn't really been done before at Python's scale (the JVM is UTF-16 based, but bypasses most OS provided services, while other tools often choose the approach of just assuming that all bytes are UTF-8 encoded, regardless of what the underlying OS claims).

In addition to the cases where blurring the binary/text distinction really did make things simpler in Python 2, we're also forcing even developers in strict ASCII-only environments to have to care about Unicode correctness, or else explicitly tell the interpreter not to worry about it. This means that Python 2 users that may have previously been able to ignore Unicode issues may need to account for them properly when migrating to Python 3.

I've written more extensively on both of these topics in *Python 3 and ASCII Compatible Binary Protocols* and *Processing Text Files in Python 3*, while **PEP 538** and **PEP 540** go into detail on the system boundary changes now being proposed for Python 3.7.

Python 3 is meant to fix Unicode, so why is <X> still broken?

The long march from the early assumptions of Anglocentric ASCII based computing to a more global Unicode based future is still ongoing, both for the Python community, and the computing world at large. Computers are still generally much better at dealing with English and other languages with similarly limited character sets than they are with the full flexibility of human languages, even the subset that has been pinned down to a particular binary representation thanks to the efforts of the Unicode Consortium.

While the changes to the core text model in Python 3 *did* implicitly address many of the Unicode issues affecting Python 2, there are still plenty of Unicode handling issues that require their own independent updates. For example, the interactive console on Windows poses some particular challenges that have yet to be satisfactorily resolved. One recurring problem is that many of these are relatively easy to work around (such as by using a graphical environment rather than the default interactive interpreter to avoid the command line limitations on Windows), but comparatively hard to fix properly (and then get agreement that the proposed fix is a suitable one).

The are also more specific questions covering the state of the *WSGI middleware interface* for web services, and the issues that can arise when dealing with *What's up with POSIX systems in Python 3?*.

Is Python 3 a better language to teach beginning programmers?

I believe so, yes, especially if teaching folks that aren't native English speakers. However, I also expect a lot of folks will still want to continue on and learn Python 2 even if they learn Python 3 first - I just think that for people that don't already know C, it will be easier to start with Python 3, and then learn Python 2 (and the relevant parts of C) in terms of the differences from Python 3 rather than learning Python 2 directly and having to learn all those legacy details at the same time as learning to program in the first place.

Note: This answer was written for Python 3.5. For Python 3.6, other potential benefits in teaching beginners include the new f-string formatting syntax, the secrets module, the ability to include underscores to improve the readability of long numeric literals, and the ordering of arbitrary function keyword arguments reliably matching the order in which they're supplied to the function call.

As noted above, Python 2 has some interesting quirks due to its C heritage and the way the language has evolved since Guido first created Python in 1991. These quirks then have to be taught to *every* new Python user so that they can

avoid them. The following are examples of such quirks that are easy to demonstrate in an interactive session (and resist the temptation to point out that these can all be worked around - for teaching beginners, it's the default behaviour that matters, not what experts can instruct the interpreter to do with the right incantations elsewhere in the program).

You can get unexpected encoding errors when attempting to decode values and unexpected decoding errors when attempting to encode them, due to the presence of decode and encode methods on both `str` and `unicode` objects, but more restrictive input type expectations for the underlying codecs that then trigger the implicit *ASCII* based encoding or decoding:

```
>>> u"\xe9".decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python2.7/encodings/utf_8.py", line 16, in decode
    return codecs.utf_8_decode(input, errors, True)
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 0:␣
↪ordinal not in range(128)
>>> b"\xe9".encode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe9 in position 0: ordinal not␣
↪in range(128)
```

Python 2 has a limited and inconsistent understanding of character sets beyond those needed to record English text:

```
>>> è = 1
  File "<stdin>", line 1
    è = 1
    ^
SyntaxError: invalid syntax
>>> print("è")
è
```

That second line usually works in the interactive interpreter, but won't work by default in a script:

```
$ echo 'print("è")' > foo.py
$ python foo.py
  File "foo.py", line 1
SyntaxError: Non-ASCII character '\xc3' in file foo.py on line 1, but no encoding␣
↪declared; see http://www.python.org/peps/pep-0263.html for details
```

The handling of Unicode module names is also inconsistent:

```
$ echo "print(__name__)" > è.py
$ python -m è
__main__
$ python -c "import è"
  File "<string>", line 1
    import è
           ^
SyntaxError: invalid syntax
```

Beginners are often surprised to find that Python 2 can't do basic arithmetic correctly:

```
>>> 3 / 4
0
```

Can be bemused by the fact that Python 2 interprets numbers strangely if they have a leading zero:

```
>>> 0777
511
```

And may also eventually notice that Python 2 has two different kinds of integer:

```
>>> type(10) is type(10**100)
False
>>> type(10) is type(10L)
False
>>> 10
10
>>> 10L
10L
```

The `print` statement is weirdly different from normal function calls:

```
>>> print 1, 2, 3
1 2 3
>>> print (1, 2, 3)
(1, 2, 3)
>>> print 1; print 2; print 3
1
2
3
>>> print 1,; print 2,; print 3
1 2 3
>>> import sys
>>> print >> sys.stderr, 1, 2, 3
1 2 3
```

And the `exec` statement also differs from normal function calls like `eval` and `execfile`:

```
>>> d = {}
>>> exec "x = 1" in d
>>> d["x"]
1
>>> d2 = {"x":[]}
>>> eval("x.append(1)", d2)
>>> d2["x"]
[1]
>>> with open("example.py", "w") as f:
...     f.write("x = 1\n")
...
>>> d3 = {}
>>> execfile("example.py", d3)
>>> d3["x"]
1
```

The `input` builtin has some seriously problematic default behaviour:

```
>>> input("This is dangerous: ")
This is dangerous: __import__("os").system("echo you are in trouble now")
you are in trouble now
0
```

The `open` builtin doesn't handle non-ASCII files correctly (you have to use `codecs.open` instead), although this often isn't obvious on POSIX systems (where passing the raw bytes through the way Python 2 does often works correctly).

You need parentheses to catch multiple exceptions, but forgetting that is an error that passes silently:

```
>>> try:
...   1/0
... except TypeError, ZeroDivisionError:
...     print("Exception suppressed")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:
...     1/0
... except (TypeError, ZeroDivisionError):
...     print("Exception suppressed")
...
Exception suppressed
```

And if you make a mistake in an error handler, you'll lose the original error:

```
>>> try:
...     1/0
... except Exception:
...     logging.exception("Something went wrong")
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'logging' is not defined
```

Python 2 also presents users with a choice between two relatively unattractive alternatives for calling up to a parent class implementation from a subclass method:

```
class MySubclass(Example):

    def explicit_non_cooperative(self):
        Example.explicit_non_cooperative(self)

    def explicit_cooperative(self):
        super(MySubclass, self).explicit_cooperative()
```

List comprehensions are one of Python's most popular features, yet they can have surprising side effects on the local namespace:

```
>>> i = 10
>>> squares = [i*i for i in range(5)]
>>> i
4
```

Python 2 is still a good language despite these flaws, but users that are happy with Python 2 shouldn't labour under the misapprehension that the language is perfect. We have made mistakes, and Python 3 came about because Guido and the rest of the core development team finally became tired of making excuses for those limitations, and decided to start down the long road towards fixing them instead.

All of the above issues have been addressed by backwards incompatible changes in Python 3. Once we had made that decision, then adding other new features *twice* (once to Python 3 and again to Python 2) imposed significant additional development effort, although we *did* do so for a number of years (the Python 2.6 and 2.7 releases were both developed in parallel with Python 3 releases, and include many changes originally created for Python 3 that were backported to Python 2 since they were backwards compatible and didn't rely on other Python 3 only changes like the new, more Unicode friendly, IO stack).

I'll give several examples below of how the above behaviours have changed in Python 3 releases, up to and including Python 3.6 (since that's the currently released version).

In Python 3, the codec related builtin convenience methods are *strictly* reserved for use with text encodings. Accordingly, text objects no longer even have a `decode` method, and binary types no longer have an `encode` method:

```
>>> u"\xe9".decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'decode'
>>> b"\xe9".encode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'encode'
```

In addition to the above changes, Python 3.4 included additional changes to the codec system to help with more gently easing users into the idea that there are different kinds of codecs, and only some of them are text encodings. It also updates many of the networking modules to make secure connections much simpler.

Python 3 also has a much improved understanding of character sets beyond English:

```
>>> è = 1
>>> è
1
```

And this improved understanding extends to the import system:

```
$ echo "print(__name__)" > è.py
$ python3 -m è
__main__
$ python3 -c "import è"
è
```

Python 3 has learned how to do basic arithmetic, replaces the surprising C notation for octal numbers with the more explicit alternative supported since Python 2.6 and only has one kind of integer:

```
>>> 3 / 4
0.75
>>> 0777
  File "<stdin>", line 1
    0777
       ^
SyntaxError: invalid token
>>> 0o777
511
>>> type(10) is type(10**100)
True
>>> 10
10
>>> 10L
  File "<stdin>", line 1
    10L
      ^
SyntaxError: invalid syntax
```

`print` is now just an ordinary function that accepts keyword arguments, rather than having its own custom (and arcane) syntax variations (note that controlling the separator between elements is a feature that requires preformatting of the string to be printed in Python 2 but was trivial to add direct support for when print was converted to an ordinary builtin function rather than being a separate statement):

```
>>> print 1, 2, 3
  File "<stdin>", line 1
    print 1, 2, 3
          ^
SyntaxError: invalid syntax
>>> print(1, 2, 3)
1 2 3
>>> print((1, 2, 3))
(1, 2, 3)
>>> print(1); print(2); print(3)
1
2
3
>>> print(1, 2, 3, sep="\n")
1
2
3
>>> print(1, end=" "); print(2, end=" "); print(3)
1 2 3
>>> import sys
>>> print(1, 2, 3, file=sys.stderr)
1 2 3
```

`exec` is now more consistent with `execfile`:

```
>>> d = {}
>>> exec("x=1", d)
>>> d["x"]
1
```

Converting `print` and `exec` to builtins rather than statements means they now also work natively with utilities that require real function objects (like `map` and `functools.partial`), they can be replaced with mock objects when testing and they can be more readily substituted with alternative interfaces (such as replacing raw print statements with a pretty printer or a logging system). It also means they can be passed to the builtin `help` function without quoting, the same as other builtins.

The `input` builtin now has the much safer behaviour that is provided as `raw_input` in Python 2:

```
>>> input("This is no longer dangerous: ")
This is no longer dangerous: __import__("os").system("echo you have foiled my cunning␣
↪plan")
'__import__("os").system("echo you have foiled my cunning plan")'
```

The entire IO stack has been rewritten in Python 3 to natively handle Unicode and (in the absence of system configuration errors), to favour UTF-8 by default rather than ASCII. Unlike Python 2, `open()` in Python 3 natively supports `encoding` and `errors` arguments, and the `tokenize.open()` function automatically handles Python source file encoding cookies.

Failing to trap an exception is no longer silently ignored:

```
>>> try:
...     1/0
... except TypeError, ZeroDivisionError:
  File "<stdin>", line 3
    except TypeError, ZeroDivisionError:
                    ^
SyntaxError: invalid syntax
```

And most errors in exception handlers will now still report the original error that triggered the exception handler:

```
>>> try:
...     1/0
... except Exception:
...     logging.exception("Something went wrong")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
NameError: name 'logging' is not defined
```

Note that implicit exception chaining is the thing I miss most frequently when working in Python 2, and the point I consider the single biggest gain over Python 3 when migrating *existing* applications - there are few things more irritating when debugging a rare production failure than losing the real problem details due to a secondary failure in a rarely invoked error path.

While you probably don't want to know how it works internally, Python 3 also provides a much cleaner API for calling up to the parent implementation of a method:

```
class MySubclass(Example):

    def implicit_cooperative(self):
        super().implicit_cooperative()
```

And, like generator expressions in both Python 2 and Python 3, list comprehensions in Python 3 no longer have any side effects on the local namespace:

```
>>> i = 10
>>> squares = [i*i for i in range(5)]
>>> i
10
```

The above improvements are all changes that *couldn't* be backported to a hypothetical Python 2.8 release, since they're backwards incompatible with some (but far from all) existing Python 2 code, mostly for obvious reasons. The exception chaining isn't obviously backwards incompatible, but still can't be backported due to the fact that handling the implications of creating a reference cycle between caught exceptions and the execution frames referenced from their tracebacks involved changing the lifecycle of the variable named in an "as" clause of an exception handler (to break the cycle, those names are automatically deleted at the end of the relevant exception handler in Python 3 - you now need to bind the exception to a different local variable name in order to keep a valid reference after the handler has finished running). The list comprehension changes are also backwards incompatible in non-obvious ways (since not only do they no longer leak the variable, but the way the expressions access the containing scope changes - they're now full closures rather than running directly in the containing scope).

As documented in **PEP 466**, the networking security changes were deemed worthy of backporting. In contrast, while it's perhaps *possible* to backport the implicit super change, it would need to be separated from the other backwards incompatible changes to the type system machinery (and in that case, there's no "help improve the overall security of the internet" argument to be made in favour of doing the work).

There are some other notable changes in Python 3 that are of substantial benefit when teaching new users (as well as for old hands), that technically *could* be included in a Python 2.8 release if the core development chose to create one, but in practice such a release isn't going to happen. However, folks interested in that idea may want to check out the

Tauthon project, which is a Python 2/3 hybrid language that maintains full Python 2.7 compatibility while backporting backwards compatible enhancement from the Python 3 series.

PEP 3151 means that Python 3.3+ has a significantly more sensible system for catching particular kinds of operating system errors. Here's the race condition free way to detect a missing file in Python 2.7:

```
>>> import errno
>>> try:
...     f = open("This does not exist")
... except IOError as err:
...     if err.errno != errno.ENOENT:
...         raise
...     print("File not found")
...
File not found
```

And here's the same operation in Python 3.3+:

```
>>> try:
...     f = open("This does not exist")
... except FileNotFoundError:
...     print("File not found")
...
File not found
```

(If you're opening the file for writing, then you can use exclusive mode to prevent race conditions without using a subdirectory - Python 2 has no equivalent. There are many other cases where Python 3 exposes operating system level functionality that wasn't broadly available when the feature set for Python 2.7 was frozen in April 2010).

Another common complaint with Python 2 is the requirement to use empty `__init__.py` files to indicate a directory is a Python package, and the complexity of splitting a package definition across multiple directories. By contrast, here's an example of how to split a package across multiple directories in Python 3.3+ (note the lack of `__init__.py` files). While technically this can be backported, the implementation depends on the new pure Python implementation of the import system, which in turn depends on the Unicode friendly IO stack in Python 3, so backporting it is far from trivial:

```
$ mkdir -p dir1/nspkg
$ mkdir -p dir2/nspkg
$ echo 'print("Imported submodule A")' > dir1/nspkg/a.py
$ echo 'print("Imported submodule B")' > dir2/nspkg/b.py
$ PYTHONPATH=dir1:dir2 python3 -c "import nspkg.a, nspkg.b"
Imported submodule A
Imported submodule B
```

That layout doesn't work at all in Python 2 due to the missing `__init__.py` files, and even if you add them, it still won't find the second directory:

```
$ PYTHONPATH=dir1:dir2 python -c "import nspkg.a, nspkg.b"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named nspkg.a
$ touch dir1/nspkg/__init__.py
$ touch dir2/nspkg/__init__.py
$ PYTHONPATH=dir1:dir2 python -c "import nspkg.a, nspkg.b"
Imported submodule A
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named b
```

That last actually shows another limitation in Python 2's error handling since import failures don't always show the full name of the missing module. That is fixed in Python 3:

```
$ PYTHONPATH=dir1 python3 -c "import nspkg.a, nspkg.b"
Imported submodule A
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'nspkg.b'
```

That said: Eric Snow *has* now backported the Python 3.4 import system to Python 2.7 as importlib2. I'm aware of at least one large organisation using that in production and being quite happy with the results :)

Python 3.3 also included some minor improvements to the error messages produced when functions and methods are called with incorrect arguments.

Out of the box, why is Python 3 better than Python 2?

The feature set for Python 2.7 was essentially locked in April 2010 with the first beta release. Since then, with a very limited number of exceptions related to network security, the Python core development team have only been adding new features directly to the Python 3 series. These new features are informed both by our experience with Python 3 itself, as well as with our ongoing experience working with Python 2 (as they're still very similar languages).

As Python 2 is a mature, capable language, with a rich library of support modules available from the Python Package Index (including many backports from the Python 3 standard library), there's no one universally important feature that will provide a compelling argument to switch for *existing* Python 2 users. Of necessity, existing Python 2 users are those who didn't find the limitations of Python 2 that lead to the creation of Python 3 particularly problematic. It is for the benefit of these users that Python 2 continues to be maintained.

For *new* users of Python however, Python 3 represents years of additional work above and beyond what was included in the Python 2.7 release. Features that may require third party modules, or simply not be possible at all in Python 2, are provided by default in Python 3. This answer doesn't attempt to provide an exhaustive list of such features, but does aim to provide an illustrative overview of the kinds of improvements that have been made. The What's New guides for the Python 3 series (especially the 3.3+ releases that occurred after the Python 2 series was placed in long term maintenance) provide more comprehensive coverage.

While I've tried to just hit some highlights in this list, it's still rather long. The full What's New documents are substantially longer.

Note: This answer was written for Python 3.5. For Python 3.6, some other notable enhancements include the new f-string formatting syntax, the secrets module, the ability to include underscores to improve the readability of long string literals, changes to preserve the order of class namespaces and function keyword arguments, type hints for named variables, and more.

Some changes that are likely to affect most projects are error handling related:

- the exception hierarchy for operating system errors is now based on what went wrong, rather than which module detected the failure (see **PEP 3151** for details).
- bugs in error handling code no longer hide the original exception (which can be a huge time saver when it happens to hard to reproduce bugs)
- by default, if the logging system is left unconfigured, warnings and above are written to sys.stderr, while other events are ignored
- the codec system endeavours to ensure the codec name always appears in the reported error message when the underlying call fails

- the error messages from failed argument binding now do a much better job of describing the expected signature of the function
- the socket module takes advantage of the new enum support to include constant names (rather than just numeric values) in the error message output
- starting in Python 3.5, all standard library modules making system calls should handle EINTR automatically

Unicode is more deeply integrated into the language design, along with a clearer separation between binary and text data:

- the `open()` builtin natively supports decoding of text files (rather than having to use `codecs.open()` instead)
- the `bytes` type provides locale independent manipulation of binary data that may contain ASCII segments (the Python 2 `str` type has locale dependent behaviour for some operations)
- the codec system has been separated into two tiers. The `str.encode()`, `bytes.decode()` and `bytearray.decode()` methods provide direct access to Unicode text encodings, while the `codecs` module provides general access to all available codecs, including binary->binary and text->text transforms (in Python 2, all three kinds can be accessed through the convenience methods on the builtin types, creating ambiguity as to the expected return types of the affected methods)
- data received from the operating system is automatically decoded to text whenever possible (this does cause integration issues in some cases when the OS provides incorrect configuration data, but otherwise allows applications to ignore more cross-platform differences in whether OS APIs natively use bytes or UTF-16)
- identifiers and the import system are no longer limited to ASCII text (allowing non-English speakers to use names in their native languages when appropriate)
- Python 3 deliberately has no equivalent to the implicit ASCII based decoding that takes place in Python 2 when an 8-bit `str` object encounters a `unicode` object (note that disabling this implicit conversion in Python 2, while technically possible, is not typically feasible, as turning it off breaks various parts of the standard library)
- Python 3.3+ now correctly handles code points outside the basic multilingual plane without needing to use 4 bytes per code point for all Unicode data (as Python 2 does)

A few new debugging tools are also provided out of the box:

- `faulthandler` allows the generation of Python tracebacks for segmentation faults and threading deadlocks (including a `-X faulthandler` command line option to debug arbitrary scripts)
- `tracemalloc` makes it possible to track where objects were allocated and obtain a traceback summary for those locations (this relies on the dynamic memory allocator switching feature added in Python 3.4 and hence cannot be backported to Python 2 without patching the interpreter and building from source
- the `gc` module now provides additional introspection and hook APIs

The concurrency support has been improved in a number of ways:

- The native coroutine syntax added in Python 3.5 is substantially more approachable than the previous "generators-as-coroutines" syntax (as it avoids triggering iterator based intuitions that aren't actually helpful in the coroutine case)
- `asyncio` (and the supporting `selectors` module) provides greatly enhanced native support for asynchronous IO
- `concurrent.futures` provides straightforward support for dispatching work to separate working processes or threads
- `multiprocessing` is far more configurable (including the option to avoid relying on `os.fork` on POSIX systems, making it possible to avoid the poor interactions with between threads and `os.fork`, while still using both multiple processes and threads)

- the CPython Global Interpreter Lock has been updated to switch contexts based on absolute time intervals, rather than by counting bytecode execution steps (context switches will still occur between bytecode boundaries)

For data analysis use cases, there's one major syntactic addition:

- Python 3.5 added a new binary operator symbol specifically for use in matrix multiplication

Notable additions to the standard library's native testing capabilities include:

- the `unittest.mock` module, previously only available as a third party library
- a "subtest" feature that allows arbitrary sections of a test to be reported as independent results (including details on what specific values were tested), without having to completely rewrite the test to fit into a parameterised testing framework
- a new `FAIL_FAST` option for `doctest` that requests stopping the doctest at the first failing test, rather than continuing on to run the remaining tests

Performance improvements include:

- significant optimisation work on various text encodings, especially UTF-8, UTF-16 and UTF-32
- a significantly more memory efficient Unicode representation, especially compared to the unconditional 4 bytes per code point used in Linux distro builds of Python 2
- a C accelerator module for the `decimal` module
- transparent use of other C accelerator modules where feasible (including for `pickle` and `io`)
- the `range` builtin is now a memory efficient calculated sequence
- the use of iterators or other memory efficient representations for various other builtin APIs that previously returned lists
- dictionary instances share their key storage when possible, reducing the amount of memory consumed by large numbers of class instances
- the rewritten implementation of the import system now caches directory listings for a brief time rather than blindly performing `stat` operations for all possible file names, drastically improving startup performance when network filesystems are present on `sys.path`

Security improvements include:

- support for "exclusive mode" when opening files
- support for the directory file descriptor APIs that avoid various symlink based attacks
- switching the default hashing algorithm for key data types to SIPHash
- providing an "isolated mode" command line switch to help ensure user settings don't impact execution of particular commands
- disabling inheritance of file descriptors and Windows handles by child processes by default
- new multiprocessing options that avoid sharing memory with child process by avoiding the `os.fork` system call
- significant improvements to the SSL module, such as TLS v1.1 and v1.2 support, Server Name Indication support, access to platform certificate stores, and improved support for certificate verification (while these are in the process of being backported to Python 2.7 as part of **PEP 466**, it is not yet clear when that process will be completed, and those enhancements are already available in Python 3 today)
- other networking modules now take advantage of many of the SSL module improvements, including making it easier to use the new `ssl.create_default_context()` to choose settings that default to providing reasonable security for use over the public internet, rather maximising interoperability (but potentially allowing operation in no longer secure modes)

Object lifecycle and resource management has also improved significantly:

- the cyclic garbage collector is now more aggressive in attempting to collect cycles, even those containing `__del__` methods. This eliminated some cases where generators could be flagged as uncollectable (and hence effectively leak memory)
- this means most objects will now have already been cleaned up before the last resort "set module globals to None" step triggers during shutdown, reducing spurious tracebacks when cleanup code runs
- the new `weakref.finalize()` API makes it easier to register weakref callbacks without having to worry about managing the lifecycle of the reference itself
- many more objects in the standard library now support the context management protocol for explicit lifecycle and resource management

Other quality of life improvements include:

- `__init__.py` files are no longer needed to declare packages - if no `foo/__init__.py` file is present, then all directories named `foo` on `sys.path` will be automatically scanned for `foo` submodules
- the new `super` builtin makes calling up to base class method implementations in a way that supports multiple inheritance relatively straightforward
- keyword only arguments make it much easier to add optional parameters to functions in a way that isn't error prone or hard to read
- the `yield from` syntax for delegating to subgenerators and iterators (this is a key part of the `asyncio` coroutine support)
- iterable unpacking syntax is now more flexible
- `zipapp` for bundling pure Python applications into runnable archives
- `enum` for creating enumeration types
- `ipaddress` for working with both IPv4 and IPv6 addresses
- `pathlib` for a higher level filesystem abstraction than the low level interface provided by
- `statistics` for a simple high school level statistics library (mean, median, mode, variance, standard deviation, etc)
- `datetime.timestamp()` makes it easy to convert a datetime object to a UNIX timestamp
- `time.get_clock_info()` and related APIs provide access to a rich collection of cross platform time measurement options
- `venv` provides virtual environment support out of the box, in a way that is better integrated with the core interpreter than is possible in Python 2 with only `virtualenv` available
- `ensurepip` ensures `pip` is available by default in Python 3.4+ installations
- `memoryview` is significantly more capable and reliable
- the caching mechanism for pyc files has been redesigned to better accommodate sharing of Python files between multiple Python interpreters (whether different versions of CPython, or other implementation like PyPy and Jython)
- as part of that change, implicitly compiled bytecode cache files are written to __pycache__ directories (reducing directory clutter) and are ignored if the corresponding source file has been removed (avoiding obscure errors due to stale cached bytecode files)
- `types.SimpleNamespace` and `types.MappingProxyType` are made available at the Python layer

- improved introspection support, based on the `inspect.signature()` API, and its integration into `pydoc`, allowing accurate signature information to be reported for a much wider array of callables than just actual Python function objects
- defining `__eq__` without also defining `__hash__` implicitly disables hashing of instances, avoiding obscure errors when such types were added to dictionaries (you now get an error about an unhashable type when first adding an instance, rather than obscure data driven lookup bugs later)
- ordered comparisons between objects of different types are now disallowed by default (again replacing obscure data driven errors with explicit exceptions)

Some more advanced higher order function manipulation and metaprogramming capabilities are also readily available in Python 3:

- the `functools.partialmethod()` function makes it straightforward to do partial function application in a way that still allows the instance object to be supplied later as a positional argument
- the `functools.singledispatch()` decorator makes it easy to create generic functions that interoperate cleanly with Python's type system, including abstract base classes
- the `contextlib.ExitStack` class makes it easy to manipulate context managers dynamically, rather than having to rely on explicit use of with statements
- The new `__prepare__` method, and associated functions in the `types` module makes it possible for metaclasses to better monitor what happens during class body execution (for example, by using an ordered dictionary to record the order of assignments)
- the updated import system permits easier creation of custom import hooks. In particular, the "source to code" translation step can be overridden, while reusing the rest of the import machinery (including bytecode caching) in a custom import hook
- the `dis.Bytecode` API and related functionality makes it easier to work with CPython bytecode

Various improvements in Python 3 also permitted some significant documentation improvements relative to Python 2:

- as the Python 3 builtin sequences are more compliant with their corresponding abstract base classes, it has proved easier to flesh out their documentation to cover all the additional details that have been introduced since those docs were originally written
- the final removal of the remnants of the legacy import system in Python 3.3 made it feasible to finally document the import system mechanics in the language reference

While many of these features *are* available in Python 2 with appropriate downloads from the Python Package Index, not all of them are, especially the various changes to the core interpreter and related systems.

While Python 2 does still have a longer tail of esoteric modules available on PyPI, most popular third party modules and frameworks either support both, have alternatives that support Python 3. or can be relatively easily ported using tools like `futurize` (part of python-future). The `3to2` project, and the `pasteurize` tool (also part of python-future) offer options for migrating a pure Python 3 application to the large common subset of Python 2 and Python 3 if a critical Python 2 only dependency is identified, and it can't be invoked in a separate Python 2 process, or cost effectively ported to also run on Python 3.

With Python 3 software collections available for both Red Hat Enterprise Linux and CentOS, Ubuntu including a fully supported Python 3 stack in its latest LTS release, and Continuum Analytics releasing Anaconda3 (a Python 3 based version of their scientific software distribution), the number of cases where using Python 2 is preferable to using Python 3 is dwindling to those where:

- for some reason, an application absolutely needs to run in the system Python on Red Hat Enterprise Linux or CentOS (for example, depending on an OS level package that isn't available from PyPI, or needing a complex binary dependency that isn't available for the Python 3 software collection and not being permitted to add additional dependencies from outside the distro)

- the particular application can't tolerate the current integration issues with the POSIX C locale or the Windows command line in environments that actually need full Unicode support
- there's a critical Python 2 only dependency that is known before the project even starts, and separating that specific component out to its own Python 2 process while writing the bulk of the application in Python 3 isn't considered an acceptable architecture

Is Python 3 more convenient than Python 2 in every respect?

