Ruby in 20 Minutes

http://www.ruby-lang.org/en/documentation/quickstart/

December 9, 2013

This is a small Ruby tutorial that should take no more than 20 minutes to complete. It makes the assumption that you already have Ruby installed. (If you don't have Ruby on your computer download and install it before you get started.)

Interactive Ruby

Open up IRB.

- If you're using Mac OS X open up Terminal and type `irb`, then hit enter.
- If you're using Linux, open up a shell and type `irb` and hit enter.
- If you're using Windows, open `fxri` from the Ruby section of your Start Menu.

```
>>
```

Ok, so it's open. Now what? Type this: `"Hello World"`

```
>> "Hello World"
=> "Hello World"
```

Ruby obeyed You!

What just happened? Did we just write the world's shortest "Hello World" program? Not exactly. The second line is just IRB's way of telling us the result of the last expression it evaluated. If we want to print out `"Hello World"` we need a bit more:

```
>> puts "Hello World"
Hello World
=> nil
```

`puts` is the basic command to print something out in Ruby. But then what's the `=> nil` bit? That's the result of the expression. `puts` always returns `nil`, which is Ruby's absolutely-positively-nothing value.

Already, we have enough to use IRB as a basic calculator:

```
>> 3+2
=> 5
```

Three plus two. Easy enough. What about three times two? You could type it in, it's short enough, but you may also be able to go up and change what you just entered. Try hitting the up-arrow on your keyboard and see if it brings up the line with `3+2` on it. If it does, you can use the left arrow key to move just after the `+` sign and then use backspace to change it to a `*` sign.

```
>> 3*2
=> 6
```

Next, let's try three squared:

```
>> 3**2
=> 9
```

In Ruby `**` is the way you say "to the power of". But what if you want to go the other way and find the square root of something?

```
>> Math.sqrt(9)
=> 3.0
```

Ok, wait, what was that last one? If you guessed, "it was figuring out the square root of nine," you're right. But let's take a closer look at things. First of all, what's `Math`?

`Math` is a built-in module for mathematics. Modules serve two roles in Ruby. This shows one role: grouping similar methods together under a familiar name. `Math` also contains methods like `sin()` and `tan()`.

Next is a dot. What does the dot do? The dot is how you identify the receiver of a message. What's the message? In this case it's `sqrt(9)`, which means call the method `sqrt`, shorthand for "square root" with the parameter of 9.

The result of this method call is the value 3.0. You might notice it's not just 3. That's because most of the time the square root of a number won't be an integer, so the method always returns a floating-point number.

What if we want to remember the result of some of this math? Assign the result to a variable.

```
>> a = 3 ** 2
=> 9
>> b = 4 ** 2
=> 16
>> Math.sqrt(a+b)
=> 5.0
```

As great as this is for a calculator, we're getting away from the traditional Hello World message that beginning tutorials are supposed to focus on. So let's go back to that.

What if we want to say "Hello" a lot without getting our fingers all tired? We need to define a method!

```
>> def h
```

```
>>   puts "Hello World!"
>> end
=> nil
```

The code `def h` starts the definition of the method. It tells Ruby that we're defining a method, and that its name is `h`. The next line is the body of the method, the same line we saw earlier: `puts "Hello World"`. Finally, the last line `end` tells Ruby we're done defining the method. Ruby's response `=> nil` tells us that it knows we're done defining the method.

Now let's try running that method a few times:

```
>> h
Hello World!
=> nil
>> h()
Hello World!
=> nil
```

Well, that was easy. Calling a method in Ruby is as easy as just mentioning its name to Ruby. If the method doesn't take parameters that's all you need. You can add empty parentheses if you'd like, but they're not needed.

What if we want to say hello to one person, and not the whole world? Just redefine `h` to take a name as a parameter.

```
>> def h(name)
>>   puts "Hello #{name}!"
>> end
=> nil
>> h("Matz")
Hello Matz!
=> nil
```

So it works. But let's take a second to see what's going on here.

What's the *`#{name}`* bit? That's Ruby's way of inserting something into a string. The bit between the braces is turned into a string (if it isn't one already) and then substituted into the outer string at that point. You can also use this to make sure that someone's name is properly capitalized:

```
>> def h(name = "World")
>>   puts "Hello #{name.capitalize}!"
>> end
=> nil
>> h "chris"
Hello Chris!
=> nil
>> h
Hello World!
=> nil
```

There are a couple of other tricks to spot here. One is that we're calling the method without parentheses again. If it's obvious what you're doing, the parentheses are optional. The other

trick is the default parameter `World`. What this is saying is "If the name isn't supplied, use the default name of World".

