Python / Ruby Comparison

Ruby and Python are both agile, interpreted languages with an object oriented design and very large communities. Although the languages share some ideas, syntax elements and have almost the same features the two communities have nothing in common. Not only because the Ruby language is "made in Japan", also because there is some kind of hatred between both communities.

Ruby people hate Python because of missing blocks, indention as syntax element, no native regular expressions and the fact that Python has some functions like `len()`, `abs()` etc. which aren't object orientated in their mind. Python people on the other side don't like Ruby because it breaks every rule of The Zen Of Python, It has much too few braces, functions don't look like first-class functions and they miss multiple inheritance :)

But what's really the truth? Most of those points aren't absolutely true or make sense if you know why the developers have chosen that kind of implementation.

Important: this document describes the current stable version of both Python and Ruby. At the moment this is Python 2.5 and Ruby 1.8. If you think I'm wrong in one point send me a mail and I'll try to correct that.

Table Of Contents

Comparison Table

Feature	Python	Ruby
Language Features		
Namespaces	yes	yes
Constants	no*	yes*
Generators	yes	yes*
Iterators	yes	yes*
Coroutines	yes	yes
Continuations	no	yes
Classes		
Multiple Inheritance	yes	no*
Interfaces	no*	no*
Class Includes*	no	yes
Nested Classes	yes	yes
Properties	yes	yes*
Operator Overloading	yes*	yes*
Functions		
First-Class Functions	yes	yes*
Anonymous Functions	yes*	yes*
Keyword Arguments	yes	no*
Closures	yes	yes
Decorators	yes	no
Collection Objects		
Tuples	yes	no
Lists	yes	yes
Hashes	yes	yes
Strings		
String Type	yes	yes
Char Type	no	no
Symbol Type	no	yes
Immutable	yes	no*
Interned	yes	no*
Heredocs	no	yes
Multiline Strings	yes	yes
Unicode Support	yes	no
Regular Expressions		
Regex Literal	no	yes
Named Groups	yes	no
Lookaheads	yes	yes
Lookbehinds	yes	no
Yes/No Pattern	yes	no
Unicode Support	yes	yes*

Details

Often it's not possible to just say yes or no. So here are some explanations to the comparsion table above.

Interfaces in Python

Currently python doesn't ship an interface system (although some libraries implement interfaces). Maybe this will change with Python3.

Ruby Class Includes

Ruby doesn't provide multiple inheritance. But it's possible to mix-in a module into a class defintion which means that methods defined in that module are part of the class then. In combintation with defined functions a mixin can add new functionallity to a class. For example, if a class has a working `.each` method, mixing in the standard library's Enumerable module gives the class sort and find methods.

Python Operator Overloading

In Python you can overload each operator using an alias function name except `and`, `or`, `not` and the assignment operator for names (not attributes).

Ruby Operator Overloading

In Ruby you overload operators by defining a method with the operator as name. You can't overload some operators like the call operator or `!`, `not`, `&&`, `and`, `||`, `or` or `.`, `!=`, `..`, `...`, `::` and some others.

Ruby Properties

Because instance variables aren't accessable from outside of a class Ruby uses getter and setter methods called `variable_name` and `variable_name=` to rebind them. In fact they are just a form of operator overloading. Because you can add, remove or modify those getter and setter methods it's very easy to implement properties although no Ruby programmer would call them properties.

Ruby First Class Functions

Ruby doesn't really have functions at all. Ruby binds functions to an object (if not defined in a class a function is bound to the module) and gives them a special syntax then. If you call a method using the name (you don't need braces) it really gets called and returns the result:

```
irb(main):001:0> def foo
irb(main):002:1>  42
irb(main):003:1> end
=> nil
irb(main):004:0> foo
=> 42
irb(main):005:0> def bar x
irb(main):006:1>  42 - x
irb(main):007:1> end
=> nil
irb(main):008:0> bar 23
=> 19
```

This makes it impossible to pass methods around. But you can get the object behind a method:

```
irb(main):009:0> bar_method = Class.method(:bar)
=> #<Method: Class(Object)#bar>
irb(main):010:0> bar_method.call 23
=> 19
```

The method `Class.method` looks up the name of the method in the current namespace and returns the object. You can then pass around that method object. So methods are objects too. But nevertheless we can't call that first-class functions. Sorry for that.

Also methods defined in methods aren't defined in the namespace of the method but in the parent namespace if the outer method is called (like in PHP). If you want an anonymous function you have to use `Proc.new`.

So: methods aren't functions. Functions are code blocks bound to an object (the `Proc` object), functions are first class, methods not.

Python Anonymous Functions

In Python functions are first-class functions which mean that the behave like any other object. If you create them you'll bind them to a variable in the current namespace. So it's on the one hand possible to create a function inside of another function so that you can call it only from that function. But that's not really anonmous since it has a name. One the other side you have the `lambda` expressions which allows you to bind expressions to anonymous functions.

Ruby Anonymous Functions

Like in Python you can have anonymous functions in Ruby too. But not using the normal method syntax, you have to use the `Proc.new` method or the `lambda` method which work basically the same (there are a few differences from what I know). And in Ruby you also have code blocks which can store some code which you can call. That's some kind of anomyous function too.