At this point in time, not quite. Python 3.5 comes much closer to this than Python 3.4 (which in turn was closer than 3.3, etc), but there are still some use cases that are more convenient in Python 2 because it handles them by default, where Python 3 needs some additional configuration, or even separate code paths for things that could be handled by a common algorithm in Python 2.

In particular, many binary protocols include ASCII compatible segments, so it is sometimes convenient to treat them as text strings. Python 2 makes this easier in many cases, since the 8-bit `str` type blurs the boundary between binary and text data. By contrast, if you want to treat binary data like text in Python 3 in a way that isn't directly supported by the `bytes` type, you actually need to convert it to text first, and make conscious decisions about encoding issues that Python 2 largely lets you ignore. I've written a separate essay specifically about this point: *Python 3 and ASCII Compatible Binary Protocols*.

Python 3 also requires a bit of additional up front design work when aiming to handle improperly encoded data. This also has its own essay: *Processing Text Files in Python 3*.

The Python 3 model also required more complex impedance matching on POSIX platforms, which is covered by a separate question: *What's up with POSIX systems in Python 3?*.

Until Python 3.4, the Python 3 codec system also didn't cleanly handle the transform codecs provided as part of the standard library. Python 3.4 includes several changes to the way these codecs are handled that nudge users towards the type neutral APIs in the codecs module when they attempt to use them with the text encoding specific convenience methods on the builtin types.

Another change that has yet to be fully integrated is the switch to producing dynamic views from the `keys`, `values` and `items` methods of dict objects. It currently isn't easy to implement fully conformant versions of those in pure Python code, so many alternate mapping implementations in Python 3 don't worry about doing so - they just produce much simpler iterators, equivalent to the `iterkeys`, `itervalues` and `iteritems` methods from Python 2.

Some of the changes in Python 3 designed for the benefit of larger applications (like the increased use of iterators), or for improved language consistency (like changing print to be a builtin function rather than a statement) are also less convenient at the interactive prompt. `map`, for example, needs to be wrapped in a `list` call to produce useful output in the Python 3 REPL, since by default it now just creates an iterator, without actually doing any iteration. In Python 2, the fact it combined both defining the iteration and actually doing the iteration was convenient at the REPL, even though it often resulted in redundant data copying and increased memory usage in actual application code.

Having to type the parentheses when using print is mostly an irritation for Python 2 users that need to retrain their fingers. I've personally just trained myself to only use the single argument form (with parentheses) that behaves the same way in both Python 2 and 3, and use string formatting for anything more complex (or else just print the tuple when using the Python 2 interactive prompt). However, I also created a patch that proves it is possible to implement a general implicit call syntax within the constraints of CPython's parsing rules. Anyone that wishes to do so is free to take that patch and turn it into a full PEP that proposes the addition of a general implicit call syntax to Python 3.5 (or later). While such a PEP would need to address the ambiguity problems noted on the tracker issues (likely by restricting the form of the expression used in an implicit call to only permit unqualified names), it's notable that the popular IPython interactive interpreter already provides this kind of implicit "autocall" behaviour by default, and many other languages provide a similar "no parentheses, parameters as suffix" syntax for statements that consist of a single function call.

Thanks are due especially to Armin Ronacher for describing several of these issues in fine detail when it comes to the difficulties they pose specifically when writing wire protocol handling code in Python 3. His feedback has been

invaluable to me (and others) in attempting to make Python 3 more convenient for wire protocol development without reverting to the Python 2 model that favoured wire protocol development over normal application development (where binary data should exist only at application boundaries and be converted to text or other structured data for internal processing). There's still plenty of additional improvements that could be made for Python 3.5 and later, though. Possible avenues for improvement previously discussed on python-dev, python-ideas or the CPython issue tracker include:

- **PEP 461** is an accepted proposal to restore support for *binary* interpolation that is to be source and semantically compatible for the use cases we actually want to support in Python 3.
- **PEP 467** is a draft proposal to clean up some of the legacy of the original Python 3 mutable `bytes` design. A related change is to better document the tuple-of-ints and list-of-ints behaviour of `bytes` and `bytearray`.
- taking the internal "text encoding" marking system added in Python 3.4 and giving either it or a more general codec type description system a public API for use when developing custom codecs.
- making it easier to register custom codecs (preferably making use of the native namespace package support added in Python 3.3).
- adding a new error handler that replaces surrogate escaped bytes with `?` characters in encoded output
- introducing a string tainting mechanism that allows strings containing surrogate escaped bytes to be tagged with their encoding assumption and information about where the assumption was introduced. Attempting to process strings with incompatible encoding assumptions would then report both the incompatible assumptions and where they were introduced.
- creating a "strview" type that uses memoryview to provide a str-like interface to arbitrary binary buffers containing ASCII compatible protocol data.

What's up with WSGI in Python 3?

The process of developing and updating standards can be slow, frustrating and often acrimonious. One of the key milestones in enabling Python 3 adoption was when the web framework developers and web server developers were able to agree on an updated WSGI 1.1 specification that at least makes it *possible* to write WSGI applications, frameworks and middleware that support Python 2 and Python 3 from a single source code base, even though it isn't necessarily easy to do so correctly.

In particular, the Python 2 `str` type was particular well suited to handling the "data in unknown ASCII compatible encoding" that is common in web protocols, and included in the data passed through from the web server to the application (and vice versa). At this point in time (March 2014), nobody has created a type for Python 3 that is similarly well suited to manipulating ASCII compatible binary protocol data. There certainly wasn't any such type available for consideration when WSGI 1.1 was standardised in October 2010.

As a result, the "least bad" option chosen for those fields in the Python 3 version of the WSGI protocol was to publish them to the web application as `latin-1` decoded strings. This means that applications need to treat these fields as wire protocol data (even though they claim to be text based on their type), encode them back to bytes as `latin-1` and then decode them again using the *correct* encoding (as indicated by other metadata).

The WSGI 1.1 spec is definitely a case of a "good enough" solution winning a battle of attrition. I'm actually hugely appreciative of the web development folks that put their time and energy both into creating the WSGI 1.1 specification *and* into updating their tools to support it. Like the Python core developers, most of the web development folks weren't in a position to use Python 3 professionally during the early years of its development, but *unlike* most of the core developers, the kind of code they write falls squarely into the ASCII compatible binary protocol space where Python 3 still had some significant ground to make up relative to Python 2 in terms of usability (although we've also converted our share of such code, just in bringing the standard library up to scratch).

What's up with POSIX systems in Python 3?

Note: This answer was written for Python 3.5. See PEP 538 and PEP 540 for discussion of some key changes now being considered for Python 3.7.

The fact that the Python 2 text model was essentially the POSIX text model with Unicode support bolted on to the side meant that interoperability between Python 2 and even misconfigured POSIX systems was generally quite straightforward - if the implicit decoding as ASCII never triggered (which was likely for code that only included 8-bit strings and never explicitly decoded anything as Unicode), non-ASCII data would silently pass through unmodified.

One option we considered was to just assume everything was UTF-8 by default, similar to the choice made by the Windows .NET platform, the GNOME GUI toolkit and other systems. However, we decided that posed an unacceptable risk of silently corrupting user's data on systems that *were* properly configured to use an encoding other than UTF-8 (this concern was raised primarily by contributors based in Europe and Asia).

This was a deliberate choice of attempting to be compatible with other software on the end user's system at the cost of increased sensitivity to configuration errors in the environment and differences in default behaviour between environments with different configurations. There are also current technical limitations in the reference interpreter's startup code that force us to rely on the locale encoding claimed by the operating system on POSIX systems.

PEP 383 added the surrogateescape error handler to cope with the fact that the configuration settings on POSIX systems aren't always a reliable guide to the *actual* encoding of the data you encounter. One of the most common causes of problems is the seriously broken default encoding for the default locale in POSIX (due to the age of the ANSI C spec where that default is defined, that default is ASCII rather than UTF-8). Bad default environments and environment forwarding in ssh sessions are another source of problems, since an environment forwarded from a client is not a reliable guide to the server configuration, and if the ssh environment defaults to the C/POSIX locale, it will tell Python 3 to use ASCII as the default encoding rather than something more appropriate.

When surrogateescape was added, we considered enabling it for *every* operating system interface by default (including file I/O), but the point was once again made that this idea posed serious risks for silent data corruption on Asian systems configured to use Shift-JIS, ISO-2022, or other ASCII-incompatible encodings (European users were generally in a safer position on this one, since Europe has substantially lower usage of ASCII incompatible codecs than Asia does).

This means we've been judiciously adding surrogateescape to interfaces as we decide the increase in convenience justifies any increased risk of data corruption. For Python 3.5, this is also being applied to `sys.stdin` and `sys.stdout` on POSIX systems that claim that we should be using `ascii` as the default encoding. Such a result almost certainly indicates a configuration error in the environment, but using ascii+surrogateescape in such cases should make for a more usable result than the current approach of ascii+strict. There's still some risk of silent data corruption in the face of ASCII incompatible encodings, but the assumption is that systems that are configured with a non-ASCII compatible encoding should already have relatively robust configurations that avoid ever relying on the default POSIX locale.

This is an area where we're genuinely open to the case being made for different defaults, or additional command line or environment variable configuration options. POSIX is just seriously broken in this space, and we're having to trade-off user convenience against the risk of silent data corruption - that means the "right answer" is *not* obvious, and any PEP proposing a change needs to properly account for the rationale behind the current decision (in particular, it has to account for the technical limitations in the startup code that create the coupling to the default locale encoding reported by the operating system, which may require a change on the scale of PEP 432 to actually fix properly).

What changes in Python 3 have been made specifically to simplify migration?

The biggest change made specifically to ease migration from Python 2 was the reintroduction of Unicode literals in Python 3.3 (in PEP 414). This allows developers supporting both Python 2 and 3 in a single code base to easily distinguish binary literals, text literals and native strings, as `b"binary"` means bytes in Python 3 and str in Python 2, `u"text"` means str in Python 3.3+ and unicode in Python 2, while `"native"` means str in both Python 2 and 3.

The restoration of binary interpolation support in Python 3.5 was designed in such as way as to also serve to make a lot of 8-bit string interpolation operations in Python 2 code "just work" in Python 3.5+.

A smaller change to simplify migration was the reintroduction of the non-text encoding codecs (like `hex_codec`) in Python 3.2, and the restoration of their convenience aliases (like `hex`) in Python 3.4. The `codecs.encode` and `codecs.decode` convenience functions allow them to be used in a single source code base (since those functions have been present and covered by the test suite since Python 2.4, even though they were only added to the documentation recently).

The WSGI update in **PEP 3333** also standardised the Python 3 interface between web servers and frameworks, which is what allowed the web frameworks to start adding Python 3 support with the release of Python 3.2.

A number of standard library APIs that were originally either binary only or text only in Python 3 have also been updated to accept either type. In these cases, there is typically a requirement that the "alternative" type be strict 7-bit ASCII data - use cases that need anything more than that are expected to do their encoding or decoding at the application boundary rather than relying on the implicit encoding and decoding provided by the affected APIs. This is a concession in the Python 3 text model specifically designed to ease migration in "pure ASCII" environments - while relying on it can reintroduce the same kind of obscure data driven failures that are seen with the implicit encoding and decoding operations in Python 2, these APIs are at least unlikely to silently corrupt data streams (even in the presence of data encoded using a non-ASCII compatible encoding).

What other changes have occurred that simplify migration?

The original migration guides unconditionally recommended running an applications test suite using the `-3` flag in Python 2.6 or 2.7 (to ensure no warnings were generated), and then using the `2to3` utility to perform a one-time conversion to Python 3.

That approach is still a reasonable choice for migrating a fully integrated application that can completely abandon Python 2 support at the time of the conversion, but is no longer considered a good option for migration of libraries, frameworks and applications that want to add Python 3 support without losing Python 2 support. The approach of running `2to3` automatically at install time is also no longer recommended, as it creates an undesirable discrepancy between the deployed code and the code in source control that makes it difficult to correctly interpret any reported tracebacks.

Instead, the preferred alternative in the latter case is now to create a single code base that can run under both Python 2 and 3. The six compatibility library can help with several aspects of that, and the python-modernize utility is designed to take existing code that supports older Python versions and update it to run in the large common subset of Python 2.6+ and Python 3.3+ (or 3.2+ if the unicode literal support in Python 3.3 isn't needed).

The "code modernisation" approach also has the advantage of being able to be done incrementally over several releases, as failures under Python 3 can be addressed progressively by modernising the relevant code, until eventually the code runs correctly under both versions. Another benefit of this incremental approach is that this modernisation activity can be undertaken even while waiting for other dependencies to add Python 3 support.

More recently, the python-future project was created to assist those developers that would like to primarily write Python 3 code, but would also like to support their software on Python 2 for the benefit of potential (or existing) users that are not themselves able to upgrade to Python 3.

The addition of the `pylint --py3k` flag was designed to make it easier for folks to ensure that code migrated to the common subset of Python 2 and Python 3 remained there rather than reintroducing Python 2 only constructs.

The landing page for the Python documentation was also switched some time ago to display the Python 3 documentation by default, although deep links still refer to the Python 2 documentation in order to preserve the accuracy of third party references (see **PEP 430** for details).

What future changes in Python 3 are expected to further simplify migration?

Most of the changes designed to further simplify migration landed in Python 3.5.

One less obviously migration related aspect of those changes is that the new gradual typing system is designed to allow Python 2 applications to be typechecked as if they were Python 3 applications, and hence many potential porting problems detected even if they're not covered by tests, or the test suite can't yet be run on Python 3.

Didn't you strand the major alternative implementations on Python 2?

Cooperation between the major implementations (primarily CPython, PyPy, Jython, IronPython, but also a few others) has never been greater than it has been in recent years. The core development community that handles both the language definition and the CPython implementation includes representatives from all of those groups.

The language moratorium that severely limited the kinds of changes permitted in Python 3.2 was a direct result of that collaboration - it gave the other implementations breathing room to catch up to Python 2.7. That moratorium was only lifted for 3.3 with the agreement of the development leads for those other implementations. Significantly, one of the most disruptive aspects of the 3.x transition for CPython and PyPy (handling all text as Unicode data) was already the case for Jython and IronPython, as they use the string model of the underlying JVM and CLR platforms.

We have also instituted new guidelines for CPython development which require that new standard library additions be granted special dispensation if they are to be included as C extensions without an API compatible Python implementation.

Python 3 specifically introduced `ResourceWarning`, which alerts developers when they are relying on the garbage collector to clean up external resources like sockets. This warning is off by default, but switched on automatically by many test frameworks. The goal of this warning is to detect any cases where `__del__` is being used to clean up a resource, such as a file or socket or database connection. Such cases are then updated to use either explicit resource management (via a `with` or `try` statement) or else switched over to `weakref` if non-deterministic cleanup is considered appropriate (the latter is quite rare in the standard library). The aim of this effort is specifically to ensure that the entire standard library will run correctly on Python implementations that don't use refcounting for object lifecycle management.

Finally, Python 3.3 converted the bulk of the import system over to pure Python code so that all implementations can finally start sharing a common import implementation. Some work will be needed from each implementation to work out how to bootstrap that code into the running interpreter (this was one of the trickiest aspects for CPython), but once that hurdle is passed all future import changes should be supported with minimal additional effort.

All that said, there's often a stark difference in the near term *goals* of the core development team and the developers for other implementations. Criticism of the Python 3 project has been somewhat vocal from a number of PyPy core developers, and that makes sense when you consider that one of the core aims of PyPy is to provide a better runtime for *existing* Python applications. However, despite those reservations, PyPy was still the first of the major alternative implementations to support Python 3 (with the initial release of their PyPy3 runtime in June 2014). The initial PyPy3 release targeted Python 3.2 compatibility, but the changes needed to catch up on subsequent Python 3 releases are relatively minor compared to the changes between Python 2 and Python 3, and the PyPy team received a funded development grant from Mozilla to bring PyPy3 at least up to Python 3.5 compatibility. Work also continues on another major compatibility project for PyPy, numpypy, which aims to integrate PyPy with the various components of the scientific Python stack.

Note: The info below on Jython and IronPython is currently quite dated. This section should also be updated to mention the new Python 3 only bytecode-focused implementations targeting the JVM (BeeWare's VOC), and JavaScript runtimes (BeeWare's Batavia)

Jython's development efforts are currently still focused on getting their currently-in-beta Python 2.7 support to a full release, and there is also some significant work happening on JyNI (which, along the same lines as PyPy's numpypy

project, aims to allow the use of the scientific Python stack from the JVM).

The IronPython folks have started working on a Python 3 compatible version, but there currently isn't a target date for a release. IronClad already supports the use of scientific libraries from IronPython.

One interesting point to note for Jython and IronPython is that the changes to the Python 3 text model bring it more into line with the text models of the JVM and the CLR. This may mean that projects updated to run in the common subset of Python 2 and 3 will be more likely to run correctly on Jython and IronPython, and once they implement Python 3 support, the compatibility of Python 3 only modules should be even better.

Aren't you abandoning Python 2 users?

We're well aware of this concern, and have taken what steps we can to mitigate it.

First and foremost is the extended maintenance period for the Python 2.7 release. We knew it would take some time before the Python 3 ecosystem caught up to the Python 2 ecosystem in terms of real world usability. Thus, the extended maintenance period on 2.7 to ensure it continues to build and run on new platforms. While python-dev maintenance of 2.7 was originally slated to revert to security-fix only mode in July 2015, Guido extended that out to 2020 at PyCon 2014. We're now working with commercial redistributors to help ensure the appropriate resources are put in place to actually meet that commitment. In addition to the ongoing support from the core development team, 2.6 will still be supported by enterprise Linux vendors until at least 2020, while Python 2.7 will be supported until at least 2024.

We have also implemented various mechanisms which are designed to ease the transition from Python 2 to Python 3. The `-3` command line switch in Python 2.6 and 2.7 makes it possible to check for cases where code is going to change behaviour in Python 3 and update it accordingly.

The automated `2to3` code translator can handle many of the mechanical changes in updating a code base, and the python-modernize variant performs a similar translation that targets the (large) common subset of Python 2.6+ and Python 3 with the aid of the six compatibility module, while *python-future* does something similar with its `futurize` utility.

PEP 414 was implemented in Python 3.3 to restore support for explicit Unicode literals primarily to reduce the number of purely mechanical code changes being imposed on users that are doing the right thing in Python 2 and using Unicode for their text handling.

One outcome of some of the discussions at PyCon 2014 was the `pylint --py3k` utility to help make it easier for folks to migrate software incrementally and opportunistically, first switching to the common subset running on Python 2.7, before migrating to the common subset on Python 3.

So far we've managed to walk the line by persuading our Python 2 users that we aren't going to leave them in the lurch when it comes to appropriate platform support for the Python 2.7 series, thus allowing them to perform the migration on their own schedule as their dependencies become available, while doing what we can to ease the migration process so that following our lead remains the path of least resistance for the future evolution of the Python ecosystem.

PEP 404 (yes, the choice of PEP number is deliberate - it was too good an opportunity to pass up) was created to make it crystal clear that python-dev has no intention of creating a 2.8 release that backports 2.x compatible features from the 3.x series. After you make it through the opening Monty Python references, you'll find the explanation that makes it unlikely that anyone else will take advantage of the "right to fork" implied by Python's liberal licensing model: we had very good reasons for going ahead with the creation of Python 3, and very good reasons for discontinuing the Python 2 series. We didn't decide to disrupt an entire community of developers just for the hell of it - we did it because there was a core problem in the language design, and a backwards compatibility break was the only way we could find to solve it once and for all.

For the inverse question relating to the concern that the existing migration plan is too *conservative*, see *But uptake is so slow, doesn't this mean Python 3 is failing as a platform?*.

What would it take to make you change your minds about the current plan?

With both the Debian/Ubuntu and Fedora/RHEL/CentOS ecosystems well advanced in their migration plans, public cloud providers offering Python 3 in addition to Python 2, major commercial end users like Facebook, Google and Dropbox migrating, and the PSF's own major services like python.org and the Python Package Index switching to Python 3, the short answer here is "That's not going to happen".

While a crash in general Python adoption might have made us change our minds, Python ended up working its way into more and more niches *despite* the Python 3 transition, so the only case that could be made is "adoption would be growing even faster without Python 3 in the picture", which is a hard statement to prove (particularly when we suspect that at least some of the growth in countries where English is not the primary spoken language is likely to be *because* of Python 3 rather than in spite of it, and that the Python 3 text model is in a much better position to serve as a bridge between the POSIX text model and the JVM and CLR text models than the Python 2 model ever was).

Another scenario that would have made us seriously question our current strategy is if professional educators had told us that Python 2 was a better teaching language, but that didn't happen - they're amongst Python 3's more vocal advocates, encouraging the rest of the community to "just upgrade already".

Wouldn't a Python 2.8 release help ease the transition?

In a word: no. In several words: maybe, but at such a high cost, the core development team consider it a much better idea to invest that effort in improving Python 3, migration tools and helping to port libraries and applications (hence why credible contributors can apply to the PSF for a grant to help port key libraries to Python 3, but PSF funding isn't available for a Python 2.8 release).

The rationale for this proposal appears to be that if backporting Python 3 changes to Python 2.6 and 2.7 was a good idea to help Python 3 adoption, then continuing to do so with a new Python 2.8 release would also be a good idea.

What this misses is that those releases were made during a period when the core development team was still in the process of ensuring that Python 3 was in a position to stand on its own as a viable development platform. We *didn't want* conservative users that were currently happy with Python 2 to migrate at that point, as we were still working out various details to get it back to feature parity with Python 2. One of the most notable of those was getting a usable WSGI specification back in 3.2, and another being the restoration of Unicode literals in 3.3 to help with migration from Python 2.

If we hadn't considered Python 3.2 to be at least back to parity with Python 2.7, *that* is when we would have decided to continue on to do a Python 2.8 release. We're even less inclined to do so now that Python 3 has several additional years of feature development under its belt relative to the Python 2 series.

There *are* parts of the Python 3 standard library that are also useful in Python 2. In those cases, they're frequently available as backports on the Python Package Index (including even a backport of the new asynchronous IO infrastructure).

There are also various language level changes that are backwards compatible with Python 2.7, and the Tauthon project was started specifically to create a hybrid runtime implementation that expanded the "common subset" of Python 2 & 3 to include those additional features.

However, I think a key point that is often missed in these discussions is that the adoption cycles for new versions of the core Python runtime have *always* been measured in years due to the impact of stable platforms like Red Hat Enterprise Linux.

Consider the following map of RHEL/CentOS versions to Python versions (release date given is the *Python* release date, and Python 2.5 was skipped due to RHEL5 being published not long before it was released in September 2006):

- 4 = 2.3 (first released July 2003)
- 5 = 2.4 (first released November 2004)
- 6 = 2.6 (first released October 2008)

- 7 = 2.7 (first released July 2010)

Now consider these Twisted compatibility requirements (going by the modification dates on the tagged INSTALL file):

- 10.0 dropped Python 2.3 in March 2010
- 10.2 dropped Python 2.4 (Windows) in November 2010
- 12.0 dropped Python 2.4 (non-Windows) in February 2012
- 12.2 dropped Python 2.5 in August 2012
- 15.4 dropped Python 2.6 in September 2015

Python 2.6 compatibility was still required more than 7 years after its original release, and didn't get dropped until well after the first CentOS 7 release was available (not to mention the earlier release of a Python 2.7 SCL).

I believe Twisted has one of *the* most conservative user bases in the Python community, and I consider this one of the main reasons we see this general pattern of only dropping support for an older release 6-7 years after it was first made available. That's also why I considered the Twisted developers a key audience for any increases in the scope of single source support in Python 3.5 (and their support for the idea was certainly one of the factors behind the planned return of binary interpolation support).

That's the way the path to Python 3 will be smoothed at this point: by identifying blockers to migration and knocking them down, one by one. The PSF has helped fund the migration of some key libraries. Barry Warsaw drove a fair amount of Python 3 migration work for Ubuntu at Canonical. Victor Stinner is working hard to encourage and support the OpenStack migration. I have been offering advice and encouragement to Bohuslav Kabrda (the main instigator of Fedora's migration to Python 3), Petr Viktorin, and other members of Red Hat's Python maintenance team, as well as helping out with Fedora policy recommendations on supporting parallel Python 2 and 3 stacks (I have actually had very little to do with Red Hat's efforts to support Python 3 overall, as I haven't needed to. Things like Python 3 support in Red Hat Software Collections and OpenShift Online happened because other folks at Red Hat made sure they happened). Guido approved the restoration of Unicode literal support after web framework developers realised they couldn't mask that particular change for their users, and he has also approved the restoration of binary interpolation support. I went through and made the binary transform codecs that had been restored in Python 3.2 easier to discover and use effectively in Python 3.4. R. David Murray put in a lot of time and effort to actually handle Unicode sensibly in the `email` module, Brett Cannon has been updating the official migration guide based on community feedback, etc, etc (I'm sure I'm missing a bunch of other relevant changes).

Outside of CPython and its documentation, Benjamin Peterson published the `six`, Lennart Regebro put together his excellent guide for porting, Armin Ronacher created `python-modernize` and Ed Schofield created `python-future`. Multiple folks have contributed patches to a wide variety of projects to allow them to add Python 3 support.

Aren't you concerned Python 2 users will abandon Python over this?

Certainly - a change of this magnitude is sufficiently disruptive that many members of the Python community are legitimately upset at the impact it has had on them.

This is particularly the case for users that had never personally been bitten by the broken Python 2 Unicode model, either because they work in an environment where almost all data is encoded as ASCII text (increasingly uncommon, but still not all that unusual in English speaking countries) or else in an environment where the appropriate infrastructure is in place to deal with the problem even in Python 2 (for example, web frameworks hide most of the problems with the Python 2 approach from their users).

Another category of users are upset that we chose to stop adding new features to the Python 2 series, and have been quite emphatic that attempts to backport features (other than via PyPI modules like `unittest2`, `contextlib2` and `configparser`) are unlikely to receive significant support from python-dev. As long as they don't attempt to present themselves as providing official Python releases, we're not *opposed* to such efforts - it's merely the case that

(outside a few specific exceptions like **PEP 466**) we aren't interested in doing them ourselves, and are unlikely to devote significant amounts of time to assisting those that *are* interested.

A third category of user negatively affected by the change are those users that deal regularly with binary data formats and had mastered the idiosyncrasies of the Python 2 text model to the point where writing correct code using that model was effortless. The kinds of hybrid binary-or-text APIs that the `str` type made easy in Python 2 can be relatively awkward to write and maintain in Python 3 (or in the common subset of the two languages). While native Python 3 code can generally simply avoid defining such APIs in the first place, developers porting libraries and frameworks from Python 2 generally have little choice, as they have to continue to support both styles of usage in order to allow their *users* to effectively port to Python 3.

However, we have done everything we can to make migrating to Python 3 the easiest exit strategy for Python 2, and provided a fairly leisurely time frame for the user community to make the transition. Full maintenance of Python 2.7 has now been extended to 2020, source only security releases may continue for some time after that, and, as noted above, I expect enterprise Linux vendors and other commercial Python redistributors to continue to provide paid support for some time after community support ends.

Essentially, the choices we have set up for Python 2 users that find Python 3 features that are technically backwards compatible with Python 2 attractive are:

- Live without the features for the moment and continue to use Python 2.7
- For standard library modules/features, use a backported version from PyPI (or create a backport if one doesn't already exist and the module doesn't rely specifically on Python 3 only language features)
- Migrate to Python 3 themselves
- Fork Python 2 to add the missing features for their own benefit
- Migrate to a language other than Python

The first three of those approaches are all fully supported by python-dev. Many standard library additions in Python 3 started as modules on PyPI and thus remain available to Python 2 users. For other cases, such as `unittest` or `configparser`, the respective standard library maintainer also maintains a PyPI backport.

The fourth choice exists as the Tauthon project, so it will be interesting to see if that gains significant traction with developers and platform providers.

The final choice would be unfortunate, but we've done what we can to make the other alternatives (especially the first three) more attractive.

Doesn't this make Python look like an immature and unstable platform?

Again, many of us in core development are aware of this concern, and have been taking active steps to ensure that even the most risk averse enterprise users can feel comfortable in adopting Python for their development stack, despite the current transition.

