Do you know Ruby Doctest?

A gem simply to provide a way to document our programs using IRB sessions

by RubyLearning
http://rubylearning.com/blog/

Do you know Ruby Doctest?

Rubydoctest is a gem that we will be installing. The purpose of the gem is simply to provide a way to document our programs using IRB sessions, resulting in a way to provide usage examples as well as knowing when an expected use fails because of some update or changes to our program or environment.

Installing Ruby DocTest

Rubydoctest can be installed by doing the following from the command line:

```
$ gem install rubydoctest
```

If you are running Windows or using RVM the above should work (pending network/internet connectivity) but you may need to use sudo to install depending on your set up.

The parts of Ruby DocTest

Rubydoctest information can be contained within a comment or a comment block.

Let's start by creating a file `hello.rb`:

```
def hello name
  "Hello " + name
end
=begin # This is the beginning of a comment block
doctest: hello " World" will return "Hello World"
>> hello "World"
=> "Hello World"
=end
```

We have our trusty method called `hello`, but we have added a comment block with doctest information inside. Other than the `doctest:` directive, it should remind you of an IRB session.
The `doctest:` directive is the human readable title of the test. If this isn't supplied you would end up seeing 'default' as the title.

To use rubydoctest:

```
USAGE: rubydoctest [options]

  rubydoctest parses Ruby files (.rb) or DocTest files
(.doctest) for irb-style
  sessions in comments, and runs the commented sessions as
tests.

  Options:
    Output Format:
      --html  - output in HTML format
      --plain - force output in plain text (no Ansi colors)

    Debug:
      --ignore-interactive - do not heed !!! special
directives
      --trace     - turn backtrace on to debug Ruby DocTest
      --debugger  - include ruby-debug library / gem
```

Let’s go ahead and run the following command to see what happens here:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello "World" will return "Hello World"
1 comparisons, 1 doctests, 0 failures, 0 errors
```

As you can see, we get passing tests. And if this were above the method definition it may even be collected during ri and rdoc generation (this process is outside the scope of this article.)

Refactoring with RubyDoctest

Once we have a simple test in place, and that test is passing, we can refactor it, with confidence that the method still passes.

I noticed that I used “String” + variable in this method, but realized that it may read a little better (and create only one object) if I use string interpolation:

```
def hello name
  "Hello #{name}"
end
```

Which I think should pass the test, but just for good measure, we want to actually test it:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello "World" will return "Hello World"
1 comparisons, 1 doctests, 0 failures, 0 errors
```

And as we can see, it passes.

Next, we will fully develop our hello method so that it has some nice features.

Using RubyDoctest in a TDD manner

We will be starting our hello method over from scratch. Going through the full life cycle of a small method, and testing along the way. It will be conversational, and demonstrative, so that you can follow along as we go.

STARTING WITH AN IDEA

It all starts with an idea: I would like a hello method that will simply state “Hello World!” In order to make this happen, and have the documentation and the tests that I would eventually like to have, I place this in my hello.rb file:

```
=begin
doctest: hello returns “Hello World!”
>> hello
=> “Hello World!”
=end
```

And to make sure that it is written correctly, we do want to run the following command and get the information back from our test:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
ERR  | hello returns "Hello World!"
       NameError: undefined local variable or method
`hello' for main:Object
         from hello.rb:3
       hello
1 comparisons, 1 doctests, 0 failures, 1 errors
```

This information, if we pause and read it, tells us exactly what we need to get it past this error. It states “undefined local variable or method ‘hello’” and from which line it was called. We have already made the assumption that we want a method, and so we can do the least amount to cause this ERR to no longer exist. (You may see it as a shade of yellow in your console.)

So let’s write the least amount of code that we can think of to get it to not error (I placed the following on lines 6 and 7, just below the comment block):

```
def hello
end
```

Once I save this, and run the rubydoctest command, I get the following:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
FAIL | hello returns "Hello World!"
       Got: nil
       Expected: "Hello World!"
         from hello.rb:4
1 comparisons, 1 doctests, 1 failures, 0 errors
```

Depending on your settings, you should get a red “FAIL” message as well as the next three lines. Regardless you will get the summary information with 1 comparisons, 1 doctests, 1 failures and 0 errors.

The error count is 0, which means that we have progressed past the yellow ERR message. We will generally always want to work past ERR messages first, then FAIL messages, until we get to OK.

Let’s go forward in our journey and get past this FAIL message.

Paying attention to the feedback we get, it states that we are getting nil when we are expecting “Hello World!”. I think it is fairly obvious that we can simply have the method return the string “Hello World!” and get this to pass. Simplistic, and exactly what we want:

```
def hello
  "Hello World!"
end
```

Simplistic and rids us of the failure in our test:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello returns "Hello World!"
1 comparisons, 1 doctests, 0 failures, 0 errors
```

Again, in your console, you may actually see a green OK. The important thing to note is the 0 failures and 0 errors.

IT CONTINUES WITH EXPANDING IDEAS

Let's think about what we have. A `hello` method that greets the world. Not too bad. But let's say that we want to be able to use it to greet the world, but also to greet someone in particular. Continuing our program documentation, we add the following:

```
doctest: hello "reader" returns "Hello reader!"
>> hello "student"
=> "Hello student!"
```

Those three lines should be added to comment block and our entire file looks like so:

```
=begin
doctest: hello returns "Hello World!"
>> hello
=> "Hello World!"
doctest: hello "reader" returns "Hello reader!"
>> hello "student"
=> "Hello student!"
=end
def hello
  "Hello World!"
end
```

When we run the doctest (which we should do any time we make a change in either the documentation or in the code itself), we should get:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello returns "Hello World!"
ERR  | hello "reader" returns "Hello reader!"
       ArgumentError: wrong number of arguments (1 for 0)
         from hello.rb:6
       hello "student"
2 comparisons, 2 doctests, 0 failures, 1 errors
```

From this we can see that the current way we want to use it by giving one argument is in error on line 6, because our method wants 0. That is what the "(1 for 0)" means. We are giving it one argument "student" and it is really what we want.

