LINUX JOURNAL
(/)

Interview with Guido van Rossum

Industry News (/taxonomy/term/22)

by James Gray on October 1, 2008

Python is the wildly popular, high-level programming language that was recently voted Favorite Scripting Language in the 2008 *Linux Journal* Readers' Choice Awards. In this interview, Python's creator Guido van Rossum shares his insights about the revolutionary new Python 3000, why the pain from backward incompatibility is worth it, what he foresees for the Python 2.6 fork, and what he's been up to lately at Google.


(/files/linuxjournal.com/linuxjournal/articles/101/10185/10185f1.resized.jpg)

Guido van Rossum

JG: By the time readers see this interview, Python 3000 (aka Py3K and Python 3.0) should be available. What is in the new version that will excite developers?

GVR: You've probably heard that Python 3000 will introduce backward-incompatible changes. That alone probably is enough to get developers excited, or at least upset. So let me emphasize first that, by and large, Python 3.0 will be the same language you've loved and used before, it's just been cleaned up a bit. You may want to contrast this with Perl 6 vs. Perl 4, where Perl 6 is a totally new language, with a completely different implementation. We're not doing anything remotely as drastic as that!

Many of the cleanups are pretty benign. For example, we're finally getting rid of string exceptions (all exceptions have to be defined as classes). There is a large class of cleanups like this, and I refer your readers to the python.org Web site for the (mostly) boring details.

Some changes *seem* controversial but actually are a big improvement, such as replacing the print statement with a print() function. The big advantage of making it a function is that we can use the familiar keyword=value syntax to specify behavioral variations like printing to a different file or suppressing the final newline. We also can add new keywords more easily. For example, in Py3k you can override the separator between items, and this makes future evolution much easier compared to evolution of a statement-based syntax. Using standard function syntax also makes it much easier to replace the built-in print function with a function of your own design. This is a common transformation over the lifetime of a program. What started out as simple print statements at some point have to become logging calls or at least redirectable to a different file, and all these changes are easier to make consistently with function calls.

There is one group of changes that is (relatively speaking) revolutionary, and at the same time, it is probably responsible for the most conversion pain, *and* for the largest sigh of relief. We're adopting a fundamentally different attitude toward Unicode. A bit of history: Python 1 supported only eight-bit strings, which were used for text and binary data alike. Python 2 kept this dual use of eight-bit strings, but added Unicode strings. This was done so as to maintain backward compatibility with Python 1, but it created a new major ambiguity. There were two ways of representing text strings, either as eight-bit strings or as Unicode strings. Moreover, the meaning of eight-bit strings remained ambiguous, as these were used for text as well as binary data.

In Python 3, we're breaking with compatibility and drawing the line differently. There will be a bytes type to be used for binary data (and *encoded* text, like UTF-8 or UTF-16), and there will be an str type to be used for text only and capable of representing all Unicode characters. The implementation of the bytes type closely resembles that of the old eight-bit string type, and the implementation of the str type is copied from the old Unicode type. The big improvement over Python 2 is that both ambiguities I mentioned above are removed. There is now a 1:1 mapping between usage (data or text) and types (bytes or str). Reports from early adopters have shown that developers really appreciate this change and are happy to pay for it. Some third-party projects, such as Django, already have adopted a convention in Python 2 that essentially is the same. All text is stored in Unicode strings, and eight-bit strings store only binary data, but Python 2 doesn't help enforce this.

There also are some other changes related to Unicode. The default source encoding is now UTF-8, identifiers can contain non-ASCII letters, and the repr() function no longer will turn all non-ASCII characters into hex escapes (it still will escape control characters of course).

JG: In retrospect, do you regret any changes that made it through to the final version?

GVR: No, I'm very happy with the outcome. I think we've struck a phenomenal balance between changing too much and changing too little. It has really helped that toward the end of the Py3k development, we switched to a time-based release schedule, so we had a clear way to stop the never-ending stream of proposals for yet more language improvements.

JG: Python 3000 is currently slower than 2.5. Will it be as fast or faster once it is seriously tuned?

GVR: I expect that by the time 3.0 is released, we'll be close to the 2.5 speed. We'll probably keep tuning it well beyond that, and if past history is any measure of future performance, we'll see continued speed improvements as new releases come out.

JG: Python 3 breaks backward compatibility with version 2.6. This is a pretty bold step for a programming language in general and in particular for one with a user base the size of Python's. The only other time I remember somebody trying this was when Microsoft went from VB6 to VB.NET, a move that has a lot of VB6 programmers still miffed six years later. Do you have concerns regarding this move?