Obviously, much of the content in the answers above regarding the viability of Python 2 as a development platform, with a clear future migration path to Python 3, is aimed at enterprise users. Government agencies and large companies are the environments where risk management tends to come to the fore, as the organisation has something to lose. The start up and open source folks are far more likely to complain that the pace of Python core development is *too slow*.

The main change to improve the perceived stability of Python 3 is that we've started making greater use of the idea of "documented deprecation". This is exactly what it says: a pointer in the documentation to say that a particular interface has been replaced by an alternative we consider superior that should be used in preference for new code. We have no plans to remove any of these APIs from Python - they work, there's nothing fundamentally wrong with them, there is just an updated alternative that was deemed appropriate for inclusion in the standard library.

Programmatic deprecation is now reserved for cases where an API or feature is considered so fundamentally flawed that using it is very likely to cause bugs in user code. An example of this is the deeply flawed `contextlib.`

`nested` API which encouraged a programming style that would fail to correctly close resources on failure. For Python 3.3, it was finally replaced with a superior incremental `contextlib.ExitStack` API which supports similar functionality without being anywhere near as error prone.

Secondly, code level deprecation warnings are now silenced by default. The expectation is that test frameworks and test suites will enable them (so developers can fix them), while they won't be readily visible to end users of applications that happen to be written in Python. (This change can actually cause problems with ad hoc user scripts breaking when upgrading to a newer version of Python, but the longevity of Python 2.7 actually works in our favour on that front)

Finally, and somewhat paradoxically, the introduction of *provisional APIs* in Python 3 is a feature largely for the benefit of enterprise users. This is a documentation marker that allows us to flag particular APIs as potentially unstable. It grants us a full release cycle (or more) to ensure that an API design doesn't contain any nasty usability traps before declaring it ready for use in environments that require rock solid backwards compatibility guarantees.

Why wasn't I consulted?

Technically, even the core developers weren't consulted: Python 3 happened because the creator of the language, Guido van Rossum, wanted it to happen, and Google paid for him to devote half of his working hours to leading the development effort.

In practice, Guido consults extensively with the other core developers, and if he can't persuade even us that something is a good idea, he's likely to back down. In the case of Python 3, though, it is our collective opinion that the problems with Unicode in Python 2 are substantial enough to justify a backwards compatibility break in order to address them, and that continuing to maintain both versions in parallel indefinitely would not be a good use of limited development resources.

We as a group also continue to consult extensively with the authors of other Python implementations, authors of key third party frameworks, libraries and applications, our own colleagues and other associates, employees of key vendors, Python trainers, attendees at Python conferences, and, well, just about anyone that cares enough to sign up to the python-dev or python-ideas mailing lists or add their Python-related blog to the Planet Python feed, or simply discuss Python on the internet such that the feedback eventually makes it way back to a place where we see it.

Some notable changes within the Python 3 series, specifically **PEP 3333** (which updated the Web Server Gateway Interface to cope with the Python 3 text model) and **PEP 414** (which restored support for explicit Unicode literals) have been driven primarily by the expressed needs of the web development community in order to make Python 3 better meet their needs.

The restoration of binary interpolation support in Python 3.5 is similarly intended to increase the size of the common subset of Python 2 and Python 3 in a way that makes it easier for developers to migrate to the new version of the language (as well as being a useful new feature for Python 3 in its own right).

If you want to keep track of Python's development and get some idea of what's coming down the pipe in the future, it's all available on the internet.

But <name> says Python 3 was a waste of time/didn't help/made things worse!

One previously popular approach to saying why Python 2 should be used over Python 3 even for *new* projects was to appeal to the authority of someone like Armin Ronacher (creator of Jinja2, Flask, Click, etc) or Greg Wilson (creator of Software Carpentry).

The piece missing from that puzzle is the fact that Guido van Rossum, the creator of Python, *and* every core developer of CPython, have not only been persuaded that the disruption posed by the Python 3 transition is worth the effort, but have been busily adding the features we notice missing from both Python 2 and 3 solely to the Python 3 series since the feature freeze for Python 2.7 back in 2010.

Where's the disconnect? Well, it arises in a couple of ways. Firstly, when creating Python 3, we *deliberately made it worse than Python 2* in particular areas. That sounds like a ridiculous thing for a language design team to do, but

programming language design is a matter of making trade-offs and if you try to optimise for everything at once, you'll end up with an unreadable mess that isn't optimised for anything. In many of those cases, we were trading problems we considered unfixable for ones that could at least be solved in theory, even if they haven't been solved *yet*.

In Armin's case, the disconnect was that his primary interest is in writing server components for POSIX systems, and cross-platform command clients for those applications. This runs into issues, because Python 3's operating system integration could get confused in a few situations:

- on POSIX systems (other than Mac OS X), in the default C locale
- on POSIX systems (other than Mac OS X), when ssh environment forwarding configures a server session with the client locale and the client and server have differing locale settings
- at the Windows command line

This change is due to the fact that where Python 2 decodes from 8-bit data to Unicode text *lazily* at operating system boundaries, Python 3 does so *eagerly*. This change was made to better accommodate Windows systems (where the 8-bit APIs use the mbcs codec, rendering them effectively useless), but came at the cost of being more reliant on receiving correct encoding and decoding advice from the operating system. Operating systems are normally pretty good about providing that info, but they fail hard in the above scenarios.

In almost purely English environments, none of this causes any problems, just as the Unicode handling defects in Python 2 tend not to cause problems in such environments. In the presence of non-English text however, we had to decide between cross-platform consistency (i.e. assuming UTF-8 everywhere), and attempting to integrate correctly with the encoding assumptions of other applications on the same system. We opted for the latter approach, primarily due to the dominance of ASCII incompatible encodings in East Asian countries (ShiftJIS, ISO-2022, GB-18030, various CJK codecs, etc). For ordinary user space applications, including the IPython Notebook, this already works fine. For other code, we're now working through the process of assuming UTF-8 as the default binary encoding when the operating system presents us with dubious encoding recommendations (that will be a far more viable assumption in 2016 than it was in 2008).

For anyone that would like to use Python 3, but is concerned by Armin Ronacher's comments, the best advice I can offer is to *use his libraries* to avoid those problems. Seriously, the guy's brilliant - you're unlikely to go seriously wrong in deciding to use his stuff when it applies to your problems. It offers a fine developer experience, regardless of which version of Python you're using. His complaints are about the fact that *writing those libraries* became more difficult in Python 3 in some respects, but he gained the insight needed to comprehensively document those concerns the hard way: by porting his code. His feedback on the topic was cogent and constructive enough that it was cited as one of the reasons he received a Python Software Foundation Community Service Award in October 2014.

The complaints from the Software Carpentry folks (specifically Greg Wilson) were different. Those were more about the fact that we hadn't done a very good job of explaining the problems that the Python 3 transition was designed to fix. This is an example of something Greg himself calls "the curse of knowledge": experts don't necessarily know what other people *don't know*. In our case, we thought we were fixing bugs that tripped up everyone. In reality, what we were doing was fixing things that *we* thought were still too hard, even with years (or decades in some cases) of Python experience. We'd waste memory creating lists that we then just iterated over and threw away, we'd get our Unicode handling wrong so our applications broke on Windows narrow builds (or just plain broke the first time they encountered a non-ASCII character or text in multiple encodings), we'd lose rare exception details because we had a latent defect in an error handler. We baked fixes for all of those problems (and more) *directly into the design* of Python 3, and then became confused when other Python users tried to tell us Python 2 wasn't broken and they didn't see what Python 3 had to offer them. So we're now in a position where we're having to unpack years (or decades) of experience with Python 2 to explain why we decided to put that into long term maintenance mode and switch our feature development efforts to Python 3 instead.

After hearing Greg speak on this, I'm actually really excited when I hear Greg say that Python 3 is no harder to learn than Python 2 for English speakers, as we took some of the more advanced concepts from Python 2 and made them *no longer optional* when designing Python 3. The Python 3 "Hello World!" now introduces users to string literals, builtins, function calls and expression statements, rather than just to string literals and a single dedicated print statement. Iterators arrive much earlier in the curriculum than they used to, as does Unicode. The chained exceptions

that are essential for improving the experience of debugging obscure production failures can present some readability challenges for new users. If we've managed to front load all of that hard earned experience into the base design of the language and the end result is "just as easy to learn as Python 2", then I'm *happy* with that. It means we were wrong when we thought we were making those changes for the benefit of beginners - it turns out English speaking beginners aren't at a point where the issues we addressed are even on their radar as possible problems. But Greg's feedback now suggests to me that we have actually succeeded in removing some of the barriers between competence and mastery, without *harming* the beginner experience. There are also other changes in Python 3, like the removal of the "__init__.py" requirement for package directories, the improvements to error messages when functions are called incorrectly, the inclusion of additional standard library modules like statistics, asyncio and ipaddress, the bundling of pip, and more automated configuration of Windows systems in the installer that should genuinely improve the learning experience for new users.

Greg's also correct that any *renaming* of existing standard library functionality should be driven by objective user studies - we learned that the hard way by discovering that the name changes and rearrangements we did in the Python 3 transition based on our own intuition were largely an annoying waste of time that modules like `six` and `future` now have to help folks moving from Python 2 to Python 3 handle. However, we're not exactly drowned in offers to do that research, so unless someone can figure out how to get it funded and executed, it isn't going to happen any time soon. As soon as someone does figure that out, though, I look forward to seeing Python Enhancement Proposals backed specifically by research done to make the case for particular name changes, including assessments of the additional cognitive load imposed by students having to learn both the new names suggested by the usability research and the old names that will still have to be kept around for backwards compatibility reasons. In the meantime, we'll continue with the much lower cost "use expert intuition and arguing on the internet to name new things, leave the names of existing things alone" approach. That low cost option almost certainly doesn't find *optimal* names for features, but it does tend to find names that are *good enough*.

The other piece that we're really missing is feedback from folks teaching Python to users in languages *other than English*. Much of the design of Python 3 is aimed at working better with East Asian and African languages where there are no suitable 8-bit encodings - you really need the full power of Unicode to handle them correctly. With suitable library support, Python 2 can be made to handle those languages at the application level, but Python 3 aims to handle them at the language and interpreter level - Python shouldn't fail just because a user is attempting to run it from their home directory and their name can't be represented using the latin-1 alphabet (or koi8-r, or some other 8-bit encoding). Similarly, naming a module in your native language shouldn't mean that Python can't import it, but in Python 2, module names (like all identifiers) are limited to the ASCII character set. Python 3 lifts the limitations on non-ASCII module names and identifiers in general, meaning that imposing such restrictions enters the domain of project-specific conventions that can be enforced with tools like pylint, rather than being an inherent limitation of the language itself.

But, but, surely fixing the GIL is more important than fixing Unicode...

With Eric Snow's publication of his intent to investigate enhancing CPython's existing subinterpreter model to provide native support for Communicating Sequential Processes based parallel execution, the discussion of Python's multicore processing support that previously appeared here has been moved out to its own *article*.

Well, why not just add JIT compilation, then?

Note: This answer was written for Python 3.5. While CPython 3.6 still doesn't ship with a JIT compiler by default, it *does* ship with a dynamic frame evaluation hook that allows third party method JITs like Pyjion to be enabled at runtime.

This is another one of those changes which is significantly easier said than done - the problem is with the "just", not the "add JIT compilation". Armin Rigo (one of the smartest people I've had the pleasure of meeting) tried to provide one as an extension module (the `psyco` project) but eventually grew frustrated with working within CPython's limitations

and even the limitations of existing compiler technology, so he went off and invented an entirely new way of building language interpreters instead - that's what the `PyPy` project is, a way of writing language interpreters that also gives you a tracing JIT compiler, almost for free.

However, while PyPy is an amazing platform for running Python *applications*, the extension module compatibility problems introduced by using a different reference counting mechanism mean it isn't yet quite as good as CPython as an *orchestration* system, so those users in situations where their Python code isn't the performance bottleneck stick with the simpler platform. That currently includes scientists, Linux vendors, Apple, cloud providers and so on and so forth. As noted above when discussing the possible future of concurrency in Python, it seems entirely plausible to me that PyPy will eventually become the default *application* runtime for Python software, with CPython being used primarily as a tool for handling orchestration tasks and embedding in other applications, and only being used to run full applications if PyPy isn't available for some reason. That's going to take a while though, as vendors are currently still wary of offering commercial support for PyPy, not through lack of technical merit, but simply because it represents an entirely new way of creating software and they're not sure if they trust it yet (they'll likely get over those reservations eventually, but it's going to take time - as the CPython core development team have good reason to know, adoption of new platforms is a slow, complex business, especially when many users of the existing platform don't experience the problem that the alternative version is aiming to solve).

While PyPy is a successful example of creating a *new* Python implementation with JIT compilation support (Jython and IronPython benefit from the JIT compilation support in the JVM and CLR respectively), the Unladen Swallow project came about when some engineers at Google made a second attempt at adding a JIT compiler directly to the CPython code base.

The Unladen Swallow team did have a couple of successes: they made several improvements to LLVM to make it more usable as a JIT compiler, and they put together an excellent set of Python macro benchmarks that are used by both PyPy and CPython for relative performance comparisons to this day. However, even though Guido gave in principle approval for the idea, one thing they *didn't* succeed at doing is adding implicit JIT compilation support directly to CPython.

The most recent attempt at adding JIT compilation to CPython is a project called Numba, and similar to `psyco`, Numba doesn't attempt to provide *implicit* JIT compilation of arbitrary Python code. Instead, you have to decorate the methods you would like accelerated. The advantage of this is that it means that Numba *doesn't* need to cope with the full dynamism of Python the way PyPy does - instead, it can tweak the semantics within the decorated functions to reduce the dynamic nature of the language a bit, allowing for simpler optimisation.

Anyone that is genuinely interested in getting implicit JIT support into the default CPython implementation would do well to look into resurrecting the speed.python.org project. Modelled after the speed.pypy.org project (and using the same software), this project has foundered for lack of interested volunteers and leadership. It comes back to the problem noted above - if you're using Python for orchestration, the Python code becoming a bottleneck is usually taken as indicating an architectural issue rather than the Python runtime being too slow.

The availability of PyPy limits the appeal of working on adding JIT compilation to CPython as a volunteer or sponsoring it as a commercial user even further - if all of the extensions an application needs are also available on PyPy, then it's possible to just use that instead, and if they *aren't* available, then porting them or creating alternatives with *cffi* or a pure Python implementation is likely to be seen as a more interesting and cost effective solution than attempting to add JIT compilation support to CPython.

I actually find it quite interesting - the same psychological and commercial factors that work against creating Python 2.8 and towards increasing adoption of Python 3 also work *against* adding JIT compilation support to CPython and towards increasing adoption of PyPy for application style workloads.

What about <insert other shiny new feature here>?

The suggestions that adding a new carrot like free threading or a JIT compiler to Python 3 would suddenly encourage users that are happy with Python 2 to migrate generally misunderstand the perspective of conservative users.

Early adopters are readily attracted by shiny new features - that's what makes them early adopters. And we're very grateful to the early adopters of Python 3 - without their interest and feedback, there's no way the new version of the language would have matured as it has over the last several years.

However, the kinds of things that attract conservative users are very different - they're not as attracted by shiny new features as they are by reliability and support. For these users, the question isn't necessarily "Why would I start using Python 3?", it is more likely to be "Why would I stop using Python 2?".

The efforts of the first several years of Python 3 deployment were about positioning it to start crossing that gap between early adopters and more conservative users. Around 2014, those pieces started falling into place, especially as more enterprise Linux vendors brought supported Python 3 offerings to market.

This means that while conservative users that are *already* using Python are likely to stick with Python 2 for the time being ("if it isn't broken for us, why change it?"), *new* conservative users will see a fully supported environment, and 3 is a higher number than 2, even if the ecosystem still has quite a bit of catching up to do (conservative users aren't going to be downloading much directly from PyPI either - they often prefer to outsource that kind of filtering to software vendors rather than doing it themselves).

Efficiently Exploiting Multiple Cores with Python

Published 21st June, 2015

Last Updated 21st June, 2015

Both the Python reference interpreter (CPython), and the alternative interpeter that offers the fastest single-threaded performance for pure Python code (PyPy) use a Global Interpreter Lock to avoid various problems that arise when using threading models that implicitly allowing concurrent access to objects from multiple threads of execution.

This approach has been the source of much debate, both online and off-, so this article aims to summarise the design trade-offs involved, and give details on some of the prospects for improvement that are being investigated.

Why is using a Global Interpreter Lock (GIL) a problem?

The key issue with Python implementations that rely on a GIL (most notably CPython and PyPy) is that it makes them entirely unsuitable for cases where a developer wishes to:

- use shared memory threading to exploit multiple cores on a single machine
- write their entire application in Python, including CPU bound elements
- use CPython or PyPy as their interpreter

This combination of requirements simply doesn't work - the GIL effectively restricts bytecode execution to a single core, thus rendering pure Python threads an ineffective tool for distributing CPU bound work across multiple cores.

At this point, one of those requirements has to give. The developer has to either:

- use a parallel execution technique other than shared memory threading

  The main alternative provided in the standard library for CPU bound applications is the multiprocessing module, which works well for workloads that consist of relatively small numbers of long running computational tasks, but results in excessive message passing overhead if the duration of individual operations is short

- move parts of the application out into binary extension modules, including wrappers for existing third party libraries

  This is the path taken by the NumPy/SciPy community, Cython users and many other people using Python as a glue language to bind disparate components together

- use a Python implementation that doesn't rely on a GIL

  While the main purpose of Jython and IronPython is to interoperate with other JVM and CLR components, they are also free threaded thanks to the cross-platform threading primitives provide by the underlying virtual machines.

- use a language other than Python for the entire application

  This is a popular approach for established systems where the problem domain is now well understood and the service's scope is stable. In these kinds of situations, efficiency-of-execution considerations start to weigh more heavily than ease-of-modification considerations in the choice of development language, which tends to count heavily against languages like Python that deliberately avoid doing any kind of inter-module consistency analysis at compile time.

  This approach also works very well for applications that happen to fall entirely within the purview of more specialised languages, such as JavaScript for web service development, Go for network services and command line applications, and Julia for data analysis.

Many Python developers find this annoying - they want to use threads to take full advantage of multicore machines *and* they want to use Python, but they have the CPython and PyPy core developers in their way saying "Sorry, we don't recommend that style of programming".

What alternative approaches are available?

Assuming that a free-threaded Python implementation like Jython or IronPython isn't suitable for a given application, then there are two main approaches to handling distribution of CPU bound Python workloads across multiple cores in the presence of a GIL. Which one will be more appropriate will depend on the specific task and developer preference.

The approach most directly supported by python-dev is the use of process-based concurrency rather than thread-based concurrency. All major threading APIs have a process-based equivalent, allowing threading to be used for concurrent synchronous IO calls, while multiple processes can be used for concurrent CPU bound calculations in Python code. The strict memory separation imposed by using multiple processes also makes it much easier to avoid many of the common traps of multi-threaded code. As another added bonus, for applications which would benefit from scaling beyond the limits of a single machine, starting with multiple processes means that any reliance on shared memory will already be gone, removing one of the major stumbling blocks to distributed processing.

The main downside of this approach is that the overhead of message serialisation and interprocess communication can significantly increase the response latency and reduce the overall throughput of an application (see this PyCon 2015 presentation from David Beazley for some example figures). Whether or not this overhead is considered acceptable in any given application will depend on the relative proportion of time that application ends up spending on interprocess communication overhead versus doing useful work.

The major alternative approach promoted by the community is best represented by Cython. Cython is a Python superset designed to be compiled down to CPython C extension modules. One of the features Cython offers (as is possible from any binary extension module) is the ability to explicitly release the GIL around a section of code. By releasing the GIL in this fashion, Cython code can fully exploit all cores on a machine for computationally intensive sections of the code, while retaining all the benefits of Python for other parts of the application.

Numba is another tool in a similar vein - it uses LLVM to convert Python code to machine code that can run with the GIL released (as well as exploiting vector operations provided by the CPU when appopriate).

This approach also works when calling out to *any* code written in other languages: release the GIL when handing over control to the external library, reacquire it when returning control to the Python interpreter. Many binary extension modules for Python already do this implicitly (especially those developed by the members of the Python community focused on data analysis tasks).

Why hasn't resolving this been a priority for the core development team?

Speaking for myself, I came to Python by way of the unittest module: I needed to write a better test suite for a C++ library that communicated with a custom DSP application, and by using SWIG and the Python unittest module I was able to do so easily. Using Python for the test suite also let me easily play audio files out of the test hardware into the DSP unit being tested. Still in the test domain, I later used Python to communicate with serial hardware (and push data through serial circuits and analyse what came back), write prototype clients to ensure new hardware systems were full functional replacements for old ones and write hardware simulators to allow more integration errors to be caught during software development rather than only after new releases were deployed to the test lab that had real hardware available.

Other current Python users are often in a similar situation: we're using Python as an orchestration language, getting other pieces of hardware and software to play nice, so the Python components just need to be "fast enough", and allow multiple *external* operations to occur in parallel, rather than necessarily needing to run Python bytecode operations concurrently. When our Python code isn't the bottleneck in our overall system throughput, and we aren't operating at a scale where even small optimisations to our software can have a significant impact on our overall CPU time and power consumption costs, then investing effort in speeding up our Python code doesn't offer a good return on our time.

This is certainly true of the scientific community, where the heavy numeric lifting is often done in C or FORTRAN, and the Python components are there to make everything hang together in a way that humans can read relatively easily.

In the case of web development, while the speed of the application server may become a determining factor at truly massive scale, smaller applications are likely to gain more through language independent techniques like adding a Varnish caching server in front of the overall application, and a memory cache to avoid repeating calcuations for common inputs before the application code itself is likely to become the bottleneck.

This means for the kind of use case where Python is primarily playing an orchestration role, as well as those where the application is IO bound rather than CPU bound, being able to run across multiple cores doesn't really provide a lot of benefit - the Python code was never the bottleneck in the first place, so focusing optimisation efforts on the Python components doesn't make sense.

Instead, people drop out of pure Python code into an environment that is vastly easier to optimise and already supports running across multiple cores within a single process. This may be hand written C or C++ code, it may be something with Pythonic syntax but reduced dynamism like Cython or Numba, or it may be another more static language on a preexisting runtime like the JVM or the CLR, but however it is achieved, the level shift allows optimisations and parallelism to be applied at the places where they will do the most good for the overall speed of the application.

Why isn't "just remove the GIL" the obvious answer?

Removing the GIL *is* the obvious answer. The problem with this phrase is the "just" part, not the "remove the GIL" part.

One of the key issues with threading models built on shared non-transactional memory is that they are a broken approach to general purpose concurrency. Armin Rigo has explained that far more eloquently than I can in the introduction to his Software Transactional Memory work for PyPy, but the general idea is that threading is to concurrency as the Python 2 Unicode model is to text handling - it works great a lot of the time, but if you make a mistake (which is inevitable in any non-trivial program) the consequences are unpredictable (and often catastrophic from an application stability point of view), and the resulting situations are frequently a nightmare to debug.

The advantages of GIL-style coarse grained locking for the CPython interpreter implementation are that it makes naively threaded code more likely to run correctly, greatly simplifies the interpreter implementation (thus increasing general reliability and ease of porting to other platforms) and has almost zero overhead when running in single-threaded mode for simple scripts or event driven applications which don't need to interact with any synchronous APIs (as the GIL is not initialised until the threading support is imported, or initialised via the C API, the only overhead is a boolean check to see if the GIL has been created).

The CPython development team have long had an essential list of requirements that any major improvement to CPython's parallel execution support would be expected to meet before it could be considered for incorporation into the reference interpreter:

- must not substantially slow down single-threaded applications
- must not substantially increase latency times in IO bound applications
- threading support must remain optional to ease porting to platforms with no (or broken) threading primitives
- must minimise breakage of current end user Python code that implicitly relies on the coarse-grained locking provided by the GIL (I recommend consulting Armin's STM introduction on the challenges posed by this)
- must remain compatible with existing third party C extensions that rely on refcounting and the GIL (I recommend consulting with the cpyext and IronClad developers both on the difficulty of meeting this requirement, and the lack of interest many parts of the community have in any Python implementation that doesn't abide by it)
- must achieve all of these without reducing the number of supported platforms for CPython, or substantially increasing the difficulty of porting the CPython interpreter to a new platform (I recommend consulting with the JVM and CLR developers on the difficulty of producing and maintaining high performance cross platform threading primitives).

It is important to keep in mind that CPython already has a significant user base (sufficient to see Python ranked by IEEE Spectrum in 2014 as one of the top 5 programming languages in the world), and it's necessarily the case that these users either don't find the GIL to be an intolerable burden for their use cases, or else find it to be a problem that is tolerably easy to work around.

Core development efforts in the concurrency and parallelism arena have thus historically focused on better serving the needs of those users by providing better primitives for easily distributing work across multiple processes, and to perform multiple IO operations in parallel. Examples of this approach include the initial incorporation of the `multiprocessing` module, which aims to make it easy to migrate from threaded code to multiprocess code, along with the addition of the `concurrent.futures` module in Python 3.2, which aims to make it easy to take serial code and dispatch it to multiple threads (for IO bound operations) or multiple processes (for CPU bound operations), the `asyncio` module in Python 3.4 (which provides full support for explicit asynchronous programming in the standard library) and the introduction of the dedicated async/await syntax for native coroutines in Python 3.5.

For IO bound code (with no CPU bound threads present), or, equivalently, code that invokes external libraries to perform calculations (as is the case for most serious number crunching code, such as that using NumPy and/or Cython), the GIL does place an additional constraint on the application, but one that is acceptable in many cases: a single core must be able to handle all Python execution on the machine, with other cores either left idle (IO bound systems) or busy handling calculations (external library invocations). If that is not the case, then multiple interpreter processes will be needed, just as they are in the case of any CPU bound Python threads.

What are the key problems with fine-grained locking as an answer?

For seriously parallel problems, a free threaded interpreter that uses fine-grained locking to scale across multiple cores doesn't help all that much, as it is desired to scale not only to multiple cores on a single machine, but to multiple *machines*. As soon as a second machine enters the picture, shared memory based concurrency can't help you: you need to use a parallel execution model (such as message passing or a shared datastore) that allows information to be passed between processes, either on a single machine or on multiple machines. (Folks that have this kind of problem to solve would be well advised to investigate the viability of adopting Apache Spark as their computational platform, either directly or through the Blaze abstraction layer)

CPython also has another problem that limits the effectiveness of removing the GIL by switching to fine-grained locking: we use a reference counting garbage collector with cycle detection. This hurts free threading in two major ways: firstly, any free threaded solution that retains the reference counting GC will still need a global lock that protects the integrity of the reference counts; secondly, switching threads in the CPython runtime will mean updating

the reference counts on a whole new working set of objects, almost certainly blowing the CPU cache and losing some of the speed benefits gained from making more effective use of multiple cores.

So for a truly free-threaded interpreter, the reference counting GC would likely have to go as well, or be replaced with an allocation model that uses a separate heap per thread by default, creating yet *another* compatibility problem for C extensions (and one that we already know from experience with PyPy, Jython and IronPython poses significant barriers to runtime adoption).

These various factors all combine to explain why it's unlikely we'll ever see CPython's coarse-graining locking model replaced by a fine-grained locking model within the scope of the CPython project itself:

- a coarse-grained lock makes threaded code behave in a less surprising fashion
- a coarse-grained lock makes the implementation substantially simpler
- a coarse-grained lock imposes negligible overhead on the scripting use case
- fine-grained locking provides no benefits to single-threaded code (such as end user scripts)
- fine-grained locking may break end user code that implicitly relies on CPython's use of coarse grained locking
- fine-grained locking provides minimal benefits to event-based code that uses threads solely to provide asynchronous access to external synchronous interfaces (such as web applications using an event based framework like Twisted or gevent, or GUI applications using the GUI event loop)
- fine-grained locking provides minimal benefits to code that uses other languages like Cython, C or Fortran for the serious number crunching (as is common in the NumPy/SciPy community)
- fine-grained locking provides no substantial benefits to code that needs to scale to multiple machines, and thus cannot rely on shared memory for data exchange
- a refcounting GC doesn't really play well with fine-grained locking (primarily from the point of view of high contention on the lock that protects the integrity of the refcounts, but also the bad effects on caching when switching to different threads and writing to the refcount fields of a new working set of objects)
- increasing the complexity of the core interpreter implementation for any reason always poses risks to maintainability, reliability and portability

It isn't that a free threaded Python implementation that complies with the Python Language and Library References isn't possible (Jython and IronPython prove that's not the case), it's that free threaded virtual machines are hard to write correctly in the first place and are harder to maintain once implemented. For CPython specifically, any engineering effort directed towards free threading support is engineering effort that isn't being directed somewhere else. The current core development team don't consider that to be a good trade-off when there are other far more interesting options still to be explored.

