MIKE HOWARD

maybe you should use Python

Mike Howard came into programming from Systems Engineering and has been stuck there. He currently makes his living doing custom software and system administration for a few small companies.

mike@clove.com

LUKE KANIES' ARTICLE "WHY YOU should use Ruby" in the April *;login:* [1] makes some really good points in Ruby's favor. While reading the article, I noticed I could make all the same points for Python.

Before getting started, I need to make two things clear.

First, this is not a criticism of Ruby. I'm not sure which is better, if either is. The important thing to me is that the features Luke talked about make programming easier to do and maintain. They should be part of *any* modern language.

I also should explain my feelings about Ruby and Python: I kind of like Ruby, but don't plan to do much coding in it. I've been writing code in Python for about five or six years now—beginning with Python 1.5.2. I like Python's terse, clear style. One thing that attracted me to it to begin with is the thing that bothers Luke—indentation is mandatory and syntactically significant. However, I'm not a Python guru—I've only written around 50,000 lines or so.

I was attracted to Ruby because of Rails. I haven't written much more than a few thousand lines, but from what I have seen, it's a nice language with a few more rough edges than Python.

The primary difference I see between the languages are:

- Their age: Python is a few years older and so has had more time to be cleaned up.
- The approach of the designers: Python is aimed at succinct code that tends to have only one method to do any given task; Ruby is more of a kitchen-sink language, and so programmers have many equivalent options.

I think the two languages are converging toward much the same feature set, but with stylistic differences. Python is gradually and carefully adding things, whereas Ruby is slowly discarding things that are redundant.

Now let's get to the points Luke Kanies brought up.

Point 1: In Ruby, Everything Is an Object

Python 1.x had two kinds of things: primitives and objects. Python 2 introduced the "new style class" and has relentlessly driven the language to the point where "everything is an object."

Within Python 2.x, old style classes still exist, even though for later releases primitives such as

integers and strings are now objects. We are told that the journey will be complete in Python 3.x.

Luke provides this example of how easy it is to get information about an object in Ruby:

```
[Class, "a string", 15, File.open("/etc/passwd")].each { |obj|
puts "'%s' is of type %s" % [obj, obj.class]
}
```

Here is essentially the same code in Python:

```
for x in [object, "string", 15, file('/etc/passwd')]:
        print "%s is a %s" % (repr(x), x.__class__)
```

which yields

```
<type 'object'> is a <type 'type'>
'string' is a <type 'str'>
15 is a <type 'int'>
<open file '/etc/passwd', mode 'r' at 0xb755f4a0> is a <type 'file'>
```

Point 2: According to Luke, in Ruby There Are No Operators: All Operations Are Defined by Functions Associated with the Objects Involved

This was not true of Python 1.x, but it is pretty much true in Python 2.4 and above. As everything becomes "an object," it will be uniformly true, in the sense it is in Ruby. That is, all operators in the language are implemented using special methods attached to the operands and if the method doesn't exist, the interpreter throws an exception.

For example, in Python, x + y is executed as x.__add__(y) or y.__radd(x)__ .

Point 3: Introspection

Introspection allows you to find information about objects as they are running. This contrasts sharply with languages in which programs must be (almost) completely specified at compile time. Both Ruby and Python have extensive support for inspecting the visible state of everything.

I'll only mention two Python features here: the builtin function dir() and the __doc__ attribute.

dir(foo) returns a list of all attributes and methods attached to its argument. That's all there is to it:

```
dir(1.0)
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__
div__', '__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
'__init__', '__int__', '__le__', '__long__', '__lt__', '__mod__', '__
mul__', '__ne__', '__neg__', '__new__', '__nonzero__', '__pos__',
'__pow__', '__radd__', '__rdiv__', '__rdivmod__', '__reduce__', '__
reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__
rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__str__', '__sub__
', '__truediv__']
```

The __doc__ attribute contains printable documentation about the object:

```
print 1.0.__doc__
float(x) -> floating point number

Convert a string or number to a floating point number, if possible.
```

Point 4: In Ruby, Many Objects Know How to Iterate Over Themselves

Rather than writing a conventional, imperative programming style loop, you can say something like this:

```
object.each { |x| do something with x }
```

This is a good thing and such facility has been added to the 2.x versions of Python, with capabilities increasing with each point release. This is a *very* light survey of what Python provides.

Python uses the existing for loop syntax similarly to the way that Ruby uses the each method. The Python for loop looks roughly like this:

```
for <list of variables> in <arbitary iterator>:
        stuff to do
```

which is equivalent to calling a block of code with parameters set to the variables in the list. So the Python for is equivalent to Luke's Ruby code.

