A comparison of object oriented scripting languages: Python and Ruby

Kaustub D.
kd@cs.washington.edu
David Grimes
grimes@cs.washington.edu

December 18, 2001

Of late, scripting languages are becoming popular amongst programmers. This is mostly due to the proliferation of the Internet, where scripting languages offer abundant opportunity, e.g. cgi scripts, string and data processing etc. The main reason for the use of scripting languages is their simplicity, power, and ease of use.

Python and Ruby are two such scripting languages which try to embody the object-oriented paradigm and make it accessible to the general script programmer. Both take diametrically opposite views in bringing the object-oriented paradigm to the procedural world of scripting. Python starts out as a procedural language and adds object-oriented features for a programmer who wants to use them, while Ruby starts out as a pure object-oriented language and then makes the code look like procedural program, thus making it accessible to the general user. In this report we compare two approaches and point out the advantages or disadvantages of each approach. We have tried to make unbiased judgments, as we were not familiar with either of the languages before the start of this project.

1 Introduction

1.1 Why scripting?

The first question most people would ask is, why do we need scripting languages in the first place? We might just always use C++ or Java! Scripting languages are designed for different kind of applications than system programming languages. They are more suitable for gluing together pre-existing

components, or for providing interfaces between two components. Scripting languages are usually typeless, which makes them far more flexible as there are no restrictions as to how things are used. Scripting languages are interpreted which allows for fast turnaround of code. They are higher-level than system programming languages, as a single instruction does a lot more work, mostly due to the fact that primitive operations in scripting languages are far more powerful. (c.f. [**?**])

Though the performance of scripting languages is much worse than system languages, it is usually a non-issue. This is because usually applications written in scripting languages are typically relatively small and performance is dominated by the larger components (written in system languages) they interconnect. Due to these features, scripting languages allow very rapid development for applications like GUIs, string processing or web scripts.

1.2 Python

Python is an object-oriented interpreted scripting language started in 1990 essentially for distributed systems management tasks, but it quickly became popular for a wide range of programming tasks. Around 1995 it became even more popular due to it's popularity in Internet related programs. It's design in the most general sense borrowed heavily from other scripting languages, particularly Perl and Tcl. Obviously it also borrowed from object oriented languages such as C++ and Smalltalk. Python also has hints of functional languages like LISP.

1.3 Ruby

Ruby is a relatively new programming language which has recently come in vogue. Though it was created in 1995 in Japan, it has come to the US only in 2000. It is an interpreted, scripting, pure object-oriented language, which can masquerade as a procedural language.

Ruby was designed to be more powerful than Perl and more object-oriented than Python. It is heavily based on Perl and is rumored to be Perl better than Perl. The main focus in its design is that object-oriented programming should be quick and easy while still retaining the power of Perl. The syntax was in part inspired by Ada, and there are lots of features borrowed from other object-oriented languages like Eiffel, Smalltalk etc.

2 Object Orientedness

In this section we differentiate between the object-oriented paradigms embodied by the two languages. We contemplate on how the various features actually affect the language, especially with respect to simplicity and ease of use.

2.1 Purity

There has been much debate if Python is truly a pure object oriented language. With respect to data representation, Python is a pure OO language, since every variable is a reference to an object. However, with respect to program design, Python allows both procedural and object oriented designs. It is possible to write functions in addition to methods. Functions do in fact operate on objects, and in fact are represented as function objects themselves. Yet, one major point in the debate is that the programmer is *required to* use functions because there is much built-in functionality in python that is implemented as functions.

In contrast, Ruby is pure object oriented language (like Smalltalk). As in Python, in Ruby everything is an object. There are no built-in types like `int` in Java. The part which is different from Python is the fact that all operations are messages to objects. i.e. there are no functions, only methods. Though it is possible in Ruby to write a method without defining a class, what Ruby does, is that it makes this method a private method of the Object class. This allows Ruby to look like a procedural language, even though it is a pure OO language.

Due to this difference, we have uniform access in Ruby, which we cant have Python. e.g. to get the absolute value of -3 you would say `-3.abs` in Ruby, while you would do `abs(-3)` in Python. Even though the difference may not be so striking, we can see that in Ruby all the method invocations will be of the first type, while in Python there will be a mixture of the first type and the second which may lead to confusion. This feature contributes to the readability and the simplicity of syntax. For a more extreme example of how code becomes ugly due to lack of uniform access, we can look at how Java solves the problem of finding the absolute value. Since `int` is not an object, the `abs` method has to be external function. But since we can't have functions in Java, we force a function like method by defining a static method `Math.abs` to do the job.