What if we want a real greeter around, one that remembers your name and welcomes you and treats you always with respect. You might want to use an object for that. Let's create a "Greeter" class.

```
>> class Greeter
>>   def initialize(name = "World")
>>     @name = name
>>   end
>>   def say_hi
>>     puts "Hi #{@name}!"
>>   end
>>   def say_bye
>>     puts "Bye #{@name}, come back soon."
>>   end
>> end
=> nil
```

The new keyword here is `class`. This defines a new class called `Greeter` and a bunch of methods for that class. Also notice `@name`. This is an instance variable, and is available to all the methods of the class. As you can see it's used by `say_hi` and `say_bye`.

So how do we get this `Greeter` class set in motion? Create a `Greeter`-object and use it:

```
>> g = Greeter.new("Pat")
=> #<Greeter:0x16cac @name="Pat">
>> g.say_hi
Hi Pat!
=> nil
>> g.say_bye
Bye Pat, come back soon.
=> nil
```

Once the `g` object is created, it remembers that the name is `Pat`. Hmm, what if we want to get at the name directly?

```
>> g.@name
SyntaxError: compile error
(irb):2: syntax error, unexpected tIVAR
        from (irb):52
```

Nope, can't do it.

Instance variables are hidden away inside the object. They're not terribly hidden, you see them whenever you inspect the object, and there are other ways of accessing them, but Ruby uses the good object-oriented approach of keeping data sort-of hidden away.

So what methods do exist for `Greeter` objects?

```
>> Greeter.instance_methods
=> ["inspect", "clone", "method", "public_methods",
    "instance_variable_defined?", "equal?", "freeze", "say_bye",
    "methods", "respond_to?", "dup", "instance_variables", "__id__",
    "eql?", "object_id", "id", "singleton_methods", "send", "taint",
    "frozen?", "instance_variable_get", "__send__", "instance_of?",
    "to_a", "type", "protected_methods", "instance_eval", "==",
    "display", "===", "instance_variable_set", "kind_of?", "extend",
    "to_s", "hash", "class", "tainted?", "=~", "private_methods",
    "say_hi", "nil?", "untaint", "is_a?"]
```

Whoa. That's a lot of methods. We only defined two methods. What's going on here? Well this is all of the methods for `Greeter` objects, a complete list, including ones defined by ancestor classes. If we want to just list methods defined for Greeter we can tell it to not include ancestors by passing it the parameter `false`, meaning we don't want methods defined by ancestors.

```
>> Greeter.instance_methods(false)
=> ["say_bye", "say_hi"]
```

Ah, that's more like it. So let's see which methods our greeter object responds to:

```
>> g.respond_to?("name")
=> false
>> g.respond_to?("say_hi")
=> true
>> g.respond_to?("to_s")
=> true
```

So, it knows `say_hi`, and `to_s` (meaning convert something to a string, a method that's defined by default for every object), but it doesn't know `name`.

But what if you want to be able to view or change the name? Ruby provides an easy way of providing access to an object's variables.

```
>> class Greeter
>>   attr_accessor :name
>> end
=> nil
```

In Ruby, you can open a class up again and modify it. The changes will be present in any new objects you create and even available in existing objects of that class. So, let's create a new object and play with its `@name` property.

```
>> g = Greeter.new("Andy")
=> #<Greeter:0x3c9b0 @name="Andy">
>> g.respond_to?("name")
=> true
>> g.respond_to?("name=")
=> true
>> g.say_hi
Hi Andy!
=> nil
>> g.name="Betty"
=> "Betty"
>> g
=> #<Greeter:0x3c9b0 @name="Betty">
>> g.name
=> "Betty"
>> g.say_hi
Hi Betty!
=> nil
```

Using `attr_accessor` defines two new methods for us: `name` to get the value, and `name=` to set it.

This greeter isn't all that interesting though, it can only deal with one person at a time. What if we had some kind of MegaGreeter that could either greet the world, one person, or a whole list of people?

Let's write this one in a file instead of directly in the interactive Ruby interpreter IRB.

To quit IRB, type "quit", "exit" or just hit Control-D.

