Truthiness

Monday, 26 July 2021 3:46 pm

Everything in Ruby is considered "truthy" except for false and nil.

Why is ability to express "true" or "false" important?

What is a boolean object?

How are booleans represented in Ruby?

What classes do the booleans true and false belong to?
`true` => TrueClass
`false` => FalseClass

What are the different uses of boolean objects?

In an if conditional, what should the test expression evaluate to?

```ruby
num = 5

if num < 5
  puts "small number"
else
  puts "large number"
end

```
*The above if-else conditional outputs "small number" because the expression `num < 5` evaluates to `true`*

**The if-else conditional executes the if branch if the test expression provided evaluates to the true object **

How can you use a method call in an if conditional?
```ruby
puts "its true" if some_method_call
```
The above will output "it's true!" if some_method_call returns a truthy object or the boolean object `true`

When using a method call as a conditional expression, what is preffered that the method call should return?

What do logical operators && and || return?

What does the && operator return?

If more than two expression are chained to &&, what will ruby return?

What does the || operator return?

What is short circuit evaluation?

When does && short cicuit?

When does || short circuit?

What does Truthiness mean?

What does ruby consider to be truthy?

Extracts from the course and book on the topic of Truthiness

Ruby Book

Notice that after if and elsif we have to put an expression that evaluates to a boolean value: true or false

In Ruby, every expression evaluates to true when used in flow control, except for false and nil.

Coding Tips

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 Ruby everything is truthy except `false` and `nil`

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

Article: The meaning of truthy and falsey (https://gist.github.com/jfarmer/2647362)

Many programming languages, including Ruby, have native boolean (true and false) data types. In Ruby they're called true and false. In Python, for example, they're written as True and False.

But oftentimes we want to use a non-boolean value (integers, strings, arrays, etc.) in a boolean context (if statement, &&, ||, etc.). So someone designing a language has to decide what values count as "true" and what count as "false."

A non-boolean value that counts as true is called "truthy," and a non-boolean value that counts as false is called "falsey."

Remember: only true and false are booleans. nil is not a boolean. 0 is not a boolean. [1,2,3] is not a boolean. The string "apple" is not a boolean. **When used in a context where a boolean is expected, Ruby evaluates them as boolean.**

## Summary of article:

Ruby, like many programming languages, has a boolean (true/false) data type. In Ruby we write "true" and "false." For convenience, though, we often want to evaluate non-boolean values (integers, strings, etc.) in a boolean context (if, &&, ||, etc.).

Ruby has to decide whether these values count as true or false. If the value isn't literally "true" but evaluates as true, we call it "truthy." Likewise, if the value isn't literally "false" but evaluates as false, we call it "falsey."

For example, 1 is "truthy." One might also say "1 evaluates to true."

In Ruby only nil and false are falsey. Everything else is truthy.

Logical (boolean) operators like &&, ||, and ! are methods. That means they return something. What do they return?

When you write a boolean expression like

```ruby
4 && 5 # This returns 5
nil && true # This returns nil
true && nil # This returns nil
true || nil # This returns true
nil || "" # This returns ""
we get a new truthy or falsey value.

```

Flowcharts and Pseudocode

Sunday, 25 July 2021 12:17 am

Variable Scope

Sunday, 25 July 2021 12:16 am

What does the variable's scope determine?
A variable's scope determines where in a program a variable is available for use.

How is a variable's scope defined in a program?
A variable's scope is defined by where the variable is initialized or created.
where a variable is initialized determines its scope

How is a variable's scope defined in Ruby?
In Ruby a variable's scope is defined by a block.

What is a block in Ruby?
A block is a piece of code following a method invocation, usually delimited by either curly braces `{}` or `do/end`

What does inner and outer scope mean?
Code inside a block is in an inner scope and code outside that block is in an outer scope with respect to that block.
the scope created by a block following a method invocation is an inner scope

Scope Rule for local variables
Inner scope can access variables initialized in an outer scope.
But Outer scope cannot access variables initialized in an inner scope.
Variables initialized in an outer scope can be accessed in an inner scope, but not vice versa

How do we know if some code delimited by do..end or {} is a block?

```ruby
for i in arr do
 a = 3
end

puts a
```

In the above code `a` is accessible outside the for..do/end because this do..end did not create a new inner scope.
for..do/end is part of the Ruby Language and not a method invocation.

