



The Making of Python

A Conversation with Guido van Rossum, Part I

by Bill Venners
January 13, 2003

Summary

Python creator Guido van Rossum talks with Bill Venners about Python's history, the influence of the ABC language, and Python's original design goals.

Guido van Rossum is the author of Python, an interpreted, interactive object-oriented programming language. In the late 1980s, Van Rossum began work on Python at the National Research Institute for Mathematics and Computer Science in the Netherlands, or *Centrum voor Wiskunde en Informatica* (CWI) as it is known in Dutch. Since then, Python has become very popular among developers, who are attracted to its clean syntax and reputation for productivity.

In this interview, which is being published in weekly installments, Van Rossum gives insights into Python's design goals, the source of Python programmer productivity, the implications of weak typing, and more. In this first installment, Van Rossum describes Python's history, major influences, and design goals.

Bill Venners: How would you describe Python to developers who have never used it?

Guido van Rossum: From one perspective you can say Python is a very high-level scripting language. From another perspective you can say it's a high-level programming language that happens to be implemented in a way that emphasizes interactivity. Python shares some characteristics with scripting languages, but also shares some characteristics with more traditional programming languages.

ABC's Influence on Python

Bill Venners: Could you give a brief history of Python?

Guido van Rossum: In the early 1980s, I worked as an implementer on a team building a language called ABC at Centrum voor Wiskunde en Informatica (CWI). I don't know how well people know ABC's influence on Python. I try to mention ABC's influence because I'm indebted to everything I learned during that project and to the people who worked on it.

ABC's design had a very clear, sharp focus. ABC was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers in any sense. During the late 1970s, ABC's main designers taught traditional programming languages to such an audience. Their students included various scientists—from physicists to social scientists to linguists—who needed help using their very large computers. Although intelligent people in their own right, these students were surprised at certain limitations, restrictions, and arbitrary rules that programming languages had traditionally set out. Based on this user feedback, ABC's designers tried to develop a different language.

Another source of frustration for ABC's designers was Basic. At the time, Basic was a fairly crippled language. Although in a sense Basic was aimed at the same audience, non-programmers using computers, it came from a different language design perspective. The Basic versions available at the time were horrible. Almost any interesting Basic program was full of low-level hacks, where one had to poke memory byte 714 to change the screen background color to yellow.

Bill Venners: I remember doing that. When were you working on ABC?

Guido van Rossum: ABC's authors started designing the language in the late 70s and early 80s. I joined the team in 1983. I think we worked until 1986 or 1987. For various reasons, the ABC project wasn't a big success. Maybe it was too early, before there was an Internet to do efficient distribution.

Python Is Born

Guido van Rossum: In 1986 I moved to a different project at CWI, the Amoeba project. Amoeba was a distributed operating system. By the late 1980s we found we needed a scripting language. I had a large degree of freedom on that project to start my own mini project within the scope of what we were doing. I remembered all my experience and some of my frustration with ABC. I decided to try to design a simple scripting language that possessed some of ABC's better properties, but without its problems.

So I started typing. I created a simple virtual machine, a simple parser, and a simple runtime. I made my own version of the various ABC parts that I liked. I created a basic syntax, used indentation for statement grouping instead of curly braces or begin-end blocks, and developed a small number of powerful data types: a hash table (or dictionary, as we call it), a list, strings, and numbers.

I took ABC's ingredients and shuffled them around a bit. Python was similar to ABC in many ways, but there were also differences. Python's lists, dictionaries, basic statements and use of indentation differed from what ABC had. ABC used uppercase for keywords. I never got comfortable with the uppercase, neither reading nor typing it, so in Python keywords were lowercase.

I think my most innovative contribution to Python's success was making it easy to extend. That also came out of my frustration with ABC. ABC was a very monolithic design. There was a language design team, and they were God. They designed every language detail and there was no way to add to it. You could write your own programs, but you couldn't easily add low-level stuff.

For example, one of the big frustrations for software developers using big mainframe computers in the 60s, 70s, and 80s was input/output (IO). All those IO systems were way too complicated. The ABC designers realized their users' confusion with IO, and decided to do something different. But I think they went overboard.

Instead of IO, where you could read a file and write a file, ABC's designers decided to just have global variables in the program. Their users already understood the concept of global variables. So ABC's designers made those global variables persistent. If you quit your session, all your global variables were saved by the system to a disk file. When you started another session, all your global variables were restored. It worked fairly well, to an extent. It is similar to the idea of a workspace, for example, in Smalltalk. There was a print statement that wrote to the screen, and an input statement that read from a keyboard, but there was no way to redirect IO to or from a file. They literally didn't have any other IO available.

