**Becoming a Rubyist with style!**

Let's take a moment to highlight some syntactical and stylistic features of Ruby that make it a very readable and expressive language. Ruby is flexible in that it allows many "shortcuts" that can make code cleaner. These "shortcuts" are initially uncomfortable for programmers to read and write, especially if they are coming from another language with stricter syntax. Since you have completed our Intro to Ruby course already, we can now take off the training wheels and begin to write code like a true Rubyist! Being a Rubyist means using the faculties of the language to your advantage. You might have experience with the programming concepts in other languages, but in this course it’ll be essential to understand how to implement these concepts in Ruby!

It's okay if you are unfamiliar with some of the methods we example below, we'll learn them overtime. However, you should be able to gather what they do from their names alone.

Proper style in our code is important because it is the mark of high quality work and an experienced programmer!

**Implicit Returns**

Methods in Ruby will automatically return the evaluation of their last executed expression. You will still need to use the explicit return keyword to do an early return.

# Less preferred

def get\_avg(num\_1, num\_2)

return (num\_1 + num\_2) / 2

end

# Preferred by a Rubyist

def get\_avg(num\_1, num\_2)

(num\_1 + num\_2) / 2

end

**Omitting parentheses for method calls with no arguments**

When calling a method without passing any arguments, we'll often drop the parentheses altogether.

def say\_hi

puts "hi"

end

# Less preferred

say\_hi()

# Preferred by a Rubyist

say\_hi

**Use single line conditionals when possible**

When we have a single line in the body of a simple if statement (that is not attached to an elsif or else), we can turn it into a one-liner:

raining = true

# Less preferred

if raining

puts "don't forget an umbrella!"

end

# Preferred by a Rubyist

puts "don't forget an umbrella!" if raining

**Use built in methods**

There are many methods in Ruby that can make your life easier. Use them:

num = 6

# Less preferred

p num % 2 == 0

# Preferred by a Rubyist

p num.even?

people = ["Joey", "Bex", "Andrew"]

# Less preferred

p people[people.length - 1]

# Preferred by a Rubyist

p people[-1]

p people.last

**Use enumerables to iterate**

There are many enumerables in Ruby that have specific use cases. These tools can really make the code read like english. Often times, you can avoid using a while loop in favor of a more readable enumerable.

# Less preferred

def repeat\_hi(num)

i = 0

while i < num

puts "hi"

i += 1

end

end

# Preferred by a Rubyist

def repeat\_hi(num)

num.times { puts "hi" }

end

Given a problem, not all enumerables are equal. Some methods will immediately solve the problem at hand elegantly.

# Less preferred

def all\_numbers\_even?(nums)

nums.each do |num|

return false if num % 2 != 0

end

true

end

# Preferred by a Rubyist

def all\_numbers\_even?(nums)

nums.all? { |num| num.even? }

end

## Common Enumerables

Ruby's robust enumerable methods are what makes it a uniquely readable and expressive programming language. Classic enumerables like each, map, and select are staples but there are more enumerables that you will want to familiarize yourself with to write even cleaner code! This is meant to be an overview of a few methods you'll find useful, so you'll want to reference the [Ruby Docs](https://ruby-doc.org/) for the complete documentation of every method available in ruby!

### all?

Return true when all elements result in true when passed into the block.

p [2, 4, 6].all? { |el| el.even? } # => true

p [2, 3, 6].all? { |el| el.even? } # => false

### any?

Return true when all at least one element results in true when passed into the block.

p [3, 4, 7].any? { |el| el.even? } # => true

p [3, 5, 7].any? { |el| el.even? } # => false

### none?

Return true when no elements of result in true when passed into the block.

p [1, 3, 5].none? { |el| el.even? } # => true

p [1, 4, 5].none? { |el| el.even? } # => false

### one?

Return true when exactly one element results in true when passed into the block.

p [1, 4, 5].one? { |el| el.even? } # => true

p [1, 4, 6].one? { |el| el.even? } # => false

p [1, 3, 5].one? { |el| el.even? } # => false

### count

Return a number representing the count of elements that result in true when passed into the block.

p [1, 2, 3, 4, 5, 6].count { |el| el.even? } # => 3

p [1, 3, 5].count { |el| el.even? } # => 0

### sum

Return the total sum of all elements

p [1, -3, 5].sum # => 3

### max and min

Return the maximum or minimum element

p [1, -3, 5].min # => -3

p [1, -3, 5].max # => 5

p [].max # => nil

**flatten**

Return the 1 dimensional version of any multidimensional array

multi\_d = [

[["a", "b"], "c"],

[["d"], ["e"]],

"f"

