Your comment Enter comment

Edit Send feedback

Preview Send feedback

Coding Tips

We've been doing a lot of code reviews, so we see a wide variety of code. We want to mention a few specific items we've seen come up over and over in doing reviews that may help you on your journey.

Dramatic Experience and Retaining Knowledge

A recent graduate was talking about how far he's come since starting from the first course and how much he's learned and how much has stuck with him. However, there were also some mistakes that he continues to make, despite having learned about them very early on.

After chatting a bit, we came to the conclusion that he was still making those basic mistakes, because he hadn't been burned badly enough. On the flip side, the things he did always remember were usually due to that problem having caused him a major headache in the past. Any time those issues came up, the burn was fresh in his mind, and he was reminded to not make that mistake again.

Part of the goal of our program is to let people realize that programming isn't really that difficult, but unfortunately, a lot of stuff just takes repetition. Maybe it takes you 2 to 3 hours to debug one little thing and then you realize "Oh, I just missed a comma" - that's your *burn*.

When you first start learning to program, it seems like there are so many rules to remember. It's hard to understand which ones are critical, and which ones aren't worth memorizing. Most of the time, when you come across yet another suggestion to follow, the reaction is "Yea, OK! Another rule I need to remember. I'll file it away... somewhere.". Until you burn a couple hours on that rule, it won't be retained for the long haul.

The only way to retain information is to pay with time. Debugging an issue for hours and hours will ensure that this problem gets *burned* into long term memory. You pay for those burns with time but they pay back with interest. If you spend 3 hours debugging, those hours are not wasted - you won't make that mistake again. We want to encourage you to think about debugging from that perspective - embrace your burns and remember their lessons.

> If you are serious about programming and you want to do it for years and maybe decades from today, then the hours you put into debugging little things are really going to help you retain knowledge for the long haul.

You'll most likely remember things that have either caused you to waste a lot of time, or have caused you some embarrassment — you decide how to pay for the experience, either through time or reputation.

Naming Things

A lot of people try to save on characters and like to use very short variable or method names. There's no need to save on characters. Choose descriptive variable and method names.

For example, we see this a lot:

```
p = gets.chomp
```

What is `p`? Obviously we're collecting some user input here, but what is it used for? Variables are named not for how they are set, but for what they actually store. Not indicating that in the name adds an additional mental check you must perform every time you see `p`. We could name it `yes_or_no`, but another problem arises if you decide to later capture a `maybe` response from the user. Typically, you don't want to hardcode possible response values into the variable name because of future uncertainty. Instead, try to capture the intent of the variable. For example, if we're trying to capture a response to determine if the game should loop again, a better name would be `play_again`. It's both descriptive as well as future-proof.

> In programming, naming things is very hard. Unfortunately, this problem isn't obvious when you write small programs, but it really impedes flow when you're working on larger programs. Try to develop a habit of thinking about how to name things descriptively.

One small exception to having descriptive variable names is when you have a very small block of code. It's less of a problem because the life or scope of that variable doesn't span more than a couple of lines.

Naming conventions

When naming things, follow Ruby conventions. Ruby is an unusual language in that you can create a variable with a capital letter, or a constant with CamelCase. Don't abuse this freedom. Adopt the style that most Rubyists follow. This will make reading your code much easier for others.

> In Ruby, make sure to use `snake_case` when naming everything, except classes which are `CamelCase` or constants, which are all `UPPERCASE`.

Using Rubocop on your code will help catch some of these issues as well.

Mutating Constants

Another common type of mistake is that people tend to change values of constants. For example:

```
CARDS = [1, 2, 3]
```

Because `CARDS` is a constant, do not add or remove values from this array. Though Ruby allows it, in general, constants should not be mutated at all.

> `CONSTANTS` should be immutable.

Methods

The instinct to extract code to a method is good. However, make sure that the method does one thing, and that its responsibility is very limited. This implies indirectly that your methods should be short (let's say, around 10 lines or so). If it's more than 15 lines long, it would probably be better to split it to 2 or 3 methods.

