

Guido van Rossum: Benevolent Dictator for Life

Adam Turoff

Abstract

I interviewed Guido van Rossum, creator of Python and leader of the Python community, also known as Python's *BDFL*, the Benevolent Dictator for Life. My goal was not to ask the same boring questions that Guido has answered many times before, but to ask the questions whose answers that you—Perl programmers—would be interested in reading.

Earlier this winter, I had the opportunity to hear Guido van Rossum speak to the New York Perl Mongers when he was in town for LinuxWorld. The discussion was quite lively, and the mood was quite civil—thankfully devoid of the holy wars that previously dominated the Perl versus Python discussion. Last month, I caught up with Guido in an email interview to follow up on some of the issues raised at that meeting. The questions I most wanted to ask related to why Python is different, and what lessons Perl programmers should be trying to learn from Python.

1 Creating Python

Programming languages come and go, yet Python is alive and well ten years after its initial release. What do you think Python got right, and what aspects of Python do you feel need improvement?

As you know, Python's predecessor was ABC, developed in the early eighties. When I created Python, I started out with strong opinions and good intuition on what was good programming language design. Much of this was rooted in the philosophy of ABC's designers, and the rest came from my own (then) 15 years of programming experience. I was lucky that I wasn't bound by a desire or need for backwards compatibility with older tools, so I could freely pick and choose only the best from other languages. I also had good firsthand experience with the areas where ABC failed, both in terms of language design (e.g. its innovative terminology was a failure, and it had too much emphasis on keeping things sorted) and in terms of implementation (ABC was a single monolithic program that was hard to extend).

I think the areas most in need of improvement are not in the language's design or implementation, but in the community support around it. For example, Python's equivalent to CPAN (the Vaults of Parnassus) could use work, and we're desperately looking for a catchy marketing slogan to explain Python's benefits in eight words or less. "import this" just doesn't strike me as a particularly good catch phrase.

What problems did you want to solve that led you to create Python? What goals do you have for Python today?

The specific problem back in 1989, was to have a programming language that would run on an operating system that was very different from Unix, and would let me write relatively small often throwaway programs that would require too much time to code in C, but weren't easily done as shell scripts either. Because of the non-Unix requirement, porting Perl wasn't an option (apart from the fact that it offended my good taste.

Today, Python has shown it's good for quite a bit more than that. It's a glue language, it's a scientific steering language, it's an extension and end-user programming language, it's a prototyping and testing language, it's a web, database and GUI language, it's used for building distributed systems consisting of hundreds of thousands of lines, and it's used in primary computer science education. My goals are to keep the language usable and make it even more usable, for all those purposes. All this without getting any group of users to revolt.

It's well known that you started Python as a language to teach computer programming. Many of your efforts over the years have involved using Python to teach programming as well. What's the best non-Python environment that you have seen for teaching programming? How close are Python and IDLE to your ideal environment for teaching programming?

Actually, while Python descended from a teaching language (ABC), it was originally not intended for that purpose. I just kept enough ideas from ABC that it turned out to be a good teaching language—in other words, teaching was in its genes.

Some of the better environments I've seen for teaching are Smalltalk (squeak) and DrScheme. Python the language is pretty close, but for teaching a parser with much better diagnostics (especially warnings while you type, unintrusively marked a la spell checking in MS Word) would be nice. IDLE the environment is pretty close too, but it needs project management, an easier way to run a script (and the script should be run in a separate process), and a better debugger.

2 Improving Python

Beginning programmers and professional programmers have two different sets of requirements for a programming language or a development environment. What does Python offer for the beginner? For the professional? For the casual programmer?

For the beginner: easy-to-learn syntax, easy-to-use data structures, and an interactive interpreter that's open to experimentation. Also a large set of readable examples in the form of the standard library, a friendly community that's eager to answer questions, good free on-line tutorials, and plenty of books to learn from.

For the pro: a large standard library and an even larger library of third party add-on modules and packages, exceptions, several layers of programming structuring devices (packages, modules, classes), a choice of several different GUI toolkits, and the ability to write your own extension modules in C, C++ or Fortran.