What does the future look like for exploitation of multiple cores in Python?

For CPython, Eric Snow has started working with Dr Sarah Mount (at the University of Wolverhamption) to investigate some speculative ideas I published a few years back regarding the possibility of refining CPython's subinterpreter support to make it a first class language feature that offered true in-process support for parallel exploitation of multiple cores in a way that didn't break compatibility with C extension modules (at least, not any more than using subinterpreters in combination with extensions that call back into Python from C created threads already breaks it).

For PyPy, Armin Rigo and others are actively pursuing research into the use of Software Transactional Memory to allow event driven programs to be scaled transparently across multiple CPU cores. I know he has some thoughts on how the concepts he is exploring in PyPy could be translated back to CPython, but even if that doesn't pan out, it's very easy to envision a future where CPython is used for command line utilities (which are generally single threaded and often so short running that the PyPy JIT never gets a chance to warm up) and embedded systems, while PyPy takes over the execution of long running scripts and applications, letting them run substantially faster and span multiple cores without requiring any modifications to the Python code. Splitting the role of the two VMs in that fashion would

allow each to be optimised appropriately rather than having to make trade-offs that attempt to balance the starkly different needs of the various use cases.

I also expect we'll continue to add APIs and features designed to make it easier to farm work out to other processes (for example, the new iteration of the pickle protocol in Python 3.4 included the ability to unpickle unbound methods by name, which allow them to be used with the multiprocessing APIs).

For data processing workloads, Python users that would prefer something simpler to deploy than Apache Spark, don't want to compile their own C extensions with Cython, and have data which exceeds the capacity of NumPy's in-memory calculation model on the systems they have access to, may wish to investigate the Dask project, which aims to offer the features of core components of the Scientific Python ecosystem (notably, NumPy and Pandas) in a form which is limited by the capacity of local disk storage, rather than the capacity of local memory.

Another potentially interesting project is Trent Nelson's PyParallel work on using memory page locking to permit the creation of "shared nothing" worker threads, that would permit the use of a more Rust-style memory model within CPython without introducing a distinct subinterpreter based parallel execution model.

Alex Gaynor also pointed out some interesting research (PDF) into replacing Ruby's Giant VM Lock (the equivalent to CPython's GIL in CRuby, aka the Matz Ruby Interpreter) with appropriate use of Hardware Transactional Memory, which may also prove relevant to CPython as HTM capable hardware becomes more common. (However, note the difficulties that the refcounting in MRI caused the researchers - CPython is likely to have exactly the same problem, with a well established history of attempting to eliminate and then emulate the refcounting causing major compatibility problems with extension modules).

Python 3 and ASCII Compatible Binary Protocols

Last Updated: 6th January, 2014

If you pay any attention to the Twittersphere (and likely several other environments), you may have noticed various web framework developers having a few choice words regarding the Unicode handling design in Python 3.

They actually have good reason to be upset with python-dev: we broke their world. Not only did we break it, but we did it on purpose.

What did we break?

What we broke is a very specific thing: many of the previously idiomatic techniques for transparently accepting both Unicode text and text in an ASCII compatible binary encoding no longer work in Python 3. Given that the web (along with network protocols in general) is *built* on the concept of ASCII compatible binary encodings, this is causing web framework developers an understandable amount of grief as they start making their first serious efforts at supporting Python 3.

The key thing that changed is that it is no longer easy to write text manipulation algorithms that can work transparently on either actual text (i.e. 2.x `unicode` objects) and on text encoded to binary using an ASCII compatible encoding (i.e. some instances of 2.x `str` objects).

There are a few essential changes in Python 3 which make this no longer practical:

- In 2.x, when `unicode` and `str` meet, the latter is automatically promoted to `unicode` (usually assuming a default `ascii` encoding). In 3.x, this changes such that when `str` (now always Unicode text) meets `bytes` (the new binary data type) you get an exception. Significantly, this means you can no longer share literal values between the two algorithm variants (in 2.x, you could just use `str` literals and rely on the automatic promotion to cover the `unicode` case).

- Iterating over a string produces a series of length 1 strings. Iterating over a 3.x bytes object, on the other hand, produces a series of integers. Similarly, indexing a bytes object produces an integer - you need to use slicing syntax if you want a length 1 bytes object.
- The `encode()` and `decode()` convenience methods no longer support the text->text and binary->binary transforms, instead being limited to the actual text->binary and binary->text encodings. The `codecs.encode` and `codecs.decode` functions need to be used instead in order to handle these transforms in addition to the regular text encodings (these functions are available as far back as Python 2.4, so they're usable in the common subset of Python 2 and Python 3).
- In 2.x, the `unicode` type supported the buffer API, allowing direct access to the raw multi-byte characters (stored as UCS4 in wide builds, and a UCS2/UTF-16 hybrid in narrow builds). In 3.x, the only way to access this data directly is via the Python C API. At the Python level, you only have access to the code point data, not the individual bytes.

The recommended approach to handling both binary and text inputs to an API without duplicating code is to explicitly decode any binary data on input and encode it again on output, using one of two options:

1. `ascii` in `strict` mode (for true 7-bit ASCII data)
2. `ascii` in `surrogateescape` mode (to allow any ASCII compatible encoding)

However, it's important to be very careful with the latter approach - when applied to an ASCII incompatible encoding, manipulations that assume ASCII compatibility may still cause data corruption, even with explicit decoding and encoding steps. It can be better to assume strict ASCII-only data for implicit conversions, and require external conversion to Unicode for other ASCII compatible encodings (e.g. this is the approach now taken by the `urllib.urlparse` module).

Why did we break it?

That last paragraph in the previous section hints at the answer: *assuming* that binary data uses an ASCII compatible encoding and manipulating it accordingly can lead to silent data corruption if the assumption is incorrect.

In a world where there are multiple ASCII *incompatible* text encodings in regular use (e.g. UTF-16, UTF-32, ShiftJIS, many of the CJK codecs), that's a problem.

Another regular problem with code that supposedly supports both Unicode and encoded text is that it may not correctly handle multi-byte, variable width and other stateful encodings where the meaning of the current byte may depend on the values of one or more previous bytes, even if the code does happen to correctly handle ASCII-incompatible stateless single-byte encodings.

All of these problem can be dealt with if you appropriately vet the encoding of any binary data that is passed in. However, this is not only often easier said than done, but Python 2 doesn't really offer you any good tools for finding out when you've stuffed it up. They're data driven bugs, but the errors may never turn into exceptions, instead just causing flaws in the resulting text output.

This was a gross violation of "The Zen of Python", specifically the part about "Errors should never pass silently. Unless explicitly silenced".

As a concrete example of the kind of obscure errors this can cause, I recently tracked down an obscure problem that was leading to my web server receiving a request that consisted solely of the letter "G". From what I have been able to determine, that error was the result of:

1. M2Crypto emitting a Unicode value for a HTTP header value
2. The SSL connection combining this with other values, creating an entire Unicode string instead of the expected byte sequence
3. The SSL connection interpreting that string via the buffer API

4. The SSL connection seeing the additional NULs due to the UCS4 internal encoding and truncating the string accordingly

This has now been worked around by explicitly encoding the Unicode value erroneously emitted, but it was a long hunt to find the problem when the initial symptom was just a 404 error from the web server.

Since Python 3 is a lot fussier when it comes to the ways it will allow binary and text data to implicitly interact, this would have been picked up client side as soon as any attempt was made to combine the Unicode text value with the already encoded binary data.

The other key reason for changing the text model of the language is that the Python 2 model only works properly on POSIX systems. Unlike POSIX, Unicode capable interfaces on Windows, the JVM and the CLR (whether .NET or mono), use Unicode natively rather than using encoded bytestrings.

The Python 3 model, by contrast, aims to handle Unicode correctly on all platforms, with the surrogateescape error handler introduced to handle the case of data in operating system interfaces that doesn't match the declared encoding on POSIX systems.

Why are the web framework developers irritated?

We knew when we released Python 3 that it was going to take quite a while for the binary/text split to be fully resolved. Most of the burden of that resolution falls on the shoulders of those dealing with the boundaries between text data and binary protocols. Web frameworks have to deal with these issues both on the network side *and* on the data storage side.

Those developers also have good reason to want to avoid decoding to Unicode - until Python 3.3 was released, Unicode strings consumed up to four times the memory consumed by 8 bit strings (depending on build options).

That means framework developers face an awkward choice in their near term Python 3 porting efforts:

- do it "right" (i.e. converting to the text format for text manipulations), and keep track of the need to convert the result back to bytes
- split their code into parallel binary and text APIs (potentially duplicating a lot of code and making it much harder to maintain)
- including multiple "binary or text" checks within the algorithm implementation (this can get very untidy very quickly)
- develop a custom extension type for implementing a str-style API on top of encoded binary data (this is hard to do without reintroducing all the problems with ASCII incompatible encodings noted above, but a custom type provides more scope to make it clear it is only appropriate in contexts where ASCII compatible encodings can be safely assummed, such as many web protocols)

I have a personal preference for the first choice as the current path of least resistance, as reflected in the way I implemented the binary input support for the `urllib.parse` APIs in Python 3.2. However, the last option (or something along those lines) will likely be needed in order to make ASCII compatible binary protocol handling as convenient in Python 3 as it is in Python 2.

The last option is still one of the options for possible future Python 3 improvements listed under *Is Python 3 more convenient than Python 2 in every respect?*.

Couldn't the implicit decoding just be disabled in Python 2?

While Python 2 *does* provide a mechanism that allows the implicit decoding mechanism to be disabled, actually trying to use it breaks the world:

```
>>> import urlparse
>>> urlparse.urlsplit("http://example.com")
SplitResult(scheme='http', netloc='example.com', path='', query='', fragment='')
>>> urlparse.urlsplit(u"http://example.com")
SplitResult(scheme=u'http', netloc=u'example.com', path=u'', query='', fragment='')
>>> import sys
>>> reload(sys).setdefaultencoding("undefined")
>>> urlparse.clear_cache()
>>> urlparse.urlsplit("http://example.com")
SplitResult(scheme='http', netloc='example.com', path='', query='', fragment='')
>>> urlparse.urlsplit(u"http://example.com")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib64/python2.7/urlparse.py", line 181, in urlsplit
    i = url.find(':')
  File "/usr/lib64/python2.7/encodings/undefined.py", line 22, in decode
    raise UnicodeError("undefined encoding")
UnicodeError: undefined encoding
```

(If you don't clear the parsing cache after disabling the default encoding and retest with the same URLs, that second call may appear to be work, but that's only because it gets a hit in the cache from the earlier successful call. Using a different URL or clearing the caches as shown will reveal the error).

This is why turning off the implicit decoding is such a big deal that it required a major version bump for the language definition: there is a *lot* of Python 2 code that only handles Unicode because 8-bit strings (including literals) are implicitly promoted to Unicode as needed. Since Python 3 removes all the implicit conversions, code that previously relied on it in order to accept both binary and text inputs (like the Python 2 URL parsing code shown above) instead needs to be updated to explicitly handle both binary and text inputs.

So, in contrast to Python 2 code above, the Python 3 version not only changes the types of the components in the result, but also changes the type of the result itself:

```
>>> from urllib import parse
>>> parse.urlsplit("http://example.com")
SplitResult(scheme='http', netloc='example.com', path='', query='', fragment='')
>>> parse.urlsplit(b"http://example.com")
SplitResultBytes(scheme=b'http', netloc=b'example.com', path=b'', query=b'',␣
↪fragment=b'')
```

However, it's also no longer dependent on a global configuration setting that controls how 8-bit string literals are converted to Unicode text - instead, the decision on how to convert from bytes to text is handled entirely within the function call.

Where to from here?

The revised text handling design in Python 3 is definitely a case of the pursuit of correctness triumphing over convenience. "Usually handy, but occasionally completely and totally wrong" is not a good way to design a language (If you question this, compare and contrast the experience of programming in C++ and Python. Both are languages with a strong C influence, but the former makes a habit of indulging in premature optimisations that can go seriously wrong if their assumptions are violated. Guess which of the two is almost universally seen as being more developer hostile?).

The challenge for Python 3.3 and beyond is to start bringing back some of the past convenience that resulted from being able to blur the lines between binary and text data without unduly compromising on the gains in correctness.

The efficient Unicode representation in Python 3.3 (which uses the smallest per-character size out of 1, 2 and 4 that can handle all characters in the string) was a solid start down that road, as was the restoration of Unicode string literal support in **PEP 414** (as that was a change library and framework developers couldn't address on behalf of their users).

Python 3.4 restored full support for the binary transform codecs through the existing type neutral codecs module API (along with improved handling of codec errors in general).

Some other possible steps towards making Python 3 as convenient a langauge as Python 2 for wire protocol handling are discussed in *Is Python 3 more convenient than Python 2 in every respect?*

But for most Python programmers, this issue simply doesn't arise. Binary data is binary data, text characters are text characters, and the two only meet at well-defined boundaries. It's only people that are writing the libraries and frameworks that *implement* those boundaries that really need to grapple with the details of these concepts.

Processing Text Files in Python 3

A recent discussion on the python-ideas mailing list made it clear that we (i.e. the core Python developers) need to provide some clearer guidance on how to handle text processing tasks that trigger exceptions by default in Python 3, but were previously swept under the rug by Python 2's blithe assumption that all files are encoded in "latin-1".

While we'll have something in the official docs before too long, this is my own preliminary attempt at summarising the options for processing text files, and the various trade-offs between them.

What changed in Python 3?

The obvious question to ask is what changed in Python 3 so that the common approaches that developers used to use for text processing in Python 2 have now started to throw `UnicodeDecodeError` and `UnicodeEncodeError` in Python 3.

The key difference is that the default text processing behaviour in Python 3 aims to detect text encoding problems as early as possible - either when reading improperly encoded text (indicated by `UnicodeDecodeError`) or when being asked to write out a text sequence that cannot be correctly represented in the target encoding (indicated by `UnicodeEncodeError`).

This contrasts with the Python 2 approach which allowed data corruption by default and strict correctness checks had to be requested explicitly. That could certainly be *convenient* when the data being processed was predominantly ASCII text, and the occasional bit of data corruption was unlikely to be even detected, let alone cause problems, but it's hardly a solid foundation for building robust multilingual applications (as anyone that has ever had to track down an errant `UnicodeError` in Python 2 will know).

However, Python 3 does provide a number of mechanisms for relaxing the default strict checks in order to handle various text processing use cases (in particular, use cases where "best effort" processing is acceptable, and strict correctness is not required). This article aims to explain some of them by looking at cases where it would be appropriate to use them.

Note that many of the features I discuss below are available in Python 2 as well, but you have to explicitly access them via the `unicode` type and the `codecs` module. In Python 3, they're part of the behaviour of the `str` type and the `open` builtin.

Unicode Basics

To process text effectively in Python 3, it's necessary to learn at least a tiny amount about Unicode and text encodings:

1. Python 3 always stores text strings as sequences of Unicode *code points*. These are values in the range 0-0x10FFFF. They *don't* always correspond directly to the characters you read on your screen, but that distinction doesn't matter for most text manipulation tasks.
2. To store text as binary data, you must specify an *encoding* for that text.
3. The process of converting from a sequence of bytes (i.e. binary data) to a sequence of code points (i.e. text data) is *decoding*, while the reverse process is *encoding*.
4. For historical reasons, the most widely used encoding is `ascii`, which can only handle Unicode code points in the range 0-0x7F (i.e. ASCII is a 7-bit encoding).
5. There are a wide variety of ASCII *compatible* encodings, which ensure that any appearance of a valid ASCII value in the binary data refers to the corresponding ASCII character.
6. "utf-8" is becoming the preferred encoding for many applications, as it is an ASCII-compatible encoding that can encode any valid Unicode code point.
7. "latin-1" is another significant ASCII-compatible encoding, as it maps byte values directly to the first 256 Unicode code points. (Note that Windows has it's own "latin-1" variant called cp1252, but, unlike the ISO "latin-1" implemented by the Python codec with that name, the Windows specific variant doesn't map all 256 possible byte values)
8. There are also many ASCII *incompatible* encodings in widespread use, particularly in Asian countries (which had to devise their own solutions before the rise of Unicode) and on platforms such as Windows, Java and the .NET CLR, where many APIs accept text as UTF-16 encoded data.
9. The `locale.getpreferredencoding()` call reports the encoding that Python will use by default for most operations that require an encoding (e.g. reading in a text file without a specified encoding). This is designed to aid interoperability between Python and the host operating system, but can cause problems with interoperability between systems (if encoding issues are not managed consistently).
10. The `sys.getfilesystemencoding()` call reports the encoding that Python will use by default for most operations that both require an encoding and involve textual metadata in the filesystem (e.g. determining the results of `os.listdir()`)
11. If you're a native English speaker residing in an English speaking country (like me!) it's tempting to think "but Python 2 works fine, why are you bothering me with all this Unicode malarkey?". It's worth trying to remember that we're actually a minority on this planet and, for most people on Earth, ASCII and `latin-1` can't even handle their *name*, let alone any other text they might want to write or process in their native language.

Unicode Error Handlers

To help standardise various techniques for dealing with Unicode encoding and decoding errors, Python includes a concept of Unicode error handlers that are automatically invoked whenever a problem is encountered in the process of

encoding or decoding text.

I'm not going to cover all of them in this article, but three are of particular significance:

- `strict`: this is the default error handler that just raises `UnicodeDecodeError` for decoding problems and `UnicodeEncodeError` for encoding problems.
- `surrogateescape`: this is the error handler that Python uses for most OS facing APIs to gracefully cope with encoding problems in the data supplied by the OS. It handles decoding errors by squirreling the data away in a little used part of the Unicode code point space (For those interested in more detail, see PEP 383). When encoding, it translates those hidden away values back into the exact original byte sequence that failed to decode correctly. Just as this is useful for OS APIs, it can make it easier to gracefully handle encoding problems in other contexts.
- `backslashreplace`: this is an encoding error handler that converts code points that can't be represented in the target encoding to the equivalent Python string numeric escape sequence. It makes it easy to ensure that `UnicodeEncodeError` will never be thrown, but doesn't lose much information while doing so losing (since we don't want encoding problems hiding error output, this error handler is enabled on `sys.stderr` by default).

The Binary Option

One alternative that is always available is to open files in binary mode and process them as bytes rather than as text. This can work in many cases, especially those where the ASCII markers are embedded in genuinely arbitrary binary data.

However, for both "text data with unknown encoding" and "text data with known encoding, but potentially containing encoding errors", it is often preferable to get them into a form that can be handled as text strings. In particular, some APIs that accept both bytes and text may be very strict about the encoding of the bytes they accept (for example, the `urllib.urlparse` module accepts only pure ASCII data for processing as bytes, but will happily process text strings containing non-ASCII code points).

Text File Processing

This section explores a number of use cases that can arise when processing text. Text encoding is a sufficiently complex topic that there's no one size fits all answer - the right answer for a given application will depend on factors like:

- how much control you have over the text encodings used
- whether avoiding program failure is more important than avoiding data corruption or vice-versa
- how common encoding errors are expected to be, and whether they need to be handled gracefully or can simply be rejected as invalid input

Files in an ASCII compatible encoding, best effort is acceptable

Use case: the files to be processed are in an ASCII compatible encoding, but you don't know exactly which one. *All* files must be processed without triggering any exceptions, but some risk of data corruption is deemed acceptable (e.g. collating log files from multiple sources where some data errors are acceptable, so long as the logs remain largely intact).

Approach: use the "latin-1" encoding to map byte values directly to the first 256 Unicode code points. This is the closest equivalent Python 3 offers to the permissive Python 2 text handling model.

Example: `f = open(fname, encoding="latin-1")`

Note: While the Windows `cp1252` encoding is also sometimes referred to as "latin-1", it doesn't map all possible byte values, and thus needs to be used in combination with the `surrogateescape` error handler to ensure it never throws `UnicodeDecodeError`. The `latin-1` encoding in Python implements ISO_8859-1:1987 which maps all possible byte values to the first 256 Unicode code points, and thus ensures decoding errors will never occur regardless of the configured error handler.

Consequences:

- data will *not* be corrupted if it is simply read in, processed as ASCII text, and written back out again.
- will never raise UnicodeDecodeError when reading data
- will still raise UnicodeEncodeError if codepoints above 0xFF (e.g. smart quotes copied from a word processing program) are added to the text string before it is encoded back to bytes. To prevent such errors, use the `backslashreplace` error handler (or one of the other error handlers that replaces Unicode code points without a representation in the target encoding with sequences of ASCII code points).
- data corruption may occur if the source data is in an ASCII incompatible encoding (e.g. UTF-16)
- corruption may occur if data is written back out using an encoding other than `latin-1`
- corruption may occur if the non-ASCII elements of the string are modified directly (e.g. for a variable width encoding like UTF-8 that has been decoded as `latin-1` instead, slicing the string at an arbitrary point may split a multi-byte character into two pieces)

Files in an ASCII compatible encoding, minimise risk of data corruption

Use case: the files to be processed are in an ASCII compatible encoding, but you don't know exactly which one. *All* files must be processed without triggering any exceptions, but some Unicode related errors are acceptable in order to reduce the risk of data corruption (e.g. collating log files from multiple sources, but wanting more explicit notification when the collated data is at risk of corruption due to programming errors that violate the assumption of writing the data back out only in its original encoding)

Approach: use the `ascii` encoding with the `surrogateescape` error handler.

Example: `f = open(fname, encoding="ascii", errors="surrogateescape")`

Consequences:

- data will *not* be corrupted if it is simply read in, processed as ASCII text, and written back out again.
- will never raise UnicodeDecodeError when reading data
- will still raise UnicodeEncodeError if codepoints above 0xFF (e.g. smart quotes copied from a word processing program) are added to the text string before it is encoded back to bytes. To prevent such errors, use the `backslashreplace` error handler (or one of the other error handlers that replaces Unicode code points without a representation in the target encoding with sequences of ASCII code points).
- will also raise UnicodeEncodeError if an attempt is made to encode a text string containing escaped bytes values without enabling the `surrogateescape` error handler (or an even more tolerant handler like `backslashreplace`).
- some Unicode processing libraries that ensure a code point sequence is valid text may complain about the escaping mechanism used (I'm not going to explain what it means here, but the phrase "lone surrogate" is a hint that something along those lines may be happening - the fact that "surrogate" also appears in the name of the error handler is not a coincidence).
- data corruption may still occur if the source data is in an ASCII incompatible encoding (e.g. UTF-16)
- data corruption is also still possible if the escaped portions of the string are modified directly

Files in a typical platform specific encoding

Use case: the files to be processed are in a consistent encoding, the encoding can be determined from the OS details and locale settings and it is acceptable to refuse to process files that are not properly encoded.

Approach: simply open the file in text mode. This use case describes the default behaviour in Python 3.

Example: `f = open(fname)`

Consequences:

- `UnicodeDecodeError` may be thrown when reading such files (if the data is not actually in the encoding returned by `locale.getpreferredencoding()`)
- `UnicodeEncodeError` may be thrown when writing such files (if attempting to write out code points which have no representation in the target encoding).
- the `surrogateescape` error handler can be used to be more tolerant of encoding errors if it is necessary to make a best effort attempt to process files that contain such errors instead of rejecting them outright as invalid input.

Files in a consistent, known encoding

Use case: the files to be processed are nominally in a consistent encoding, you know the exact encoding in advance and it is acceptable to refuse to process files that are not properly encoded. This is becoming more and more common, especially with many text file formats beginning to standardise on UTF-8 as the preferred text encoding.

Approach: open the file in text mode with the appropriate encoding

Example: `f = open(fname, encoding="utf-8")`

Consequences:

- `UnicodeDecodeError` may be thrown when reading such files (if the data is not actually in the specified encoding)
- `UnicodeEncodeError` may be thrown when writing such files (if attempting to write out code points which have no representation in the target encoding).
- the `surrogateescape` error handler can be used to be more tolerant of encoding errors if it is necessary to make a best effort attempt to process files that contain such errors instead of rejecting them outright as invalid input.

Files with a reliable encoding marker

Use case: the files to be processed include markers that specify the nominal encoding (with a default encoding assumed if no marker is present) and it is acceptable to refuse to process files that are not properly encoded.

Approach: first open the file in binary mode to look for the encoding marker, then reopen in text mode with the identified encoding.

Example: `f = tokenize.open(fname)` uses PEP 263 encoding markers to detect the encoding of Python source files (defaulting to UTF-8 if no encoding marker is detected)

Consequences:

- can handle files in different encodings
- may still raise UnicodeDecodeError if the encoding marker is incorrect
- must ensure marker is set correctly when writing such files

- even if it is not the default encoding, individual files can still be set to use UTF-8 as the encoding in order to support encoding almost all Unicode code points
- the `surrogateescape` error handler can be used to be more tolerant of encoding errors if it is necessary to make a best effort attempt to process files that contain such errors instead of rejecting them outright as invalid input.

Python, Enumerations and "Good Enough"

Note: To provide feedback on this essay, use the issue tracker or the DISQUS comments below.

Site last built: Jul 18, 2017 (Change history)

PEP 435 adds support for explicit enumerations in the Python standard library. One valid criticism of the accepted design is that there are some awkward compromises in the definition syntax, most notably the fact that you have to choose between explicit identification of the enum values in the class based syntax and the repetition of the class name and the use of strings in the functional declaration API. Andrew Cooke's review of the accepted PEP provides a good overview of that criticism.

Why standardise enums at all?

The idea of standardising enumerations has been kicking around for years. What finally tipped the balance this time? The major factor was a combination of Guido being sufficiently interested to rule on the many and varied debatable aspects of `Enum` runtime behaviour, and his declaration at the PyCon US 2013 language summit that we should just adopt Barry Warsaw's `flufl.enum` package wholesale as a "good enough" solution and be done with it (as **PEP 435** describes, that didn't end up happening, but having a concrete starting point like that helped focus the extensive subsequent design discussions).

The other motivating factor, however, was the ability to use enumerations to improve various error messages emitted by the standard library. One of the significant downsides of magic integers is the fact that they often result in cryptic error messages, unless the library authors take special care to translate numeric values back to their string equivalents when creating the error message. This limitation also applies to logging messages and value introspection when debugging. The additional name information in the representation of enumeration values provides that easier interpretation of otherwise generic values for free.

In almost all of the cases where we're interested in this, however, the enumeration values are NOT arbitrary: instead, we're exposing values defined in POSIX or IETF standards or by the underlying operating system, or ones where we're exposing some internal implementation detail of the interpreter in a more robust and implementation independent way.

The other cases that we may update are ones that currently use `range` to generate the values, or else map variables to their own name as strings. For these cases, the **PEP 435** functional API is seen as being at least an improvement over the status quo, even though it remains somewhat inelegant.

Requirements for the standard enum design

The core requirements for the standard enum design were based on a few standard library use cases:

- exposing externally defined constants like those in `errno` and `socket`
- exposing non-enum constants like those in `opcode` in a more user friendly way
- exposing arbitrary constants like some of those in `inspect`

The goal on the definition side was definitely "usable" rather than "beautiful" or "elegant". Since a genuinely elegant declaration syntax was considered a "nice to have" rather than a "must have" for the design, it isn't especially surprising that we failed to achieve it.

The declaration syntaxes in **PEP 435** are designed to support the first two use cases by letting us write ordinary Python code in the body of a class based enum declaration (including easy definition of aliases, since the externally defined constants we want to support often include aliases) and the shorthand functional API handles the "we don't care about the values" use case well enough for us to consider it acceptable as the initial implementation.

But *why* isn't elegance of declarations important?

Elegance of declarations *is* important, it just isn't the only consideration. When designing additions to the Python standard library, we don't consider "don't worry, it's magic" to be an acceptable explanation for how the new feature works (the zero-argument form of super() in Python 3 is a notable exception, and that violation of our usual principles has been the direct cause of a number of annoying issues due directly to its current conceptual incoherence relative to the rest of the language semantics).

The addition of enumerations represents only the second use of a custom metaclass in the standard library (the first was Abstract Base Classes), and we think it's important that any deviations made from standard class behaviour are absolutely essential to the consistency of the runtime behaviour of enumerations (this includes items like having member definition order match iteration order, and being able to define integer-based enumerations that are transparently interoperable with the integers they replace).

