;login:

THE MAGAZINE OF USENIX & SAGE

June 2002 volume 27 • number 3

inside:

PROGRAMMING

Python or Perl: Which is Better?

by Kragen Sitaker

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

python or perl: which is better?

by Kragen Sitaker

Kragen Sitaker is a multilingual hacker who's used UNIX since 1992, presently consulting on server-side Web software development in San Francisco. See *http://pobox.com/~kragen/* for more.

kragen@pobox.com

[Editors' Note: Kragen Sitaker wrote this reply to the standard question, "Which is better, Python or Perl? And why?" He has graciously granted us permission to reprint it.]

Python has a read-eval-print loop, with history and command-line editing, which Perl doesn't.

Python has a bigger standard library, which tends to be better designed. For example,

- it's possible to write signal handlers that work reliably without a deep knowledge of the implementation (and I'm not convinced that a deep knowledge of the implementation is sufficient in Perl).
- time.time() and time.sleep() speak floating-point.
- open() raises exceptions when it fails. Every day, people post on *comp.lang.perl.misc* asking why their programs are failing, and it's because some file is missing or unreadable, and they can't tell what's wrong because they've forgotten to check the return code from open(). This doesn't happen in Python.
- os.listdir() omits "." and ".."; they're always there, so you can include them by writing [".",".."] + os.listdir(), but you almost never want them there. Perl's readdir doesn't omit them, which is a very frequent source of bugs in programs that use readdir.

On the other hand, Perl has a much bigger nonstandard library – CPAN – and some of its standard library is better designed: system() and popen() can take a list of arguments to avoid invoking the shell, meaning all characters are safe.

Python makes it sort of a pain to build types that act like built-in lists or strings or dictionaries or files, and (as of 2.0, with "x in y" now having a meaning when y is a dict) it's impossible to build something that acts both like a dictionary and like a list. It's relatively easy to build something that acts like a function.

But Perl makes it a pain to *use* types that act like those things, and it's impossible to build types that act like functions. But you don't need to build types that act like functions, because you have Scheme-style closures.

Python's syntax is far, far better.

Except that it's indentation-sensitive, which makes it slightly harder to cut-and-paste code and offends the kind of people who unthinkingly adhere to stupid traditions for no good reason. (It might offend other kinds of people, too, but I know it offends this kind of people.)

Perl has a C-like plethora of ways to refer to data types, which brings with it lots of confusion and dumb bugs. It also makes things like arrays of arrays and dicts of dicts slightly more confusing.

Perl has lots of implicit conversions, which hide typing errors and silently give incorrect results. Python has almost none, which leads to slightly more verbose code (for the explicit conversions) and occasional fatal exceptions (when you forgot to convert). (Unfortunately, Python has some implicit conversions and is getting more.)

On the other hand, Python overloads + and * to do different, though vaguely related, things for strings and numbers; Perl makes the distinction explicit, calling Python's string + and * as . and x instead. Also, there is a certain variety of implicit conversion – namely, from fixnums to bignums – that Python doesn't yet do, but should. (Python 2.2 does it.)

Python's variables are local by default; Perl's are global by default. Perl's policy is unbelievably stupid. Worse, in Perl, the normal ways to make variables local don't apply to filehandles and dirhandles, so you have to use special tricks for them.

Perl can be (and, for me, always is) configured to require that all variables be declared and local. Python has no way to require variable declarations, although reading an uninitialized variable is a runtime error, not a runtime warning.

Perl implicitly returns the value of the last expression in a routine. Python doesn't. This is a point in Python's favor most of the time, although it makes very short routines verbose. (Although Python has lambda, which lets you write those very short routines in the Perl style.

Perl has nested lexical scopes, which means that occasionally your variables disappear when you don't want them to, but more often, they aren't around to cause trouble when you don't want them to. They also make it really easy to write functions that return functions as Scheme-style lexical closures. In Python, writing functions that return functions is painful; you must explicitly list all of the values you want to close the inner function over, and if you want to keep callers from accidentally blowing your closure data by passing too many arguments or keyword arguments with the wrong names, you need to write a class with `__init__` and `__call__`. Also, in Python, if you want statements in your closure, you can't write it inline – you have to write `def foo()` and then refer to `foo` later.

