# Strong versus Weak Typing

## A Conversation with Guido van Rossum, Part V

by Bill Venners with Frank Sommers
February 10, 2003

**Summary**
Python creator Guido van Rossum talks with Bill Venners about the robustness of systems built with strongly and weakly typed languages, the value of testing, and whether he'd fly on an all-Python plane.

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 six 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 Part I: The Making of Python, Van Rossum describes Python's history, major influences, and design goals.
- In Part II: Python's Design Goals, Van Rossum talks about Python's original design goals—how he originally intended Python to "bridge the gap between the shell and C," and how it eventually became used on large-scale applications.
- In Part III: Programming at Python Speed, Van Rossum discusses the source of Python's famed programmer productivity and the joys of exploring new territory with code.
- In Part IV: Contracts in Python, Van Rossum discusses the nature of contracts in a runtime typed programming language such as Python.

In this installment, Van Rossum discusses the robustness of systems built with strongly and weakly typed languages, the value of testing, and whether he'd fly on an all-Python plane.

**Bill Venners**: I once asked James Gosling about programmer productivity. His answer, which I consider to be the strong-typers view of weak typing, was, "There's a folk theorem out there that systems with very loose typing are very easy to build prototypes with. That may be true. But the leap from a prototype built that way to a real industrial-strength system is pretty vast."

**Guido van Rossum**: The leap is also a folk theorem as far as I can tell. That the leap from prototype to system is so big is just as poorly documented as the ease of prototyping.

**Bill Venners**: Gosling also said, "Anything that tells you about a mistake earlier not only makes things more reliable because you find the bugs, but the time you don't spend hunting bugs is time you can spend doing something else." When I asked Josh Bloch about strong and weak typing, he said,

"It's always beneficial to detect programming errors as quickly as possible."

**Guido van Rossum**: Of course. I'm not arguing with that.

**Bill Venners**: Josh Bloch continued, "There's no doubt that you can prototype more quickly in an environment that lets you get away with murder at compile time, but I do think the resulting programs are less robust. I think that to get the most robust programs, you want to do as much static type checking as possible."

That all sounds fine to me in theory. It makes sense that the sooner I find programming errors the better. Strong typing helps me find errors at compile time. Weak typing makes me wait until a runtime exception is thrown. The trouble is that I use Mailman, a mailing list manager written completely in Python. Mailman works fine. It isn't a prototype. It's an application, and it seems quite robust to me. So where does theory miss practice?

**Guido van Rossum**: That attitude sounds like the classic thing I've always heard from strong-typing proponents. The one thing that troubles me is that all the focus is on the strong typing, as if once your program is type correct, it has no bugs left. Strong typing catches many bugs, but it also makes you focus too much on getting the types right and not enough on getting the rest of the program correct.

Strong typing is one reason that languages like C++ and Java require more finger typing. You have to declare all your variables and you have to do a lot of work just to make the compiler happy. An old saying from Unix developers goes something like, "If only your programs would be correct if you simply typed them three times." You'd gladly do that if typing your programs three times was enough to make them work correctly, but unfortunately it doesn't work that way.

All that attention to getting the types right doesn't necessarily mean you don't have other bugs in your program. A type is a narrow piece of information about your data. When you look at large programs that deal with a lot of strong typing, you see that many words are spent working around strong typing.

The container problem is one issue. It's difficult in a language without generics to write a container implementation that isn't limited to a particular type. And all the strong typing goes out the door the moment you say, "Well, we're just going to write a container of `Object`s, and you'll have to cast them back to whatever type they really are once you start using them." That means you have even more finger typing, because of all those casts. And you don't have the helpful support of the type system while you're inside your container implementation.

Python doesn't require you to write the cast, and its containers are completely generic. So it has the plus side of generic containers without the downside. It doesn't have the plus side that the C++ folks claim to get with their templates and other generics. But in practice that mechanism turns out to be very cumbersome. Even compiler writers have difficulty getting templates to work correctly and efficiently, and the programmers certainly seem to have a lot of trouble learning how to use it correctly. Templates are a whole new language that has enormous complexity.