In the case of the class-based declaration syntax, the long standing descriptor protocol means having class attributes behave differently when accessed through the class than they do when accessed in the class body is perfectly normal class behaviour. Thus, we're comfortable with implicitly wrapping explicitly assigned names to turn them into enumeration members, and handling descriptors differently from other values.

However, having references to missing names implicitly create new enumeration members is *not* something we're comfortable with as part of a standard language feature. We did consider the idea (and, as noted in the PEP, it's certainly feasible to implement enums that way), but it makes it impossible to write normal code in the class body (any typo would implicitly create a new enum member instead of reporting a `NameError` as expected).

Why would anyone want to write normal code in the body of an enumeration? One reason is because that's the way you create enumerations with custom behaviour: by defining methods in their class definition, just as you do for other classes. If references to missing names implicitly created a new enumeration member, then making a typo in a default value in a method definition would behave very strangely.

More importantly, we plan to tighten up the formal specification for `locals()` so that it can be manipulated in the class body to define enumerations programmatically. For several of the standard library use cases (where the enum represents an externally defined mapping of names to values) this is far more important than the ability to concisely define enumerations where we don't care about the values (which is largely covered by the functional API anyway).

The pedagogical aspect of requiring explicit assignments is that allowing implicit creation of enumeration values elevates the "don't worry it's magic" factor well beyond what we consider necessary. With the current design, the code in the body reads like normal Python code, the same as any other class. With implicit creation, enumerations behave wildly differently from anything else in Python. Yes, it *can* be done, but that doesn't mean it *should* (at least, not as the standard incarnation of the syntax).

The question of whether or not to allow aliasing by default was a close-run thing, eventually decided by Guido opting for easier support for POSIX and IETF standards (which often include aliases) over easier detection of typos when entering values directly. While I briefly thought we could use a little more magic to support aliasing without supporting independently binding two different names to the same value, that turned out to be more problematic than I expected so we're sticking with Guido's original decision.

However, it may still be possible add a simple class decorator that makes it easy to ensure there a no accidental aliases when they are not desired.

Support for alternate declaration syntaxes

That said, something we're deliberately aiming to do with the **PEP 435** enum implementation is to make the `enum.EnumMeta` *metaclass* amenable to customisation. Metaclasses are ultimately just classes (albeit ones with a specific use case in Python's data model), so you can subclass and tweak them in order to change their behaviour, as long as they were designed with that kind of tweaking in mind. In the case of enums, we plan to rely on that to let people create their own variations on enum *declaration* syntax, while largely retaining the runtime semantics of the standard enumerations.

Personally, I expect to see variants that enable the following behaviours:

- Autonumbered enums with a sentinel value (such as Ellipsis). This is used as an `enum.EnumMeta` subclassing test case in the test suite for the reference implementation and allows code like:

```
class Color(AutoNumberedEnum):
    red = ...
    green = ...
    blue = ...
```

  A relatively straightforward variant of this would use the "= ..." notation to mean "use the enum member's qualified name as its value".

- Implicit enums that *don't* really support normal code execution in the class body, and allow the above to be simplified further. It's another variant of the autonumbered example in the test suite, but one that diverges substantially from normal Python semantics: merely *mentioning* a name will create a new reference to that name. While there are a number of ways to get into trouble when doing this, the basic concept would be to modify `__prepare__` on the metaclass to return a namespace that implements `__missing__` as returning a custom sentinel value and overrides __getitem__ to treat repeating a name as an error:

```
class Color(ImplicitEnum):
    red
    green
    blue
    green # This should trigger an exception
```

  When the metaclass is putting the class together, it then looks at all the entries set to the sentinel value and then either numbers them in order (if using integers as values) or else sets each of them to the qualified name of the member (if using strings as values)

- Extensible enums, that make it easier to include elements of another enum inside a larger one. One feature of `flufl.enum` that was lost in the journey to the standard library is the ability to inherit enum members from a parent enum, as a consequence of making it so that standard enum members are actually instances of the corresponding enum.

  This change makes it easy to add new behaviour to enums - you just define methods in the enum definition. However, the combination of inheriting members *and* adding additional behaviour is incoherent - you can't do both and get a sensible result, as you either don't actually inherit the members (as you want to add additional behaviour, and thus wrap them in a different type) *or* you use the inherited members, which then don't support the additional added behaviours.

  The other problem with enum extension-through-inheritance is that one of the standard expectations of class inheritance is that the base class be usable wherever an instance of the parent class is expected. However, one of the assumptions of enumerations is that `isinstance(x, MyEnum)` implies `x in MyEnum` and vice-versa, and that's automatically violated as soon as you add members in a subclass (the members of the subclass will satisfy the first condition, but not the second).

  PEP 435 addressed this by adding the restriction that you simply can't subclasse an enumeration that already has defined members (this is a similar restriction to the one that Java places on their enumerations).

I suspect that extensible enums are going to require a slightly different abstraction, closer to the `flufl.enum` model, where the group members are aggregated from multiple independent underlying enumerations. For example, something like:

```
class MoreColors(AggregateEnum, extends=Color):
    cyan = ...
    magenta = ...
```

To some degree, this "customise the metaclass if you want something different" approach *is* indeed a copout - we're providing a lowest-common-demoninator enum implementation, and leaving it to people to add their own syntactic sugar on top if they really want to. On the other hand, this is an approach the Python core development team has been using successfully for a *long* time: providing a basic initial implementation, and then seeing how that initial approach is used in the real world before tweaking it in future versions.

Improving the functional APIs

We're *not* happy with the current state of the functional APIs for either named tuples and enumerations. However, rather than being limited to either of those specific use cases, the limitations of those APIs are symptomatic of a deeper language design problem relating to the creation of arbitrary objects that know their own names and their locations in the import namespace, along with the inability to cleanly specify lists of identifiers in a way that is visible to and checked by the compiler.

Thus, any improvements to these APIs will likely be based on addressing those broader design problems. It doesn't make sense to hold up standardisation of enumerations for the resolution of those much harder design problems though, particularly when the immediately available workarounds aren't *that* ugly.

This is also the reason we're not going to provide an `ImplicitEnum` implementation in the standard library at this time - it's not yet clear if that's the right answer to the problem.

I did like the idea of using the members qualified names as their values for the enum functional API, though, so I filed an issue suggesting we change to that after the core implementation of the PEP has been put in place. However, as I noted when ultimately rejecting that change, the current approach works well with any likely concrete subclasses (specifically numbers and strings), whereas the same cannot be said for using the member names (that would break as soon as you tried to use the functional API with a numeric subclass).

CHAPTER 2

Python Concepts

Years ago (around the Python 2.5 time frame) I wrote a draft manuscript for a Python User's Reference that aimed at providing a middle ground between the tutorial and the full language reference. The idea was to cover everything that was included in the language reference, but in a way that was designed to help Python developers understand what the interpreter was doing on their behalf, rather than being aimed at developers trying to produce a compliant interpreter implementation.

That book deal never went anywhere, so I asked for the copyright on the manuscript to be returned, and donated the draft to the PSF. Converting the ODF files to reStructuredText has been an idle thought ever since, but my motivation to do so has dropped ever lower as the contents of the manuscript become increasingly out of date.

So, instead, this page will host an ad hoc series of articles that dive deep into some esoteric element of the Python language definition, and attempt to explain it in ways that may be more intuitive than the official version.

Traps for the Unwary in Python's Import System

Python's import system is powerful, but also quite complicated. Until the release of Python 3.3, there was no comprehensive explanation of the expected import semantics, and even following the release of 3.3, the details of how `sys.path` is initialised are still somewhat challenging to figure out.

Even though 3.3 cleaned up a lot of things, it still has to deal with various backwards compatibility issues that can cause strange behaviour, and may need to be understood in order to figure out how some third party frameworks operate.

Furthermore, even without invoking any of the more exotic features of the import system, there are quite a few common missteps that come up regularly on mailing lists and Q&A sites like Stack Overflow.

This essay only officially covers Python versions back to Python 2.6. Much of it applies to earlier versions as well, but I won't be qualifying any of the explanations with version details before 2.6.

As with all my essays on this site, suggestions for improvement or requests for clarification can be posted on BitBucket.

The missing __init__.py trap

This particular trap applies to 2.x releases, as well as 3.x releases up to and including 3.2.

Prior to Python 3.3, filesystem directories, and directories within zipfiles, *had* to contain an `__init__.py` in order to be recognised as Python package directories. Even if there is no initialisation code to run when the package is imported, an empty `__init__.py` file is still needed for the interpreter to find any modules or subpackages in that directory.

This has changed in Python 3.3: now any directory on `sys.path` with a name that matches the package name being looked for will be recognised as contributing modules and subpackages to that package.

Consider this directory layout:

```
project/
    example/
        foo.py
```

Where `foo.py` contains the source code:

```
print("Hello from ", __name__)
```

With that layout and the current working directory being `project`, Python 2.7 gives the following behaviour:

```
$ python2 -c "import example.foo"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named example.foo
```

While Python 3.3+ is able to import the submodule without any problems:

```
$ python3 -c "import example.foo"
Hello from  example.foo
```

The __init__.py trap

This is an all new trap added in Python 3.3 as a consequence of fixing the previous trap: if a subdirectory encountered on `sys.path` as part of a package import contains an `__init__.py` file, then the Python interpreter will create a *single directory* package containing only modules from that directory, rather than finding all appropriately named subdirectories as described in the previous section.

This happens *even if* there are other preceding subdirectories on `sys.path` that match the desired package name, but do not include an `__init__.py` file.

Let's take the preceding layout, and add a second project that *also* has an `example` directory, but includes an `__init__.py` file in it, as well as a `bar.py` file with the same contents as `foo.py`:

```
project/
    example/
        foo.py
project2/
    example/
        __init__.py
        bar.py
```

If we explicitly add the second project to `PYTHONPATH`, we find that Python 3.3+ can't find `example.foo` either, as the directory containing `__init__.py` file prevents the creation of a multi-directory namespace package:

```
$ PYTHONPATH=../project2 python3 -c "import example.bar"
Hello from  example.bar
$ PYTHONPATH=../project2 python3 -c "import example.foo"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named 'example.foo'
```

However, if we remove the offending `__init__.py` file, Python 3.3+ will automatically create a namespace package and be able to see both submodules:

```
$ rm ../project2/example/__init__.py
$ PYTHONPATH=../project2 python3 -c "import example.bar"
Hello from  example.bar
$ PYTHONPATH=../project2 python3 -c "import example.foo"
Hello from  example.foo
```

This complexity is primarily forced on us by backwards compatibility constraints - without it, some existing code would have broken when Python 3.3 made the presence of `__init__.py` files in packages optional.

However, it is also useful in that it makes it possible to explicitly declare that a package is closed to additional contributions. All of the standard library currently works that way, although some packages may open up their namespaces to third party contributions in future releases (the key challenge with that idea is that namespace packages can't offer any package level functionality, which creates a major backwards compatibility problem for existing standard library packages).

The double import trap

This next trap exists in all current versions of Python, including 3.3, and can be summed up in the following general guideline: "Never add a package directory, or any directory inside a package, directly to the Python path".

The reason this is problematic is that every module in that directory is now potentially accessible under two different names: as a top level module (since the directory is on `sys.path`) and as a submodule of the package (if the higher level directory containing the package itself is also on `sys.path`).

As an example, Django (up to and including version 1.3) used to be guilty of setting up exactly this situation for site-specific applications - the application ends up being accessible as both `app` and `site.app` in the module namespace, and these are actually two *different* copies of the module. This is a recipe for confusion if there is any meaningful mutable module level state, so this behaviour was eliminated from the default project layout in version 1.4 (site-specific apps now always need to be fully qualified with the site name, as described in the release notes).

Unfortunately, this is still a *really* easy guideline to violate, as it happens automatically if you attempt to run a module inside a package from the command line by filename rather than using the `-m` switch.

Consider a project & package layout like the following (I typically use package layouts along these lines in my own projects - a lot of people hate nesting tests inside package directories like this, and prefer a parallel hierarchy, but I favour the ability to use explicit relative imports to keep module tests independent of the package name):

```
project/
    setup.py
    example/
        __init__.py
        foo.py
        tests/
            __init__.py
            test_foo.py
```

What's surprising about this layout is that all of the following ways to invoke `test_foo.py` *probably won't work* due to broken imports (either failing to find `example` for absolute imports like `import example.foo` or `from example import foo`, complaining about relative imports in a non-package or beyond the top-level package for explicit relative imports like `from .. import foo`, or issuing even more obscure errors if some other submodule happens to shadow the name of a top-level module used by the test, such as an `example.json` module that handled serialisation or an `example.tests.unittest` test runner):

```
# These commands will most likely *FAIL* due to problems with the way
# the import state gets initialised, even if the test code is correct

# working directory: project/example/tests
./test_foo.py
python test_foo.py
python -m test_foo
python -c "from test_foo import main; main()"

# working directory: project/example
tests/test_foo.py
python tests/test_foo.py
python -m tests.test_foo
python -c "from tests.test_foo import main; main()"

# working directory: project
example/tests/test_foo.py
python example/tests/test_foo.py

# working directory: project/..
project/example/tests/test_foo.py
python project/example/tests/test_foo.py
python -m project.example.tests.test_foo
python -c "from project.example.tests.test_foo import main; main()"
```

That's right, that long list is of all the methods of invocation that are quite likely to *break* if you try them, and the error messages won't make any sense if you're not already intimately familiar not only with the way Python's import system works, but also with how it gets initialised. (Note that if the project exclusively uses explicit relative imports for intra-package references, the last two commands shown may actually work for Python 3.3 and later versions. Any absolute imports that expect "example" to be a top level package will still break though).

For a long time, the only way to get `sys.path` right with this kind of setup was to either set it manually in `test_foo.py` itself (hardly something novice, or even many veteran, Python programmers are going to know how to do) or else to make sure to import the module instead of executing it directly:

```
# working directory: project
python -c "from example.tests.test_foo import main; main()"
```

Since Python 2.6, however, the following also works properly:

```
# working directory: project
python -m example.tests.test_foo
```

This last approach is actually how I prefer to use my shell when programming in Python - leave my working directory set to the project directory, and then use the `-m` switch to execute relevant submodules like tests or command line tools. If I need to work in a different directory for some reason, well, that's why I also like to have multiple shell sessions open.

While I'm using an embedded test case as an example here, similar issues arise any time you execute a script directly from inside a package without using the `-m` switch from the parent directory in order to ensure that `sys.path` is initialised correctly (e.g. the pre-1.4 Django project layout gets into trouble by running `manage.py` from inside a

package, which puts the package directory on `sys.path` and leads to this double import problem - the 1.4+ layout solves that by moving `manage.py` outside the package directory).

The fact that most methods of invoking Python code from the command line break when that code is inside a package, and the two that do work are highly sensitive to the current working directory is all thoroughly confusing for a beginner. I personally believe it is one of the key factors leading to the perception that Python packages are complicated and hard to get right.

This problem isn't even limited to the command line - if `test_foo.py` is open in Idle and you attempt to run it by pressing F5, or if you try to run it by clicking on it in a graphical filebrowser, then it will fail in just the same way it would if run directly from the command line.

There's a reason the general "no package directories on `sys.path`" guideline exists, and the fact that the interpreter itself doesn't follow it when determining `sys.path[0]` is the root cause of all sorts of grief.

However, even if there are improvements in this area in future versions of Python, this trap will still exist in all current versions.

Executing the main module twice

This is a variant of the above double import problem that doesn't require any erroneous `sys.path` entries.

It's specific to the situation where the main module is *also* imported as an ordinary module, effectively creating two instances of the same module under different names.

As with any double-import problem, if the state stored in `__main__` is significant to the correct operation of the program, or if there is top-level code in the main module that has undesirable side effects if executed more than once, then this duplication can cause obscure and surprising errors.

This is just one more reason why main modules in more complex applications should be kept fairly minimal - it's generally far more robust to move most of the functionality to a function or object in a separate module, and just import that module from the main module. That way, inadvertently executing the main module twice becomes harmless. Keeping main modules small and simple also helps to avoid a few potential problems with object serialisation as well as with the multiprocessing package.

The name shadowing trap

Another common trap, especially for beginners, is using a local module name that shadows the name of a standard library or third party package or module that the application relies on. One particularly surprising way to run afoul of this trap is by using such a name for a *script*, as this then combines with the previous "executing the main module twice" trap to cause trouble. For example, if experimenting to learn more about Python's `socket` module, you may be inclined to call your experimental script `socket.py`. It turns out this is a really bad idea, as using such a name means the Python interpreter can no longer find the *real* socket module in the standard library, as the apparent socket module in the current directory gets in the way:

```
$ python -c 'from socket import socket; print("OK!")'
OK!
$ echo 'from socket import socket; print("OK!")' > socket.py
$ python socket.py
Traceback (most recent call last):
  File "socket.py", line 1, in <module>
    from socket import socket
  File "/home/ncoghlan/devel/socket.py", line 1, in <module>
    from socket import socket
ImportError: cannot import name socket
```

The stale bytecode file trap

Following on from the example in the previous section, suppose we decide to fix our poor choice of script name by renaming the file. In Python 2, we'll find that still doesn't work:

```
$ mv socket.py socket_play.py
$ python socket_play.py
Traceback (most recent call last):
  File "socket_play.py", line 1, in <module>
    from socket import socket
  File "/home/ncoghlan/devel/socket.py", line 1, in <module>
    # Wrapper module for _socket, providing some additional facilities
ImportError: cannot import name socket
```

There's clearly something strange going on here, as we're seeing a traceback that claims to be caused by a *comment* line. In reality, what has happened is that the cached bytecode file from our previous failed import attempt is still present and causing trouble, but when Python tries to display the source line for the traceback, it finds the source line from the standard library module instead. Removing the stale bytecode file makes things work as expected:

```
$ rm socket.pyc
$ python socket_play.py
OK!
```

This particular trap has been largely eliminated in Python 3.2 and later. In those versions, the interpreter makes a distinction between standalone bytecode files (such as `socket.pyc` above) and cached bytecode files (stored in automatically created `__pycache__` directories). The latter will be ignored by the interpreter if the corresponding source file is missing, so the above renaming of the source file works as intended:

```
$ echo 'from socket import socket; print("OK!")' > socket.py
$ python3 socket.py
Traceback (most recent call last):
  File "socket.py", line 1, in <module>
    from socket import socket
  File "/home/ncoghlan/devel/socket.py", line 1, in <module>
    from socket import socket
ImportError: cannot import name socket
$ mv socket.py socket_play.py
$ python3 socket_play.py
```

Note, however, that mixing Python 2 and Python 3 can cause trouble if Python 2 has left a standalone bytecode file lying around:

```
$ python3 socket_play.py
Traceback (most recent call last):
  File "socket_play.py", line 1, in <module>
    from socket import socket; print("OK!")
ImportError: Bad magic number in /home/ncoghlan/devel/socket.pyc
```

If you're not a core developer on a Python implementation, the problem of importing stale bytecode is most likely to arise when renaming Python source files. For Python implementation developers, it can also arise any time we're working on the compiler components that are responsible for generating the bytecode in the first place - that's the main reason the CPython `Makefile` includes a `make pycremoval` target.

The submodules are added to the package namespace trap

Many users will have experienced the issue of trying to use a submodule when only importing the package that it is in:

```
$ python3
>>> import logging
>>> logging.config
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'config'
```

But it is less common knowledge that when a submodule is loaded *anywhere* it is automatically added to the global namespace of the package:

```
$ echo "import logging.config" > weirdimport.py
$ python3
>>> import weirdimport
>>> import logging
>>> logging.config
<module 'logging.config' from '/usr/local/Cellar/python3/3.4.3/Frameworks/Python.
↪framework/Versions/3.4/lib/python3.4/logging/config.py'>
```

This is most likely to surprise you when in an `__init__.py` and you are importing or defining a value that has the same name as a submodule of the current package. If the submodule is loaded by *any* module at any point after the import or definition of the same name, it will shadow the imported or defined name in the `__init__.py`'s global namespace.

More exotic traps

The above are the common traps, but there are others, especially if you start getting into the business of extending or overriding the default import system.

I may add more details on each of these over time:

- the weird signature of `__import__`
- the influence of the module globals (`__import__`, `__path__`, `__package__`)
- issues with threads prior to 3.3
- the lack of PEP 302 support in the default machinery prior to 3.3
- non-cooperative package portions in pre-3.3 namespace packages
- sys.path[0] initialisation variations
- more on the issues with pickle, multiprocessing and the main module (see PEP 395)
- `__main__` is not always a top level module (thanks to `-m`)
- the fact modules are allowed to replace themselves in sys.modules during import
- `__file__` may not refer to a real filesystem location
- since 3.2, you can't just add `c` or `o` to get the cached bytecode filename

Else Clauses on Loop Statements

Python's loop statements have a feature that some people love (Hi!), some people hate, many have never encountered and many just find confusing: an `else` clause.

This article endeavours to explain some of the reasons behind the frequent confusion, and explore some other ways of thinking about the problem that give a better idea of what is *really* going on with these clauses.

Reasons for Confusion

The major reason many developers find the behaviour of these clauses potentially confusing is shown in the following example:

```
>>> if [1]:
...     print("Then")
... else:
...     print("Else")
...
Then
>>> for x in [1]:
...     print("Then")
... else:
...     print("Else")
...
Then
Else
```

The `if <iterable>` header looks very similar to the `for <var> in <iterable>` header, so it's quite natural for people to assume they're related and expect the `else` clause to be skipped in both cases. As the example shows, this assumption is incorrect: in the second case, the else clauses triggers even though the iterable isn't empty.

If we then look at a common `while` loop pattern instead, it just deepens the confusion because *it* seems to line up with the way we would expect the conditional to work:

```
>>> x = [1]
>>> while x:
...     print("Then")
...     x.pop()
... else:
...     print("Else")
...
Then
Else
>>> if x:
...     print("Then")
... else:
...     print("Else")
...
Else
```

Here, the loop runs until the iterable is empty, and then the `else` clause is executed, just as it is in the `if` statement.

A different kind of `else`

So what's going on? The truth is that the superficial similarity between `if <iterable>` and `for <var> in <iterable>` is rather deceptive. If we call the `else` clause on an `if` statement a "conditional else", then we can look to `try` statements for a different *kind* of `else` clause, a "completion clause":

```
>>> try:
...     pass
... except:
...     print("Then") # The try block threw an exception
... else:
...     print("Else") # The try block didn't throw an exception
```

```
...
Else
```

With a completion clause, the question being asked has to do with how an earlier suite of code *finished*, rather than checking the boolean value of an expression. Reaching the `else` clause in a `try` statement means that the try block actually completed successfully - it didn't throw an exception or otherwise terminate before reaching the end of the suite.

This is actually a much better model for what's going on in our `for` loop, since the condition the `else` is checking for is whether or not the loop was explicitly terminated by a `break` statement. While it's not legal syntax, it may be helpful to mentally insert an `except break: pass` whenever you encounter a loop with an associated `else` clause in order to help remember what it means:

```
for x in iterable:
    ...
except break:
    pass # Implied by Python's loop semantics
else:
    ...  # No break statement was encountered

while condition:
    ...
except break:
    pass # Implied by Python's loop semantics
else:
    ...  # No break statement was encountered
```

What possible use is the current behaviour?

The main use case for this behaviour is to implement search loops, where you're performing a search for an item that meets a particular condition, and need to perform additional processing or raise an informative error if no acceptable value is found:

```
for x in data:
    if acceptable(x):
        break
else:
    raise ValueError("No acceptable value in {!r:100}".format(data))

... # Continue calculations with x
```

But how do I check if my loop never ran at all?

The easiest way to check if a `for` loop never executed is to use `None` as a sentinel value:

```
x = None
for x in data:
    ... # process x
if x is None:
    raise ValueError("Empty data iterable: {!r:100}".format(data))
```

If `None` is a legitimate data value, then a custom sentinel object can be used instead:

```
x = _empty = object()
for x in data:
    ... # process x
if x is _empty:
    raise ValueError("Empty data iterable: {!r:100}".format(data))
```

For `while` loops, the appropriate solution will depend on the details of the loop.

But couldn't Python be different?

Backwards compatibility constraints and the general desire not to change the language core without a compelling justification mean that the answer to this question is likely always going to be "No".

The simplest approach for any new language to take to avoid the confusion encountered in relation to this feature of Python would be to just leave it out altogether. Many (most?) other languages don't offer it, and there are certainly other ways to handle the search loop use case, including a sentinel based approach similar to that used to detect whether or not a loop ran at all:

```
result = _not_found = object()
for x in data:
    if acceptable(x):
        result = x
        break
if result is _not_found:
    raise ValueError("No acceptable value in {!r:100}".format(data))

... # Continue calculations with result
```

Closing note: Not so different after all?

Attentive readers may have noticed that the behaviour of `while` loops still makes sense regardless of whether you think of their `else` clause as a conditional else or as a completion clause. We can think of a `while` statement in terms of an infinite loop containing a `break` statement:

```
while True:
    if condition:
        pass # Implied by Python's loop semantics
    else:
        ... # While loop else clause runs here
        break
    ... # While loop body runs here
```

If you dig deep enough, it's also possible to relate the completion clause constructs in `try` statements and `for` loops back to the basic conditional else construct. The thing to remember though, is that it is only `while` loops and `if` statements that are checking the boolean value of an expression, while `for` loops and `try` statements are checking whether or not a section of code was aborted before completing normally.

However, digging to that deeper level doesn't really provide much more enlightenment when it comes to understanding how the two different forms of `else` clause work in practice.

CHAPTER 3

Various Ideas for Python and CPython

My ability to generate ideas for ways in which Python could potentially be improved in various areas vastly outstrips the time I'm willing to spend following up on them, kicking them around with other community members and turning them into suggestions that may actually prove to be a genuine enhancement to the language rather than just useless cruft new Python programmers have to learn in order to understand code other programmers might write.

As a result, there are plenty of ideas I've had that may actually have some merit, but that I don't have the time to explore. Some of these are going to be *bad* ideas that justifiably won't survive the gauntlet of python-ideas and python-dev review, while others may make sense as esoteric PyPI modules rather than as standard library or core language additions.

Beyond that, there are many fascinating things *other* people are working on, where I may be able to contribute some thoughts, but don't have the time to get more directly involved. Those noted below are all core development related - there are many fascinating things going on in the community and education space, as well as at higher levels of the application stack (especially in web development and scientific computing) but in those spaces I truly am just an observer (albeit one that is enthusiastically cheering on those doing the work).

Development Philosophy

This sections covers my general development philosophy as it applies to Python and CPython in particular.

Python's Users

One of the reactions we sometimes get to various things we work on for Python is one of incredulity: "Why on Earth are you wasting your time on X? Problem Y is far more important!".

The answer is typically "because the relative importance of X and Y depends on your point of view, and I currently consider X to be the more important of the two" (although "because I find working on X to be more fun and interesting than working on Y" is another common reason).

Assessing the relative importance of different changes to the language involves making assumptions regarding the way the language is used. As core developers, we actually have a pretty decent view into how it gets used in practice, and many of us also have the experience and contacts to know how it gets used by the silent majority of developers

operating behind corporate and government firewalls (of both the civilian and military variety) rather than participating in the open source development community.

However, we don't make a habit of articulating the different users we're trying to design for, so it's not surprising that the common answer of "this is not for you" isn't recognised in advance.

The various sections of this essay try to go over the different groups I keep in mind when working on CPython, my current perspective on how well I think those groups are served by the Python community (as of 18 March 2013), and where I see support for those groups currently heading.

Optimise for Maintenance

One of Guido's key insights into language design is that code is read far more often than it is written, so it makes more sense to optimise for software maintenance than it does to optimise for starting from an empty file. This means that readability is highly prized in the language design, and brevity is favoured only insofar as it improves readability.

Individual Automation

These users represent the classic "scripting" use case that gives scripting languages their name. Traditionally associated with system administrators, this role encompasses any individual that uses Python to automate aspects of their own activities, without necessarily sharing those tools with anyone else.

The line between individual automation and system integration is a complex one. Often a script may start life as an individual automation tool and then later become a critical workflow tool as it is shared amongst a group.

There are many characteristics of CPython (such as the large standard library) that are useful for this group

System Integrators

Scientists and Data Explorers ————————————s

Internet Application Developers

Corporate Internal Developers

Educators and Students

Archived Articles

These are articles that I wrote to help organise my thoughts on a particular topic, but am not currently actively maintaining:

Incremental Plans to Improve Python Packaging

Note: This essay is not the most recent source of packaging information. It is retained as a potentially interesting recording of my thinking when I first volunteered to be the BDFL-Delegate for packaging related PEPs :)

For more recent information, refer to the Python Packaging Authority's Python Packaging User Guide.