In what two areas are the local variable scoping rules encountered?
method definition and method invocation with a block.

What is the relation of a block to a method invocation?
The block following the method invocation is actually an argument being passed into the method

What is an important aspect of a block related to variable scope?
They create a new scope for local variables.

How can we effect a local variable initialized in an outer scope inside a do..end block following a method invocation?
A do..end block following a method invocation creates an inner scope for local variables. An outer scope variable can be both accessed and reassigned in an inner scope.
This means that when we instantiate variables in an inner scope, we have to be very careful that we're not accidentally re-assigning an existing variable in an outer scope.

How do local variables initialized in peer method invocations behave? What does this behavior entail?
Peer blocks cannot reference variables initialized in other blocks. This means that we could use the same variable names in peer blocks. However, they are not the same variables.

How do nested block behave with respect to local variable scope?
Nested blocks follow the same rules of inner and outer scoped variables. However, when dealing with nested blocks, our usage of what's "outer" or "inner" is going to be relative.

What is variable shadowing? How can it be prevented?
When a block parameter has the same name as a outer scoped local variable, it prevents access to the outer scope variable inside the block. IT 'shadows' the outer scoped variable. And we are only able to access the block parameter's value.
Variable shadowing also prevents us from making changes to the outer scoped variables.
To prevent variable shadowing, block parameters should be given a name different from outer scoped variables,

What are the scoping rules with regards to method definitions?
Method definitions are self contained with respect to local variables. Methods can only access variables that were initialized inside the method or that are defined as parameters.

What is inner and outer scope for method definitions?
A method definition has no notion of "outer" or "inner" scope -- you must explicitly pass in any parameters to a method definition.

What if a local variable and a method were to share the same name?
Ruby will first search for the local variable, and if it is not found, then Ruby tries to find a method with the given name.
If neither local variable nor method is found, then a NameError message will be thrown.

What are the scoping rules inside a method definition?
The rules of scope for a method invocation with a block remain in full effect even if we're working inside a method definition.

More Variable Scope

How does a block relate to a method invocation?

any method can be called with a block, but the block is only executed if the method is defined in a particular way.

## The way that an argument is used, whether it is a method parameter or a block, depends on how the method is defined.

## blocks and methods can interact with each other; the level of that interaction is set by the method definition and then used at method invocation.

## When invoking a method with a block, we aren't just limited to executing code within the block; depending on the method definition, the method can use the return value of the block to perform some other action.

For example The Array#map method is defined in such a way that it uses the return value of the block to perform transformation on each element in an array

Method definitions cannot directly access local variables initialized outside of the method definition, nor can local variables initialized outside of the method definition be reassigned from within it

A block can access local variables initialized outside of the block and can reassign those variables. We already know that methods can access local variables passed in as arguments, and now we have seen that

## methods can access local variables through interaction with blocks.

we can think of method definition as setting a certain scope for any local variables in terms of the parameters that the method definition has, what it does with those parameters, and also how it interacts (if at all) with a block.

We can then think of method invocation as using the scope set by the method definition. If the method is defined to use a block, then the scope of the block can provide additional flexibility in terms of how the method invocation interacts with its surroundings.

## Summary

- The def..end construction in Ruby is method definition

- Referencing a method name, either of an existing method or subsequent to definition, is method invocation

- Method invocation followed by {..} or do..end defines a block; the block is part of the method invocation

- Method definition sets a scope for local variables in terms of parameters and interaction with blocks

- Method invocation uses the scope set by the method definition

Variable References and Mutability

Saturday, 24 July 2021 8:12 pm

Objects and Variables

What is an object?

An object is a bit of data that has some sort of state or value and associated behavior.

How are objects assigned to variables?

objects can be assigned to variables.

```
`greeting = "Hello"`
=> "Hello"
```

What does the above code tell ruby?