Also, Python 2.x has list comprehensions, which reduce the need for really simple inline functions (for map and filter), and also are a very nice language feature in their own right.

The Perl parser gives better error messages for syntax errors.

Perl optimizes better.

Python has an event loop in the standard library. Perl has POE, which isn't in the standard library.

Perl has while (<>). Python doesn't, although it has `fileinput.input()`, which seems to be broken for interactive use. (It doesn't hand the lines to the loop until it's read 8K, and it requires you to hit ^D twice to convince it to stop reading and once more to end the loop.)

Strings in Python are immutable and pass-by-reference, which means that passing large strings around is fast, but appending to them is slow, and it's possible to intern so that string compares are blazingly fast. Strings in Perl are mutable and pass-by-value, which means that passing large strings around is slow, but appending to them is fast, and comparing them is slow.

Python lists don't auto-extend when you try to assign to indices off the end. Perl lists do. This is generally a point in Python's favor.

Perl autovivifies things, so you can say things like $x->{$y}->[$z]++, which will make a hash for $x if there isn't one already, an array for $x->{$y} if there isn't one already, and an element for $x->{$y}->[$z] if there isn't one already, before incrementing it from its default value of zero. Doing this in Python is painful. However, Python allows tuples as hash/dict keys, which lessens the need for this; you can write:

```
if not x.has_key((y, z)): x[y, z] = 0
x[y, z] = x[y, z] + 1
```

Python's variables are local by default; Perl's are global by default. Perl's policy is unbelievably stupid.

Python has this icky (x,) syntax to create a tuple of one item. Perl doesn't have this problem.

Python treats strings as sequences, so most of the list and tuple methods work on them, which makes some code much terser. You have to use substr() or split() in Perl.

Python requires you to use triple-quoted strings to have multi-line strings. Perl has here-docs, but it also lets ordinary strings cross line endings.

Perl indicates the ends of ranges in two ways: the index one past the end of the range, and the index of the last element in the range. The ".." notation uses the second; @foo in scalar context uses the first; etc. Python consistently uses the index one past the end of the range, which is confusing for new users.

Python has this icky (x,) syntax to create a tuple of one item. Perl doesn't have this problem.

On the other hand, Perl has cryptocontext bugs: expressions evaluate to different, and possibly unrelated, things in scalar and list context. This rarely bites me any more, but it used to.

Perl's context-dependency in function evaluation leads to brittleness problems in ways that are difficult to explain; it leads to difficulties in wrapping functions, à la Emacs advice.

Python has reasonable exception handling; you can catch just the exceptions you expect to have happen. Perl has "die" and if you want, you can eval {} and then regex-match $@ to see if it was the exception you wanted, and if not, die $@. The usual upshot is that Perl programs that catch some exceptions usually end up catching all exceptions and continuing in the face of exceptions that should be fatal.

On the other hand, it's still too easy to write a program that does that in Python, too, so people do.

Python gives you backtraces when there are exceptions, from which you are more likely to be able to find the error than from Perl die messages, because they have more information; but Perl die messages are likely to tell you where the error is more quickly, for the same reason. Perl has "croak" which lets you decide which level of the call stack to accuse of causing the error, and can give you backtraces if you want them.

The Python syntax for referring to things in another module is terse enough that people actually use it. Perl's syntax for the same thing is uglier ($math::pi instead of math.pi), and Perl module names are longer, so people tend to import things from the other modules into their own namespace. This makes Perl programs harder to understand.

However, in both Perl and Python you can specify which names can be imported from your module into someone else's namespace, but you can't specify which names can be referred to from another module (e.g., as math.pi, $math::pi). The consequence is that in Perl programs you can usually tell which names are internal to the module and which ones are used from other modules, and in Python programs you usually can't. (Without looking at the other modules, that is.)