As of March 2013, the Python packaging ecosystem is currently in a rather confusing state. If you're starting a new Python project, should you use the `distutils` system in the standard library to publish your software? Or should you use `setuptools`? If you decide to use `setuptools`, should you download the PyPI distribution that is actually *called* `setuptools`, or should you download the competing `distribute` distribution, which *also* provides `setuptools`?

If you're on Linux, it's quite likely your operating system already *comes* with a pre-installed copy of `setuptools`. Perhaps you can use that?

The above options all use a Python script called `setup.py` to configure the build and distribution software. You'll see plenty of essays on the internet decrying this use of an executable configuration system, and promoting the use of static configuration metadata instead. So, perhaps the answer is to avoid all of them and use `distutils2` (or `d2to1`), with a `setup.cfg` file, or `bento` with its `bento.info` file instead?

Even on the *installation* side, things are confusing. `setuptools` and `distribute` both provide a utility called `easy_install`. Should you use that? But there are websites telling you that `easy_install` is bad and you should use `pip` instead!

Or perhaps everything above is wrong, and you should be using `hashdist`, `conda` or `zc.buildout` to create fully defined stacks of dependencies, including non-Python software?

HALP! MAKE IT STOP!

TL;DR

The current situation *is* messy, especially sinces it makes it hard for users to find good, clearly authorative, guides to packaging Python software. While many people are actively working on improving the situation, it's going to take a while for those improvements to be fully deployed.

Note: This section is outdated. For more recent information, refer to the Python Packaging Authority's Python Packaging User Guide.

If all you're after is clear, simple advice for right now, today, as of March 2013, this is as clear as it gets:

- use distribute to build and package Python distributions and publish them to the Python Package Index (PyPI). The `setuptools` and `distribute` projects are in the process of merging back together, but the merger isn't complete yet (I will update this essay as soon as that changes).
- use pip to install Python distributions from PyPI
- use virtualenv to isolate application specific dependencies from the system Python installation
- use zc.buildout (primarily focused on the web development community) or hashdist and conda (primarily focused on the scientific community) if you want fully intregrated software stacks, without worrying about interoperability with platform provided package management systems
- if you're on Linux, the versions of these tools provided as platform specific packages should be fine for most purposes, but may be missing some of the latest features described on the project websites (and, in Fedora's case, calls `pip` `pip-python` due to a naming conflict with an old Perl tool).

Unfortunately, there are a couple of qualifications required on that simple advice:

- use `easy_install` or zc.buildout if you need to install from the binary `egg` format, which `pip` can't currently handle
- aside from using `distribute` over the original `setuptools` (again, keeping in mind that those two projects are in the process of merging back into a single `setuptools` project), and `pip` over `easy_install` whenever possible, try to ignore the confusing leftovers of slanging matches between developers of competing tools,

as well as information about upcoming tools that are likely still months or years away from being meaningful to anyone not directly involved in developing packaging tools

The Quick Start section of the Hitchhiker's Guide to Packaging provided by the `distribute` team is still a decent introduction to packaging and distributing software through the Python Package Index. However, the rest of that guide includes a lot of opinions and plans that don't quite match up with reality (this is being cleaned up as part of the current packaging ecosystem improvement efforts).

After the quickstart, your best options for learning more are likely the distribute project documentation and the standard library's own guide to Distributing Python Modules (Legacy version).

A (long) caveat on this essay

I'm currently the "BDFL-Delegate" for packaging related Python PEPs. It's important to understand that being a BDFL-Delegate does *not* mean I automatically get my way - even Guido van Rossum doesn't automatically get his way in regards to Python, and he's Python's actual Benevolent Dictator for Life! This is a large part of why we call him a *benevolent* dictator - most of the time he only needs to invoke his BDFL status to cut short otherwise interminable arguments where there are several possible answers, any of which qualifies as "good enough".

Instead, being a BDFL-Delegate means I get to decide when "rough consensus" has been achieved in relation to such PEPs. I'm trusted to listen to feedback on PEPs that are being proposed for acceptance (including any where I am both author and BDFL-Delegate) and exercise good judgement on which criticisms I think are valid, and need to be addressed before I accept the PEP, and which criticisms can be safely ignored (or deferred), deeming the contents of the PEP "good enough" (or at least "good enough for now").

In the case of packaging PEPs, I have identified a core set of projects whose involvement I consider essential in assessing any proposals (in particular the updated metadata standard described in **PEP 426**):

- CPython - masters of distutils, the default Python build system
- setuptools - originators of the dominant 3rd party Python build system
- distribute - popular, recommended (and Python 3 compatible) setuptools fork (this project and setuptools are in the process of merging back together)
- pip - Python installer that addresses many of the design flaws in easy_install
- distlib - distribution tools infrastructure, originally part of distutils2
- PyPI - the central index for public Python packages

In addition, there are other projects whose developers provide valuable additional perspectives:

- zc.buildout - powerful and flexible application deployment manager
- hashdist/conda - a "software stack" management system, tailored towards the scientific community, which needs to deal with arcane build requirements and a large user community that is interested in software solely as a tool rather than in its own right
- distutils2 - alternate build system that replaces setup.py with setup.cfg and the last major attempt at bringing order to the Python packaging ecosystem
- bento - an experimental alternative build system

Why those projects? A few different reasons:

1. Almost all of the listed projects have representatives that are active on distutils-sig and catalog-sig, the primary mailing lists for discussing changes that affect the overall Python packaging ecosystem (hashdist/conda are currently an exception, but I'm hoping that will change at some point).
2. CPython needs to be involved because support for the new standards should be part of the Python 3.4 standard library (just as it was previously proposed that distutils2 would be added to the 3.3 standard library).

3. distlib needs to be involved as that is the project to extract the core distribution management infrastructure from distutils2 that *almost* made it into Python 3.3. It serves as the reference implementation for the new metadata format proposed in PEP 426, will likely be proposed as the basis of any support for the new formats in Python 3.4, and may hopefully be used as part of other distribution tools prior to inclusion in the standard library (as a real world usability test for the API).
4. PyPI needs to be involved, in order to act as an effective and efficient publisher of the richer metadata set
5. Five of the projects (setuptools, distribute, hashdist, distutils2, bento) provide build systems that are usable with *current* versions of Python, rather than requiring an upgrade to Python 3.4. If a new metadata standard is to see widespread adoption, all of them need to be able to generate it.
6. Eight of the projects (setuptools, distribute, pip, zc.buildout, conda, distutils2, distlib) provide or rely on dependency resolvers and other tools that consume metadata at installation time. If a new metadata standard is to see widespread adoption, all of them need to be able to correctly retrieve and process that metadata from the package index, source and binary archives, as well as the target installation environment.
7. Four of the projects (setuptools, distribute, distutils2, distlib) provide tools for accessing distribution metadata at runtime. If a new metadata standard is to see widespread adoption, all of them need to be able to retrieve and process that metadata from the execution environment.
8. Between them, these projects and their derivatives, cover the vast majority of the current Python packaging ecosystem. If they collectively endorse an updated metadata standard, it has a good chance of succeeding. If they reject it, then it really doesn't matter if python-dev nominally accepts it (and, in fact, python-dev would be wrong to do so, as we have unfortunately learned the hard way).

The Phases of Distribution

One component severely lacking in the status quo is a well-defined model of the phases of distribution. An overall packaging system needs to be able to handle several distinct phases, especially the transitions between them. For Python's purposes, these phases are:

- Development: working with source code in a VCS checkout
- Source Distribution: creating and distributing a source archive
- Building: creating binary files from a source archive
- Binary Distribution: creating and distributing a binary archive
- Installation: installing files from a binary archive onto the target system
- Execution: importing or otherwise running the installed files

The setuptools distribution covers *all six* of those phases. A key goal of any new packaging system should be to cleanly decouple the phases and make it easier for developers to choose the right tool for each phase rather than having one gigantic project that handles everything internally with poorly defined data interchange formats. Having a single project handle everything should still be *possible* (at least for backwards compatibility, even if for no other reason), it just shouldn't be required.

distutils isn't much better, since it is still an unholy combination of a build system *and* a packaging system. Even RPM doesn't go that far: it's "build system" is just the ability to run a shell script that invokes your *real* build system. In many ways, distutils was really intended as Python's equivalent of `make` (or perhaps `make` + `autotools`), so we're currently in the situation Linux distributions were in before the creation of dedicated package management utilities like `apt` and `yum`.

It isn't really a specific phase, but it's also desirable for a meta-packaging system to define a standard mechanism for invoking a distribution's automated test suite and indicate whether or not it passed all its tests.

A Meta-Packaging System

My goal for Python 3.4 is to enable a solid *meta-packaging* system, where we have multiple, cooperating, tools, each covering distinct phases of distribution. In particular, a project's choice of build system should NOT affect on end user's choice of installation program.

In this system, there are a few key points where interoperability between different tools is needed:

1. For binary distribution, an installation tool should be able to unpack and install the contents of the binary archive to the appropriate locations, *without* needing to install the build system used to create the archive.
2. For source distribution, an installation tool should be able to identify the appropriate build tool, download and install it, and then invoke it in a standard fashion, *without* needing any knowledge of any particular build systems.
3. The central package index needs to accept and publish distribution metadata in a format that is easy to consume
4. Installation tools need to store the distribution metadata in a standard format so other tools know where to find it and how to read it.

The development phase and the execution phase are the domain of build tools and runtime support libraries respectively. The interfaces they expose to end users in those phases are up to the specific tool or library - the meta-packaging system only cares about the interfaces between the automated tools.

The `wheel` format

The binary `wheel` format, created by Daniel Holth, and formally specified in **PEP 427**, is aimed at solving two problems:

- initially, acting as a cache format for `pip`, allowing that tool to avoiding having to rebuild packages from source in each virtual environment
- eventually, as build tools gain the ability to publish wheels to PyPI, and more projects start to do so, as a way to support distribution of Python software that doesn't require the invocation of `./setup.py install` on the target system

This is a critical step, as it finally allows the build systems to be systematically decoupled from the installation systems - if `pip` can get its hands on a `wheel` file for a project, it will be possible to install it, even if it uses some arcane build tools that only run on specific systems.

In many respects, `wheel` is a *simpler* format than the setuptools egg format. It deliberately avoids all of the features of eggs (or, more accurately, `easy_install`) which resulted in runtime modifications to the target environment. Those were the features that people disliked as being excessively magical, and which limited the popularity of the format.

In two respects, wheel is *more* complex than the egg format. Firstly, the compatibility tagging scheme used in file names (defined in **PEP 425**) is more comprehensive, allowing the interpreter implementation and version to be clearly specified, along with the Python C ABI requirement, and the underlying platform compatibility.

Secondly, the wheel format allows *multiple* target directories to be defined, as is supported by the `distutils` installation operation. This allows the format to support correctly spreading files to appropriate directories on a target system, rather than dropping all files into a single directory in violation of platform standards (although the wheel format *does* also support the latter style).

Python distribution metadata v2.0

My own efforts are currently focused primarily on **PEP 426**, the latest version of the standard for Python distribution metadata. My aim with this latest version of the metadata is to address the issues which prevented widespread adoption of the previous version by:

- deciding on appropriate default behaviour for tools based on the experiences of other development language communities
- supporting additional features of setuptools/distribute that were missing from the previous version of the standard
- engaging with the distribute and setuptools developers to ensure both of those projects (or, as is now more likely, the post-merger setuptools) are able to start emitting the new metadata format within a reasonable period of time after the standard is accepted
- simplifying backwards compatibility with those same two projects (just one after the merger) by adding a recommendation for installation tools to correctly generate legacy versions of the metadata that those two projects will be able to easily read

I also plan to design this standard to use JSON as the on-disk serialisation format. There are four reasons for this:

- Over time, the original simple Key:Value format has grown various ad hoc extensions to support structured data that doesn't fit the simple key-value format. Some fields are "multi-use", some allow embedded environment markers, one is a space separated sequence of values. Switching to JSON means structured data is supported simply and cleanly, without these ad hoc complexities in the parsing rules.
- to completely replace the existing `./setup.py install` idiom, **PEP 426** is going to need to define a post-install hook, and conversion to a more structured format makes it easier to pass the metadata to the registered hook
- **PEP 376** currently ignores the existence of import hooks completely: it is only correctly defined for Python distributions that are installed to the filesystem. Fixing that will require a structured metadata representation that can be returned from an appropriate importer method.
- TUF (The Update Framework), is an intriguing approach proposed for adding a usable end-to-end security solution to the Python packaging ecosystem. One feature of TUF is the ability to embed arbitrary JSON metadata describing "targets", which, in Python's case, would generally mean source and binary archives for distributions.

Converting the earlier versions of PEP 426 (which still use the old key:value format as a basis) to a useful platform-neutral JSON compatible metadata format is actually fairly straightforward, and Daniel Holth already has a draft implementation of the bdist_wheel distutils command that emits a preliminary version of it.

Secure metadata distribution

In the wake of the rubygems.org compromise, a topic of particular interest on catalog-sig is the definition of a reliable, usable, end-to-end security mechanism that allows end users the option of either trusting PyPI to maintain the integrity of distributed packages, *or* maintaining their own subset of trusted developer keys.

While I'm not actively working on this myself, I'm definitely interested in the topic, and currently favour the concept of adopting The Update Framework, a general purpose software updating architecture, designed to protect from a wide variety of known attack vectors on software distribution systems. I particularly like the fact that TUF may not only address the end-to-end security problem, but also provide a *far* superior metadata publication system to that provided by the current incarnation of the PyPI web service.

A number of the TUF developers are now active on catalog-sig, attempting to devise an approach to securing the *existing* PyPI metadata, which may then evolve over time to take advantage of more of TUF's features.

A Bit of Python Packaging History

The `packaging` module (based on the `distutils2` project) was slated for inclusion in Python 3.3. However, it was ultimately removed, as the lead developers of the project felt it was not yet sufficiently mature.

Following that decision, the entire approach being taken to enhancing Python's packaging ecosystem has been in the process of being reassessed. This essay is part of my own contribution to that reassessment, and the reasoning

described here is the reason I decided to offer to take on the role of BDFL delegate for any PEPs related to the packaging ecosystem.

This essay also serves as a clear declaration of my vision for how I think we can avoid repeating the mistakes that limited the overall effectiveness of the `distutils2` effort, and make further improvements to the Python packaging ecosystem. If this effort is successful, then improved software distribution utilities should become one of the flagship features of Python 3.4.

How did we get here?

(This section is painted in fairly broad strokes, both because the details don't really matter, and also because I don't want to go double check everything I would have to in order to get the details right)

Python's packaging history largely starts with the inclusion of the `distutils` project into the standard library. This system was really built to handle distribution of source modules and simple C extensions, but ended up being pushed well beyond that task. I was lucky enough to meet Greg Ward at PyCon US 2013, and he has posted a great write-up of the early history of distutils as part of his post-conference review.

Another key piece of the puzzle was the creation of the Python Package Index to serve as a central repository for Python packages that could be shared by the entire community, without being coupled to any particular operating system or platform specific packaging format.

One notable enhancement was Phillip Eby's `setuptools`, which became popular after he created it as part of the work he was doing for OSAF. This was subsequently forked to create the `distribute` project (like `setuptools` itself, the `distribute` distribution installs both the `setuptools` and `pkg_resources` modules on to the target system.

The distutils project suffered from being poorly defined and documented in many ways. In particular, the phases of distribution were not well documented and the main "metadata" file used to drive the process was a full-fledged Python script. This contrasts with other packaging systems, such as RPM, where the main metadata file may *contain* executable code, but is not itself executable.

setuptools took that already complicated system, and then layered *more* complications on top (up to and including monkey-patching the standard library distutils pacakge when imported). This limited the adoption of setuptools to those users that *really* needed the features it provided.

Many other parts of the Python community didn't see the necessity, and instead rejected setuptools as an opaque blob of magic that they didn't want anywhere near their systems. setuptools has also suffered PR problems due to its close association with `easy_install`, the default behaviour of which violated many users and system administrators assumptions about how a language specific packaging tool should behave.

The misbehaviour of `easy_install` also gave the associated "egg" binary format a poor reputation that it really didn't deserve (although that format does have some genuine problems, such as being difficult to transform into platform specific binary formats, such as RPM, in a way that complies with typical packaging policies for those platforms, as well as failing to adequately convey compatibility limitations in the egg filenames. Both of these deficiencies are addressed at least to some degree by the recently approved `wheel` format).

The setuptools project also inherited many of the distutils documentation problems, although it does at least provide reasonable documentation for most of its file formats (the significant formats on that page are `requires.txt`, `entry_points.txt` and the overall egg format itself). By contrast, even today, you won't find a clear specification of the expected contents of a Python `sdist` archive.

The more recent `pip` project builds on the setuptools defined metadata and provides similar functionality to `easy_install`, but does so in a way that is far more palatable to a wider range of Python users.

The way setuptools was written also coupled it tightly to internal details of the standard library's distutils package. This coupling, along with some significant miscommunication between the setuptools and distribute developers and the core development team, had effectively frozen feature development within distutils itself for a few years, as a request to

avoid all refactoring changes in maintenance releases managed to morph into a complete ban on new distutils features for a number of releases.

The `distribute` project was created as a fork of setuptools that aims to act as a drop-in replacement for setuptools, with much clearer documentation and a broader developer base. However, this project is limited in its ability to move away from any undesirable default behaviours in setuptools, and the naming creates confusion amongst new users.

These issues led to the creation of the `distutils2` project, as a way to start migrating to an updated packaging infrastructure. As the core development team largely wasn't concerned about cross platform packaging issues, the burden of guiding the packaging improvement effort landed on a small number of heads (mostly Tarek Ziadé and Éric Araujo, and they became core developers in large part *because* they were working on packaging and the rest of us were just happy that someone else had volunteered to handle the job).

The `distutils2` developers did a lot of things right, including identifying a core issue with setuptools and easy_install, where behaviour in certain edge cases (such as attempting to interpret nonsensical version numbers) resulted in *some* kind of answer (but probably not the answer you wanted) rather than a clear error. This lead to the creation of a number of PEPs, most notably **PEP 345** (v1.2 of the metadata standard) and **PEP 386** (the versioning scheme for metadata v1.2), in an attempt to better define the expected behaviour in those edge cases. This effort was also responsible for the creation of the standard installation database format defined in **PEP 376**, which is what allows `pip`, unlike `easy_install`, to support uninstallation of previously installed distributions.

At the PyCon 2011 language summit, the decision was made to adopt distutils2 wholesale into Python 3.3 as the `packaging` package. At Éric Araujo's recommendation, that decision was reversed late in the Python 3.3 release cycle, as he felt the distutils2 code, and the PEPs it was based on simply weren't ready as the systematic fix that was needed to convince the community as a whole to migrate to the new packaging infrastructure.

In the ensuing discussion, many good points were raised. This essay started as my attempt to take a step back and *clearly define the problem that needs to be solved*. Past efforts have tried to work from a goal statement that consisted of little more than "fix Python packaging", and we can be confident that without a clearer understanding of the problems with the status quo, we aren't going to be able to devise a path forward that works for all of these groups:

- currently satisfied distutils users
- currently satisfied setuptools/distribute users
- users that are not happy with either setuptools *or* distutils

Another significant recent development is that the setuptools and distribute developers are currently working on merging the two projects back together, creating a combined setuptools distribution that includes the best aspects of both of these tools. The merger will also make it easier to make incremental changes to the default behaviour (especially of `easy_install`) without abruptly breaking anyone's tools.

Our Q & A panel at PyCon US 2013 is worth a look for interested folks.

My Position

I've been trying to ignore this problem for years. Since working at Red Hat, however, I've been having to deal with the impedance mismatch between RPM and Python packaging. As valiant as the efforts of the distutils2 folks have been, I believe their approach ultimately faltered as it attempted to tackle both a new interoperability standard between build tools and installation tools (switching from `./setup.py install` to `pysetup install project`) *at the same time* as defining a new archiving and build tool (switching from `./setup.py sdist` to `pysetup sdist project`). This created a very high barrier to adoption, as the new metadata standards were only usable after a large number of projects changed their build system. The latter never happened, and the new version of the metadata standard never saw significant uptake (as most build tools are still unable to generate it).

My view now is that it is *necessary* to take it for granted that there will be multiple build systems in use, and that `distutils`, `setuptools` and `distribute` really aren't that bad as *build* systems. Where they primarily fall down is as installation tools, through the insidious `./setup.py install` command.

That means my focus is on the developers of build tools and installation tools, to *transparently* migrate to a new metadata format, without needing to bother end users at all. Most Python developers should be able to continue to use their existing build systems, and with any luck, the only observable effect will be improved reliability and consistency of the installation experience (especially for pre-built binaries on Windows).

Some Thoughts on Asynchronous Programming

Some of the feedback I sent to Guido regarding **PEP 3156** didn't make the cut for inclusion in the PEP itself. I still consider it useful background and explanatory info, but that PEP's already going to be massive, so it makes sense that he'd prefer to keep the PEP text aimed at those that already understand the specific problems he is trying to solve.

As with all essays on these pages, feedback is welcome via the issue tracker or Twitter. If you want to comment on **PEP 3156** itself, use the python-ideas mailing list.

A Bit of Background Info

The term "Asynchronous I/O" is used to refer to two distinct, but related, concepts. The first of these concepts is an execution model for network programming, where the scalability of an I/O bound application is governed by the number of open socket connections that can be handled in a single OS process rather than by the number of concurrent OS level threads. This approach can significantly improve the scalability of an application, as most POSIX based operating systems can effectively manage thousands or tens of thousands of open socket connections without any significant tuning of process options, but only hundreds of threads (with the default size of the C stack being a key culprit - consuming the resources of an entire thread to wait for an I/O operation can waste a whole lot of memory). Disk I/O can also be scaled in this manner, but it's substantially less common to do so (since disk latency is typically orders of magnitude better than network latency).

The second of these concepts is a programming model based on explicit cooperative multi-threading where yield points are visible locally (rather than the pre-emptive multi-threading provided by OS level threads or implicit cooperative multi-threading where any function call or magic method invocation may hide a suspension point). One key goal of this explicit programming model is to change the nature of the bugs typically seen in an application. Rather than (potentially subtle) correctness bugs due to incorrect manipulation of data structures shared between preemptively scheduled threads, or inadvertently yielding control when a data structure is in a partially modified state, applications and libraries using this explicit programming model see performance bugs where an erroneous call to a synchronous API blocks the entire application. (Note that pre-emptively multi-threaded applications can still see the later kind of bug if a blocking call is made while holding a lock on a critical data structure). Another perceived benefit is that this model better matches the reality of event based programming: every event is dealt with immediately, and either translated into a response (whether that's a network message or a UI update) based on information already available locally or else into waiting for a different event (via some kind of callback API).

The key problem with this explicitly asynchronous programming model, of course, is that if an operation starts as synchronous, converting it to asynchronous requires modifying every point that calls it to yield control appropriately when necessary.

The Stackless Python project, and the greenlets library it inspired, aim to provide the first benefit, while retaining the standard synchronous programming model for application level code. This is a hugely powerful technique, as it allows the scalability benefits to be gained without needing to rewrite the entire application stack. Object-Relational-Mappers, for example, usually assume that it is OK to query or write to a database as a side effect of attribute retrieval or modification. Greenlets can often be used to implicitly turn such attribute access operations into asynchronous I/O operations, by replacing the underlying database access APIs (if the lowest layer is written in Python rather than C, it may even be possible to do so via monkey-patching), while switching to explicit asynchronous programming would require rewriting the entire ORM. Using explicit asynchronous programming also prevents entirely much of the syntactic sugar provided by ORMs, such as implicitly loading of data from the database when retrieving an attribute from an object.

PEP 3156, however, like the Twisted networking engine and the Tornado web server, is aimed at providing *both* benefits: an explicitly asynchronous programming model based on cooperative multi-threading where the suspension points are clearly marked in the individual functions and scalability is limited by the number of concurrent I/O operations supported per process rather than the number of OS level threads.

I think this quote from Guido in PEP 343 (the PEP that added the `with` statement) is also relevant to the asynchronous IO PEP:

> But the final blow came when I read Raymond Chen's rant about flow-control macros. Raymond argues convincingly that hiding flow control in macros makes your code inscrutable, and I find that his argument applies to Python as well as to C.

When writing implicitly asynchronous code, you have to assume that you may lose control of the execution at any point, since even something as innocous as retrieving an attribute from an object may suspend the thread of control. By contrast, with explicitly asynchronous code, it is safe to assume that you have sole access to shared data structures between suspension points.

While the inherent duplication between the synchronous programming model and the asynchronous programming model is unlikely to ever be eliminated, the aim of **PEP 3156** is to help reduce the unnecessary duplication between the asynchronous programming frameworks, as well as to provide improved asynchronous programming capabilities as part of the standard library, with an easier migration path to third party projects like Twisted and Tornado. This improved asynchronous infrastructure should also benefit greenlets-based synchronous-to-asynchronous adapters, as there should be a richer asynchronous ecosystem to draw from when implementing the networking side of frameworks like `gevent` (more on that below).

Furthermore, Guido's PEP aims to take full advantage of the improved support added to the language for using generators as coroutines in PEP 342 and PEP 380, as well as aligning with the API for the OS level parallel execution techniques supported by the `concurrent.futures` standard library module added in PEP 3148. I've occasionally spoken of the changes to Python's generators over the years as a way to make writing Twisted code less painful, and see the new PEP as a natural continuation of that effort.

Gevent and PEP 3156

If you look at gevent's monkey patching code, you can see that one of the key features it provides is the ability to act as a "synchronous to asynchronous adapter": taking code that assumes a synchronous blocking model and running it based on asynchronous IO instead.

From an application point of view, that's an amazing capability, and it allows some things that are impossible in an explicitly asynchronous model (such as implicitly suspending inside a magic method, which is needed for features like lazy loading of attributes from the database in an ORM).

Where even a framework like `gevent` can benefit from the transport and protocol infrastructure that will be exposed by **PEP 3156** is that, as with other projects like Twisted and Tornado, it will become easier to avoid reinventing the wheel on the *asynchronous I/O* side of things.

There will thus be 3 models for integrating asynchronous and synchronous code:

- Thread pools: **PEP 3156** will allow operations to be passed to separate threads, allowing blocking operations to be executed without suspending the main thread. This will allow explicitly asynchronous code to take advantage of existing blocking operations without blocking the main loop.
- Blocking: one of the capabilities anticipated in **PEP 3156** is the ability to effectively block on an asynchronous operation, running the event loop until the operation completes. This won't give any scalability benefits, but should allow synchronous applications to take advantage of at least some asynchronous transport and protocol implementations without needing to rewrite them as synchronous operations.
- Implicit asynchronous operations: `gevent` will be able to share elements of the IO stack with other asynchronous frameworks, while still allowing *gevent*' users to write apparently synchronous code.

Using Special Methods in Explicitly Asynchronous Code

One challenge that arises when writing explicitly asynchronous code is how to compose it with other elements of Python syntax like operators, for loops and with statements. The key to doing this effectively is the same as that adopted when designing the `concurrent.futures.as_completed()` iterator API: these other operations should always return a Future or coroutine object, even if the result of the operation happens to be available immediately. This allows the user code to consistently retrieve the result via `yield from`. The implementation of `__iter__` on Future objects and coroutines is such that they will return immediately if the result is already available, avoiding the overhead of a trip through the event loop.

Naming conventions

The examples below follow Guido's convention in NDB, where it is assumed that synchronous and asynchronous versions of operations are offered in the same namespace. The synchronous blocking versions are considered the "normal" API, and the asynchronous variants are marked with the `_async` suffix.

If an API is entirely asynchronous (as in **PEP 3156** itself) then the suffix may be dispensed with - users should assume that all operations are asynchronous. In such an API, marking any synchronous operations API with a `_sync` suffix may be desirable, but I don't know of any real world usage of that convention.

Asynchronous conditional expressions

While loops and if statements are a very simple case, as it's merely a matter of using an asynchronous expression in place of the normal boolean query:

```
while (yield from check_async()):
    # check_async() always returns a Future or coroutine
    # The loop will suspend if necessary when evaluating the condition
```

Asynchronous Iterators

Asynchronous iterators work by producing Futures or coroutines at each step. These are then waited for explicitly in the body of the loop:

```
for f in iterator_async():
    # Each iteration step always returns a Future or coroutine immediately
    # Retrieving the result is then flagged as a possible suspension point
    x = yield from f
```

For example, this approach is useful when executing multiple operations in parallel, and you want to process the individual results as they become available:

```
for f in as_completed(operations):
    result = yield from f
    # Process the result
```

This is very similar to the way the existing concurrent.futures module operates, with the `f.result()` call replaced by the explicit suspension point `yield from f`.

