**Blocks**

Let's take a closer look at main feature of the Ruby language: **blocks**. Until this point you have been utilizing blocks with enumerable methods like each, map, and times. A block is a chunk of code that is passed into a method to be executed. We explored blocks in previous sections, but now we'll want to take a peek under the hood and use blocks in our methods. First, let's do a quick refresher.

**Block Basics**

There are two ways to pass a block into a method.

We can use {...} brace syntax for blocks that only contain a single line of code:

[1,2,3].map { |num| -(num \* 2) } # => [-2, -4, -6]

Or, we can use do...end syntax for multiline blocks:

[1,2,3].map do |num|

doubled = num \* 2

-doubled

end # => [-2, -4, -6]

Brace {...} blocks and do...end blocks are functionally equivalent, we just prefer do...end for blocks that contain many lines.

Blocks can accept parameters if we name them between pipes (|param\_1, param\_2, etc.|).

We know that when we pass a block into map, map will execute the block, passing in every element of the array one by one. It will take each evaluation of the block and make that value an element of the new array that the map method will return.

Blocks are somewhat similar to methods in that both can contain lines of code as well as take in parameters. However, an important distinction to make is that the return keyword pertains to *methods*, not *blocks*. Let's take a look at a common pitfall:

# Correct:

def double\_eles(arr)

arr.map do |ele|

ele \* 2

end

end

double\_eles([1,2,3]) #=> [2,4,6]

# Incorrect:

def double\_eles(arr)

arr.map do |ele|

return ele \* 2

end

end

double\_eles([1,2,3]) #=> 2

Looking at the incorrect implementation of double\_eles, we use the return keyword within the block. If we use return we will be returning out of the entire double\_eles method on the first iteration of map. In other words, using return in a block will not make the block evaluate to the return value. The return will force the outer method to evaluate to the return value. This is a huge reason why we should treat *blocks* and *methods* as somewhat distinct concepts.

**Using Methods as Blocks**

It is very, very common to have blocks that take an argument and call a single method. For example:

["a", "b", "c"].map { |str| str.upcase } #=> ["A", "B", "C"]

[1, 2, 5].select { |num| num.odd? } #=> [1, 5]

Ruby allows us to use cleaner syntax when we have simple blocks that follow the above pattern. Let's refactor the above example to use this shortcut:

["a", "b", "c"].map(&:upcase) #=> ["A", "B", "C"]

[1, 2, 5].select(&:odd?) #=> [1, 5]

You may find this syntax quite strange. Let's notice a few things about how we are using map. We call map and pass in a single argument, &:upcase. :upcase is a symbol referring to the String#upcase method. We use & to convert this "method" into an object that we can pass into map. In Ruby, we cannot directly pass a method into another method, so we need to use the & operator. In the next lecture we'll explore the intricacies of &.

For now, here's a hard and fast rule you can use to optimize some blocks. If you are calling a method like map, passing a block that has this general structure:

array.map { |block\_param| block\_param.method }

Then you can rewrite it as:

array.map(&:method)

When employing this trick, be aware of what &:method you are using. The method you choose should be compatible with your data:

["a", "b", "c"].map(&:upcase) # => ["A", "B", "C"]

[1, 2, 3].map(&:upcase) # NoMethodError: undefined method 'upcase' for Integer

The second map is invalid because we can't use upcase on numbers!