In ruby, how would you describe the relationship between greeting and the string object "Hello"?

`greeting` references or points to the string object.

`greeting` is bound to the String object.

What is the act of doing this greeting = "Hello" known as?

String object represented by the literal 'Hello' is assigned to the variable `greeting`

How is the variable greeting able to reference the string object "Hello"?

It does so by storing the object id of the string object "Hello"

What is the #object_id method used for?

Every object in Ruby has a unique object id, and that object id can be retrieved simply by calling `#object_id` on the object in question. Even literals, such as numbers, booleans, `nil`, and Strings have object ids

Variable Reassignment

What happens if you assign a variable to another variable?

It causes the second variable to reference the same object as the first variable. Both variables will store the same object id.

Since both variables are associated with the same object, using either variable to mutate the object is reflected in the other variable.

What does reassignment to a variable do to that variable? And what doesn't it do?

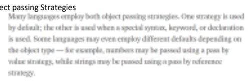
Reassignment to a variable doesn't mutate the object referenced by that variable;

Reassignment binds a different object to the variable.

What does reassignment do to the original object?

The original object is merely disconnected from the variable. It is no longer accessible if there are no other variables pointing to it.

Object passing Strategies



Many languages employ both object passing strategies. One strategy is used by default; the other is used when a special syntax, keyword, or declaration is used. Some languages may even employ different defaults depending on the object type — for example, numbers may be passed using a pass by value strategy, while strings may be passed using a pass by reference strategy.

Mutable and Immutable Objects

What is Mutability and Immutability?

In Ruby which object types are immutable?

`+=`, `*=` and other operators like these, what do these do?

How does a class establish itself as immutable?

By not providing any methods that mutate the object.

What is one characteristic of Mutable Objects?

They are objects of a class that permits changes to the object's state in some way.

What if mutation is permitted by setter methods, is it still a mutable object?

What is a setter method?

What is the signature of setter methods?

What is the array element setter method?

What concept does this code demonstrate?

What concept does this picture demonstrate?

When we assign to `a[1]` what are we actually doing?

Object Passing - Introduction

When we pass an object as an argument to a method, the method can mutate the object or leave it unchanged. What does this ability to mutate an object passed as an argument depend on?

What does the pass by value object passing strategy mean?

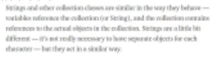
What does the pass by reference object passing strategy mean?

Initial Mental Model for Object Passing Strategy in Ruby

Immutable Objects: What Object passing Strategy does ruby employ for immutable Objects?

Mutable Object: What Object passing Strategy does ruby employ for immutable Objects?

Collections Behavior



Strings and other collection classes are similar in the way they behave — variables reference the collection (or String), and the collection contains references to the actual objects in the collection. Strings are a little bit different — it's not really necessary to have separate objects for each character — but they act in a similar way.

Summary

- Ruby variables are merely references to objects in memory; that is, a variable is merely a name for some object.
- Multiple variables can reference the same object, so mutating an object using a given variable name will be reflected in every other variable that is bound to that object.
- We've also learned that assignment to a variable merely changes the binding; the object the variable originally referenced is not mutated. Instead, a different object is bound to the variable.
- We've also learned that certain object types, primarily numbers and Booleans but also some other types, are immutable in Ruby — unchanging; many other objects are mutable— changeable.
- If you attempt to change an immutable object, you won't succeed— at best, you can create a new object, and bind a variable to that object with assignment.
- Mutable objects, however, can be mutated without creating new objects.
- Finally, we've learned a bit about what pass by value and pass by reference mean. We've established a mental model that says that Ruby is pass by value for immutable objects, pass by reference otherwise.
- This model isn't perfect, but it can be used to help determine whether the object associated bound to an argument will be mutated.

Ruby Object's Mutating and Non Mutating Methods

Monday, 26 July 2021 8:56 pm

Mutating and Non Mutating Methods

What are non-mutating methods?
What are mutating methods?
What is important when deciding whether a method is mutating or non-mutating?
What are the different types of mutating method?

Are there any methods that are mutating for immutable Objects?

