Ruby for YNelson Users in 20 minutes

September 1, 2013

For `YNelson` users, basic Ruby syntax is necessary. This document provides the Ruby syntax primer for `YNelson` users. **This document is better done in one session, as the provided code samples rely on each other.** If you are familiar with Ruby, you do not need to read this document at all. Those who want more thorough introduction to the language, I recommend http://www.rubyist.net/~slagell/ruby/index.html, or any of the many Ruby textbooks.

Variables and Constants

In Ruby, everything is an *object*. Objects can be assigned to *variables* or *constants*. Ruby constants must always start with capital letter. Variables starting with small letter are *local variables*. (Other types of variables are *instance variables*, *class variables* and *global constants*; this is not important at the moment.)

```
alpha = 1
#=> 1
beta = [1, 2]
#=> [1, 2]
Gamma = { x: 1, y: 2, z: 3 }
#=> {:x=>1, :y=>2, :z=>3}
```

You can check this using `defined?` operator:

```
defined? alpha
#=> "local-variable"
defined? Gamma
#=> "constant"
```

Methods

Different classes respond to different *methods*, and respond to them differently:

```
beta.size
#=> 2
Gamma.size
#=> 3
Gamma.keys
#=> [:x, :y, :z]
Gamma.values
#=> [1, 2, 3]
beta.keys
#=> NoMethodError: undefined method ‘keys’ for [1, 2]:Array
```

Methods can be defined by `def` keyword:

```
def average( a, b )
  ( a + b ).to_f / 2
end
#=> nil
average( 2, 3 )
#=> 2.5
```

In the code example above, '`to_f`' method performs conversion of an integer into a floating point number, which is not important.

Classes

Every object belongs to some *class* (object type):

```
alpha.class
#=> Fixnum
beta.class
#=> Array
Gamma.class
#=> Hash
```

New classes can be defined with `class` keyword. The methods defined inside the class will become the *instance methods* of that class:

```
class Dog
  def speak!
    puts "Bow wow!"
  end
end
#=> nil
Pochi = Dog.new
#=> #<Dog:0x9c214ac>
Pochi.speak!
#=> Bow wow!
class Cat
  def speak!
    puts "Meow"
  end
end
#=> nil
Tama = Cat.new
#=> #<Cat:0x98efb80>
Tama.speak!
#=> Meow
```

These two classes now represent respectively dogs and cats in your irb session. In the code above, you could notice '`new`' method, used to create instances from the defined classes, and '`puts`' method, used to simply print characters on the screen.

Strings, Symbols, Arrays and Hashes

For `YPetri` users, it will be especially necessary to learn more about *strings*, *symbols*, *arrays*, *hashes*, and how to define and read *closures* (aka. *anonymous functions*). Strings and symbols are among the most basic Ruby objects, while arrays and hashes are important in understanding *argument passing* to methods and closures. **Understanding argument passing and closure writing is essential in using YPetri DSL.**

Strings

A string is simply a sequence of characters, which can be defined using single or double quotes (' or "):

```
my_string = 'Hello world!'
#=> "Hello world!"
my_string.class
#=> String
```

Strings are mutable (can be changed):

```
my_string.object_id
#=> 81571950
7.times do my_string.chop! end
#=> 7
my_string
#=> "Hello"
my_string.object_id
#=> 81571950
```

Above, you can newly notice `times` method, `do ... end` block, and `chop!` method that removes the last character from `my_string` 7 times, until only `"Hello"` remains. But the important thing is that as `object_id` method shows, `my_string` is still the same object (same *object id*), although the contents is changed.

```
my_string << "Pochi!"
#=> "Hello Pochi!"
my_string.object_id
#=> 81571950
```

Again, << operator changed the contents, but the object id remained the same.