The MegaGreeter Class

```
#!/usr/bin/env ruby

class MegaGreeter
  attr_accessor :names
  # Create the object
  def initialize(names = "World")
    @names = names
  end

  def say_hi # Say hi to everybody
    if @names.nil?
      puts "..."
    elsif @names.respond_to?("each")
      # @names is a list of some kind, iterate!
      @names.each do |name|
        puts "Hello #{name}!"
      end
    else
      puts "Hello #{@names}!"
    end
  end

  def say_bye # Say bye to everybody
    if @names.nil?
      puts "..."
    elsif @names.respond_to?("join")
      # Join the list elements with commas
      puts "Goodbye #{@names.join(", ")}.  Come back soon!"
    else
      puts "Goodbye #{@names}.  Come back soon!"
    end
  end

end

puts "__FILE__=#{__FILE__}"   # name of this file
puts "$0=#{$0}"               # name of running ruby script

if __FILE__ == $0
  namelist = ["Albert", "Brenda", "Charles", "Dave", "Joe"]
  mg = MegaGreeter.new
  mg.say_hi
  mg.say_bye
  mg.names = "John" # Change name to be "John"
  mg.say_hi
  mg.say_bye
  # Change the name to an array of names
  puts namelist.join("; ")
  mg.names = namelist
  mg.say_hi
```

```
  mg.say_bye
  mg.names = nil # Change to nil
  mg.say_hi
  mg.say_bye
end
```

Save this file as `ri20min.rb`, make it an executable by typing `chmod u+x ri20min.rb`. This works on Mac OS X and on Linux. Now run the script as `ri20min.rb`. The output should be:

```
Hello World!
Goodbye World.  Come back soon!
Hello Zeke!
Goodbye Zeke.  Come back soon!
Hello Albert!
Hello Brenda!
Hello Charles!
Hello Dave!
Hello Englebert!
Goodbye Albert, Brenda, Charles, Dave, Englebert.  Come back soon!
...
...
```

So, looking deeper at our new program, notice the initial lines, which begin with a hash mark `(#)`. In Ruby, anything on a line after a hash mark is a comment and is ignored by the interpreter. The first line of the file is a special case, and under a Unix-like operating system tells the shell how to run the file. The rest of the comments are there just for clarity.

Our `say_hi` method has become a bit tricker:

```
# Say hi to everybody
def say_hi
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("each")
    # @names is a list of some kind, iterate!
    @names.each do |name|
      puts "Hello #{name}!"
    end
  else
    puts "Hello #{@names}!"
  end
end
```

It now looks at the `@names` parameter to make decisions. If it's `nil`, it just prints out three dots. No point greeting nobody, right?

If the `@names` object responds to `each`, it is something that you can iterate over, so iterate over it and greet each person in turn. Finally, if `@names` is anything else, just let it get turned into a string automatically and do the default greeting.

Let's look at that iterator in more depth:

```
@names.each do |name|
  puts "Hello #{name}!"
```

```
end
```

`each` is a method that accepts a block of code, then runs that block of code for every element in a list, and the bit between `do` and `end` is just such a block. A block is like an anonymous function. The variable between pipe characters is the parameter for this block.

What happens here is that for every entry in a list, `name` is bound to that list element, and then the expression

```
puts "Hello #{name}!"
```

is run with that name.

Most other programming languages handle going over a list using the for loop, which in C looks something like:

```
for (i=0; i<number_of_elements; i++)
{
  do_something_with(element[i]);
}
```

This works, but isn't very elegant. You need a throw-away variable like `i`, have to figure out how long the list is, and have to explain how to walk over the list. The Ruby way is much more elegant, all the housekeeping details are hidden within the each method, all you need to do is to tell it what to do with each element. Internally, the each method will essentially call yield `"Albert"`, then yield `"Brenda"` and then yield `"Charles"`, and so on.

The real power of blocks is when dealing with things that are more complicated than lists. Beyond handling simple housekeeping details within the method, you can also handle setup, teardown, and errors, all hidden away from the cares of the user.

```
# Say bye to everybody
def say_bye
  if @names.nil?
    puts "..."
  elsif @names.respond_to?("join")
    # Join the list elements with commas
    puts "Goodbye #{@names.join(", ")}.  Come back soon!"
  else
    puts "Goodbye #{@names}.  Come back soon!"
  end
end
```

The `say_bye` method doesn't use `each`, instead it checks to see if `@names` responds to the `join` method, and if so, uses it. Otherwise, it just prints out the variable as a string. This method of not caring about the actual type of a variable, just relying on what methods it supports is known as "Duck Typing", as in "if it walks like a duck and quacks like a duck". The benefit of this is that it doesn't unnecessarily restrict the types of variables that are supported. If someone comes up with a new kind of list class, as long as it implements the join method with the same semantics as other lists, everything will work as planned.

So, that's the `MegaGreeter` class, the rest of the file just calls methods on that class. There's one final trick to notice, and that's the line:

```
if __FILE__ == $0
```

`__FILE__` is the magic variable that contains the name of the current file. `$0` is the name of the file used to start the program. This check says "If this is the main file being used"? This allows a file to be used as a library, and not to execute code in that context, but if the file is being used as an executable, then execute that code.

So that's it for the quick tour of Ruby. There's a lot more to explore, the different control structures that Ruby offers; the use of blocks and yield; modules as mixins; and more. I hope this taste of Ruby has left you wanting to learn more.