Ruby Keyword Arguments

Ruby doesn't have keyword arguments but some libraries (as Rails) use hashes for that purpose. While it doesn't provide all features like in Python. For Ruby2 keyword arguments are on the todo list.

Constants in Ruby and Python

In fact neither Ruby nor Python have real constants. But it a variable starts with an upper case letter in Ruby the interpreter will warn everytime you bin the variable new. In Ruby constants have an own lookup which works different than the local variable / method lookup. In Python you just bind a value to a name.

Ruby Generators and Iterators

If you know iterators from other languages like Python or PHP you might wonder why it doesn't really exist as such. Most objects support methods like `.each` which is passed a code block that is executed for each item in the sequence (ie: implemented for arrays). In combination with continuations you can implement iterators and generators.

Ruby Strings

In Ruby strings are mutable (like in PHP or C). Because of that strings are a lot heaver than in Python for example where all strings are interned. So you have a second data type for symbols which work like Python strings, except that they don't provide string manipulation functions.

Ruby Unicode Regular Expressions

Because unicode isn't supported at the moment by Ruby it doesn't mean that you can't use them in regular expressions. In fact there is utf-8 support in the current Ruby Regular Expression engine.

Other Differences

There are also other differences between Ruby and Python. Here a list of things that are completely different :-)

Boolean Values

In Python you have a boolean data type called `bool` which is a subclass of `int` and which can either be `True` or `False`. Those two values are the only instances of this class, they behave like singletons. Every Python object can either be `True` or `False` which is defined by overriding the `__nonzero__` method of an object. Per default all empty objects (empty lists, dicts, tuples, strings etc) are `False`. Same for `0` and `0.0`.

In Ruby the situation is completely different. There is no boolean data type but there are two singletons (`true`, instance of `TrueClass` and `false`, instance of `FalseClass`). Everything but `nil` and `false` is `true`. It's also not overrideable.

Namespaces

Ruby and Python have different ideas about namespaces. In Python there is a stack of namespaces, everytime you assign a name it's saved in the highest namespace. In Ruby you have implicit method lookup, an own namespace for local variables, an `@`-prefix for instance variables, and `@@` prefix for class variables and a `$`-prefix for global variables. And global variables are really global. Each module has access to them. That's something that doesn't really exist in Python where the only way to define a inter module global is to patch the special `__builtin__` module.

Iterables

Ruby and Python have different concepts of Iteration. In Python an object is iterable if it defines a method `__iter__` which returns in iterator. An iterator is an object with an `__iter__` method that returns a reference to itself and a `next` method which returns the next item or raises `StopIteration`.

In Ruby you normally iterate passing a code block to the `.each` method of an object which is called with the item for each item in the sequence.

Whereas both work nearly the same for most of the users Python programmer will find the Ruby way of iteration darn limited because they are used to wrap iterators. (Python ships an module called itertools for example) In fact it's possible to emulate the Python iterator/generator system by using callcc. The generator.rb module in the Ruby distribution does that for example.

Object Orientation

Haaa! The best part. Because it's also the most discussed one. Python and Ruby are both object orientated. Nevertheless most people call Python **not** object orientated which isn't the case. But Python has a different idea of it. In Ruby the list of methods which are defined per default for the plainest object (`Object`) looks like this:

```
irb(main):001:0> Object.methods
=> ["inspect", "autoload?", "send", "class_eval", "clone",
    "public_methods", "method", "protected_instance_methods", "__send__",
    "private_method_defined?", "equal?", "freeze", "methods", "instance_eval",
    "dup", "include?", "private_instance_methods", "instance_variables",
    "extend", "protected_method_defined?", "const_defined?", "instance_of?",
    "name", "eql?", "public_class_method", "object_id", "hash", "id", "new",
    "singleton_methods", "taint", "frozen?", "instance_variable_get",
    "constants", "kind_of?", "to_a", "ancestors", "display",
    "private_class_method", "const_missing", "type", "autoload",
    "<", "protected_methods", "<=>", "instance_methods", "==",
    "method_defined?", "superclass", ">", "===", "instance_variable_set",
    "instance_method", "const_get", "is_a?", ">=", "respond_to?", "to_s",
    ">=", "module_eval", "class_variables", "allocate", "class",
    "public_instance_methods", "tainted?", "=~", "private_methods",
    "public_method_defined?", "__id__", "nil?", "untaint", "included_modules",
    "const_set"]
```

In Python like this:

```
>>> dir(object)
['__class__', '__delattr__', '__doc__', '__getattribute__', '__hash__',
 '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__str__']
```

That in fact looks different. Let's have a look at the input line of both snippets (I used irb for the first, and the Python shell for the latter). In Ruby you can use the method `methods` which is defined for each object to get a list of *methods*. In Python I can use the builtin function `dir()` to get a list of all *members* of an object. It's possible to override the `methods` method in Ruby, but not the behavior of the `dir()` function which is just meant for object introspection.

