Enter comment

Your comment
Edit Send feedback
Preview Send feedback

Coding Tips 2

Now that you have a little bit more programming experience under your belt, it's time to talk about a few more coding tips to watch out for.

Using new lines to organize code

If you've been running Rubocop, you'll notice that one of the offenses it looks for is extraneous lines in your code. At first, this may seem very nit-picky. However, new lines are actually important visual cues in your program.

Just like how writers use paragraphs to organize related sentences, we (that is, programmers) also need to learn to organize chunks of code to make it easier to read.

> Making your code readable is of paramount importance, not only for others, but for future self.

Sometimes, that future self is just a few hours or days later.

At first, it's going to be difficult to develop a feel for how to use new lines in your programs. Over time, you'll start to get a feel for it, especially if you read a lot of existing code.

Here's an example:

```
# bad

name = ''
puts "Enter your name: "
loop do
  name = gets.chomp
  break unless name.empty?
  puts "That's an invalid name. Try again:"
end
puts "Welcome #{name}!"
puts "What would you like to do?"
```

Use some new lines to separate the different concerns in the code.

```
# better

name = ''

puts "Enter your name: "
```

```
loop do
  name = gets.chomp
  break unless name.empty?
  puts "That's an invalid name. Try again:"
end

puts "Welcome #{name}!"
puts "What would you like to do?"
```

Visually, you can quickly see where the `name` variable is initialized. You can also see that this small code snippet is roughly divided into 3 parts:

1. variable initialization
2. user input and validation
3. using the variable

If there's a bug or feature related to any of the above, you'd know where to start reading.

Should a method return or display?

This is something we talked about previously, but ties into a larger concept:

> Understand if a method returns a value, or has side effects, or both.

What we mean by "side effects" there is either displaying something to the output or mutating an object. For example, the below methods have side effects:

```
# side effect: displays something to the output
# returns: nil

def total(num1, num2)
  puts num1 + num2
end

# side effect: mutates the passed-in array
# returns: updated array

def append(target_array, value_to_append)
  target_array << value_to_append
end
```

And here's an example of a method with no side effects:

```
# side effect: none
# returns: new integer

def total(num1, num2)
  num1 + num2
end
```

Paying attention to methods -- both existing ones in the standard Ruby library, as well as ones you write yourself -- and their side effects vs return values is critical to writing code that's well organized and easy to read.

In general, if a method has both side effects and a meaningful return value, it's a red flag. Try to avoid writing methods that do this, as it will be very difficult to use these methods in the future.

Name methods appropriately

One way to help yourself remember what each method does is to choose good method names. If you have some methods that output values, then preface those methods with `display_` or `print_`. For example, if you

see a method named `print_total`, you can be certain it will output a total and not return anything. Following this convention will save you from constantly looking at the implementation to recall how to use a method.

> If you find yourself constantly looking at a method's implementation every time you use it, it's a sign that the method needs to be improved.

All this goes back to one bit of advice: a method should do one thing, and be named appropriately. If you can treat a method as a "black box", then it's a well designed method.

You should be able to use a method called `total` and understand that it returns a value, and a method called `print_total` returns nil, without looking at the implementation of either. On the other hand, if there's a method called `total!`, then it's a sign that there is some side effect somewhere.

Don't mix up those concerns. Don't write a method that mutates, outputs and returns a meaningful value. Make sure your methods just do one of those things.

Don't mutate the caller during iteration

Suppose we have an array of strings and we want to iterate over that array and print out each element. We could do something like this:

```
words = %w(scooby doo on channel two)
words.each {|str| puts str}
```

That's very typical code. Now let's mutate the elements in the array and add a "!" after each word.

```
words = %w(scooby doo on channel two)
words.each {|str| str << '!'}
puts words.inspect        # => ["scooby!", "doo!", "on!", "channel!", "two!"]
```

Note that we aren't mutating the `words` object. That is, we aren't adding to or removing elements from `words`. But we're in fact mutating each element within the `words` array. This is also pretty typical Ruby.

Now suppose we do something weird in the `each` code block, and we want to remove the element as we're iterating.

```
words = %w(scooby doo on channel two)
words.each {|str| words.delete(str)}
puts words.inspect        # => ["doo", "channel"]
```

That is very strange -- shouldn't every element be deleted? We should be expecting an empty array, or even perhaps `nil`. The return value `["doo", "channel"]` may result in some confusion, but that's expected. The lesson here is:

> Don't mutate a collection while iterating through it.