Symbols

Unlike strings, symbols are immutable – they never change. They are written with colon (:):

```
:Pochi.class
#=> Symbol
```

Arrays

As seen earlier, they can be defined with square brackets `[]`. Square brackets are also used to address the array elements, counting from 0.

```
my_array = [ Pochi, Tama ]
#=> [#<Dog:0x9c214ac>, #<Cat:0x98efb80>]
my_array[0]
#=> #<Dog:0x9c214ac>
```

Negative numbers can be used to address the elements from the end of the array:

```
my_array[-1]
#=> #<Cat:0x98efb80>
my_array[-2]
#=> #<Dog:0x9c214ac>
```

Hashes

As for hashes, there are two ways of defining them. The first way uses *Ruby rocket* (=>):

```
h1 = { Pochi => "dog", Tama => "cat" }
#=> {#<Dog:0x9c214ac>=>"dog", #<Cat:0x98efb80>=>"cat"}
h1[ Tama ]
#=> "cat"
h1[ Pochi ]
#=> "dog"
```

The second way is possible only when the keys are symbols. It is done by shifting the colon to the right side of the symbol:

```
h2 = { dog: Pochi, cat: Tama }
#=> {:dog=>#<Dog:0x9c214ac>, :cat=>#<Cat:0x98efb80>}
h2[:dog]
#=> #<Dog:0x9c214ac>
```

Code blocks and Closures

Code blocks, or simply *blocks*, are pieces of code enclosed by do / end pair, or by curly brackets {}. Code blocks can be passed to methods:

```
[1, 2, 3, 4].map { |n| n + 3 }
#=> [4, 5, 6, 7]
my_array.each do |member| member.speak! end
#=> Bow wow!
    Meow
```

In the first case, 'map' method was passed a block specifying addition of 3. In the second case, 'each' method was passed a block calling speak! method on the array elements. Please note the pipe, or vertical line charecters (|), that delimit the block arguments (both blocks above happen to have only one argument). Code blocks can be understood as anonymous functions – a way of specifying an operation, when one does not want to write a method for it. Their semantics corresponds to *lambda calculus*.

Return values

Code blocks (and actually, all Ruby statements) have return value. With code blocks, the return value will typically be the last statement:

```
[1, 2, 3, 4].map { |v|
  v + 3    # this value will be ignored
  v - 1    # last value of the block will be returned
}
#=> [0, 1, 2, 3]
```

Closures

A block packaged for future use is called a *closure*. Ruby closures come in two flavors: proc and lambda. They are created by passing a block to the proc / lambda keyword:

```
my_proc = proc do |organism| organism.speak! end
#=> #<Proc:0x952674c@(irb):136>
my_lambda = lambda do |organism| organism.speak! end
#=> #<Proc:0x942faf0@(irb):137 (lambda)>
```

Once defined, they can be reused in code. Notice the ampersand (`&`) indicating block reuse:

```
my_array.each &my_proc
#=> Bow wow!
    Meow
my_array.each &my_lambda
#=> Bow wow!
    Meow
```

Closures can also be called alone, a little bit like methods:

```
my_proc.call( Pochi )
#=> Bow wow!
my_lambda.call( Tama )
#=> Meow
```

Instead of `call` keyword, you can just use dot before the parenthesis to call closures:

```
my_proc.( Tama )
#=> Meow
my_lambda.( Pochi )
#=> Bow wow!
```

Differences between `proc` and `lambda` closures are minor. For `YNelson` users, the most noticeable difference will be, that `proc` less finicky about its arguments than `lambda`:

```
my_proc.( Tama, "garbage" )
#=> Meow
my_lambda.( Tama, "garbage" )
#=> ArgumentError: wrong number of arguments (2 for 1)
```

Passing arguments

Earlier, we have defined method `average`, expecting two arguments. If wrong number of arguments is supplied, `ArgumentError` will ensue:

```
average( 3, 5 )
#=> 4
average( 3, 5, 8 )
#=> ArgumentError: wrong number of arguments (3 for 2)
```

Obviously, this is not a very nice behavior when it comes to averages. It is a general situation, that when calling more advanced methods, we need to modify their behavior, or pass more complicated structures to them. This is seen eg. with `YNelson::Transition` constructors, and will be further encountered in `YCell` and `YChem` DSLs. Furthermore, `YNelson` users have to be able to write their own closures, because that is how *functions* of *functional transitions* are specified. In other words, `YNelson` **users have to master argument passing from both user and programmer side**. There is no way around this. With functional Petri nets, one cannot avoid writing functions. It is possible to avoid using `YNelson`, but it is not possible to avoid learning to write functions. Every simulator of functional Petri nets brings with itself some sort of function language, which one has to learn. With `YNelson`, this is the language of Ruby closures.