Assignment

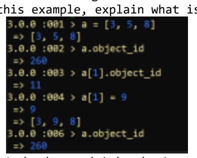
Is assignment mutating? =

What's happening in the below code?
What's happening in the below code example?
What does assignment do?
What does assignment do to the original object referenced by the variable?
What do *=, +=, -=, /=, %= do?
What does the following code do?

Re Assignment

Mutating Methods

What is a mutating method?
A method is said to be mutating with respect to an argument or it calling object if it mutates its value in the process. If the variable references the same object before and after the method is called and changes the state of the object, then the method is said to be mutating with respect to that object.
Is indexed assignment mutating? Which objects use indexed assignment?
Index assignment i.e. #[]= is mutating with respect to its calling object.
It is used by strings, arrays and hashes. It doesn't change the binding of the object.
Why is indexed assignment mutating while simple assignment is not?
Indexed assignment is a method that a class must supply if it needs indexed assignment.
In this example, explain what is happening to ary?



```
3.0.0 :001 > a = [3, 5, 8]
 => [3, 5, 8]
3.0.0 :002 > a.object_id
 => 260
3.0.0 :003 > a[1].object_id
 => 11
3.0.0 :004 > a[1] = 9
 => 9
 => [3, 9, 8]
3.0.0 :006 > a.object_id
 => 260
```

In the above code indexed assignment was used to mutate the array `a`. As we can see that
The object id of `a` remains the same but the state of the object it references has changed.
This means that the []= method mutated the array. However, the object referenced by a[1] has changed. Although the object referenced by the second array element was changed i.e. reassigned to a different object, but `a` still references the same array object but with a different state.

Concatenation is Mutating

What ways are there for String and Array Concatenation? Which are mutating?
#<< , concat and += are used for string and array concatenation.
#<< and #concat are mutating for both strings and arrays. While += is non mutating for both. += only reassigns the target array to a new array.

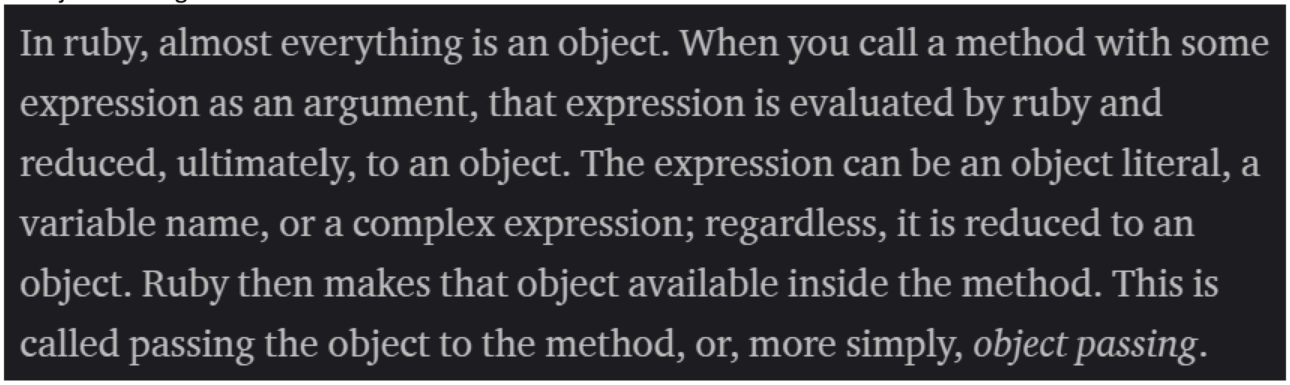
Setters are Mutating

What is the difference between indexed assignment and setter methods?
Indexed assignment and setter methods both use the something= syntax.
While indexed assignment replaces elements of a collection, setter alter the state of the object.