Another major problem with Python is that since there is no special syntax to distinguish methods and functions, methods are basically imple-

mented as functions which have a first parameter `self` which is a reference to the object on which to call the method. Generally, this is done implicitly by OO languages, but in Python it must be done explicitly by the programmer. This makes it look as if the object oriented support was tacked onto Python rather than inherent in the design.

2.2 Inheritance Model

Python's inheritance model allows for multiple inheritance. A classic issue with multiple inheritance is name conflicts between super-classes. Python "solves" this simply by having the programmer annotate the order of super-classes, and then search class by class for the name until it is found. However there are some caveats to this approach. Most importantly it may be impossible to order the super-classes such that you get the desired member references. Another possible problem is that many programmers may not used to the idea of precedence in inheritance declarations.

In contrast, Ruby features single inheritance only. This is a design decision to avoid the complexities that result due to multiple inheritance. To add support for multiple inheritance, Ruby has the concept of modules. A module is like a partial class definition. Modules can be used as mix-ins in classes. (This is similar to interfaces in Java, except that in interfaces you cannot implement methods.)

When a class includes a module, it is almost as if the module is a super-class of the class. Ruby creates a proxy class from the module and adds it in the inheritance chain as the direct super-class of the class that included it. If many modules are included, then all of them are put one after another in an inheritance chain.

2.3 Access Protection/Information Hiding

Like in Java, Ruby supports private, protected and public types of access to the elements of a class. By default all methods are public except the `initialize` method, and all instance variables are private.

Python has a weird solution for this problem. Although it initially had no access protection a mechanism has been added called *name mangling* which renames class members starting with "__" (two underscores) to "_<classname>__<original_name>". Thus references to members outside the class cannot get the same mangled names simply by referencing `__foo`. Note that in fact it is still possible to get at the class member from outside simply by doing the name mangling yourself. Python doesn't support the

notion of protected class members at all. The main reason for lack of access protected in Python is probably the very simple class member representation scheme which is simply a single dictionary mapping names to methods or objects. This dictionary is stored in the class field `__dict__` which exists in all objects.

2.4 Dynamic Typing

In both Ruby and Python, all types are dynamic. So you do not need to give types to variables when you initialize them, and there is no question of declaring variables. The advantage of dynamic typing is that it is much more flexible and naturally allows for polymorphism. It also allows for simpler and clearer syntax. The disadvantage of dynamic typing is that the compiler cannot statically check whether there will be typing errors in the program, which could result in "method missing" errors at run-time.

From the above discussions, we see that Ruby emerges as the winner, hands down, when it comes to using the object-oriented paradigm. This gives Ruby much of its simplicity and ease of use.

3 Look and Feel

3.1 Syntax

One of the main advantages of Ruby is the simplicity and readability of its syntax. Ruby code is far less cryptic than most other programming languages. The block are delimited by begin and end like in Ada, with each statement on a single line. This reduces a lot of syntactic clutter like {}; etc. In many places () can be omitted from method calls where the context is obvious. This helps in further reducing the clutter. The fact that Ruby is a pure OO language also contributes to the readability of its syntax.

Python also seeks to reduce general syntactic clutter in order to improve readability. One idea Python employs is blocking by indentation in a fashion similar to Haskell. This enforces what it generally considered desirable practice, and results in slightly more compressed code. Python also requires single line expressions thus eliminating the need for an end of expression mark (";").

As both Python and Ruby were designed from the ground up as simple scripting languages, both are very comparable in syntactic simplicity. That said there are certainly differences in readability due to other more funda-

mental differences in the object-orientedness, as we have discussed earlier.

3.2 Learnability

Another advantage of both Ruby and Python is their learnability. For most programmers, the principle of least surprise ensures that the syntax will usually mean what you expect it to mean (from experience with C, Perl etc.). And it is quite possible to start writing Ruby or Python code within a few minutes of learning about the syntax.

Both Ruby and Python also allow for fast prototyping due to the numerous built-in libraries. It also turns out that the code is far more compact than the corresponding C++ or Java code due to some of the inherent features of the language.

Python claims an additional high learnability due to it's ability to accommodate both procedural and object oriented paradigms seamlessly, although we will saw the price of this in the previous section.

4 Basic Language Features

In this section we will have a look at how the basic language features are implemented in both the languages. We will also point out some of the issues which arise due to the design choices, and propose solutions in some cases.