Asynchronous Context Managers

Asynchronous context managers are able to cope with blocking operations on entry to a with statement by implementing them as a Future or coroutine that produces a context manager as its result. The __enter__ and __exit__ methods on this context manager must themselves be non-blocking:

```
with (yield from cm_async) as x:
    # The potentially blocking operation happens in cm_async.__iter__
    # The __enter__ and __exit__ methods on the result cannot
    # suspend execution
```

Alternatively, a Future or coroutine may be returned from __enter__, similar to the usage of asynchronous iterators:

```
with cm_async as f:
    # The potentially blocking operation happens in f.__iter__
    x = yield from f
    # The __exit__ method on the CM still cannot suspend execution
```

For example, either of these models may be used to implement an "asynchronous lock" that is used to control shared access to a data structure even across operations which require handing control back to the event loop.

However, it is not currently possible to handle operations (such as database transactions) that may need to suspend execution in the __exit__ method. In such cases, it is necessary to either adopt a synchronous-to-asynchronous adapter framework (such as gevent) or else revert to the explicit try statement form:

```
x = yield_from cm.enter_async()
try:
    ...
except Exception as ex:
    cm.handle_error_async(ex)
else:
    cm.handle_success_async()
```

Asynchronous Operators

The approach described above generalises to other operators, such as addition or attribute access: rather than returning a result directly, an API may be defined as returning a Future or coroutine, to be turned into a concrete result with `yield from`:

```
add_async = objA + objB
add_result = yield from add_async
```

In practice, it is likely to be clearer to use separate methods for potentially asynchronous operations, making it obvious through naming conventions (such as the `_async` suffix) that the operations return a Future or coroutine rather than producing the result directly. Synchronous-to-asynchronous adapters also have a role to play here in allowing code that relies heavily on operator overloading to interact cleanly with asynchronous libraries.

Additional Asynchronous Syntax

The `yield` and `yield from` keywords apply directly to the subsequent expression. As noted above, this means they can't easily be used to affect the operation of magic methods invoked implicitly as part of other syntax. Accordingly, that you means you can't have an asynchronous context manager that needs to suspend in `__exit__`, nor can you easily write an asynchronous comprehension.

However, a new keyword should allow certain subexpressions in `for` loops, `with` statements and comprehensions to be flagged for invocation as `yield from expr` rather than using the result of the expression directly.

Using `yielding` as the example keyword, usage would look like the following:

```
for x in yielding async_iterable:
    # Current generator may be suspended each time __next__ is called.
    # Semantics are exactly the same as the workaround noted above:
    # the __next__ is expected to return a Future or coroutine, which
    # the interpreter will then invoke with "yield from". The difference
    # is that in this case the invocation is implied by the "yielding"
    # keyword, avoid the need to spell out the temporary varibale

# Similarly, in the following comprehensions, the current generator
# may be suspended as each value is retrieved from the iterable
x = [x for x in yielding async_iterable]
y = {x for x in yielding async_iterable}
z = {k:v for k, v in yielding async_iterable}

with yielding async_cm as x:
    # Current generator may be suspended when __enter__ and __exit__
    # are called. As with _iter__, for __enter__, the semantics are the
    # same as in the workaround noted above, except that the temporary
    # variable is hidden in the interpreter.
    # Unlike the workaround, the dedicated syntax means __exit__ is
    # also invoked asynchronously, so it can be used to implement
    # asynchronous database transactions.
```

Dubious Ideas

This section is for ideas that would require changes to the Python language definition. All such proposals are deemed dubious by default - ideas that would make the core Python language definition better are vastly outnumbered by those that would make it worse :)

Suite Expressions

python-ideas: http://mail.python.org/pipermail/python-ideas/2011-December/013003.html

This idea had an unusual start to life as a "devil's advocate" style response to a long, rambling trollerific post to python-dev.

It ended up being a reasonably coherent write-up of a "delimited suites for Python" proposal that didn't make me run screaming in horror, and several other people had similar reactions, so I decided to include a cleaned up version of it here.

If anyone does decide to take this idea and run with it, try running `from __future__ import braces` at the interactive prompt to get some idea of how big a can of worms you're going to be opening (however, also remember that Guido's views on the topic have moderated somewhat over the years)

Background

Python's whitespace based delineation of suites is one of its greatest strengths. It aligns what the human reader perceives with what the computer is actually executing, reducing the frequency of semantic errors due to mismatches between the use of separate block delimiters and the human readable indentation.

However, this benefit comes at quite a high price: it is effectively impossible to embed arbitrary Python statements into any environment where leading whitespace is *not* significant, including Python's own expression syntax.

It can be argued that this restriction has led directly to the introduction of "expression friendly" variants of several Python top level constructs (for example, lambda expressions, conditional expressions and as a contributing factor in creating the various forms of comprehension).

It is also one of the reasons Python-based templating languages almost always create their own custom syntax - embedding Python's own whitespace sensitive statement syntax into environments where leading whitespace is either ignored or forms a significant part of the template output is a formidable challenge.

In other languages, this kind of issue is handled by using explicit suite and statement delimiters (often braces and semi-colons, respectively) to allow full suites to be used as expressions.

Rationale

To elaborate on the points made in the Background section, the reason significant leading whitespace can be problematic is due to two main circumstances:

1. Attempting to transport it through a channel that either strips leading and trailing whitespace from lines, or else consolidates certain whitespace sequences into a single whitespace character (generally sequences of spaces and tabs becoming a single space). Python source code simply cannot be passed through such channels correctly - if they don't offer an escaping mechanism, or that mechanism is not applied correctly, the code *will* be corrupted. Explicitly delimited code, on the other hand, can be passed through without semantic alteration (even if the details of the whitespace change) and a pretty printer can fix it at the far end.
2. Attempting to transport it through a channel where leading whitespace already has another meaning. This comes up primarily with templating languages - your whitespace is generally part of the template output, so it becomes problematic to break up your Python code across multiple code snippets while keeping the suite groupings clear. With explicit delimiters, though, you can just ignore the template parts, and pretend the code snippets are all part of a single string.

A delimited syntax allows definition of a standard way to pass Python source code through such channels. Since Python expression syntax is one such medium, this then leads immediately to the possibility of supporting multi-line lambdas (amongst other things).

Proposal

While Python uses braces for another purpose (dictionary and set definitions), it is already the case that semi-colons (`;`) can be used as statement terminators, both optionally at the end of any simple statement, and also to combine multiple simple statements into a single larger statement (e.g. `x += y; print(x)`).

It seems that this existing feature could be combined with a brace-based notation to create an unambiguous "suite expression" syntax that would enjoy the same semantics as ordinary Python suites (i.e. doesn't create a new scope, doesn't directly affect control flow), but allows *all* Python statements to be embedded inside expressions.

Currently, the character sequence `{:` is a Syntax Error: you are attempting to end a compound statement header line while an opening brace remains unmatched, or else trying to build a dictionary without specifying the key value. This creates an opportunity to re-use braces for a suite expression syntax without conflicting with their use for set and dictionary construction.

Specifically, it should be possible to create a variant of the top-level Python syntax that:

1. Explicitly delimits suites using the notation `{:` to open the suite and `:}` to end it
2. Requires the use of `;` to separate simple statements (i.e. newline characters would not end a statement, since we would be inside an expression)

3. Allows compound statements to be used as simple statements by requiring that all subordinate suites also be suite expressions (i.e. leading whitespace would not be significant, since we would be using subexpressions to define each suite)

This would be sufficient to have a version of Python's syntax that is both compatible with the existing syntax and could be embedded in whitespace-insensitive contexts without encountering problems with suite delineation. However, with one additional change, this new format could also be used to define "suite expressions" that could be used meaningfully anywhere Python currently accepts an expression:

4. Uses the value of the last statement executed in the suite as the result of the overall suite expression (since return statements would affect the containing scope)

This would finally allow the oft-requested "multi-line lambdas", since the body of the lambda could now be a suite expression.

Examples

Raise expressions:

```
x = y if y is not None else {: raise ValueError("y must not be None!") :}
```

Try expressions:

```
x = {: try {: y.hello} except AttributeError {: "world!"} :}
```

With expressions:

```
data = {: with open(fname) as f {: f.read() :} :}
```

Embedded assignments:

```
if {: m = pat.search(data); m is not None :}:
    # do something with m
else:
    # No match!
```

In-order conditional expressions:

```
x = {: if a {:b:} else {:c:} :}
```

One-line accumulator function:

```
def acc(n=0): return lambda (i) {: nonlocal n; n += i; n :}
```

A Python-based templating engine ([*1*]):

```
<% if danger_level > 3 {: %>

<div class="alert">
  <% if danger_level == 5 {: %>EXTREME <% :} %>DANGER ALERT!
</div>

<% :} elif danger_level > 0 {: %>

<div>Some chance of danger</div>

<% :} else {: %>
```

```
<div>No danger</div>

<% :} %>

<% for a in ['cat', 'dog', 'rabbit'] {: %>

<h2><%= a %></h2>
<p><%= describe_animal(a) %></p>

<% :} %>
```

Implicit Context Managers

Boredom & Laziness: http://www.boredomandlaziness.org/2011/01/some-goals-for-python-33.html

Basic concept is to add an optional `__cm__` method that relates to context managers the way `__iter__` relates to iterators.

I still don't have a pithy name like "iterable" for "objects with an implicit context manager", but the passage of time has at least cemented "context manager" as referring specifically to objects with `__enter__` and `__exit__` methods.

AST Metaprogramming

python-ideas: http://mail.python.org/pipermail/python-ideas/2011-April/009765.html

Past Ideas

These are archived ideas that were either elaborated into full PEPs, or which I now actively consider to be a bad idea:

Skipping the With Statement Body

Boredom & Laziness: http://www.boredomandlaziness.org/2011/01/some-goals-for-python-33.html

(Note: I'm no longer a fan of this idea. contextlib.ExitStack should provide a way to achieve some of the same benefits programmatically in `__enter__` without needing to change the context management protocol, at least as far as resource cleanup goes. It should also make it easier to create context managers that can be combined with an if statement to only execute if the resource is acquired successfully)

Basic concept was to add an optional `__entered__` method to the context management protocol that gets executed *inside* the scope of the try block, so any exceptions it raises are seen by the context managers `__exit__` method.

Standard Library Preview Namespace

python-ideas: http://mail.python.org/pipermail/python-ideas/2011-August/011317.html

This isn't originally my idea (that honor goes to Dj Gilcrease), it's just something I think would be really valuable and that I have some definite ideas about.

Guido's Verdict

Eli Bendersky ran with this idea and wrote it up as PEP 408. As you can see if you click on that link, Guido rejected the PEP in favour of slightly relaxing the rules for stdlib inclusion: if we're not 100% sure of an addition, even after it has been battle-tested and received widespread approval on PyPI, we now have the option to add it anyway, with a documented warning that the inclusion of the library is provisional. It will remain at least for that version, but there's a slim chance it will be removed, or experience backwards incompatible API tweaks in the next feature release. (The most likely outcome, however, is that it will remain unchanged for the next release and simply lose the "provisional" tag)

Guido's chosen way forward is actually very similar to the way Google now provides experimental modules on App Engine under their final names with an "Experimental!" tag on their documentation (after an earlier failed experiment with a dedicated "labs" namespace). It's also virtually identical to the way that Red Hat provides some product features as tech previews (i.e. without being covered by the usual support guarantees).

In all cases the intended plan is that the incorporated module or feature will, in fact, end up satisfying the normal backwards compatibility guidelines. The documented warning just gives us an out where we have the *option* of making a backwards incompatible change if we deemed it necessary.

The new policy was written up as PEP 411.

The Namespace

Red Hat have a concept of "tech previews" that they use for features that they don't believe are fully mature yet, but that they want to provide to customers in order to gather early feedback.

I think `__preview__` would be a great term for this namespace.

Why not __future__?

Python already has a "forward-looking" namespace in the form of the `__future__` module, so it's reasonable to ask why that can't be re-used for this new purpose.

There are two reasons why doing so not appropriate:

1. The `__future__` module is actually linked to a separate compiler directives feature that can actually *change* the way the Python interpreter compiles a module. We don't want that for the preview namespace - we just want an ordinary Python package.

2. The `__future__` module comes with an express promise that names will be maintained in perpetuity, long after the associated features have become the compiler's default behaviour. Again, this is precisely the *opposite* of what is intended for the preview namespace - it is almost certain that all names added to the preview will be removed at some point, most likely due to their being moved to a permanent home in the standard library, but also potentially due to their being reverted to third party package status (if community feedback suggests the proposed addition is irredeemably broken).

The Benefits for python-dev

Currently, we're *really* reluctant to add new interfaces to the standard library. This is because as soon as they're published in a release, API design mistakes get locked in due to backwards compatibility concerns.

By gating all major API additions through the preview namespace for at least one release, we get one full release cycle of community feedback before we lock in the APIs with our standard backwards compatibility guarantee.

This is similar to the way the `sets` module was used to gather broad API feedback before the final API of the `set` and `frozenset` builtins was determined.

We can also start integrating preview modules in with the rest of the standard library early, so long as we make it clear to packagers that the preview modules should *not* be considered optional. The only difference between preview APIs and the rest of the standard library is that preview APIs are explicitly exempted from the usual backwards compatibility guarantees)

The Benefits for End Users

For future end users, the broadest benefit lies in a better "out-of-the-box" experience - rather than being told "oh, the standard library tools for task X are horrible, download this 3rd party library instead", those superior tools are more likely to be just be an import away.

For environments where developers are required to conduct due diligence on their upstream dependencies (severely harming the cost-effectiveness of, or even ruling out entirely, much of the material on PyPI), the key benefit lies in ensuring that anything in the preview namespace is clearly under python-dev's aegis from at least the following perspectives:

- licensing (i.e. redistributed by the PSF under a Contributor Licensing Agreement)
- testing (i.e. the module test suites are run on the python.org buildbot fleet and results published via http://www.python.org/dev/buildbot)
- issue management (i.e. bugs and feature requests are handled on http://bugs.python.org)
- source control (i.e. the master repository for the software is published on http://hg.python.org)

Those are the things that should allow the preview modules to be used under any existing legal approvals that allow the use of Python itself (e.g. in a corporate or governmental environment).

The Rules

New modules added to the standard library spend at least one release in the `__preview__` namespace (unless they are using a largely pre-defined API, such as the new `lzma` module, which generally follows the API of the existing `bz2` module).

API updates to existing modules may also be passed through this namespace at the developer's discretion. In such cases, the module in the preview namespace should use `from original import *` so that users never need to include both versions.

Adding this preview namespace doesn't mean that the floodgates suddenly open for the addition of arbitrary modules and packages to the standard library. All of the existing criteria regarding "best of breed" projects, sufficient API stability and general project maturity to cope with an 18 month release cycle, etc would still apply. Also, as Ethan Furman once put it, the standard library philosophy is "batteries included", not "nuclear reactors included". Some projects are simply too big and too complicated to become part of the standard library - in such cases, it is better for the standard library to define standard interfaces that allow third party projects to interoperate effectively, rather than trying to do everything itself (e.g. `wsgiref`, `memoryview()`).

All the preview namespace is intended to do is lower the risk of locking in minor API design mistakes for extended periods of time. Currently, this concern can block new additions, even when the python-dev consensus it that a particular addition is a good idea in principle.

The Candidates

For Python 3.3, there are a number of clear current candidates:

- regex
- daemon (PEP 3143)

- ipaddr (PEP 3144)

Other possible future use cases include such things as:

- improved HTTP modules (e.g. requests)
- HTML 5 parsing support (e.g. html5lib)
- improved URL/URI/IRI parsing
- a standard image API (PEP 368)
- encapsulation of the import state (PEP 368)
- standard event loop API (PEP 3153)
- a binary version of WSGI for Python 3 (e.g. PEP 444)
- generic function support (e.g. simplegeneric)

Feature Release Cadence

This will be a PEP proposing a simpler alternative to :pep‘407‘ and **PEP 413** that merely releases alpha versions of the upcoming feature release early.

The meaning of an “alpha” release will be clarified to say that all stability guidelines for *existing* APIs remain in force, but all APIs that have not yet been part of a release must be considered provisional, as defined in **PEP 411**.

Type Neutral Codec API

(Note: this article was substantially rewritten after some initial feedback from Armin Ronacher. As always, old versions are available on BitBucket)

One of the complaints with Python 3 is that it broke the old idiom for many text-to-text and binary-to-binary transforms: the `encode()` and `decode()` methods of 8-bit and Unicode string objects.

In Python 2, these methods were fairly thin shells around the type-neutral `codecs` module. Both 8-bit and Unicode strings had both methods and the type of the return value was based on the specific encoding passed in.

In Python 3, these convenience methods have instead been incorporated directly into the text model of the language. Text strings only have an `encode()` method, and that method can only be used with codecs that produce bytes objects. Similarly bytes and bytearray objects only have a `decode()` method which can only be used with codecs that produce string objects.

For example (Python 2.7):

```
>>> x = u'Hello World!'.encode("rot-13").encode("koi8-r").encode("bz2")
>>> x
'BZh91AY&SY]\xc2\xf0\xb7\x00\x00\x01\x97\x80`\x00\x00\x10\x02\x00\x12\x000 ␣
↪\x001\x06LA\x06\x98\x9a\x166$\x1et\xf1w$S\x85\t\x05\xdc/\x0bp'
>>> x.decode("bz2").decode("koi8-r").decode("rot-13")
u'Hello World!'
```

If you try the first or last step of that chain in Python 3, it fails:

```
>>> x = "Hello World!".encode("rot-13").encode("koi8-r").encode("bz2")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: encoder did not return a bytes object (type=str)
>>> x = "Hello World!".encode("koi8-r").encode("bz2")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'encode'
```

This means the old text-to-text and binary-to-binary transforms can only be accessed via the type neutral `codecs` module APIs. To make matters even more annoying, the shorthand aliases for most of those codecs are still missing, even though the codecs themselves were restored for Python 3.2.

There is a suggestion that this be replaced directly with a similarly method-based `transform`/`untransform` API, but I'm now convinced that's a bad idea being considered only due to the precedent set by Python 2. Instead, I believe it makes more sense to take a step back and consider a fully type-neutral solution, just like the `codecs` module itself.

The simple alternative I plan to propose is introducing a pair of top level functions in the codecs module that are type neutral alternatives to the type restricted str and bytes convenience functions. The semantics would be equivalent to these pure Python versions:

```
def encode(input, encoding, errors='strict'):
    encoder = getencoder(encoding)
    result, len_consumed = encoder(input, errors)
    if len_consumed < len(input):
        ... # Copy str.encode behaviour for this case
    return result

def decoder(input, encoding, errors='strict'):
    decoder = getdecoder(encoding)
    result, len_consumed = decoder(input, errors)
    if len_consumed < len(input):
        ... # Copy bytes.decode behaviour for this case
    return result
```

Getting Cute with Codec Pipelines

Armin assures me the following example isn't all that useful in practice, but it was a fun exercise in exploring what is possible when working directly with the codecs API.

Below is a sketch of a simple `CodecPipeline` that works on both Python 2 and Python 3. It accepts an arbitrary number of codec names as positional parameters, as well as the error handling scheme as a keyword-only parameter:

```
import codecs
class CodecPipeline(object):
    """Chains multiple codecs into a single encode/decode operation"""
    def __init__(self, *names, **kwds):
        self.default_errors = self._bind_kwds(**kwds)
        encoders = []
        decoders = []
        self.codecs = names
        for name in names:
            info = self._lookup_codec(name)
            encoders.append(info.encode)
            decoders.append(info.decode)
        self.encoders = encoders
        decoders.reverse()
        self.decoders = decoders

    def _bind_kwds(self, errors=None):
        if errors is None:
            errors = "strict"
```

```
        return errors

    def _lookup_codec(self, name):
        # Work around for http://bugs.python.org/issue15331 in 3.x
        try:
            return codecs.lookup(name)
        except LookupError:
            return codecs.lookup(name + "_codec")

    def __repr__(self):
        names = self.codecs
        errors = self.default_errors
        if not names:
            return "{}(errors={!r})".format(type(self).__name__, errors)
        return "{}({}, errors={!r})".format(type(self).__name__,
                                            ", ".join(map(repr, names)),
                                            errors)

    def encode(self, input, errors=None):
        """Apply all encoding operations in the pipeline"""
        if errors is None:
            errors = self.default_errors
        result = input
        for encode in self.encoders:
            result, __ = encode(result, errors)
        return result

    def decode(self, input, errors=None):
        """Apply all decoding operations in the pipeline"""
        if errors is None:
            errors = self.default_errors
        result = input
        for decode in self.decoders:
            result,__ = decode(result, errors)
        return result
```

And using it in Python 2 looks like this:

```
>>> cp = CodecPipeline("rot-13", "koi8-r", "bz2")
>>> cp
CodecPipeline('rot-13', 'koi8-r', 'bz2', errors='strict')
>>> cp.encode(u'Hello World!')
'BZh91AY&SY]\xc2\xf0\xb7\x00\x00\x01\x97\x80`\x00\x00\x10\x02\x00\x12\x000 ␣
↪\x001\x06LA\x06\x98\x9a\x166$\x1et\xf1w$S\x85\t\x05\xdc/\x0bp'
>>> cp.decode(cp.encode(u'Hello World!'))
u'Hello World!'
```

Python 3 looks almost identical, aside from the lack of the `u` prefix on the string literals (and, in Python 3.3, such prefixes are once again legal on the input front).

```
>>> cp = CodecPipeline.from_chain("rot-13", "koi8-r", "bz2")
>>> cp
CodecPipeline('rot-13', 'koi8-r', 'bz2', errors='strict')
>>> cp.encode('Hello World!')
'BZh91AY&SY]\xc2\xf0\xb7\x00\x00\x01\x97\x80`\x00\x00\x10\x02\x00\x12\x000 ␣
↪\x001\x06LA\x06\x98\x9a\x166$\x1et\xf1w$S\x85\t\x05\xdc/\x0bp'
>>> cp.decode(cp.encode(u'Hello World!'))
'Hello World!'
```

String Views

python-ideas: http://mail.python.org/pipermail/python-ideas/2011-December/012993.html

CHAPTER 4

Using the Python Kerberos Module

I'm currently integrating Kerberos authentication support into a custom Pulp client and have completely failed to find any *good* documentation on how to use the kerberos module.

I managed to find a basic example, which makes reference to "another example in the python-kerberos package", which I assume is a reference to the final test case in the package. I also looked at the XML-RPC wrapper implemented in kobo.

Rather than just documenting this for my own use, I decided to write up and publish what I figured out. Besides, it gives me an excuse to try out Kenneth Reitz's famous requests module :)

Note: After I originally wrote this article, Kenneth accepted a pull request that added Kerberos authentication support directly to `requests`. With the refactored 1.0 release, that support has been moved out to a separate requests-kerberos project.

All examples in this document are from a Python 2 interactive session. As far as I know, there is not yet a Python 3 compatible version of the kerberos module available.

Kerberos Basics

When setting up Kerberos authentication on a server, there are two basic modes of operation. The simplest from a client implementation point of view just uses Basic Auth to pass a username and password to the server, which then checks them with the Kerberos realm. That's not the case I'm interested in, since it just looks like ordinary Basic Auth from the client side.

The case I am interested in is the one where the client has a preexisting Kerberos ticket and we want to pass *that* to the server automatically without the user needing to reenter their password. The relevant HTTP authorization protocol is called "Negotiate".

The basic flow of a typical Kerberos authentication is as follows:

- Client sends an unauthenticated request to the server

- Server sends back a 401 response with a `WWW-Authenticate: Negotiate` header with no authentication details
- Client sends a new request with an `Authorization: Negotiate` header
- Server checks the `Authorization` header against the Kerberos infrastructure and either allows or denies access accordingly. If access is allowed, it *should* include a `WWW-Authenticate: Negotiate` header with authentication details in the reply.
- Client checks the authentication details in the reply to ensure that the request came from the server

This article doesn't cover server side authentication, as I just use mod_auth_kerb to handle that side of things and set up the application to accept the `REMOTE_USER` setting from Apache. One useful undocumented trick that `mod_auth_kerb` supports is the KrbLocalUserMapping option (which strips the realm details from the value stored in `REMOTE_USER`).

The Role of the Python Kerberos Module

From a client point of view, the kerberos module handles two tasks:

- Figuring out the value to send in the `Authorization` field
- Checking Kerberos level authentication of the response provided by the server

The kerberos module does this by exposing the GSS API - this is an ugly interface, but it *does* work.

The Initial Request and Response

This part doesn't involve the kerberos module at all, just a basic HTTP request:

```
>>> import requests
>>> r = requests.get("https://krbhost.example.com/krb")
>>> r.status_code
401
>>> r.headers["www-authenticate"]
'Negotiate, Basic realm="Example Realm"'
```

This example uses a fictional host and realm. This fictional host accepts either Negotiate (i.e. Kerberos tickets) or direct username/password authentication.

As the same header occurs multiple times in the response, `requests` reports it as a comma separated list. This isn't very convenient, so we'll write a helper to split out the auth headers more cleanly:

```
>>> def www_auth(response):
...     auth_fields = {}
...     for field in response.headers.get("www-authenticate", "").split(","):
...         kind, __, details = field.strip().partition(" ")
...         auth_fields[kind.lower()] = details.strip()
...     return auth_fields
...
>>> www_auth(r)
{'negotiate': '', 'basic': 'realm="GlobalSync: Kerberos Login"'}
```

That means we can now easily detect when the client should reply with a Kerberos authenticated connection. For example, a host may provide two entry points, one configured to use `mod_auth_kerb` for preauthentication of users, while the other handles authentication entirely at the application level:

```
>>> r = requests.get("https://krbhost.example.com/krb/")
>>> r.status_code == 401 and www_auth(r).get('negotiate') == ''
True
>>> r = requests.get("https://krbhost.example.com/api/")
>>> r.status_code == 401 and www_auth(r).get('negotiate') == ''
False
```

If we accessed the `"https://krbhost.example.com/krb/"` URL with a web browser, it would forward the Kerberos ticket if available (and the browser is configured to do so), otherwise it would pop up a password dialog, using the realm info from the `WWW-Authenticate: Basic` header as the dialog title (at least, that's what Firefox does - I assume other browsers are similar)

The Kerberos Authenticated Request

Now we know we want to send a Kerberos authenticated request to the server, the `kerberos` module comes into play. While this is a very thin wrapper around a C API, it *does* at least turn failures into exceptions (rather than setting the return code) so we'll ignore that value:

```
>>> __, krb_context = kerberos.authGSSClientInit("HTTP@krbhost.example.com")
>>> kerberos.authGSSClientStep(krb_context, "")
0
>>> negotiate_details = kerberos.authGSSClientResponse(krb_context)
>>> headers = {"Authorization": "Negotiate " + negotiate_details}
>>> r = requests.get("https://krbhost.example.com/krb/", headers=headers)
>>> r.status_code
200
>>> r.json
["example_data"]
```

You *can* set additional GSS flags in the call to `authGSSClientInit` but I haven't found any need to for simple client authentication via Kerberos.

Authenticating the reply from the server

While we can just trust SSL to ensure the integrity of the response from the server, we can also complete the Kerberos handshake and use it to further authenticate the reply from the server:

```
>>> kerberos.authGSSClientStep(krb_context, www_auth(r)["negotiate"])
1
>>> kerberos.authGSSClientClean(krb_context)
1
```

As with other calls, these should throw an exception if they fail, so even though the return code is passed through from C, it should never be anything other than 1 at the Python level.