GVR: I think you may have forgotten about Perl 6.

My understanding is that VB.NET was actually fundamentally different from VB6, much more so than Python 3 differs from Python 2. Most of the differences in Python 3 are relatively close to the surface. In particular, we've made a conscious choice *not* to radically change the underlying implementation. If I understand correctly, VB.NET uses a completely different virtual machine (based on the new .NET technology) from VB6. This is not the case for Python 3. We started Py3k as a branch of the Python 2 VM and gradually modified it to support the new language. But, most implementation details are exactly the same, and up to this date, we routinely merge changes from the trunk (which will be released as Python 2.6) into the Py3k branch.

I certainly don't want to underestimate the cost for developers of the transition from Python 2 to Py3k. We have been thinking about this transition for at least two years now, and we have several parallel strategies in place to make developers comfortable with the change.

First of all, Python 2 will be fully supported for a long time in parallel with Python 3. My personal expectation is that there will be a period of at least three to five years where developers have complete freedom to choose between Python 2 or Python 3, getting the same level of support. There will be new releases of Python 2, starting with 2.6, in parallel with the Python 3 releases.

Second, we have designed a specific two-prong transition strategy. The first prong of this strategy is the release of Python 2.6 simultaneously with the 3.0 release. 2.6 will be backward compatible with 2.5, but it also will contain an *optional* set of warnings that alert you about a variety of issues in your program that will break if and when you port it to Py3k. These warnings are issued only when specifically requested via a command-line option, so that they are not an impediment toward upgrading from 2.4 or 2.5 to 2.6, regardless of whether you are planning to port your code over to 3.0. In addition, 2.6 also will contain some back-ported 3.0 features, which we hope will encourage people to start using 2.6 in a way that will reduce the pain when they are ready for 3.0.

The second prong of the transition strategy is a source code conversion tool that we call 2to3. This tool handles most of the small syntactic changes you encounter when converting Python 2 code to Py3k. For example, it automatically translates print statements into print() function calls, turns Unicode literals (such as `u"..."`) into regular string literals, strips the trailing L from long integer literals, and so on. It also does a decent (though not perfect) job of converting calls to popular dictionary methods like .keys() and .iterkeys() into their Py3k equivalent.

The two prongs complement each other nicely. The 2to3 tool takes care of the syntactic changes, and the Py3k warnings in Python 2.6 handle those changes that a purely syntactic tool cannot handle easily. Because Python is such a dynamic language, conversions that require information about the type of a variable or attribute generally cannot be automated. The 2to3 tool leaves these alone, but there is enough overlap between the 2.6 and 3.0 languages that, in general, it will be possible to change your source code in such a way that it still is compatible with Python 2.6 (and usually with older versions as well), produces no Py3k warnings, and can be translated safely to valid Python 3.0 source code using the 2to3 tool.

JG: Also, how complex do you think that the upgrade process to Python 3000 will be?

GVR: I think I've given a decent indication of the complexity in my answer to the previous question. The general work flow for a conversion could be as follows:

1. Start with code that works under Python 2.4 or 2.5 and has a good test suite.
2. Port to Python 2.6. This should be straightforward. Try to run the test suite under Python 2.6, resolve issues found, and repeat until all tests pass. Python developers have used this process for years with the transition to each Python version, and the expectation is that there won't be many changes to make.
3. Turn on Py3k warnings and run the test suite again. Resolve issues reported, and repeat until all tests pass without warnings.
4. Run the 2to3 tool over your source code, including your test suite, and run the converted test suite under Python 3.0. If there are issues, *don't fix them here*, but fix them in the 2.6 code base, and repeat starting from step 3.

In terms of revision control, you most likely will be maintaining two branches of your code long term: the 2.6 version and the 3.0 version. Changes to the 2.6 version should be merged to the 3.0 version using the 2to3 tool.

JG: What kind of feedback have you gotten from the early adopters of Python 3000 thus far?

GVR: We've heard everything from pure excitement to extreme fear. Given the magnitude of the change, we can't expect everybody to be happy, but the general trend is one of cautious optimism. As expected, most developers are happy with most of the new features. Although almost everyone has a pet peeve or two, those appear to be mostly outliers, and there aren't any changes that stand out as unwanted by many.

JG: Have any large projects already been converted to Python 3000, and what have the results been?

GVR: It's too early to say. We've only just released the first betas of 2.6 and 3.0, and so far, the focus of third-party developers, especially of large packages, has been on 2.6 over 3.0.