It's pretty interesting that the amout of per default methods in Ruby is 73. That's really a huge number of methods which are taken, eg: method names you can't use for something else without destroying functionallity. For example you can't really name a method `display` for something else than printing the value of the object to an output buffer. Basically you would be able to do that but someone else working with that class would expect a different behavior.

Now let's have a look at the Python class. Because Python objects use the same namespace for methods and members in that list are members *and* methods. The underlines indicate that those methods are special and defined in the Python specification. But normally you wouldn't call them because there are other ways to do that: the builtin functions. Which do more than just calling the attribute methods etc.

For example those things do exactly the same:

```
>>> class Foo(object): pass
...
>>> foo = Foo()
>>> type(foo)
<class '__main__.Foo'>
```

```
>>> Foo = type('Foo', (object,), {})
>>> foo = Foo.__call__()
>>> foo.__class__
<class '__main__.Foo'>
```

And that's basically where the difference between Ruby and Python is. In Ruby you would say: an object has a length if it has an method called `length` and is iterable if it has a method called `each`. In Python you would say: An object has a length if it has a method called `__len__` which I can call using `len(obj)` and is iterable if it provides a method called `__iter__` which returns a valid iterator.

Many people don't like those builtin accessor functions for the special methods in Python but once you learned to live with them you don't have an problem with them. In fact they do more than just calling those internal methods. For example the `iter()` function which returns the iterator of an object can give you an iterator also for object which don't have a method called `__iter__` as long as it has a method called `__len__` and a method to get an item for an index, called `__getitem__`.

Metaclasses

Ruby and Python have strong support for metaprogramming. And the support for metaprogramming is part of the language itself and not added by extension libraries like in PHP.

In Python the class of a new class is resolved after the class definition. Python looks for a `__metaclass__` variable in either the class definition, the global namespace of the current module or if it finds nothing, it uses the metaclass of the first class the new class is inheriting from. If the user doesn't inherit from a class it uses the default metaclass which is `classobj` for current Python versions. In Python 3 this will become `type`.

In Ruby you have a special syntax to get the singleton of an object which you can use to modify the class of a class.

Of Methods and Functions

In Ruby it's easy. You define methods using the `def` keyword like you define functions and methods in Python. Methods aren't first-class but you can get the Method object behind the method easily. Additionally you can create anonymous functions using the `Proc.new` method which have a method `call` for calling it. Ruby provides an variable `self` which automatically points to the instance of the class the method is defined in.

In Python you just have functions. But the metaclass (the class of a class in fact) of classes wraps functions in method proxy objects. That means a function inside of a class (without applied decorators like `classmethod` or `staticmethod`) is passed an first parameter (usually called `self`) which points to the current instance of the class. That explicit first parameter is quite irritating at the beginning but makes it possible to pass functions and methods around without problems.

You can for example bind an instance method of an object to another class (self would still point to the instance), or a class method to an instance (self would point to the class), or you can bind the function behind the instance method to another class and self would point to that other class etc.

Syntax

If you compare Ruby and Python code they look similar. In fact they don't have anything in common but some keywords they share. In Ruby the syntax is much more dynamic. Because you can call functions without parenthesis Ruby has a very low amount of statements (like print/import/exec in Python) but functions. Also Ruby has much more possibilities defining a string (including recursive string patterns and heredocs). There is also a difference in expressions. In Ruby everything an expression whereas Python doesn't see assignments as expression. (A lambda in Python can't assign something to an attribute for example)

The Python syntax is defined in the language reference which means that Python doesn't rely on the original C-Implementation. The Python syntax is also very strict in comparison with Ruby where the meaning of an syntax element (for example heredocs vs. lshift operator, fancy strings vs modulo operator, regular expressions vs. division operator) is often guessed (the Ruby interpreter keeps track of defined methods and variables so that it can lower the amount of wrong matches).

Performance

That's a hot topic. Because Python and Ruby are awfully slow in comparison with other languages. But if you compare Python with Ruby, Python wins. Currently. Together with Murphy from the German Ruby forum we found out that some parts of Ruby are badly implemented (for example the `.index` method of strings is always slower than a regular expression) which means that not the dynamic nature of Ruby leads to a slow language. Also there is no bytecode caching at the moment which will be there with Ruby2. So I think that in the next few years Ruby has a good chance to get faster than the current Python C implementation.

The Python community on the other side knows about a project called pypy which tries to write a Python interpreter in Python itself which is able to translate a restricted Python subset (called rPython) to other languages like C, javascript or into .net assemblies. While the interpreted version of pypy is slower than the C implementation for the moment, compiled rPython code is faster.

Documentation

Documentation is very important for the success of a language. And often people complain (at least I hear people complain) about the Ruby documentation. While that's true for the C code of Ruby, the Ruby documentation for the standard library very good. There is not a single document about the Ruby Specification in English. It's impossible to find out what's a bug in the parser and what not because nobody ever wrote down how the parser is suppposted to parse. But now, where Ruby is part of the web2.0 movement I suppose that will improve in the next two years.

On the other side is Python. The documentation is complete for both the language itself, the standard library and inline comments in the C code.

Community

Ruby and Python have a very active and friendly community (although the Ruby community a bit more active and a bit more friendy :D)