# When Is A Dynamic Programming Language Not Dynamic?

Sam Wilmott
sam@wilmott.ca
www.wilmott.ca

August 17, 2004

## 1 A Bit Of History

Dynamic programming languages have been around almost as long as electronic computers. Development started on the Lisp language in 1956, only a couple of years after development started on Fortran, the first high-level programming language. Dynamic languages with an Algol-based syntax date back to the mid-1960's with Lisp2 — at the same time that other popular dynamic languages such as Logo and Smalltalk were under development.[1] The most successful of the early Algol syntax dynamic languages was Icon, dating from the late 1970's. Icon and Smalltalk, and to some extent Logo, represent the end of the early popularity of dynamic programming languages: if anything, they increased in popularity with the introduction of early personal computers. Lisp's popularity seems a constant — I'm certainly a fan. And Basic just won't go away.

The early dynamic programming languages were largely eclipsed by the object-oriented programming languages in the late 1980's. Programmers turned into software engineers. And dynamic programming languages and their merits were pushed to the background — relegated to the role of minor use scripting utility languages. But almost simultaneously with this change, a new generation of dynamic programming languages emerged, represented most popularly by Tcl, Perl and Python, with Python being a child of the Algol syntax dynamic language family.

So here we are again.

## 2 What Makes Dynamic Languages Work?

The object-oriented programming languages of the last two decades (C++, Java, C#, Eiffel) have emphasized the requirements of software engineering. They

[1] And my first programming language, the dynamic programming language APL.

aim to fix as many of the properties of a program as possible — to be checked by the compiler. The idea is to minimize the few surprises at run time — typically for the software's users. Users don't like surprises.

These static model languages have been quite successful. But it's in the nature of design and implementation that every decision made narrows further choices. The constraints that these languages place on programs can make things harder than they need be.

Dynamic programming languages work because they minimize the barriers placed in the way of programmers. You can do anything you want, as it were. And typically you can do so with a minimum of fuss — no inconvenient declarations, program organization and other bookkeeping.

Dynamic programming languages are especially useful for trying out new ideas. They can be the origin of new ways of programming. After all, although object oriented originated in a static language Simula, it was popularized by a dynamic language Smalltalk. I've been using Python to try out new forms of pattern matching and XML processing for this reason.[2]

Performance has traditionally been an issue for static software engineering — users don't like slow programs. Dynamic languages have always had a reputation for being slow. In part this reputation is only partially justified. Back in the 1960's there were APL programs that were running faster than Fortran programs. In part, who cares? Machines have got wicked fast.

## 3 Fun Is Fun

One consequence of dynamic languages' "do anything you want" is that programming in a dynamic language is fun. Programmers can get right down to what they want to do.

Now there's nothing wrong with having fun. But there is a down side to having fun. It can make one overlook the shortcomings of dynamic languages in general:

- It's not easy to make programs in dynamic languages reliable. It's possible, just not easy.
- Programming techniques that work well in dynamic languages tend not to scale.

And of dynamic languages in particular:

- Care has to be taken that a dynamic language stays true to its roots. The less dynamic a dynamic programming language becomes, the less fun you'll have in the end.

[2] <http://www.wilmott.ca/python>

# 4 The Lisp Lesson

There's one programming language that will have a small but solid community of fans a hundred years from now. It'll be a member of the Lisp family. What member exactly we don't know, although Scheme is a possibility.

What makes Lisp different?

- It's the earliest of the dynamic languages.
- It's minimalist approach to syntax is partially a result of dealing with implementation issues circa 1958, but it's also because of Lisp's attempt at unifying as many issues as possible: most importantly unifying its model of program structure and of data structure.
- It's small but complete. A Lisp family member like Scheme is more of a foundation for programming than a programming language. Scheme has just about everything you'd ever want in terms of "can I do this?"
- Features are added to the language by being implemented in the language. It is in some sense the ultimate extensible language.