Initially, the <arbitary iterator> was any sequence—a list, tuple, or string. This has been extended in Python 2.x to anything that satisfies the "iterator protocol" (see below).

In addition, several interesting additions to Python make it easier to use this construction in more compact, yet clear ways.

LIST COMPREHENSIONS

List Comprehensions, added in 2.1, are lists defined by one or more sequences. List Comprehensions generalized the ideas of map(), zip(), and friends. For example,

```
[x*x/2.0 for x in range(1,1000) if x%3 == 0]
```

GENERATORS

Generators were added in 2.2. A generator looks like a function except that it contains the keyword yield instead of return. A generator returns an object that has a next() method. Calling the next method either returns a value or raises the StopIteration exception. (StopIteration is what now stops the for loop in Python.)

```
def myrange(bot, top):
        while bot < top:
                yield bot
                bot += 1

for x in myrange(1,20):
        whatever
```

GENERATOR EXPRESSIONS

Generator Expressions, added in 2.4, are essentially lazy list comprehensions. That is, a list comprehension realizes the entire list, but a generator expression just returns the next element. This allows simple iteration over infinite sequences. To write one, replace the square brackets with parentheses:

```
(x*x/2.0 for x in range(1,1000) if x%3 == 0)
(line for x in file('/etc/passwd') if x[0] != '#')
```

ITERATOR PROTOCOL

Iterator Protocol, added in 2.2, is a general method for objects to become iterators. This provides functionality similar to Ruby's each method. In brief, if an object defines two special methods, it can replace the list component of a for loop. These methods are:

- __iter__(self), which returns a function that acts as an iterator
- next(self), which returns the next item from the object or raises the StopIteration exception

The imminent release of Python 2.5 will continue this trend and promises support for co-routines by expanding the capabilities of generators and unifying Python's exception-handling code.

Iterators are really cool. They make code both compact *and* easy to understand.

Point 5: Ruby Has Code Blocks

Ruby code blocks are similar to anonymous functions that can be passed to methods for execution. I say they are similar because the parameters of a Ruby code block are existing local variables; those local variables are then used within the block and their values are changed as a result of the block's execution [2]. Ruby code blocks can be arguments of methods, can be assigned to variables, etc.

Python functions can be manipulated similarly. They can be passed to and returned from functions. They can be assigned to variables. And they can be executed in loops that are controlled by iterators. Python function parameters are always local variables, so they are less prone to unintended side effects.

In Python, def (function definition) is an executable statement that returns a function. This makes it easy to write closures:

```
>>> def foo(x):
...         def tmp(y):
...                 return y < x
...         return tmp
...
>>> a = foo(10)
>>> a
<function tmp at 0xb7562b1c>
>>> a(4)
True
>>> a(12)
False
```

Summary

I agree strongly with Luke that the features he outlined in [1] are excellent features that should be in all modern programming languages. I also agree that they are good reasons to use Ruby or Python.

REFERENCES

[1] Luke Kanies, "Why You Should Use Ruby," *;login:* (April 2006).

[2] Dave Thomas, *Programming Ruby: The Pragmatic Programmers Guide,* 2nd ed. (Pragmatic Programmers, 2005), p. 51. This scope issue is being debated within the Ruby community and will probably change in some subsequent release.

[3] See www.python.org for documentation for all releases, as well as the code. Many features (implemented and proposed) are described in the PEPs.

[4] David M. Beazley, *Python, Essential Reference,* 3rd ed. (Sams Publishing, 2006).

OSDI '06

7th USENIX Symposium on Operating Systems Design and Implementation

Sponsored by USENIX, The Advanced Computing Systems Association, in cooperation with ACM SIGOPS

Join us in Seattle, WA, November 6–8, 2006, for the 7th USENIX Symposium on Operating Systems Design and Implementation. OSDI brings together professionals from academic and industrial backgrounds in what has become a premier forum for discussing the design, implementation, and implications of systems software. Sessions include:

- Runtime Reliability Mechanisms
- OS Implementation Strategies
- Distributed Storage and Locking
- Program Analysis Techniques
- And more

See the full program at **www.usenix.org/osdi06**. Don't miss an outstanding program covering the best systems software research and practice. Register online by October 16, 2006, and save!

OSDI '06 is co-located with the 3rd USENIX Workshop on Real, Large Distributed Systems (WORLDS '06), which will take place on November 5. The Second Workshop on Hot Topics in System Dependability (HotDep '06) will be held on November 8, immediately following OSDI '06. See www.usenix.org/events for more information and to register online.

Nov. 6–8, 2006, Seattle, WA

www.usenix.org/osdi06