4.1 Ruby

4.1.1 Classes and Objects

In Ruby you can declare a new class as follows, which creates an object of type Class at run-time, with the superclass `super`. The name of the

```
class classname [ < super ]
  body
end
```

class *has to* start with a capital letter. Within the body of a class, the statements are executed as the definition is read, while the methods are stored in the method table associated with the class. This means that you can execute statements within a class without actually having them in any method. These statements are run exactly once when the class is defined. It is not sure whether this is an acceptable feature. It does help Ruby act in

a procedural sort of way. But we think that in general it is not a good idea to just allow code to be written outside of methods. A better alternative would be to have the main function which Ruby will execute.

Ruby's classes are not closed. So you can add methods to classes later on. This means that you can also add methods to the built-in classes or worse, redefine their methods. It may not be a good idea, to allow programmers direct access to internal details of important classes like Object or Kernel. A way of getting over the problem could be defining packages containing classes, and allowing the user to add methods to classes in the same package.

A new object of this class is created by calling the `new` method of the class. This creates the new object, and calls the `initialize` method with all the arguments passed to `new`. The `initialize` method is by default a private method of the class. If the class overrides `new`, without calling `super`, then no objects of that class can ever be created!

Ruby also allows you to freeze objects by calling the `freeze` method on them. The values inside a frozen object cannot be changed.

4.1.2 Variables

Ruby has three kinds of variables: local, instance and global. Ruby differentiates between different variables and constants by having them to be lexically different. Instance variables begin with `@`, globals with `$` and constants with a capital letter.

Though global variables are in general considered harmful, the main advantage offered is to allow access of the environment and system variables. Another use is Perl like regular expression matching. To allow for debugging of programs, Ruby allows global variables to be traced. i.e. you can specify a procedure object with each global which gets invoked every time the global is changed.

There is an issue with the use of instance variables in Ruby. Since there are no declarations in Ruby, instance variables are never declared. Instead they are created whenever they are initialized within a method! Since all the methods of a class are not located in the same place, this is bound to cause problems as far as readability of code is concerned. We think that instance variables should be declared, (but not initialized) within a class just to allow for better readability of programs.

4.1.3 Methods and Operators

In Ruby a method is defined as shown below, where `arg` denotes the arguments,

```
def methodname [ ( [arg [=val]] [, *vararg] [, &blockarg] )]
  body
end
```

some of which may be assigned default values. `vararg` denotes the remaining arguments which would be put in an array, while `blockarg` denotes the block which may be passed to this method as an argument. (We will look at blocks later). In Ruby, each method returns the value of the last executed statement in the method. (there is also a `return` statement if needed.)

In Ruby all operators are methods, and hence each class can define its own operators. Any of these operators or methods could be further redefined anywhere in the program. Hence operator overloading is trivial.
An example of Ruby code is given below to illustrate classes and methods:

```
class Doubler
  def initialize name
     @name = name
  end

  attr_writer: name;

  def double x
    print "Hi my name is ", @name, \
          ", and I'll double your variable", "\n"
    2*x
  end
end

foo = Doubler.new "Fred!"
foo.double 2
```

OUTPUT:

```
Hi my name is Fred, and I'll double your variable
4
```

Ruby also offers convenience methods to do attribute reading and writing. For example, the `attr_writer` defined in the above class lets us do: `foo.name = "Bob"`. Note: this is not accessing the attribute directly, but using the setter method to access the attribute transparently.

A problem encountered while doing method dispatching, is the fact that Ruby syntax allows a local variable and a method name to look lexically alike. How does Ruby decide whether the current reference is a variable

or a method dispatch? To disambiguate between a variable and a method, Ruby keeps a track of all the symbols seen so far which have been assigned to, assuming that these symbols are variables. Whenever it encounters a symbol being used as if it could either be a method or a variable, it looks in the list of variables to see if it was assigned before. If it was, then it treats it as a variable, else it considers it to be a method.

4.1.4 Standard Types

Ruby does support quite a few literal types. This includes Numbers, Strings, Ranges, Regexp amongst others. The Numbers are of the following types: Floats, FixNums and BigNums. BigNums are infinite precision integers, and Ruby converts FixNums to BigNums and vice-versa transparent to the user.

4.2 Python

4.2.1 Classes and Objects

The following code creates a class object in the Python interpreter, and instantiates an object of type *classname*.

`class` *classname [(superclass1, superclass2, ..., superclassN)]:*
  body1
  body2
  ...
  bodyN

class_instance = classname()

Very similarly to Ruby, Python allows for arbitrary statements in the body of a class. However, unlike Ruby, Python classes are closed after a block is read. The class however can be modified as described in the section on reflection. In fact it is possible to redefine classes at run-time (without a warning), even the primitive object classes, can be redefined within a module. As such a thing seems certainly dangerous, it is important to not pick common names for user defined classes. We feel this places additional burden on the programmer to manage a possibly large name-space.