# It's Runtime, Not Weak, Typing

**Guido van Rossum**: One of the things I like about Python is that everything happens at runtime. Python's compile-time complexity is almost nonexistent compared to that complexity of languages like C++ or even Java, which is much simpler than C++ at compile time. Although, doesn't Sun plan to add generics to Java?

**Bill Venners**: Yes.

**Guido van Rossum**: Well, they have to, because it is a strongly typed language, and you can't do certain things with strong typing otherwise.

Weak typing is not really a fair description of what's going on in Python. It's really runtime typing because every object is labeled with a type.

**Bill Venners**: Every object in Python does have a type.

**Guido van Rossum**: Every object does have a type. It is a string, an integer, a floating point number, a dictionary, or whatever.

**Bill Venners**: But method parameters and return values don't have types.

**Guido van Rossum**: Those variables don't have types. Runtime typing works differently, because you can easily make a mistake where you pass the wrong argument to a method, and you will only find out when that method is actually called. On the other hand, when you find out, you find out in a very good way. The interpreted language tells you exactly this is the type here, that's the type there, and this is where it happened. If you make a mistake against the type system in C or C++, and it only happens at runtime, you're in much worse shape. So you can't simply say strongly typed languages are better than runtime typed languages or vice versa, because you have a whole tradeoff of different parts.

**Bill Venners**: Let me try to restate what you said. Basically, as a strong typer, I may believe it is always better to detect errors sooner rather than later. You agree, but you also say I must weigh the cost of the mechanism used to detect those errors sooner. One cost of the strong typing in Java is that I have to finger type all my variables' types. Another cost is I may have three or four iterations of compiling, because I've screwed up something that may not really matter to my program's proper execution. You're saying in a strongly typed language I'm wrestling with things that take up my time.

Gosling said, "Anything that tells you about a mistake earlier not only makes things more reliable because you find the bugs, but the time you don't spend hunting bugs is time you can spend doing something else." You're saying, if you don't have to actually do all the extra work to make the strong typing compiler happy...

**Guido van Rossum**: You still have time to do something else. In Python, because your program is shorter, you can actually test it much sooner than when you're writing a large Java or C program.

You can also easily write a small piece of code in Python and see that it works. You can do very fine-grained unit testing. Bottom-up program development works really well. You can easily start with only a vague idea of what you want, start to implement it, and see how well it works so far. You can then change many things around. If you have a strongly typed language, changing things around becomes much more painful.

In a strongly typed language, when you change to a different data structure, you will likely have to change the argument and return types of many methods that just pass these things on. You may also have to change the number of arguments, because suddenly you pass the information as two or three parts instead of one. In Python, if you change the type of something, most likely pieces of code that only pass that something around and don't use it directly don't have to change at all.

# The Importance of Unit Testing

**Bill Venners**: What is your opinion of unit testing? How important is unit testing in achieving overall system robustness?

**Guido van Rossum**: I like unit testing a lot when you're in the early phases of writing your application—especially when you write the application from the bottom up, as a library of components that your higher-level application can later use. If you follow a rule that you need a unit test for every function, and that you must unit test everything that's part of the function specification, then you prevent many mistakes that otherwise would be much harder to debug. If you have a large program that doesn't work because of a bug in a function, you'll have a hard time finding the buggy function. The function can return a result that is slightly off. That result gets passed on and used in

another calculation, which returns a slightly wrong result. Then 17 steps later you have a negative result that you didn't expect could possibly be negative. Tracking that down could be really hard.

If you do unit testing, you test each function individually. Doing it properly, using the eXtreme Programming attitude, you write your unit tests and your actual function implementation at the same time—when what you should test is fresh in your mind.

I always recommend both black-box and white-box testing. A black box test is written only to the interface, but a white-box test is written with knowledge of the implementation. If you know that you're using a nasty implementation trick, make your unit test also contain a torture test for your implementation's end cases. If you know you're doing something completely different when your input is more than 512 bytes long, for example, I'd recommend you write a unit test to test a few small values, then test 510, 511, 512, 513, and 514. Make sure all those values work and that you don't have an end case where the transition between one algorithm and another introduces a bug.