Optional arguments

Arguments with prescribed default value are optional. Let us write an improved `average` method that can accept either 2 or 3 arguments:

```
def average( a, b, c=:pochi )
  # If c argument was not given, :pochi symbol will be assigned
  # to c by default.
  if c == :pochi then # only 2 arguments were supplied
    ( a + b ).to_f / 2
  else # 3 arguments were supplied
    ( a + b + c ).to_f / 3
  end
end
#=> nil
average( 3, 5 )
#=> 4
average( 3, 5, 8 )
#=> 5.333333333333333
average( 1, 2, 3, 4 )
#=> ArgumentError: wrong number of arguments (4 for 3)
```

The default value for `c` argument is prescribed using single equals sign (`=`). Apart from that, you can notice `if` ... `then` ... `else` ... `end` statement, which needs no explanation, equality test (double equals sign, `==`), used to test whether `c` contains `:pochi` symbol (indicating missing value), and comment character (octothorpe aka. sharp, `#`). Comment character `#` causes all characters until the end of the line to be ignored by Ruby. All code lines, exception the obvious ones, should have comments.

Variable-length argument lists

We will now improve our `average` method, so that it can calculate averages of any number of arguments. For this, we will use asterisk (`*`) syntactic modifier, also known as *splash*. The asterisk will cause a method to collect the arguments into an array. Let's try it out first:

```
def examine_arguments( x, *aa )
  puts "x is a #{x.class}."
  puts "aa is #{aa.class} of #{aa.size} elements."
end
#=> nil
```

Method examine arguments takes one normal argument (`x`), and collects the rest of the arguments into an array (`aa`), thanks to the splash modifier. (Apart from that, you can notice string interpolation using `#{ ... }` notation in the above code.) Then it prints the class of `x`, class of `aa` (which should be an array), and the number of elements after `x`.

```
examine_arguments( 1 )
#=> x is a Fixnum.
    aa is Array of 0 elements.
    nil
examine_arguments( :hello, :pochi, 3, 5, "garbage" )
#=> x is a Symbol.
    aa is Array of 4 elements.
    nil
```

With this, we can go on to define our improved average method:

```
def average( *aa )
  aa.reduce( :+ ).to_f / aa.size
end
#=> nil
average 3, 5, 7, 11
#=> 6.5
```

You can also newly notice `reduce( :+ )` method, used to calculate the sum of the `aa` array. To also practice closures, let us define a lambda doing the same as the `average` method above:

```
avg = lambda { |*aa| aa.reduce( :+ ).to_f / aa.size }
#=> #<Proc:0x9dbd220@(irb):208 (lambda)>
avg.( 11, 7, 5, 3 )
#=> 6.5
```

Named arguments

The main purpose of named arguments is to make the interface (or DSL) easier to remember, and the code easier to read. Easy-to-read code is a crucial requirement for scalable development. In Ruby methods, named arguments can be specified as hash pairs in the method call:

```
def density( length: 1, width: 1, height: 1, weight: 1 )
  weight.to_f / ( length * width * height )
end
#=> nil
density( length: 2, width: 2, height: 2, weight: 10 )
#=> 1.25
```

The above method calculates mean density of boxes of certain height, width, length and weight. Double splash (`**`) can be used to collect all the options in a hash. Let's use it to define a closure that does exactly the same thing as the method `density` we have just defined, in a slightly different way:

```
dens_closure = -> **nn do
  nn[:weight].to_f / ( nn[:length] * nn[:width] * nn[:height] ) end
#=> #<Proc:0x9a5d60c@(irb):241 (lambda)>
dens_closure.( length: 2, width: 2, height: 2, weight: 10 )
#=> 1.25
```

Above, note the alternative syntax for lambdas: `-> arg do ... end` is the same as `lambda do |arg| ... end`. Having hereby introduced the named arguments, let us notice hash-collecting behavior for square bracket (`[]`) array constructor syntax.

Hash-collecting behavior of square brackets

In more complicated method argument structures, it can be advantageous to take use of the hash-collecting by square brackets. It is normal for curly braces to create hashes:

```
h = { length: 2, width: 3, height: 4 }
#=> {:length=>2, :width=>3, :height=>4}
h.class
#=> Hash
```

However, square brackets, that generally create arrays, are also able to collect hashes just like the argument fields with named arguments:

```
a0 = [ 1, 2, 3 ]
#=> [1, 2, 3]
a0.class
#=> Array
a1 = [ 1, 2, 3, length: 2, width: 3, height: 4 ]
#=> [1, 2, 3, {:length=>2, :width=>3, :height=>4}]
a1.class
#=> Array
a1.map &:class
#=> [Fixnum, Fixnum, Fixnum, Hash]
a1[-1]
#=> {:length=>2, :width=>3, :height=>4}
```

In other words, if there are any trailing `key / value` pairs inside square brackets, they will be collected into a hash, which will become the last element of the array. This possibility to mix ordered elements with `key / value` pairs is used eg. in `YCell enzyme` constructor method.