Here are some guidelines on how to write good methods:

- It should not display something to the output as well as return a value.
- Decide whether the method should return something with no side effects *or* only perform side effects with no return. If you do both, it's likely that you won't remember what it does when you need to use the method. The method name should reflect whether it has side effects or not (for example, many methods in the standard Ruby library end with a `!` to signify side effects).
- In Ruby, we would not say `return_total`, it would be just `total` - returning a value is implied. Further, we would not expect a `total` method to have side effects or print a value out. We would expect a method called `total` to be defined something like this:

```
def total(cards) # [1, 2, 3]
end
# => Integer
```

From the name `total` you know this method should return an integer (the value of total). If instead this method mutates cards, it then becomes confusing. You will always have to remember this and someone else trying to read your code will have to dig into to the method to understand its behavior. If you find yourself always looking at the method implementation, it's a sign that the method is not named appropriately, or that it's doing more than one simple thing.

Methods should be at the same level of abstraction

This is a little hard in the beginning because you have to develop a feel for it over time. Usually, methods take some input and return an output. We should be able to just copy and paste the method into irb and test it. When working with a method, you should be able to mentally extract the method from the larger program, and just work with the method in isolation. You should be able to feed this method inputs, and expect it to give some outputs. When you have a method like this, you can use it without thinking about its implementation. Working this way helps compartmentalize your focus, an important skill when working on large codebases.

If the methods in your program are correctly compartmentalized, it makes programming much easier, especially on larger programs. That is why when you read good code, the methods are all at the same layer of abstraction.

For example, given the four methods below, which one stands out?

- `deal()`
- `hit()`
- `stay()`
- `iterate_through_cards()`

The last one, `iterate_through_cards`, is not at the same abstraction level as the other methods. The other methods are in the language of the game — verbs that are used only for this game. They all specify "what" to do, but leave the implementation details to the actual method. You shouldn't care about the implementation when you use the method. The last method is a programmer concern — iterating through cards. It's "how" something needs to be done. Again, this is going to be hard in the beginning, but pay attention to how your methods are organized, and whether you can look at a list of methods a week later and still understand how to use them without studying their implementation.

Method names should reflect mutation

```
def update_total(total, cards)
end
# total is changed
```

When we see a method called `update_total`, we assume that the parameter passed in to it will be mutated. Therefore, we wouldn't expect to use this method like this: `total = update_total(total, cards)`. In other words, we wouldn't expect the `update_total` method to return a value. The less you have to remember, and the less other people have to remember while looking at your code, the better. Use naming conventions, even in your own code, to signify which types of methods mutate vs which methods return values.

The more you have to think about the method, the harder it is to use it. If it is performing a side effect and at the same time returning a value, this makes debugging and using the method very hard.

Your goal should be to build small methods that are like LEGO blocks: they should be stand-alone pieces of functionality that you can use to piece together larger structures. You don't want these methods to be mentally taxing to use. Interesting structures are comprised of many hundreds or thousands of atomic lego pieces. Likewise, large programs are comprised of hundreds or thousands of small methods (later, we'll talk about organizing methods into classes and objects).

There are, however, some methods that are convoluted because the logic is complex. It's likely a sign that you don't quite understand the problem well enough to break it down into well-compartmentalized pieces. That's fine. But you will understand the problem better as you dig into the code more and more, and as your understanding becomes more clear, refactor your code to reflect that growing clarity.

This is a very similar process to writing. Your first draft is almost exploratory, dumping out ideas all over the place. As your narrative comes into more focus, the structure of your piece can become more organized and clean.

Displaying Output

Sometimes, you'll have methods that only display things. For example:

```
def welcome
  puts "welcome"
end
```

This is fine, but it's not obvious whether a method called `welcome` returns a string, or outputs strings directly. One way to resolve this is to help yourself remember and prefix all methods that output values with something like `print_`, `say_` or `display_`. This will require some discipline and it's important that you only output values in these methods. Don't mutate parameters or return values.

Methods are like black-boxes. It takes some stuff (input) and returns some value (output) to you. They should be very contained and you should know what they do without having to look up the implementation. Coding will be much easier if you follow these general guidelines.

Remember that code not only has to work properly, but must also be read easily — both by others as well as your future self.