For the casual programmer: a syntax that's easy to remember, a large standard library with pre-built solutions, a vast amount of documentation, and real power under the hood when you need it.

What major features have been added to Python over the last few releases? What major features are slated for inclusion in the next few releases?

In 2.0, we've added list comprehensions, augmented assignment, a real garbage collector, string methods, Unicode and XML support, and a new, more Perl-compatible regular expression engine. In 2.1, we added nested scopes and weak references, amongst others. In 2.2, we added iterators, generators, the ability to subclass (most) built-in types, and unified ints with long ints. For more on all these changes, I recommend the series of articles by Andrew Kuchling, "What's new in Python 2.x", with URLs:

`["http://www.amk.ca/python/2.%s/" % x for x in "012"]1`

¹or, just start at <http://www.amk.ca/python/>

(BTW, that was a list comprehension, introduced in 2.0.²)

In 2.3, we plan to do mostly consolidation work, and add to the library. There are a lot of proposals under consideration, and some of them will make it into 2.3, some will make it into 2.4 or later, and some will be shot down or replaced by something better. I've written a highly personal and subjective review of most of the proposals recently, on view at

<http://www.python.org/doc/essays/pepparade.html>

I'm also thinking about adding a new 'bool' type to the language, which will make it a little easier for programmer to express their intention of using a variable to hold a truth value. To my surprise, it brought out a lot of controversy: the responses cover the entire spectrum, from "at last, but you're not going far enough" to "who needs it" and "this would ruin the language". So I'm still thinking about that one.

3 Python and Perl

One of the things that distinguishes Python when compared to Perl is that Python has been implemented multiple times: CPython, JPython/Jython, Stackless Python, etc. Have multiple implementations of Python been a goal for the Python language, or is it an interesting side effect of Python's standard implementation? Where would you like these alternative implementations to take Python?

These really are only two implementations, Jython and CPython: Stackless is just an add-on for CPython. But there's a third implementation, Python for .NET. There are also several projects aiming at translating Python to C code, which will eventually become separate implementations in their own right, with different semantics in corner cases.

It wasn't an original goal to have multiple language implementations, but I've always resented language features that were too closely tied to a particular implementation technique. I've actively encouraged the development of alternate versions, in particular Jython. I've enjoyed the opportunity it gave me to become conscious of the difference between accidents of an implementation from the intentions of the language designer, and it has helped clarify issues that might change in future versions of CPython (like the reliance on reference counts). It has affected the design of some newer features. For example, we decided that generators would use a new keyword, so they would be recognizable by the parser, which was a requirement for a 100% pure Java translator.

I hope the alternatives take Python to places that CPython cannot reach. Jython is an obvious example; the C translators will eventually break through the speed barrier.

What lessons do you feel Python can teach Perl programmers (both as a language and as an environment)?

I think you'd have to ask a converted Perl programmer; I don't know enough Perl to be able to say for sure. But I guess that it wouldn't hurt to emphasize the importance of readable code for projects that extend over time or space (e.g. number of developers). Too much cleverness in the parser can turn against you.

²or, just start at <http://www.amk.ca/python/>

What lessons do you feel Perl can teach Python programmers (both as a language and as an environment)?

Sometimes, speed matters. Regular expressions are handy. String interpolation (the ability to substitute variables into strings) is useful. CPAN rules.

4 Thoughts about Python

What lessons has Python taught you about the design of a programming language?

Don't be afraid to follow your intuition. At least, that worked for me.

What use of Python has made you most proud to be recognized as the Python's creator?

That's a tie between two opposite ends of the spectrum: at one end, the existence of a very large system written in Python, Zope, which now supports a company with a growing number of consulting customers while the software itself is open source; and at the other end, the Python programs written by middle and high schoolers, especially girls—a very underrepresented group in our geek world.

5 Resources

The Python Homepage – <http://www.python.org/>

Jython, the Java implementation of Python – <http://www.jython.org>

IDLE, an Integrated DeveLopment Environment for Python – <http://www.python.org/idle/>

Squeak, a cross-platform Smalltalk implementation – <http://www.squeak.org>

DrScheme, a beginner-friendly environment for the Scheme programming language; part of the PLT Scheme project – <http://www.plt-scheme.org/>