JG: Is there a chance that there might be a rogue fork of the 2.x line, and would this bother you?

GVR: I don't expect any “rogue” forks to happen. The Python community tends to prefer consensus over conflict, at least in the long term.

JG: What was the process by which changes were accepted or rejected in the upgrade process?

GVR: We started out by setting some basic parameters for the upgrade, in PEP 3000: the goal was primarily to fix early design mistakes and clean up situations where two ways to do something had evolved out of a desire to improve the language while also maintaining backward compatibility (for example, new-style vs. classic classes). This was a powerful argument to keep many of the more radical change proposals out of the door.

The rest was a matter of long community discussions with the occasional tie-cutting by yours truly in case a consensus remained elusive. I have an incredibly subtle set of gut feelings for judging the most “Pythonic” solution to any one issue, keeping a precarious balance between pragmatics and principles. But, I have tried to use this only after ample discussion had clarified motivations and use cases for proposed changes.

JG: Were there any changes you wanted that were rejected, or any that you didn't want that were accepted?

GVR: That's hard to say. I certainly have proposed things that were rejected, but in the end, I always ended up agreeing with the rejection—and, ditto in the other direction.

JG: How are your synapses currently firing regarding Python 4000 and beyond?

GVR: I hope I'll be in retirement by then!

JG: Our Publisher Emeritus and your old friend Phil Hughes asked me to ask you, “Is Django [the high-level Python Web framework] as cool as it appears?”

GVR: Oh yes, it is. (And hi, Phil!) I like it because it strikes a very Pythonic balance between theory and practice, and because the organization of the project is very similar to that of Python itself. The Django developers run an excellent open-source project, listening carefully to their users and contributors, without being distracted by “feature-itis”.

JG: KDE 4.x has abandoned the classic desktop for Plasma, which supports writing scripted add-ons, or applets, in a number of programming languages. Do you see a role for Python in this space?

GVR: This is the first I've heard of this, so I'd rather not make any rash comments. I hope that if Plasma becomes popular, its developer makes it scriptable using Python.

JG: What interesting trends have you seen lately in the development of the Python community?

GVR: I'm very happy with the influx of new developers in the past year or so. This has really enriched the community with new ideas and new areas of expertise, and removed the pressure from some of the old hands who have been keeping things running for many years.

Another, quite unrelated, but also hugely exciting, trend is the activity in the PyPy Project. As you may remember, PyPy started out as an attempt to write a portable Python interpreter in Python, made fast by the use of a Python-specific JIT. Most PyPy developers are in Europe, and with two years of EU (European Union) funding, the project has made tremendous progress. As agreed ahead of time, the EU funding ended after two years, but recently Google has started funding some specific PyPy activities, and I am excited that these will eventually make PyPy a viable alternative to CPython.

JG: You have been working for Google now for almost three years. Can you divulge what they've had you working on, or is it top secret? Also, is Python subject to Google's 80/20 rule—the one that allows employees to spend 20% of their time on personal projects that are potentially worthwhile to the business—or do you have a different arrangement?

GVR: It's no secret that my first Google project was Mondrian, an internal Web tool for collaborative code reviews using Perforce. Since last November, I've been working on Google App Engine, an exciting project that allows Web developers to run scalable Python Web applications on Google's powerful infrastructure. (In the future, other languages also will be supported.)

I have written an App Engine demo that reuses some components of Mondrian and refactors them into a code review tool for Subversion. With Google's permission, I have released this as open source. You can see it working at codereview.appspot.com (https://codereview.appspot.com), and you can find a link to the source code there as well.

I don't have a 20% project per se, but I have Google's agreement that I can spend 50% of my time on Python, with no strings attached, so I call this my “50% project”.

JG: Thanks so much for your insights, Guido, and good luck with the new Python!

James Gray is *Linux Journal* Products Editor and a graduate student in environmental sciences and management at Michigan State University. A Linux enthusiast since the mid-1990s, he currently resides in Lansing, Michigan, with his wife and cats.

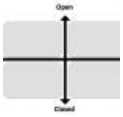
Shawn Powers (/users/shawn-powers)

(/content/getting-sticky-it)

FreeDOS Is 23 Years Old, and Counting (/content/freedos-23-years-old-and-counting)

Jim Hall (/users/jim-hall)

(/content/freedos-23-years-old-and-counting)



From vs. to + for Microsoft and Linux (/content/vs-microsoft-and-linux)

Doc Searls (/users/doc-searls)

(/content/vs-microsoft-and-linux)