4.2.2 Variables

Python's basic variable types are: local, instance, and global. Local variables are local to a function or method execution and are deleted after the function or method has returned. Like Ruby, these variables don't need to be declared, only initialized. Instance variables operate in a standard way with the syntax *object.instanceVar*. This syntax is required even within a method declaration using the method parameter `self` as the object. Although it requires additional typing it does clear up the ambiguity between local and instance variables. Yet then we have global variables, which are global in the scope of a module (or name-space).

Python has rules regarding name-spaces and scoping, although these rules have changed from version to version. One difficult area is that function scopes are treated textually, so even if a function is passed out of a module and into another the name-space will map into the original module. Python has traditionally been dynamically bound, but there is some movement in the development community to try to implement static bound variables. Here static bound means at script read-in time, instead of compile time.

4.2.3 Functions, Methods and Operators

In the most basic sense Python functions are methods are defined identically, with the only difference being that methods are passed the called-object as the first parameter.

```
def double (x):
  return 2*x

class Doubler:
  name="Fred"
  def double(self, x):
    print "Hi my name is ", self.name, \
          ", and I'll double your variable"
    return 2*x

fred = Doubler()
fred.double(2)
double(3)
```

OUTPUT:

```
Hi my name is Fred, and I'll double your variable
```

```
4
6
```

Thus when the call fred.double(2) is made the interpreted passes `fred` as `self`.

Operators are simply shortcut to methods. For instance the '+' operator ends up calling "__add__(self, x)" on the object with which it associated. Thus operator overloading is very simple.

4.2.4 Standard Types

Python's data model provides the programmer with various high-level of abstraction data types. Due to the flexibility afforded to Python by it's dynamic typing the large number of primitive types found in C or Java is collapsed to just four "primitive objects": word-sized integers, infinite precision integers, word-sized floating point numbers, and strings. In this sense it's very similar to Perl which also has very few primitive types. Since python represents all data types as objects there is little significance in differentiating between various sized integer or floating point numbers, as the size of the base object overwhelms these small differences. A single character type would also not be of significance since it is a special case of the string in a dynamically typed language.

5 Advanced Language Features

In this section we will look at some of the advanced features of Ruby and Python from the programming languages point of view and compare between the two languages.

5.1 Higher-Order Functions

Both Python and Ruby feature higher-order functions which represented as objects. Python uses things called lambda expressions while Ruby uses procedure objects. Below we can see how the same function is written as an object in the two languages.

Ruby

```
p = proc do |x|
      print "Hello ",x ,"\n"
    end

p.call "Fred!"
```

Python

```
p = lambda x:
      print "Hello ", x
p("Fred!")
```

In Ruby the `call` method inside a procedure object executes the method. While in Python, an anonymous function object is created.

5.2 Blocks in Ruby

Ruby has a very powerful concept called *code blocks*. It is nothing but a bunch of statements between braces or a `do/end` pair. In the previous subsection, you saw an example of a block being converted to a procedure object by the method `proc`. The block usually follows a method invocation. A block may take optional arguments `a1, a2, ...` etc. and may return a

methodinvocation `do | a1, a2, ... |`
  statements
`end`

value (which is the value returned by the last statement executed). The power of blocks comes from the fact that these blocks are passed to the method which is invoked. The method may either take the block as an argument and convert it to a procedure object (the &`blockarg` argument in method), or it can invoke the block using the `yield` statement, which may optionally pass arguments inside a block. An example of this is as follows:

```
class Array
  def each
    for i in 0..(size-1) do
      yield self[i]
    end
  end
end

[1,2,3].each { | x | print x*x,"\n" }
```

Output:
```
1
```

```
4
9
```

The above code defines the each method, which goes through all the elements of an array and invokes the block on the current element. Hence we print out the squares of all the numbers.

A block may contain local variables or variables defined outside the block but present in the scope of the block. Even if the original environment containing containing the block disappears, the block still retains all the variables and their values. Hence the block object forms a closure in itself.

5.3 Iterators Vs List Comprehension

Due to the power of blocks, Ruby supports constructs called iterators. An iterator goes over every element of any container class and may do any processing on it. In fact in the last subsection, you saw an example of the `each` iterator which went over all elements in an Array. Another example, we look at the collect iterator, which applies a function to all the elements and collects all the results back into an array. (Similar to *map* in functional languages).

```
[1,2,3].collect { |x| x*x }
```

Output: `[1,4,9]`

If we compare Ruby iterators to iterators in Java, we find that Ruby iterators are nothing but methods supported by the container classes. While in Java, Iterators are sophisticated helper classes which need to built around container objects.