Perl lets you trivially build dicts out of lists, which is good, because lists are easy to compute. Python doesn't, although you can write imperative loops to do the same thing.

Perl lets you easily splice lists into other lists (functionally, in list literals).

You can't slice dicts in Python, although you can use 2.x listcomps to get almost the same effect; Perl's @thing{qw(foo bar baz)} becomes [thing[k] for k in 'foo bar baz'.split()]. I'm not sure whether this is better or worse; I think they're both pretty unreadable.

Perl has "last LABEL" and "next LABEL"; Python doesn't. This is stupid of Python, although I don't need multi-level break often. I can get two-level break by moving the nested loops into a new function and using "return" and two-level continue by one-level break.

When Perl converts aggregate data types into strings (e.g., for printing), it turns any references into ugly strings. When Python does, it recursively prints what is pointed to (which fails if the structure is cyclic).

In both Perl and Python, the rules for what counts as true and what counts as false in conditional expressions are needlessly complicated.

In Python, loop conditions that have side effects end up needing to be hidden in functions, or you have to write an N-and-a-half-times loop — which there's no syntactic construct for, so you have to kludge it with "while 1:" and "break".

In Python, you can iterate over multiple sets of items at once:

```
for number, name in [(0, 'zero'), (1, 'one'), (2, 'two')]: pass
```

You can't do that in Perl, although you can do the equivalent if you have an iterator function which returns the tuples one at a time:

```
while (($number, $name) = next_num_name) { }
```

Python also has the zip/map(None,...) function to make this easier, and map() can take multiple sequences, which it iterates over in parallel.

Python has built-in arbitrary precision arithmetic. Perl has it in a nonstandard library.

Python has built-in named, default, and variadic parameters; Perl lets you do all those things yourself, which means that every Perl library that uses named parameters does it differently, and none of them has syntax as nice as Python's f(x=3, y=5) syntax.

Perl has class methods; Python doesn't, although, unfortunately, it is adding them in 2.2.

Perl unifies classes with modules; Python doesn't. So in Python, you can't import a class directly, the way you can in Perl; you can import it from the module it lives in, or you can import its module and get it from there. In Perl, the module is the class. On the other hand, in Python, modules are unified with files, and in Perl they aren't; this usually results in more verbosity in Perl.

Python lets you create classes at runtime with the same ease, or lack thereof, that you can create functions. Perl doesn't allow you to create classes at runtime as easily. This is arguably excessive flexibility that leads to excessive cleverness and unmaintainable code.

Perl will destruct any objects left around at program exit, possibly resulting in destructing objects that hold pointers to already destructed objects; Python doesn't destruct them at all. Both of these approaches suck.

In both Perl and Python, the rules for what counts as true and what counts as false in conditional expressions are needlessly complicated.

Python's sort() and reverse() are in-place only, which means they don't work on immutable sequences, and often makes your code more complicated; this is moronic. Perl did the right thing here.

Perl's split uses a regex; Python's standard split doesn't. Perl is better here. However, Python's standard split defaults to the right separator (whitespace) when you just specify a string, and Perl's split, by default, discards trailing empty fields, which Python's doesn't.

Python's built-in comparison routines do recursive lexical comparison of similar data structures, so you can sort a list of lists or tuples straightforwardly; and you can sort records by some computed key by forming tuples of the key and the record, then sorting the tuples. If you try to sort complex data types in Perl, it will sort them by memory address.

Python's regular expression library is easier to understand than Perl's and uses mostly compatible syntax (although Perl keeps adding features). Perl's regular expressions return subexpressions by mutating global variables $1, $2, etc., which have their state saved and restored in hard-to-understand ways, and they don't mutate those variables if they don't match. Python's regular expression match operator returns a "match object," which, if the regex failed to match, is None; or, if the match succeeded, has a method to fetch numbered subexpressions of the matched text.

See also *http://mail.python.org/pipermail/python-list/1999-August/009693.html* and *http://www.amk.ca/python/writing/warts.html*.