You can, however, mutate the individual elements within that collection, just not the collection itself. Otherwise, you'll get unexpected behavior.

Variable shadowing

Variable shadowing occurs when you choose a local variable in an inner scope that shares the same name as an outer scope. It's incredibly easy to make this mistake, and essentially prevents you from accessing the outer scope variable from an inner scope.

Let's see an example. Suppose we have an array of names, and we want to append a last name to them. We could write some code like this:

```
name = 'johnson'
```

```
['kim', 'joe', 'sam'].each do |name|
  # uh-oh, we cannot access the outer scoped "name"!
  puts "#{name} #{name}"
end
```

The problem is that we've clobbered the outer scoped `name` variable. Within the `each` code block, the `name` variable represents the elements in the array - "kim", "joe", or "sam".

Note that the below is *not* variable shadowing:

```
name = 'johnson'

['kim', 'joe', 'sam'].each do |fname|
  name = fname
end
```

The above code is accessing the outer scope `name` variable and re-assigning it. After the `each` block, the `name` will be set to "sam".

Be careful about choosing appropriate block variables (the thing between the `| |`) when working with blocks. If you pick a name that is identical to an outer scope variable, variable shadowing will prevent you from using the outer scope variable.

This is another reason you should run Rubocop on your code; it'll catch this error for you.

Don't use assignment in a conditional

We recommend that you never use assignment in a conditional. It's not clear whether you meant to use `==` or if you indeed meant to do assignment.

```
# bad

if some_variable = get_a_value_from_somewhere
  puts some_variable
end

# good

some_variable = get_a_value_from_somewhere
if some_variable
  puts some_variable
end
```

It may work, but it's extremely confusing and others reading your code aren't 100% confident if it's a bug or if it's on purpose. Avoid at all times.

However, you'll sometimes still see experienced Rubyists do this. For example, here's a `while` loop that iterates through a collection:

```
numbers = [1, 2, 3, 4, 5]

while num = numbers.shift
  puts num
end
```

The `Array#shift` method removes and returns the first element in the array. When there's nothing to remove, `shift` returns `nil`. This loop takes advantage of that fact to serve as the loop termination condition.

While it works, the problem is that this code is hard to read, and future programmers (remember, including your future self) can't be 100% confident that this is what you meant to do. Did you mean `num == numbers.shift`? Is this a bug, or is it written on purpose?

As a convention, if you must do this, wrap the assignment in parentheses. This will signify to future programmers that you know what you're doing and this is done on purpose.

```
numbers = [1, 2, 3, 4, 5]

while (num = numbers.shift)
  puts num
end
```

However, we still recommend that you don't do this.

Use underscore for unused parameters

Suppose you have an array of names, and you want to print out a string for every name in the array, but you don't care about the actual names. In those situations, use an underscore to signify that we don't care about this particular parameter.

```
names = ['kim', 'joe', 'sam']
names.each { |_| puts "got a name!" }
```

Or, if you have an unused parameter when there are multiple parameters:

```
names.each_with_index do|_, idx|
  puts "#{idx+1}. got a name!"
end
```

Gain experience through struggling

The final tip in this assignment is about dealing with struggling. There are two things that beginners feel at this stage:

- want to know the "best" or "right" way to do something, and want to learn the "best practice"
- too much time being wasted on debugging and not doing things correctly the first time

To the first point: it's less impactful to learn "best practices" without first learning *why* they are best practices. This leads to the second point, which is you must learn to be ok with struggling through the "bad" or sub-optimal practices first. That's not wasting time, that's generating experience. Becoming a good developer means experiencing and solving a lot of weird issues.

In other words, don't memorize "best practices", but spend enough time programming to the point where you understand the context for those practices. If you do that, you'll be able to lean on your past experience to draw on, rather than try to retrieve a bullet list from an old reading months or years ago.

Therefore, don't be fearful of violating rules or afraid to make mistakes. But keep an eye out for improvements. Coding is like writing -- there are syntactical rules, but there are also creative ways of expression.

We can't say this enough: spend the time programming. Learn to debug through problems, struggle with it, search for the right terms, play around with the code, and you'll be able to transform into a professional developer. Because that's exactly what professional developers do on a daily basis.

Mark this assignment as complete

Rubocop Rock Paper Scissors

Formatting Help

- Supports Github Flavored Markdown
  ****bold****, ~~strike through~~ `` `single line of code` ``
- Supports Emoji