In many ways, Lisp is the measure of dynamic languages. Except for all those parentheses. There have been various attempts to repackage Lisp, to deparenthesize it:

- One of the earliest attempts to give Lisp an Algol-syntax was Lisp2. Hardly anybody paid it any attention.
- The Logo programming language is a Lisp family member with few parentheses. It uses a "prefix" operator syntax.
- It could be argued that Smalltalk is really a Lisp family member, with its high dynamic syntax.

None of these attempts have left a lasting impression on the programming language community. But if you're trying to design a new dynamic programming language, it's still worth considering starting with a Lisp family member, expressing your special requirements in Lisp terms, and then repackaging in a more friendly syntax. The alternative is to take some other language's syntax as a starting point, which invariably imposes design decisions from that language that may not apply to your new language.

So why is Lisp important? From the above it's definitely worth keeping in mind. But why important? Because of the way Lisp makes you think about things. It's a fine focus for language ideas. Over the years, there's one thing that programmers have told me over and over again: learning Lisp has a greater impact on the way they program than any other factor.

# 5 Language Extension

A prime benefit of dynamic languages is the ease with which one can extend the language: add operations and even control structures, using just the language itself.

In every programming language, features can roughly be divided into two classes:

- Application oriented features: those used in common programming tasks.
- Tool-building features: those used primarily in implementing facilities for use by other programmers.

There's hardly a firm line between these two categories: application oriented features are used commonly when building tools, and various features straddle the divide between these two categories. For example:

- Operator overloading is definitely a tool building functionality. It's not something to be done casually.
- Coroutines are largely a tool building functionality — they are a major aid when writing parsers for various kinds of specification and data encoding languages (like XML) — although they can be of great use in application development too.
- Objects are largely an application feature — if only because of their ubiquity – but are important for tool building too.

Popular dynamic languages in the past have had a strong set of tool building facilities — language extension has been a big issue for these languages. This is an area I see less understanding and less support of in the current generation of dynamic programming languages.

# 6 Python

The programming language I'm using most these days is Python. It's a nice language for doing prototyping an scripting work. I've been recommending it to my friends as a complement to whatever "serious" programming language (Java, C#) they're using these days. I'm having fun.

However I've run into a number of stumbling blocks, and they're mostly in attempting to write language extensions using the language (i.e. writing modules). In particular:

- The set of operators one can define is predefined and limited. It's very much oriented to numerical operations, which in these days of text processing is unfortunate. On the other hand, it's a boon to be able to overloading the slicing (a.k.a. subscripting) operation.

- One can't add overloaded operations to predefined or built-in types. For the kind of application tools I'm interested in building, defining specialized operations on the string type would help a lot.

- I needed coroutines. I really needed coroutines.

I'm also concerned with Python's apparent ongoing "let's add another feature" direction of language development. It's a sign of solving symptoms of problems rather than solving the problems themselves.

There's something fundamentally wrong with a programming language that's continually developing. It's fun for the language developers maybe, but it's bad for the language's users: how to do something is always changing.

There's also something fundamentally wrong with a programming language that doesn't have coroutines. (Yes, this a hobby horse of mine.) Coroutines are a fundamentatal control flow mechanism, like "if" statements/expressions and functions. Why doesn't every general purpose language have them? (I've actually got an answer to that, but that's another paper.)

Having been involved in the development of programming languages all of my working life, I'm particularly concerned with the tendency to consider a programming language as just a set of features. There's got to more organization to the language than that or it just accumulates features without really increasing its functionality in any significant way. I know, I've been there.

Python's pretty good in the "reflectivity" part it's dynamic language features. But:

- It's missing basic functionality like coroutines. (There I go again.)

- Its object model is static. Not a good fit for the language. A prototyped object model would much enhance the language, and be a better fit. Things like retrofitting methods to existing types would then be possible, for example.

- Operator overloading needs to be made more general.

Progress is being made in these directions by two alternate Python implementations: Christian Tismer's "Stackless Python"[3] and Mark Hahn's "Prothon" project[4]. But it would be nice if these efforts were part of the "main stream" Python development, so that their value could be known and used by a larger audience.

[3] <http://www.stackless.com>
[4] <http://www.prothon.org>