Wrapping this up in a helper class

Here's a simple class that can help make this a bit easier to use in a client without making any assumptions about the HTTP interface being used:

```
class KerberosTicket:
    def __init__(self, service):
        __, krb_context = kerberos.authGSSClientInit(service)
        kerberos.authGSSClientStep(krb_context, "")
        self._krb_context = krb_context
        self.auth_header = ("Negotiate " +
                            kerberos.authGSSClientResponse(krb_context))
    def verify_response(self, auth_header):
        # Handle comma-separated lists of authentication fields
        for field in auth_header.split(","):
            kind, __, details = field.strip().partition(" ")
            if kind.lower() == "negotiate":
                auth_details = details.strip()
                break
        else:
            raise ValueError("Negotiate not found in %s" % auth_header)
        # Finish the Kerberos handshake
        krb_context = self._krb_context
        if krb_context is None:
            raise RuntimeError("Ticket already used for verification")
        self._krb_context = None
        kerberos.authGSSClientStep(krb_context, auth_details)
        kerberos.authGSSClientClean(krb_context)
```

And an example of using it with `requests`:

```
>>> krb = KerberosTicket("HTTP@krbhost.example.com")
>>> headers = {"Authorization": krb.auth_header}
>>> r = requests.get("https://krbhost.example.com/krb/", headers=headers)
>>> r.status_code
200
>>> krb.verify_response(r.headers["www-authenticate"])
>>>
```

CHAPTER 5

General Linux Notes

I know, I know, that URL up there says *Python* notes. This is just a convenient way to post some more general Linux articles. If you want at least a tenuous connection, Linux is the platform I personally favour for software development, and it's what I use when I'm hacking on CPython :)

Rebuilding a Fedora system on new hardware

Note: this was written for Fedora 17 in July 2012. Who knows what future versions will bring in terms of system migration utilities...

For a while I've been running my ASUS Zenbook as a combined gaming/development laptop by running Windows as the base OS and Fedora under VMWare workstation.

I realised recently that I didn't really care about using it for gaming any more, so I decided to simplify things and just run Fedora directly on bare metal (that decision was simplified by the fact that the Zenbooks have been out for a while now, and the open source driver situation is much improved from what it was at their original release).

Since the "Oh, I forgot to reinstall X" process is rather annoying, I decided to find a way to relatively quickly build up the reinstalled system. It turns out there are some rather useful commands in the `yum-utils` package that can be abused for this purpose. It *is* messing around with debug commands though, so no guarantees that this won't result in a broken target system (however, the whole point is to do this when that target system is completely new, so no great loss if it ends up needing to be reinstalled).

Step 1 was moving the VM to my desktop machine. If this had been an actual reinstall rather than a virtual to physical migration I would have instead just copied the various files described later out to external storage as archive files, and then unpacked those archives after doing the reinstallation. As it is, `scp` and `rsync` are my preferred tools.

Step 2 was reinstalling Fedora directly on the Zenbook. Nothing special there - I used the Fedora 16 live USB I still had from the original install, then used `preupgrade` to bring the system up to Fedora 17. (I considered this easier than relearning the live USB creation process, but creating a live USB isn't particularly difficult)

Step 3 was to temporarily allow SSH access to the notebook:

```
$ sudo service sshd start
$ sudo iptables -I INPUT -p tcp --dport 22 -j ACCEPT
```

I then checked this from the VM running on the destop:

```
$ ssh <zenbook>
```

Step 4 was to ensure the current system was fully up to date and create a file describing the full package state (for anyone else following this, don't forget to install the `yum-utils` package if it isn't already installed):

```
$ sudo yum update -y
$ yum-debug-dump
```

Step 5 was to ensure the repo config on the bare metal install matched that on the VM:

```
$ scp /etc/yum.repos.d/* root@<zenbook>:/etc/yum.repos.d
$ scp /etc/pki/rpm-gpg/* root@<zenbook>:/etc/pki/rpm-gpg
```

Step 6 duplicated my entire home directory, including the dump of the package state created in Step 4 above:

```
$ rsync -avz ~/ <zenbook>:~
```

Step 7 was to do a quick sanity check that everything was copied over correctly by running the following three commands that check disk usage on both machines (the first checks the total usage in your home directory, the second, the normally visible files and directories, and the last, the hidden files and directories with a bit of trickery to avoid attempting to recurse into the parent directory reference stored at `..`):

```
$ du -hs ~
$ du -hs ~/*
$ du -hs ~/.[^.]*
```

Step 8 was to restore the package state from the old system on the new one:

```
$ sudo yum-debug-restore <dump file>
```

I could probably have just restarted the X server at this point, but I figured I may as well just restart the whole machine :)

I expect I'll still need a few more tweaks, as I come across any additional configuration tweaks needed under `/etc/`, as well reinstallation of any components that weren't installed using `yum` (I don't recall putting any packages into the system Python with `pip`, but it's always possible I have done so and simply forgotten about it.

CHAPTER 6

Python Conferences

Inspired by Danny Greenfield, I've started keeping conference notes in reStructured text on Bitbucket.

PyCon Australia 2012

Hobart Tasmania, August 18-19

What to Build and How to Build It

Mark Ramm: Answering questions with Python

Previous SourceForge technical lead TurboGears 2 technical lead Current Juju technical lead

Don't waste your life

- 9 out of 10 projects fail
- 9 out of 10 startups fail
- Sturgeon's Law!

Building products - failing for technical reasons or poor execution - failing because it wasn't actually useful

Try to build things people want!

Market Research vs Total Bullshit

Not all product managers bother to do research

Even when they do, results can be questionable

Management culled at SourceForge

XKCD: Stand back, I'm going to try: SCIENCE!

Test Driven Management

In dev, tests provide measurable progress

Can you apply the same idea to management?

3 screenshots from SF download page - updated with large download button - redesigned (ended up slightly worse) - current design is actually measured

First two redesigns had no metrics. Final redesign had metrics to track how often people made it through to get the files they wanted.

Scientific Management

Document assumptions Work with the team to learn what works Don't just make shit up!

Scientific method - ask a question - do background research - construct a hypothesis - test hypothesis through experiment (or, observation if necessary) - analyse results and potentially reformulate hypothesis - repeat!

Science requires that your hypothesis is falsifiable. It isn't about proving we're right, it's about proving we're "Not Wrong Yet"

(Popper quote) The role of evidence is to:

Arguing based on opinions can easily go nowhere. Arguing based on objective data can be

Metrics for Pirates

Dave McClure

(these are for a public facing website)

Acquisition: users reaching your site Activation: getting visitors to go further Retention: getting vistitors to come back Referral: getting visitors to recommend the site Revenue: getting visitors to pay you!

Tools - how do you gather data? Tests - *what* data do you measure?

Examples

Zappo's: selling shoes on the internet! Took photos of shoes in stores Set up the online store Bought at retail and shipped them Low overhead way to test if people would buy shoes online Return policy dealt with "What if the shoes don't fit?" Figured all this out *before* expending a lot of capital

Pets.com: spent the money first, it didn't work!

Building a great product

Traditional: - Great idea - build it - market it - profit!

Not so much: this often fails

The Structure of Scientific Revolutions

Normal progress (refining the existing models) vs revolution (rebuilding the models on a new foundation)

Optimisation (doing an existing thing better) vs Discovery (building something new)

Customer development

Searching for a busines - discover customers - validate customers (will/can they pay?)

Growing a business - create customers - scale company

Desirable, feasible, viable

- is it worth doing?
- can it be done?
- is there a business model?

Figure out which of these is the biggest risk for any idea

A "Pivot" is a change in direction during the customer discovery and validation phase

Closing

More Popper: great scientists have bold ideas, but are highly critical of their own ideas. They try to find whether their ideas are right by trying first to see whether the are, perhaps, not wrong.

Q & A

Data vs knowledge - e.g. visitor -> download conversion may go down if the search page is bad - can refine an experiment (e.g. only use referrers that are almost certainly trying to download) - balancing trade-offs (e.g. ad revenue at source forge)

Low volume testing? - how do you test when you don't have traffic yet? - initial focus needs to be on the acquisition step - need a lot of data to distinguish between 2% and 3% conversion rate - but 0% vs 1% is easier to figure out (and is where you can start)

How to explain the need for this? - jackass answer: get the hell out! - explaining the scientific method to someone that doesn't get it is hard - in the discovery phase, "it doesn't matter" is a common answer

My Thoughts

Wondering how you could apply this in an intranet context...

Web Templating Battle

Ivan Teoh (PretaWeb - government CMS provider)

Comparing 5 Python web templating systems

Intro

Web templates - separating presentation from content

5 to be reviewed:

- Django
- Chameleon

- Jinja2
- Diazo and XSLT
- Mako

Django templating

Text based

Customisation in Python code

Separate variable interpolation syntax ({{, }}) and tag syntax ({%, %})

Also have filters etc

Django-specific

Chameleon

HTML/XML based

Compiles to Python bytecode

Language is "page templates", originally from Zope

Python is the default expression langage

Used with Pyramid, Zope, Plone, Grok, standalone

Commands embedded into HTML attributes. Also allows arbitrary Python snippets in 2.0

Jinja2

text based (syntax very similar to Django templates)

compiles down to Python bytecode

Diazo and XSLT

XML rules files

Combines web designer template with developer backend

Uses CSS classes and XSLT to drive the transforms

Various directives identify the pieces (e.g. identifying the theme file)

No looping constructs, just use XSLT

From Plone 4.2 (this came up in the Q&A)

Mako

Embeds Python directly in the template (i.e. Python server pages)

Uses <%, %> to identify code blocks

Summary

No real battle, no real winner

Driven by different use cases & history

Q & A

5 chosen are under active development

Diazo seemed very complicated relative to others. Designed to support a drag-and-drop theme designer for use by CMS clients. Still a little complicated to use from the developer side to set it up.

My Thoughts

Reasonable overview, but hard to get into any detail in 25 minutes.

Would have been nice to have explicit pros/cons for each one.

The Lazy Dev's Guide to Testing Your Web API

Ryan Kelly

Love/Hate Testing?

Quality assurance = diminishing returns

Early testing efforts rewarded with great improvements

Tests get harder, improvements get more obscure

Process improvement!

Better testing tools to do more with less effort

Invest in laziness!

Be pro-actively lazy! (And I clearly need to re-read Moving Pictures!)

What does Ryan know?

- Mozilla web services (Firefox Sync Server)
- Tools (WebTest, WSGIProxy, FunkLoad)
- enthusiast, not yet expert

WSGI

- webtest = WSGI for humans
- drive a WSGI application in a temporary in-process environment

Demo

Using webtest to ensure the API operates as expected

Many helpers to perform useful function tests

WSGIProxy

In-memory WSGI "app" that maps WSGI requests to HTTP requests

Can easily run webtest based functional tests against an out-of-process web server

Repurpose functional tests as integration tests and deployment tests

Can pick up anything that goes wrong during deployment

Can use as a basis for simple load testing

FunkLoad

Wants to be a monolithic "do everything" test framework

Standard mode of use requires that tests be implemented specifcally for FunkLoad (this annoys Ryan, just demoing it to illustrate the point)

Can gather benchmarking data

Use benchmarking tools, as they deal with problems you may not have thought of yet (e.g. only recording data while under full load, ignoring ramp up and ramp down periods)

Nice report generation tools

FunkLoad + WebTest

Write an adapter between webtest API and FunkLoad API

Can then reuse *any* of your webtest based tests as a FunkLoad test

Can use to get differential results to see how your response times are going

Distributed Mode

Spin up clients on multiple hosts

Aggregate benchmarks from multiple clients

Other Approaches

Write FunkLoad tests, use WSGI-Intercept to redirect to in-process WSGI application

Q & A

Just some specific questions about FunkLoad and making in-process WSGI calls on a live application

My Thoughts

Not a lot to add! Some of the ideas here seem similar to those in django-sanetesting, but it in a way that doesn't drive the inheritance model of the test suite. I may look at migrating the PulpDist tests over to webtest (depending on how complex it is to get the in-process Django environment up and running)

Debugging Live Python Web Applications

Graham Dumpleton (mod_wsgi author, New Relic)

Amjith (New Relic, will be presenting this at DjangoCon US0

(Many details missing - will link to slideshare once the slides are up)

Debugging

Can't always duplicate problems in a devel environment

Debugging live servers can be dangerous - crash the site - corrupt data - expose customer data

Tools to manage risk - use software to restrict access - script changes - test beforehand

Passive monitoring

Safest approach

Needs to be set up in advance

Log file analysis tools

Capture Python exceptions (e.g. Sentry)

Monitor the server as whole

Application monitoring (this is what Graham does for a living at New Relic) (For open source custom equivalents, Graphite counters are a good option)

Web page performance analysis - resource timing spec coming from W3C

Reaching the limit

Passive monitoring eventually gets to opaque blocks

Can add more monitoring hooks, *if* you can change the code

New Relic supports configuration and monkey patching at runtime, and custom setups can do this, too.

Problem is you have to deploy a new build to make this happen

Thread Sampling and Profiling

Full profiling is too slow for production

Periodic sampling can pick up big consumers of time

Targeted profiling can profile a function *sometimes* to minimise overhead to the point of allowing it in production (just needs a fairly simple decorator/context manager)

Still requires redeployment to gather more data

Browser tools

Can query a bit more, but still can't change the server

Interactive debuggers

Can give full access, but can break the running application

New tool: ispyd

https://github.com/GrahamDumpleton/wsgi-shell (originally wsgi only, hence the misnamed repo)

Limited access

Configure plugins to provide specific commands

Limits access (but you can enable the 'console' option for the Python plugin if you want)

Can change configuration on the fly (e.g. New Relic plugin)

See the slides for more :)

This looks really cool with a lot of potential

Summary

Monitoring is essential to detect the existence of problems, and to know where to start looking.

Create defined, controlled mechanisms to perform *detailed* queries for deeper dives into production problems that can't be reproduced in development

Hopes ispyd may catch on as a standard approaching for live debugging in a controlled manner (this is the first real public announcement)

Virtual Robotic Cars

Tennessee Leuwenburg (sp?)

Why?

Loves robotics as an intellectual playspace

Submitted the talk partly as a motivation/excuse to spend more time playing

Real World

Brief history of auto racing

Many current "self-driving car" efforts

DARPA Grand Challenge video

Urban Grand Challenge video ($40k radar array!)

Traxxas X01 (video) - 100 MPH remote control car - controlled via iPhone

Range of technology - drive-by-wire(less) - remote drive-by-wire(less) - supervised autonomous (ABS, - fully autonomous

Few real world races - DARPA Grand Challenge - DARPA Urban Challenge - Audi Pikes Peak

Virtual World

TORCS - The Open Racing Car Simulator

TORCS video

Something anyone can do

Similar abstractions to a real robotic car - distance and other sensors - speed, steering, braking controls - noisy sensors - complex environments - open-ended design and problem definitions

Can go as far as you want in learning

Yearly TORCS competition - current entries Java & C++

Now has Python bindings, so can write car control systems in Python

pyScrcClient - uses a socket to talk to the TORCS server

Udacity course - programming a robotic car in 7 weeks

The TORCS Vehicles

Sensors - 20 range-finders - current angle to track bearing

Controls - accelerator - brake - gear changes

Trigonometry to get from distance sensors to an actual picture of the world

Getting Started

Just follow the track center line!

Beyond that - steering and path planning - acceleration and braking - collision avoidance (for racing) - strategy (beyond the scope of the talk)

No compass sensor, so first step is to build a motion model for the car itself

TORCS updates every 0.02 seconds

Places an upper limit on your processing time

Evil maths!

More trig to work out the expected centre point of a turn.

Simplify the model of the car to a bicycle rather than worrying about the four wheels

How to model:

- assume you know the car positionss
- see what happens
- draw something so you can see what your car is "seeing"

Yakkety Sax goes with everything!

(Oops, wrong version of the slides, opened right version to get correct embedded video)

Localisation

Given a map, use it work out where you are

Feedback loop between deriving the map from sensors, and using the results of your sensors to determine where you are

Local vs Global planning

Global: mapping + localisation

Local: collision avoidance, ABS, etc

SLAM and Filtering

Can use "mapping runs" to build up hypothesis maps

Can use track distances to filter particles after a complete lap (known location)

Great free resources online

Virtual Robotic Car Racing lets you explore and exploit most of the related algorithms

Resources

See slides for links (I'll add a link to the slides once they're up)

Q & A

(missed the first couple)

Polulu - remote control chassis with reversible wheels and an Arduino built in to embody your questions

Can download and run bots from the competition to see how good (or bad) you are.

No GPS in the sim software, so you can't "pre-plan" too much (and competition uses novel tracks)

No accelerometer info in TORCS (which seems odd, since this is the first sensor you would add to a real version)

Strategy depth is immense! (e.g. blocking lines to prevent following cars using them)

My Thoughts

Sounds like an interesting way to explore various AI topics.

PiCloud

Amit Saha

http://echorand.me/2012/08/17/pyconau-2012-talk-on-picloud/

Intro

Commercial cloud service

"Worker process" oriented rather than hosting oriented

20 core hours per month included in free account

Python API "import cloud"

REST API for more general access

Transfers dependencies automatically

Pre-initialised with NumPy/SciPy

Can tailor requested compute resources to a particular workload

Can run in a mode that uses multiprocessing locally rather than the cloud

Example

`cloud.call` is the most basic API!

Also the candidates you would expect, like `cloud.call`

(problems with the video demo failing to display properly - sorted out by using a different player)

(My comment: you could probably map a concurrent.futures executor to this API pretty easily)

IPython notebook provided in repo for talk (see link above)

Other capabilities

Directed-acyclic graph of job dependencies (to pass data between jobs, map/reduce, etc)

Persistent data: cloud.files API to move data to/from the cloud and update it in jobs

More compex example: pyevolve

Shows automatic deployment of dependencies

Identifies and pushes Python files that are referenced locally

Retrieves CSV files created pyevolve

Publishing APIs via REST

Can designate a Python function to expose as a REST API

Environment

Automatic deployment only works for pure Python modules

Environments let you tailor what it is installed - non-Python tools - extension modules - Ubuntu-based

Management APIs

Query job status, list jobs, etc.

Resources

More links! (again, see link at top)

Q & A

Any other services like this? (Don't know)

Security - SSH authentication

My Comments

Looks pretty interesting. Definitely aimed at the scientific crowd rather than the webhosting crowd, thus the different emphasis and "worker process" style API.

PyCon US 2013

Santa Clara, California, March 13-21

Python Language Summit

Note: while a few procedural decisions were made (such as the Discussions-To: header becoming more significant for PEPs), this was more an information sharing session than it was about making decisions on any particular topic.

State of PyPy

Armin Rigo and Maciej Fijalkowski

- PyPy 2.0 not far away
- PyPy/RPython split in progress. Separated directories in the main PyPy repo, will move to separate repos some time post-2.0 (requires a lot of fixes to related tools)
- ARM support in progress, will chat to Trent regarding better access to ARM machines through Snakebite
- making good progress on Py3k support

State of Jython

Phillip Jenvey

- Jython 2.7b1 released last month
- Will look at 3.3 support after 2.7 is released
- Java 7 "invoke dynamic" support doesn't actually work yet (JRuby tried it), but once it is working and Jython has been updated to use it, then it should make for substantial performance improvements

State of IronPython

Jeff Hardy

- Already supports 2.7
- Jeff's made some attempts at Python 3 support, hopes to take another look this year
- Not many contributors, definitely welcome more
- Now Apache licensed on Github (shared repo with IronRuby)

Packaging Eco-System

Me

- Previous effort involved a lot of good work, but various factors have limited adoption in practice
- Current efforts are focused on decoupling the build toolchains from the installation tools
- Will be giving the "Discussions-To" header in PEPs more significance: the announcement of acceptance of a PEP with that set will happen on the named list and copies will not be sent to python-dev.
- PEPs involving standard library changes will still have to happen on python-dev
- Need better documentation for the overall packaging and distribution tools ecosystem. I've started such a thing, but at the moment it is just a forlorn issue sitting alone in a "python-meta-packaging" repo I created under the PSF's BitBucket account.

XML and Security

Brett Cannon

- Many security issues inherent in the XML spec.
- Hard to decide how to update the standard library appropriately
- "Secure-by-default" is highly desirable, but some things are inherently dangerous (e.g. pickle, XML)
- May settle for readily available "safer XML parsing" config options that frameworks may choose to enable by default
- Also some communication issues with being clear on what is currenty blocking CPython point releases

Tulip and enhanced async programming support in the standard library

Guido van Rossum

- PEP 380 almost made it into 2.7. Integration issues (such as the lack of unit tests and docs) and the language moratorium meant it ended up being delayed until 3.3
- Non-blocking socket support and asyncore exist, but not a great foundation for robust async IO infrastructure
- Twisted and Tornado show how event based async IO can be succesful in Python
- Guido still doesn't like callback based programming :)
- Aim to create a universal event loop API for Tornado/Twisted/et al to interoperate
- Also aim to make it possible to write yield-from based async code

- Read PEP 3156 and related discussions, we mostly just rehashed those for the benefit of those that hadn't been following along through the many, many threads on python-ideas :)

Parallelizing the Python Interpreter

Trent Nelson

- Allow CPython internals to be executed from multiple threads
- Minimise required changes
- Initial attempt on hg.python.org/trent (px branch)
- Only works on Vista+ (relies heavily on Windows features, Trent has ideas on how to adapt it to *nix)
- Wants to get it working as a proof of concept first, then clean up and add *nix based solution
- May end up as a Stackless style persistent fork/derived implementation for a long while
- Example of a CPU-bound task based on tulip style async API, able to exploit all cores
- GIL still present, no fine-grained locks, no STM
- Intercept "thread-sensitive calls" - anything the GIL protects. (refcounts, object allocator, free lists, interpreter globals, etc)
- "Normal" threads behave as they do now
- Declared "parallel threads" do something different
- Low overhead then becomes about detecting whether or not you're in a parallel thread *really* fast.
- Windows and POSIX both offer ways to detect thread identity based on a single memory read
- Parallel threads only incref/decref when the parallel context is created/destroyed, so it is possible to cope with the fact that the main thread is effectively ignoring any synchronisation mechanisms
- Main thread *stops* while the parallel threads are running, so it can't steal things out from underneath the parallel threads
- Wraps objects with async-protected equivalents
- Ultimately, current version is highly experimental, and it's not yet clear if it can be made sufficiently robust to be useful in general.

(There are some promising notions here that may fit with some vague ideas I've had regarding subinterpreters, but there are a lot of real problems with the current approach, especially relating to references to mutable containers that are modified after the parallel context starts. May still be worth pursuing for the benefif of platforms where multiple processes are a significant problem for performance especially memory usage. I suggested Trent look into subinterpreters and the Rust memory model for ways this could be hardened against the many possible segfault inducing behaviours in the current imlementation)

Snakebite

Trent Nelson

- Set up to provide interesting architectures and OSes for open source projects to test against
- Currently heavily reliant on Trent's time, interested in exploring ways to make it more open to external contributions (donate to PSF?)
- AIX, HP-UX, still red on CPython buildbots, some others are only greendue to extensive environment setup to get CPython building properly

- Trent is interested in finding ways to make this more useful to the community
- Perhaps set up databases for easier database testing?
- Ad hoc BuildBot farms for testing experimental forks?
- Currently pre-built machines on bare metal (mostly more esoteric OSes and architectures)

Argument Clinic

Larry Hastings, Nick Coghlan

- Introspection on builtin and extension functions is currently close to useless
- Builtin and extension functions are already too hard to write, adding signature data as well isn't a reasonable option
- Solution: add an in-place DSL that generates in-place C to be checked in.
- PEPs 436 (Larry) and 437 (Stefan Krah) are competing flavours of the DSL
- Both PEPs agree on the general concept of adding a preprocessor step to reduce the complications involved in adding and updating builtin and extension module functions and methods
- Both PEPs also agree on checking the *preprocessed* modules with both the input and generated output into source control, so the custom preprocessor isn't needed to build Python from a source checkout
- Stefan's PEP pushes for a more Python-inspired syntax for the signature definition itself, whereas Larry's PEP is more Javadoc inspired (with fewer @ symbols and more indentation)
- Since the PEPs are in agreement on most points, Larry, Guido and I will get together at some point this week to try to thrash out something Guido likes in terms of the DSL syntax details

CFFI

Alex Gaynor, Armin Rigo, Maciej Fijalkowski

- cffi competes with both ctypes and SWIG (for C only, not C++)
- unlike ctypes, transparent to the JIT on PyPy (and hence much faster)
- generally slightly faster than ctypes on CPython (due to module generation step)
- replaces ctypes for ABI access to shared libraries
- provides an easy way to generate C extensions given a subset of the C API details (thus replacing some uses of SWIG and Cython)
- Needs some work to clarify the API and more clearly separate the "create an extension module" step from the "load from cached extension module" step
- Dependencies are pycparser and PLY for the higher level typesafe API, libffi for callback handling and the ABI layer of the API (which is just as unsafe and prone to segfaults as ctypes)
- If cffi, and hence pycparser and PLY, are added to the stdlib, all 3 will be public. We may make use of the "provisional API" status.
- Will reconsider proposal once some of the feedback has been addressed, but the idea of adding it certainly seems reasonable

Cross-compilation

Matthias Klose

(I confess I wasn't really listening to this part, I was playing catch-up on Stefan Krah's draft argument DSL PEP he sent me shortly before I left Australia for PyCon US)

- CPython 3.3 and 2.7 both support cross-compilation (e.g. x86_64 to ARM)
- still a few issues in various regards
- looking to propose additional more invasive changes to the build process to potentially make this easier

Test Facilities

Robert Collins

- stdlib test facilities are focused on in-process testing
- cross-platform and cross-process and parallel testing becoming more important
- easier to drop into a debugger (especially a remote debugger!)
- Robert has a stack that can do this for ordinary unittest-based tests
- Michael is interested in evolving unittest itself as needed, but need to figure out appropriate things to do

Enums in the standard library

Barry Warsaw

- Feature set of `flufl.enum` is pretty good
- Don't want to implement the superset of all third party enum libraries
- Precedent set by bool is for enums to seamlessly interoperate with integers
- Highly desirable for any stdlib enum solution to be usable as a replacement for the constants in the socket and errno libraries without a backwards compatibility break
- Guido doesn't want to have 2 similar enum types in the stdlib, and he wants one that can be used in socket and errno (he called this behaviour `bdfl.enum`, to contrast with `flufl.enum`)
- Guido is OK with different enum types comparing equal, and requiring explicit type checks to limit an API to accepting only particular enum types
- As a proponent of labelled values over any form of enum, Guido's stated preference for "enum as labelled int" (following the precedent set by bool) actually works for me
- Barry is in favour of bitmask support for flufl.enum anyway, which is the other element (other than comparisons) needed for a solid proposal that is interoperable with integers
- Guido also made the point that this is a case where "good enough" will likely be enough to kill off most third party enums over time

Requests

Larry Hastings

- With Kenneth Reitz declaring a stable API for requests with 1.0, he's interested in offering it for stdlib inclusion in 3.4

- chardet and urllib3 vendored dependencies are a concern for incorporation, particularly with tulip/PEP 3156 also coming in Python 3.4
- a tulip-backed requests would be much easier to include (as well as a validation of tulip's support for writing synchronous front ends to the async tulip backend.

Legacy Modules

Nick Coghlan

- Better indicate deprecated libraries in the table of contents
- Maybe by a separate section in the ToC, or just by appending "(deprecated)" to the section titles

Things we didn't cover

These didn't get covered because I forgot to put them on the agenda. I'll probably be chatting to people about them during the week anyway:

- PEP 422 (simple customisation of class creation)
- PEP 432 (CPython interpreter initialization)
- Unicode improvements (change stream encodings, better encoding specification in subprocess, restore type agnostic convenience access to the codecs module)

Brisbane Python User Group - PyCon US Review

About me

- Nick Coghlan (Twitter: @ncoghlan_dev)
- CPython core developer (commit privileges ~2005)
- Red Hat toolsmith (since Jun 2011)

A litany of links

I can't do a 2500-person, 9 day, global conference justice in a short talk, so this is mostly a collection of links to sites and articles related to the conference.

Awesomeness

A nice collection of quotes and links: http://thisispycon.com/

The basics

Almost all the slides and videos for the talks and tutorials are freely available online

- Tutorial schedule: https://us.pycon.org/2013/schedule/tutorials/
- Talk schedule: https://us.pycon.org/2013/schedule/talks/
- Talk & tutorial slides: https://speakerdeck.com/pyconslides
- Talk & tutorial videos: http://www.pyvideo.org/category/33/pycon-us-2013

My PyCon

- Python Language Summit: *Python Language Summit*
- Post conference report: http://community.redhat.com/change-the-future-one-small-slice-of-pycon-us-2013/
- Plans for Python packaging: *Incremental Plans to Improve Python Packaging*

Changing the future

- Raspberry Pi: http://raspberry.io/
- Diversity & outreach: http://adainitiative.org/2013/03/pycon-feminist-hacker-lounge-report-out/
- Revamped python.org: http://preview.python.org/

CHAPTER 7

About These Notes

These notes are written using Sphinx and published to ReadTheDocs via BitBucket. Feedback is welcome on the BitBucket project issue tracker.

CHAPTER 8

Indices and tables

- genindex
- search

Index

P