]

p multi\_d.flatten # => ["a", "b", "c", "d", "e", "f"]

## Symbols

Ruby has an additional data type that is similar to Strings, called **Symbols**. Let's explore what differentiates a Symbol from a String, and how to use them in our code. In Ruby, we can denote a symbol using a colon (:) before writing characters. Where a string is wrapped in quotes, a symbol just has a leading colon. Both strings and symbols contain many characters, but they are not equivalent. Symbols are immutable and cannot change, while strings are mutable and can change

str = "hello" # the string

sym = :hello # the symbol

p str.length # => 5

p sym.length # => 5

p str[1] # => "e"

p sym[1] # => "e"

p str == sym # => false

# a string is different from a symbol!

### Symbols are Immutable

The most apparent difference between strings and symbols is that strings are mutable, while **symbols are immutable**. This means that string can be "changed", but a symbol can never be "changed":

str = "hello"

sym = :hello

str[0] = "x"

sym[0] = "x"

p str # => "xello"

p sym # => :hello

The utility of a symbol comes from the fact that it can never change over time. The technical implication of this is that a symbol only needs to be "created" once. There is no need to create "copies" of symbol because we can be certain that it will not change over the course of our programs. Operations such as comparing two symbols is very fast and efficient compared to regular strings.

Under the hood, each time we reference a literal string, Ruby will allocate a piece of our machine's memory to store that string. More memory must always be allocated for a new string, even if it is a duplicate value, because strings are mutable! We must track changes to the strings separately, so we need to store the two instances of the string in distinct memory locations.

Talk of memory locations is pretty abstract, but an easy way to witness this is to use Ruby's object\_id method. This will return the memory address of some data. Notice how duplicate value strings will be stored at different memory locations:

"hello".object\_id # => 70233443667980

"hello".object\_id # => 70233443606440

"hello".object\_id # => 70233443438700

If we don't intend to mutate the string, we can use a symbol to save some memory. A symbol value will be stored in exactly one memory location:

:hello.object\_id # => 2899228

:hello.object\_id # => 2899228

:hello.object\_id # => 2899228

Because of these characteristics, symbols are often used to act as unique identifiers in our code. We'll be able to ensure the the identifier will remain intact, without change, while also being efficient with memory.

### Symbols as hash keys

We'll see the preference of using of symbols in a few places in Ruby. For now, one common way to a symbol is as the key in a hash:

my\_bootcamp = { :name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"] }

p my\_bootcamp # => {:name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"]}

p my\_bootcamp[:color] #=> "red

When initializing a hash with symbol keys, Ruby offers a shortcut. We can drop the rocket (=>) and move the colon (:) to the right of the symbol:

my\_bootcamp = { name:"App Academy", color:"red", locations:["NY", "SF", "ONLINE"] }

p my\_bootcamp # => {:name=>"App Academy", :color=>"red", :locations=>["NY", "SF", "ONLINE"]}

p my\_bootcamp[:color] #=> "red

This shortcut is only allowed when initializing the symbols in the hash. When getting a value from the hash after initialization, we must always put the colon on the left like normal. hash[:key] is the correct syntax. Writing hash[key:]is invalid.

## Default Arugments

As you are writing methods there are times where you may want to make an argument optional. In this scenario, we can assign a default value in the parameter list:

# Let's make num an optional parameter.

# By default, num will have the value of 1

def repeat(message, num=1)

message \* num

end

p repeat("hi") # => "hi"

p repeat("hi", 3) # => "hihihi"

The repeat method above has an optional num argument. If we call repeatwithout explicitly passing in a value for num, num will be implicitly passed in with the value 1. This is useful for implementing methods with a default behavior.

We are free to use any default value for an optional argument, so the possibilities are endless. A fairly common design pattern is to set an arg to nil by default and have logic based on that scenario:

def greet(person\_1, person\_2=nil)

if person\_2.nil?

p "Hey " + person\_1

else

p "Hey " + person\_1 + " and " + person\_2

end

end

greet("Chao") # => "Hey Chao"

greet("Chao", "Arittro") # => "Hey Chao and Arittro"

To avoid confusion, it's best practice to have optional parameters listed after the required ones. If we stick to this convention, we can always expect arguments to be taken in the same order we pass them in. So avoid writing code like this:

def greet(person\_1="default", person\_2)

p person\_1 + " and " + person\_2

end

greet("Chao") # => "default and Chao"

The method above is not intuitive because although "Chao" is first argument passed in, person\_2 will be assigned "Chao". Avoid this by only assigning default values at the end of the parameter list.