You may think that if we fix the error, the first test with either error or fail, and you would be right. When going through this process, you want to be careful to make small changes, as well as concentrating on the feature you are working on. So we will work on test hello "reader" returns "Hello reader!" until it passes. Until that happens, we will ignore the first test.

How might we get out of our error condition? Perhaps like this?

```
def hello name
  "Hello World!"
end
```

Notice how we added the argument of name to our method. We could have used something instead of name, perhaps arg or argument, but that simply states what it is, not what we should pass. In this way, what we call the argument helps to document our method as well.

As we have made a change, it is time to run rubydoctest again and see if we got past the error:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
ERR  | hello returns "Hello World!"
       ArgumentError: wrong number of arguments (0 for 1)
         from hello.rb:3
       hello
FAIL | hello "reader" returns "Hello reader!"
       Got: "Hello World!"
       Expected: "Hello student!"
         from hello.rb:7
2 comparisons, 2 doctests, 1 failures, 1 errors
```

As you can see, we went from ERR to FAIL, which is where we want to be. Ignoring the first test, we go from FAIL to OK (pass) in the simplest change we can think of:

```
def hello name
  "Hello student!"
end
```

Simply replacing ‘World’ with ‘student’, I think, should get this test to pass:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
ERR  | hello returns "Hello World!"
       ArgumentError: wrong number of arguments (0 for 1)
         from hello.rb:3
       hello
 OK  | hello "reader" returns "Hello reader!"
2 comparisons, 2 doctests, 0 failures, 1 errors
```

There we go. Now we can look for any ERR messages as they are critical to allowing our program to run. It is complaining about wrong number of arguments, this time in reverse of what our second test was complaining about a few moments ago.

Do we want to change the test? It is something we could consider. I think I like having the ability to have default behavior of no argument to say greet the world. And so I would leave the test as it is.

Changing the method itself to accept 0 or more arguments is likely the right thing to do here. Let’s give that a shot:

```
def hello *name
  “Hello student!”
end
```

Adding the * to the argument will give us the ability to accept 0 or more arguments. Let’s test and see if we are still in an error condition:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
FAIL | hello returns "Hello World!"
       Got: "Hello student!"
       Expected: "Hello World!"
         from hello.rb:4
 OK  | hello "reader" returns "Hello reader!"
2 comparisons, 2 doctests, 1 failures, 0 errors
```

Seems to be OK, as we are no longer in an error condition. And a fail condition leads us where to go. We are getting “Hello student!” when we don’t give any argument. Looking above at our program, we want something a little more dynamic. Let’s use name variable to give us what we want in the string.

```
def hello *name
  "Hello " + name + "!"
end
```

It looks good to me, let’s see what our tests tell us?

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
ERR  | hello returns "Hello World!"
       TypeError: can't convert Array into String
         from hello.rb:3
       hello
ERR  | hello "reader" returns "Hello reader!"
       TypeError: can't convert Array into String
         from hello.rb:6
       hello "student"
2 comparisons, 2 doctests, 0 failures, 2 errors
```

As I know that prior to this change, the additional feature passed, we will ignore any change to that, and continue to make our default behavior work.

What is Ruby telling us? That we can't convert Array into String on line 3. It must be the +() method. Let's use .to_s method to convert the name variable explicitly to a string:

```
def hello *name
  "Hello " + name.to_s + "!"
end
```

and see if that takes us away from the error:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
FAIL | hello returns "Hello World!"
       Got: "Hello []!"
       Expected: "Hello World!"
         from hello.rb:4
FAIL | hello "reader" returns "Hello reader!"
       Got: "Hello [\"student\"]!"
       Expected: "Hello student!"
         from hello.rb:7
2 comparisons, 2 doctests, 2 failures, 0 errors
```

Good, we are out of error mode again, and back to FAIL mode. We can deal with that, and we got some more information. We got what appears to be an empty `Array` as indicated by the [] in the string. I would guess it is because of the *name that we added. Let's try a default value instead. Still not changing any tests… but paying attention to the feedback from the first test.

```
def hello name='World'
  "Hello " + name.to_s + "!"
end
```

That gets us out of fail and into OK (pass) for each test as indicated below:

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello returns "Hello World!"
 OK  | hello "reader" returns "Hello reader!"
2 comparisons, 2 doctests, 0 failures, 0 errors
```

REFACTORING IN THE GREEN

You might think we are done. But we are in ‘green’ mode and so are free to refactor our code. And looking at our code I see some things that can be improved.

```
def hello name='World'
  "Hello + name + “!"
end
```

Let’s use remove the `.to_s` method as we know we are going to get a string. Of course, we test after refactoring to make sure we are still good.

```
rubydoctest hello.rb
=== Testing 'hello.rb'...
 OK  | hello returns "Hello World!"
 OK  | hello "reader" returns "Hello reader!"
2 comparisons, 2 doctests, 0 failures, 0 errors
```

And we see we are still golden.

And on second thought, we can use string interpolation which to me makes it more readable, but also creates only one `String` object.

```
def hello name='World'
  "Hello #{name}!"
end
```

And of course, we run rubydoctest again to make sure we didn't break anything, and we see that we are good.

If you have any questions, feel free to leave a comment on my blog post **here**.