Around that same time, personal computers became available. Personal computers had all this wonderful packaged software that dealt in files. There was a spreadsheet file, a word processor file, a graphics editor file. The ABC users wanted to write little ABC programs that took something from their word processor file and pushed it back into the spreadsheet, or the other way around, but they couldn't because of the limitation on IO.

Bill Venners: They wanted to massage files.

Guido van Rossum: They wanted to massage data, and the data just happened to be in files. It made things difficult that the language didn't have files as a concept.

Extensibility in Python

Guido van Rossum: What made the lack of file support in the ABC language worse was that it wasn't easy to extend ABC. You couldn't say, "This language is implemented in C, so let's just add another function to the standard library that does open a file." ABC had no concept of a standard library. It had built-in commands, but the parser knew about them. It had built-in functions that were very much integrated in the runtime. Adding to the language's standard functionality was very difficult.

For Python, I felt extensibility was obviously a great thing to have. I already knew we would want to use Python on different platforms. I knew we wanted to use Python on Amoeba, the operating system we were developing, and on UNIX, the operating system we were using on our desktops. I knew we would also want to support Windows and Macintosh. I realized that each of those systems had certain functionality that was consistent everywhere, like the standard IO library in C—but there was also different functionality. If you wanted to draw bits on the screen on a Windows system, you had to use different code and a different programming model than you would on a Macintosh or on Unix.

I came up with a flexible extensibility model for Python. I said: "We'll provide a bunch of built-in object types, such as dictionaries, lists, the various kinds of numbers, and strings, to the language. But we'll also make it easy for third-party programmers to add their own object types to the system."

ABC also didn't have namespaces, because it was intended as a relatively small scale language. It only had functions and procedures. You couldn't group them together. Later they added a namespace mechanism, but I thought it was fairly clumsy. By then I had some experience with Modula-2 and Modula-3, so I decided the module would be one of Python's major programming units.

I decided Python would have two different kinds of modules: You can write modules in Python, but you can also write a module entirely in C code. And such a module can make new types and new objects available. That turned out to be a successful idea, because immediately my CWI colleagues, the users, and I started writing our own extension modules. The extension modules let you do all sorts of things: communicate with graphics libraries, data flow libraries, and all sorts of file formats.

Bill Venners: So if I write a module in C, I can use it from my Python program and the types look just like Python types?

Guido van Rossum: That's right. In Python, the way to use a module is always through import statements. Python's import works slightly different from Java's import, but it has the same idea behind it. You import some module name, and the system uses several different ways to locate that module. There's a search path. It looks for files of different types. If you're looking for import foo, it will eventually find either a file foo.py or foo.so (or foo.dll on Windows). foo.py is a piece of Python source code. The Python source is parsed and interpreted. That makes functions and/or classes available to the program. foo.so, or foo.dll, is a piece of precompiled machine code. It is usually implemented in C or C++, but some people use Fortran to write their extensions that will link to large Fortran libraries. The way you use a precompiled machine code module is, from the Python point of view, exactly the same. You import it. You can list the module's contents to see what's inside. Or, if you're already familiar with the module, you can just start using it.

Next Week

Come back Monday, January 20 for Part II of this conversation with Python creator Guido van Rossum.

Resources

Python.org, the Python Language Website:

<http://www.python.org/>

Introductory Material on Python:

<http://www.python.org/doc/Intros.html>

Python Tutorial:

<http://www.python.org/doc/current/tut/tut.html>

Python FAQ Wizard:

<http://www.python.org/cgi-bin/faqw.py>

Guido van Rossum's home page:

<http://www.python.org/~guido/>

Other Guido van Rossum Interviews:

<http://www.python.org/~guido/interviews.html>

Talk back!

Have an opinion? Be the first to [post a comment](#) about this article.

About the author

Bill Venners is president of Artima Software, Inc. and editor-in-chief of Artima.com. He is author of the book, *Inside the Java Virtual Machine*, a programmer-oriented survey of the Java platform's architecture and internals. His popular columns in JavaWorld magazine covered Java internals, object-oriented design, and Jini. Bill has been active in the Jini Community since its inception. He led the Jini Community's ServiceUI project that produced the ServiceUI API. The ServiceUI became the de facto standard way to associate user interfaces to Jini services, and was the first Jini community standard approved via the Jini Decision Process. Bill also serves as an elected member of the Jini Community's initial Technical Oversight Committee (TOC), and in this role helped to define the governance process for the community. He currently devotes most of his energy to building Artima.com into an ever more useful resource for developers.