Arity

Every closure and every method has arity, which is basically the number of input arguments. (Closures with 0 arguments are null*ary*, with 1 argument un*ary*, with 2 arguments bin*ary*, with 3 arguments tern*ary* etc. – therefrom *arity*.)

```
doubler = lambda { |a| a * 2 }
#=> #<Proc:0xa19b5b8@(irb):1 (lambda)>
doubler.call( 3 )
#=> 6
doubler.arity
#=> 1
adder = lambda { |p, q| p + q }
#=> #<Proc:0xa27d940@(irb):6 (lambda)>
adder.call( 5, 6 )
#=> 11
adder.arity
#=> 2
scaler = lambda { |number, p, q| number * (q / p) }
#=> #<Proc:0xa2825e4@(irb):7 (lambda)>
scaler.call( 10, 2, 3 )
#=> 15
scaler.arity
#=> 3
constant_function = lambda { 42 }
#=> #<Proc:0xa2825e4@(irb):7 (lambda)>
constant_function.call
#=> 42
constant_function.arity
#=> 0
```

Closures / methods with variable length arguments indicate this by reporting negative arity:

```
summation = lambda { |*array| array.reduce( :+ ) }
#=> #<Proc:0xa296ddc@(irb):9 (lambda)>
summation.call( 1, 2, 3, 4 )
#=> 10
summation.arity
#=> -1
array_scale = lambda { |*a, coeff| a.map { |e| e * coeff } }
#=> #<Proc:0xa2a9edc@(irb):12 (lambda)>
array_scale.call( 1, 2, 3, 4, 7 )
#=> [7, 14, 21, 28]
array_scale.arity
#=> -2
```

Return value

The last statement in a closure / method becomes the return value. In methods and lambda-type closures, return statement can also be used explicitly:

```
divider = lambda { |u, v|
            if v == 0 then
              return :division_by_zero # explicit return statement
            end
            u / v # implicit return value - last statement of the closure
          }
#=> #<Proc:0xa21e878@(irb):15 (lambda)>
divider.call( 15, 3 )
#=> 5
divider.call( 15, 0 )
#=> :division_by_zero
experimental_closure = proc {
                         1      # this value will be ignored
                         3      # this value will be ignored, too
                         42     # this value will be discarded as well
                         41 }   # this value will be returned
#=> #<Proc:0xa249460@(irb):28>
experimental_closure.call
#=> 41
experimental_lambda = lambda {
                        1          # this value will be ignored
                        return 3   # this value will be returned
                        7          # execution will never get here at all
                      }
#=> #<Proc:0xa3200dc@(irb):38 (lambda)>
experimental_lambda.call
#=> 3
```

Return value arity

It is possible to return more than one value. For example:

```
multiplication_table = lambda { |number|
                         [1, 2, 3, 4, 5]
                           .map { |element| element * number }
```

```
                        }
#=> #<Proc:0xa36a0d8@(irb):55 (lambda)>
```

This method returns 5 values. We can receive them by using a simultaneous assignment statement:

```
by_one, by_two, by_three, by_four, by_five = multiplication_table.call( 7 )
#=> [7, 14, 21, 28, 35]
by_one
#=> 7
by_two
#=> 14
by_five
#=> 35
```

Or we can simply collect them in an array:

```
collection = multiplication_table.( 3 )
#=> [3, 6, 9, 12, 15]
```

In `YNelson`, it sometimes becomes necessary to write closures with higher return arity (returning more than one value). This is normally done by returning an array. Also, lambda return statement can be used to return multiple values:

```
constant_vector = lambda { return 1, 2, 3 }
#=> #<Proc:0xa3cb338@(irb):72 (lambda)>
x, y, z = constant_vector.call
#=> [1, 2, 3]
x
#=> 1
y
#=> 2
z
#=> 3
```