Test - DYPL

Daniel Sundstrom
daniel@sykewarrior.com

March 16, 2012

1 Question 1

There are some major differences in the philosophy underpinning the design and the choices made for each language. In short, these differences can be described as; *Multiple inheritance vs. Mixins*, *Blocks vs. Functions* and *Built-in Functions vs. Global Object*. These differences will be explored in depth below.

1.1 Multiple inheritance vs. Mixins

One of the major differences in design between Ruby and Python is how the language handles class-based behavior modification, both at runtime and at write time. Ruby favors a module-based approach, where specific methods[1] are grouped together in named modules, which in turn are included in the desired classes to provide the behavior. Modules can be included both as class methods and as instance methods, depending on the keyword used for inclusion. This is illustrated in the example below.

```
module RangedWeapons
  def attack_with_bow
    puts "#{@name} attackes with bow!"
  end
end

class Ninja
  include RangedWeapons
  def initialize(name)
      @name = name
  end
  def attack
    puts "#{@name} attacks with sword!"
  end
end

n = Ninja.new('Shinobin')
n.attack                # => prints "Shinobin attacks with sword!"
n.attack_with_bow       # => prints "Shinobin attacks with bow!"
```

[1] Which, in Ruby, of course includes attributes since they are implemented as methods

As the example shows, this allows for composing the aggregate behavior for a class into separate, re-usable components (since a module can be included in any number of classes). Python, on the other hand, supports the same behavior through a limited form of multiple inheritance[2] as shown below:

```
class RangedNinja():
  def attack_with_bow(self):
    print self.name + " attacks with bow!"

class Ninja(RangedNinja):
  def __init__(self, name):
    self.name = name
  def attack(self):
    print self.name + " attacks with sword!"

n = Ninja("Shinobin")
n.attack()                  # => prints "Shinobin attacks with sword!"
n.attack_with_bow()         # => prints "Shinobin attacks with bow!"
```

As clearly shown in the examples above, both of these styles support the same effect - but point to interesting differences in design philosophy. One of the basic tenets of OOP[3] is its suitability and ability to map and represent real-world systems and relationships, and this is also one of basic foundation of object-oriented modeling as taught and discussed today[6][1]. The relationship of RangedNinja in regard to Ninja is more of superset-subset rather than a formal relationship. Ruby's usage of modules to support this behavior relates more closely to the concept of behavior rather than relationship. While multiple inheritance is a powerful tool[4], the presence of it in Python indicates a philosophy centered on inheritance rather than composition. The philosophy of Ruby, on the other hand, is to loosely couple shared groups of behavior with the desired classes - thus compositing the complete behavior of the class. Good example of where this is a more desired philosophy is when implementing a Data-Context-Interaction pattern.

On a larger scale this points to a difference in philosophy concerning the goals of the language, more specifically:

> "The principle of least surprise means principle of least my surprise. And it means the principle of least surprise after you learn Ruby very well. - Yukihiro Matsumoto"[4]

Multiple inheritance is known to be quite complex to use correctly, while providing very powerful tools for specializing instance-behavior. Composition, on the other hand, is quite simple to understand and use while not being as "strict" in the computer science sense of the word.

[2] Resolution is depth-first, left-to-right
[3] Object-Oriented Programming
[4] Though it comes at a steep price, since it increases complexity of implementation severely

1.2 Blocks vs. Functions

Another difference in philosophy between Ruby and Python is the treatment of closures[5]. Where Python supports closures in two major ways - lambdas and unbound functions - Ruby provides a richer set of tools for using closures - procs, lambdas, blocks and method-objects[6]. However, Python's lambdas are severely limited since they can only contain expressions and not statements, while Ruby allows any valid construct inside a closure. The difference this makes is apparent below:

```
# Ruby

[1,2,3,4,5,6,7,8,9].filter do |item|
  local_variable = SomeObject.interesting_method(item)
  return local_variable
end

# The block will have its own lexical scope and will not leak
puts local_variable # => Will raise a MethodNotFound
```

while the corresponding pythonic way of doing a filter looks something like

```
# Python

multiples_of_3 = filter(lambda x: x % 3 == 0, \
[1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Both languages support the notion of unbound functions, even though this is much simpler in Python than in Ruby (where it is actually quite complex), as in method objects not currently bound to an object and which can be bound at a later time to a suitable object. The main difference, however, is apparent when discussing blocks, and their nonexistence in Python. Python has an expressly stated goal from its most prolific core member that

> "Explicit is better than implicit." - Tim Peters[2]

This reflects in the choice of closure-support since the preferred way of handling them in Python - unbound methods - is a very explicit way since one decouples a bound method, making it an unbound method, passes it around (since functions are first-class objects in Python) and then binds them to a scope on execution. At all times do one act upon an Method-object. In Ruby, however, there's a notion of callable objects, that is - objects with respond to the method *call*. These includes all the types mentioned above (and could thus also include user-defined objects implementing the same API). One can, for example, implicitly convert a method argument into a callable object using the following syntax:

[5] A closure allows a function to execute outside its immediate lexical scope

[6] It can be argued that procs and lambdas are in essence the same thing, but that falls outside the scope of this text (and is plain wrong, see the arity-handling!)

```
def block_method(&block)
    yield 'ninja'
end

block_method do |ninja_string|
  puts "Yield:#{ninja_string}"
end
```

Note how the implicitly defined block is converted into an object at method-execution and is implicitly yielded to by the method body. This allows Ruby to use closures as dynamic callbacks for other methods, something that is possible in Python but require the use of external libraries to allow higher-order functions.

Ruby supports a very implicit style of scope management for both variables, and in order to support a more functionally oriented style of programming, while Python eschews implicitly declared scopes in lieu of making the process explicit and binding it to a single object and a construct.

1.3 Built-in Functions vs. Global Object

One major design difference between Ruby and Python is how to implement and make available basic and necessary functions and methods to a default scope, i.e. the scope where code is executed unless explicitly executed in another scope. This includes *eval* and *puts* in Ruby and *file* and *print* in Python. Ruby chooses to implement these as instance-methods on a global object[7] while Python builds them into the interpreter. This has numerous implications, for example - adding new expressions in Ruby is a matter of opening up one of Object, Kernel or BaseObject (which are all present in the inheritance chain by default) and adding new instance methods - while adding new functions on that level in Python is not possible (they must be explicitly referenced). This reflects a difference in where the languages comes from. Python, which originally was intended as "[..] bridge the gap between the shell and C."[5] whereas Ruby comes from "[...] concentrate on the fun and creative part of programming when they use Ruby."[3]. Since built-in functions on an interpreter level encourages an expression-oriented style of coding (much like Lisp), something which might feel natural to someone used to C-style procedural coding, it's natural that Python should choose interpreter-level functions. Ruby, on the other hand, springing from a designer who explicitly wants to escape the then traditional style of coding has a design which encourages a completely different style of coding.

[7] Much like the window-prototype in regular Javascript

2 Question 2

3 Question 3

4 Question 4

5 Question 5

References

[1] S. Isoda. Object-oriented real-world modeling revisited. *Journal of Systems and Software*, 59(2), 2001.

[2] T. Peters. The zen of python, Aug 2004.

[3] B. Stewart. An interview with the creator of ruby, Nov 2001.

[4] B. Venners. The philosophy of ruby - interview with yukihiro matsumoto, Sep 2003.

[5] B. Venners. Python's design goals - interview with guido van rossum, Jan 2003.

[6] N. Wirth. Good ideas, through the looking glass. Technical report, Swiss Federal Institute of Technology - Department of Computer Science, 2005.