Python achieve a similar effect by using list comprehensions, similar to those in Haskell:

```
numbers = [1,2,3]
[x*x for x in numbers]
```

Output: `[1,4,9]` We feel list comprehensions are a great feature, and add a lot of expressiveness to a language. The only problem with them in Python is that they are relatively newcomers to the language, thus one cannot rely on them to be in all implementations.

5.4 Singleton Methods

Ruby supports another interesting concept: Singleton methods. This allows Ruby objects to form classes just for themselves without cluttering up the class name-space. i.e. you can add methods to single objects. For example,

```
foo1 = Doubler.new("Fred");
foo2 = Doubler.new("Jove");
def foo2.name
  @name
end
foo2.name
foo1.name
```

Output:

```
"Jove"
undefined method `name' for #<Doubler:0x401d8610> (NoMethodError)
```

6 Other Language features

In this section we briefly list other minor features which both the languages support, and make brief comments about them.

- **Arrays and hashes** Ruby supports Arrays and associative arrays (Hash). Python has Lists and Tuples which are the same as arrays.
- **Memory management** Ruby has a mark and sweep garbage collector, while Python has a reference counting garbage collector.
- **Parallel assignment** Assignments can be done parallely, saving the need for temporary variables. `a,b = b,a`
- **Multi-threading** Threading is optional in python but available on many common platforms with some threading support. Ruby claims to do OS independent threading (even on MS-DOS!)
- **Regular expressions** While python has support for regular expression through a library object, Ruby has extensive regular expression support similar to Perl.
- **Exception handling** Both languages support exception handling like in Java.
- **Variable length arguments** Methods can take variable length arguments.
- **Dynamic loading** Can dynamically load extension libraries.
- **Extension modules** It is easy to write extension modules in other languages like C, Tcl, Perl etc.
- **Package support** Python has package support.

- **Serialization** Both languages support serialization and marshaling.
- **Portability** Both languages are portable across most platforms including Linux, Windows, MS-DOS, most Unixes, Mac, BeOS, OS/2 etc.
- **Reflection** Both languages support reflection, which can be useful for debugging, tracing etc.
- **Libraries** Both Ruby and Python have extensive built-in libraries.
- **Documentation** Both languages allow you to generate automatic documentations.

7 Conclusion

As we have seen in this report, that pureness in Ruby does tend to simplify the syntax and the readability of the code. But does it really help productivity? Certainly the fact that there are no primitive, non-object types helps both languages immensely compared to a language like Java. In Java a library writer must write an additional method for each primitive type if they want a container class to be completely general. This is seen in many of the standard Java libraries.

Even though Ruby claims to be have uniform access, there are still some quirks about it. For example, if I want to print a string to stdout, in Ruby I have to do:
`print "hello world!"`
What I would have really liked to do was
`"hello world!".print`
or even
`out.print "hello world!"` (out is the output stream)
Naively, there doesn't seem to be any difference, but the fact is that printing on the screen is a property of the string, or maybe of the output stream, but it definitely should not be a method in *my class* which does the printing. In fact such a method does make Ruby nothing different from a procedural language. And according to me this is against the philosophy of Ruby being a pure object-oriented language. In fact it is not just print which is the trouble, there are many useful methods in the Object class, which I can use as if they are functions, simply contradicting the fact that Ruby supports uniform access (which is one of Ruby's major selling point).

Overall there is not much different between Ruby and Python, though Ruby offers some cleaner syntax due to its object oriented model. Also Ruby

is equipped with lot more features than Python, though you may never use all of them. The advantage of Python is that since it has been around for a much longer time, there are a lot of libraries readily available for Python, which are not so readily available for Ruby. The disadvantage of Ruby is the fact that it is very poorly documented. Our advice is that you choose Ruby, if you are new to both the languages, otherwise weigh the option of continuing with language you are using. While both languages have nice features, there is nothing revolutionary about either.

References

Scripting: Higher Level Programming for the 21st Century
John K. Ousterhout
(This article appears in IEEE Computer magazine, March 1998)

Programming Ruby: The Pragmatic Programmer's Guide
http://www.rubycentral.com/book

The Ruby User's Guide
http://www.ruby-lang.org/

The Python Tutorial
http://www.python.org/doc/current/tut/tut.html
Guido van Rossum and Fred L. Drake, Jr., editor

The Python Reference Manual
http://www.python.org/doc/current/ref/ref.html
Guido van Rossum and Fred L. Drake, Jr., editor