# Interactively Testing Java with Jython

**Bill Venners**: What's a good use for Jython?

**Guido van Rossum**: A good use for Jython is interactive testing. Even in a pure Java shop, you can use Jython to interactively test your Java classes. It's nice to be able to write unit tests, but sometimes you just want to poke at your system. You want to import your class and see what happens when you call a method with a particular parameter. Maybe you're not yet in the unit-testing phase, you're in the debugging phase. You're trying to determine which things work and which things don't work.

With Jython, you can just start the interactive interpreter, import your Java classes directly into Jython, instantiate classes as instances or even subclass the classes, and start typing away. You can see the results directly rather than having to type a program, compile it, link it, and run it. If you want to also see what it prints for 5 instead of for 4, you can just type that in the next line of the interpreter. If you want to tabulate the results for input values 1 to 10, you can just create a for loop around your Java call. That experimental approach is very helpful.

# Testing Large-Scale Systems

**Frank Sommers**: How do you test the correctness of large-scale Python systems—the kinds of systems you are working on currently?

**Guido van Rossum**: One thing my team and I do a lot, which is actually an area where Java programmers sometimes use Python, is write small tests that interact with the system. For unit tests, you can look at the function specification. This goes in. This comes out. You provide test cases to verify that the function works right. But when you want to test a large system, you often have to improvise.

How to test a larger system is much less obvious, because you may not be able to test all possible inputs or states of the system. In Python, you can easily write a little Python program, or sometimes a large one, that helps you test by providing test input or test data and monitoring what comes out.

Testing a Python program is especially easy, because you also have introspection. You can quite easily intercept output from Python methods. You can quite easily substitute one class that includes test instrumentation for another class. In Python, you can write a generic instrumentation class that acts as a proxy for another class, without knowing the interface of the other class. The proxy class is completely generic. That means if you change the underlying class, you don't need to change the proxy class, even though it implements the same type. The proxy just passes everything on but maybe logs all the method calls, or certainly the important or interesting ones.

# Weak and Strong Personalities

**Bill Venners**: To what extent do you think the choice between using a strongly and weakly typed language to build systems has to do with personality? I've met people who just seem to be weak typing kind of people. I know many people who loved Smalltalk, for example. They consider their Smalltalk days a time when they had it best. I heard one ex-Smalltalker complain of "getting spanked by compile-time errors." Such people feel compile-time errors are slowing them down. On the other hand, I've also met many people who like getting compile-time errors. They see the compiler as a friend helping them find errors sooner. To what extent is choosing the right tool a problem of finding the best fit for the programmer's personality?

**Guido van Rossum**: It could very well be a problem of personality. It's interesting that you mention Smalltalk. I've noticed Python to be a good fit for ex-Smalltalk programmers who are currently forced to write Java code. Many find Python a nice alternative that gets much closer to what they apparently like best. Of course, there are so many different application areas. Writing software for a spacecraft is very different from writing software for telephone exchanges. But you may be right with your personality theory.

# Flying with Python

**Bill Venners**: Speaking of spacecraft, would you be comfortable enough with the robustness of Python systems to fly on an airplane in which all the control software was written in Python?

**Guido van Rossum**: That depends much more on the attitude of the design team that built it than on the language the team chose. There are situations where doing part of the software in Python makes much more sense than doing it in any other language, even if it must have the reliability requirements of a spacecraft or air traffic control.

**Bill Venners**: Why?

**Guido van Rossum**: You'll never get all the bugs out. Making the code easier to read and write, and more transparent to the team of human readers who will review the source code, may be much more valuable than the narrow-focused type checking that some other compiler offers. There have been reported anecdotes about spacecraft or aircraft crashing because of type-related software bugs, where the compilers weren't enough to save you from the problems.

# Next Week

Come back Monday, February 17 for the final installment of this conversation with Python creator Guido van Rossum. If you'd like to receive a brief weekly email announcing new articles at Artima.com, please subscribe to the Artima Newsletter.

# Talk Back!

Have an opinion about strong and weak typing, testing, or robustness? Discuss this article in the News & Ideas Forum topic, Strong versus Weak Typing

# Resources

Python.org, the Python Language Website:
http://www.python.org/

James Gosling's comments on weak typing:
http://www.artima.com/intv/gosling319.html

Josh Bloch's comments on weak typing:
http://www.artima.com/intv/bloch19.html

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

Interviews | Discuss | Print | Email | Screen Friendly Version | Previous | Next