Miscellaneous Tips

- Don't prematurely exit the program. All your methods should be at the same mental scope and not just randomly exit the program. Your program should probably only have one exit point.
- Watch your indentation. 2 spaces, not tabs. This violation burns Rubyists' eyes. Verify your indentation after you pushed to Github.
- Name your methods from the perspective of using them later. That is, think about how you would like to invoke them. Think about the implementation later. For example, if you have an array of `cards`, and you want to write a method to find the ace, your method should be called `find_ace`, and you can use it like this: `ace = find_ace(cards)`. You shouldn't name it `find_ace_from_cards`, because you'd be invoking it

like this: `find_ace_from_cards(cards)`. When Rubyists see a method like that, they think "what else can you find an ace from besides cards? tiles?". The easier your code is to read, the easier it is to debug and maintain.

- Know when to use a "do/while" vs a "while" loop. Here's an example:

  ```
  while answer.downcase != 'n' do
    puts "Continue? (y/n)"
    answer = gets.chomp
  end
  ```

  When running this, ruby will throw an exception of "undefined local variable or method 'answer'". To correct this, we have to initialize answer before the while statement, like this:

  ```
  answer = ''
  while answer.downcase != 'n' do
    puts "Continue? (y/n)"
    answer = gets.chomp
  end
  ```

That certainly would work, but a slightly better implementation could be to use a "do/while" loop:

```
loop do
  puts "Continue? (y/n)"
  answer = gets.chomp
  break if answer.downcase == 'n'
end
```

Here, the entire code is contained in the loop, and it's slightly easier to reason with. You could even do without the `answer` variable and use the user's input (i.e. `gets.chomp`) in the if condition directly, but using `answer` is fine -- remember, clarity over terseness.

Truthiness

Why do we say "truthiness" instead of just true or false? The reason is because in Ruby, like most programming languages, more than just true evaluates to true in a conditional. In fact, Ruby is very liberal about what "true" means. Here's one simple rule to remember:

> In ruby, everything is truthy except `nil` and `false`.

Because of this, we don't have to compare an expression to `true` or `false`, and can rely on the expression's "truthiness" directly. Example:

```
if user_input == true

# could be just

if user_input
```

Sometimes, you're trying to guard against `nil` value, and you'd write: if `user_input != nil`. But if your intent is to guard against `nil`, you can just as well do `if user_input`, since a `nil` value will evaluate to `false`. That is, the "truthiness" of `nil` is `false`.

This seems all very simple and intuitive, but it can lead to dangerous bugs. For example, take this code:

```
if some_variable = 2
  puts "some_variable is 2"
else
  puts "some_variable is not 2"
end
```

The above code will always output "some_variable is 2" even if `some_variable` is not. The reason is because the expression `some_variable = 2` is assigning the value `2` to `some_variable`, instead of doing a comparison. Because only `nil` and `false` evaluate to false, the expression `some_variable = 2` has a truthiness of true. Therefore, the statement `if some_variable = 2` always evaluates to true. Avoid this type of "assignment within a conditional" code. Others reading this code won't know for sure if it's intentional or a typo.

Approach to learning

Learning to program takes some focus and attention. It takes a lot of repetition over a long period. One of the surprising aspects about learning to program is that it's not a sequential process. You can't "master Ruby", then "master Rails", then "master testing". You'll likely be revisiting old topics over and over, and, through dramatic experience, certain things will be *burned* into long term memory.

The first time you are exposed to a new topic, it's going to be daunting. The second time, it's going to be easier and the third time even easier.

The tips mentioned here may not make sense now, but that's ok. Over time, hopefully you'll see many of these topics surface again, and it'll make more sense the second time around. Don't be demoralized if you do it once and you can't remember most of it. That's normal. Keep moving forward, and don't be afraid to spend time gaining valuable experience.

You marked this topic or exercise as completed.
Assignment: Mortgage / Car Loan Calculator
Variable Scope

Formatting Help

- Supports Github Flavored Markdown
  ****bold****, ~~strike through~~ `` `single line of code` ``
- Supports Emoji
- Use ```` ```ruby ```` before a code block and end with ```` ``` ````, for example:
  ````
  ```ruby
  class Foo
  end
  ```
  ````

×