## Option Hashes

If you have a method that accepts a hash as an argument, you can omit the braces when passing in the hash:

def method(hash)

p hash # {"location"=>"SF", "color"=>"red", "size"=>100}

end

method({"location"=>"SF", "color"=>"red", "size"=>100})

# this also works:

method("location"=>"SF", "color"=>"red", "size"=>100)

This can really clean things up when you have other arguments before the hash:

def modify\_string(str, options)

str.upcase! if options["upper"]

p str \* options["repeats"]

end

# less readable

modify\_string("bye", {"upper"=>true, "repeats"=>3}) # => "BYEBYEBYE"

# more readable

modify\_string("bye", "upper"=>true, "repeats"=>3) # => "BYEBYEBYE"

Combining this with the default arguments we covered in the previous section can make our code even more flexible:

def modify\_string(str, options={"upper"=>false, "repeats"=>1})

str.upcase! if options["upper"]

p str \* options["repeats"]

end

modify\_string("bye") # => "bye"

modify\_string("bye", "upper"=>true, "repeats"=>3) # => "BYEBYEBYE"

## Splat Operator

There are few different ways to use the splat (\*) operator in Ruby. Let's explore each of them so we can add them to our programming tool belt.

### Using splat to accept additional arguments

Ruby methods are pretty strict in that we must pass in the exact number of arguments that a method expects. If we pass in too many, we will receive an error:

def method(arg\_1, arg\_2)

p arg\_1

p arg\_2

end

method("a", "b", "c", "d", "e") # ArgumentError: wrong number of arguments (given 5, expected 2)

Building upon the code above, if we want our method to have the ability to accept at least two arguments with potentially more, we can add a splat parameter. The additional arguments will be gathered into an array for us to use as we see fit:

def method(arg\_1, arg\_2, \*other\_args)

p arg\_1 # "a"

p arg\_2 # "b"

p other\_args # ["c", "d", "e"]

end

method("a", "b", "c", "d", "e")

If we pass in exactly two arguments, then other\_args will be an empty array:

def method(arg\_1, arg\_2, \*other\_args)

p arg\_1 # "a"

p arg\_2 # "b"

p other\_args # []

end

method("a", "b")

Notice that in any scenario, the arguments are passed in positionally. This means that in the example above, arg\_1 is assigned "a", arg\_2 is assigned "b", and there is no additional data being passed, so other\_args is empty.

As a best practice, we should use splat at the end of the parameter list to avoid confusion. So avoid writing code like this:

# Avoid doing this, it's confusing:

def method(\*other\_args, required\_arg)

p other\_args # ["a", "b"]

p required\_arg # "c"

end

method("a", "b", "c")

### Using splat to decompose an array

We can also use splat to decompose or unpack elements of an array. Let's say we had an array containing some elements, but we wanted each individual element to become an argument:

def greet(first\_name, last\_name)

p "Hey " + first\_name + ", your last name is " + last\_name

end

names = ["grace", "hopper"]

greet(names) # ArgumentError: wrong number of arguments (given 1, expected 2)

The code above does not work because we are passing in the full array as the first\_name, making last\_name a missing argument. Thankfully we can use a splat to unpack this array:

def greet(first\_name, last\_name)

p "Hey " + first\_name + ", your last name is " + last\_name

end

names = ["Grace", "Hopper"]

greet(\*names) # => "Hey Grace, your last name is Hopper"

When using splat to unpack an array, you can imagine that the \* will remove the brackets ([]) that enclose the array. This leaves us with a simple comma separated list, perfect for passing in arguments. If you imagine \* as removing the brackets around an array, we can figure out some other creative ways to use this tool:

arr\_1 = ["a", "b"]

arr\_2 = ["d", "e"]

arr\_3 = [ \*arr\_1, "c", \*arr\_2 ]

p arr\_3 # => ["a", "b", "c", "d", "e"]

### Using splat to decompose a hash

We can use a double splat (\*\*) to perform a similar unpacking of a hash's key-value pairs. Double splat will only work with hashes where the keys are symbols:

old\_hash = { a: 1, b: 2 }

new\_hash = { \*\*old\_hash, c: 3 }

p new\_hash # => {:a=>1, :b=>2, :c=>3}