Object Passing in Ruby - Pass by Value or Pass by Reference

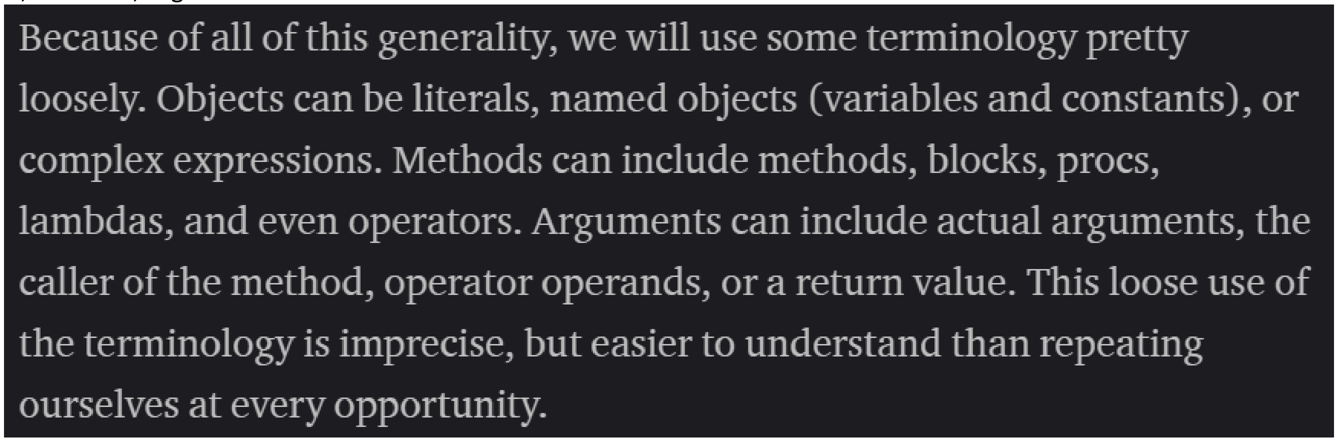
Saturday, 31 July 2021 6:22 pm

What is Object Passing?



In ruby, almost everything is an object. When you call a method with some expression as an argument, that expression is evaluated by ruby and reduced, ultimately, to an object. The expression can be an object literal, a variable name, or a complex expression; regardless, it is reduced to an object. Ruby then makes that object available inside the method. This is called passing the object to the method, or, more simply, *object passing*.

Objects, Methods, Arguments?



Because of all of this generality, we will use some terminology pretty loosely. Objects can be literals, named objects (variables and constants), or complex expressions. Methods can include methods, blocks, procs, lambdas, and even operators. Arguments can include actual arguments, the caller of the method, operator operands, or a return value. This loose use of the terminology is imprecise, but easier to understand than repeating ourselves at every opportunity.

Objects == literals, variables, complex expressions
Methods == methods, blocks, procs, lamdas and operators
Arguments == arguments, the caller of the method, operator operands or a return value.

What is the evaluation strategy used by Ruby?

Ruby uses strict evaluation strategy. According to this strategy, every expression is evaluated and converted to an object before it is passed along to a method.

What are the two most common strict evaluation strategies?

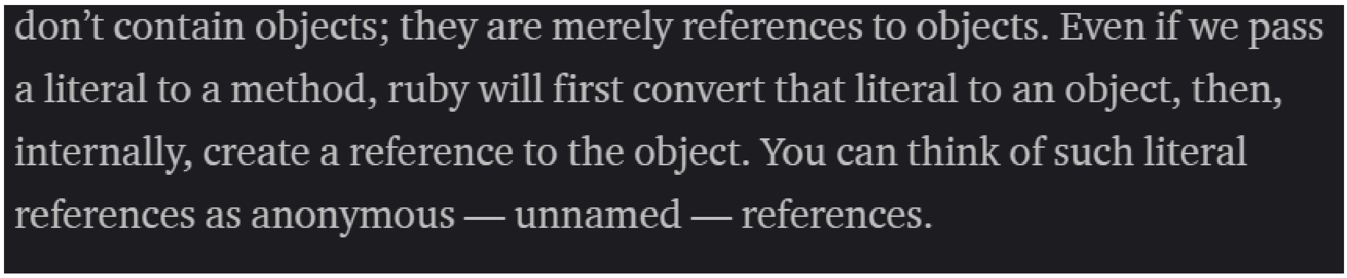
Pass by value and pass by reference. Collectively known as object passing strategies.

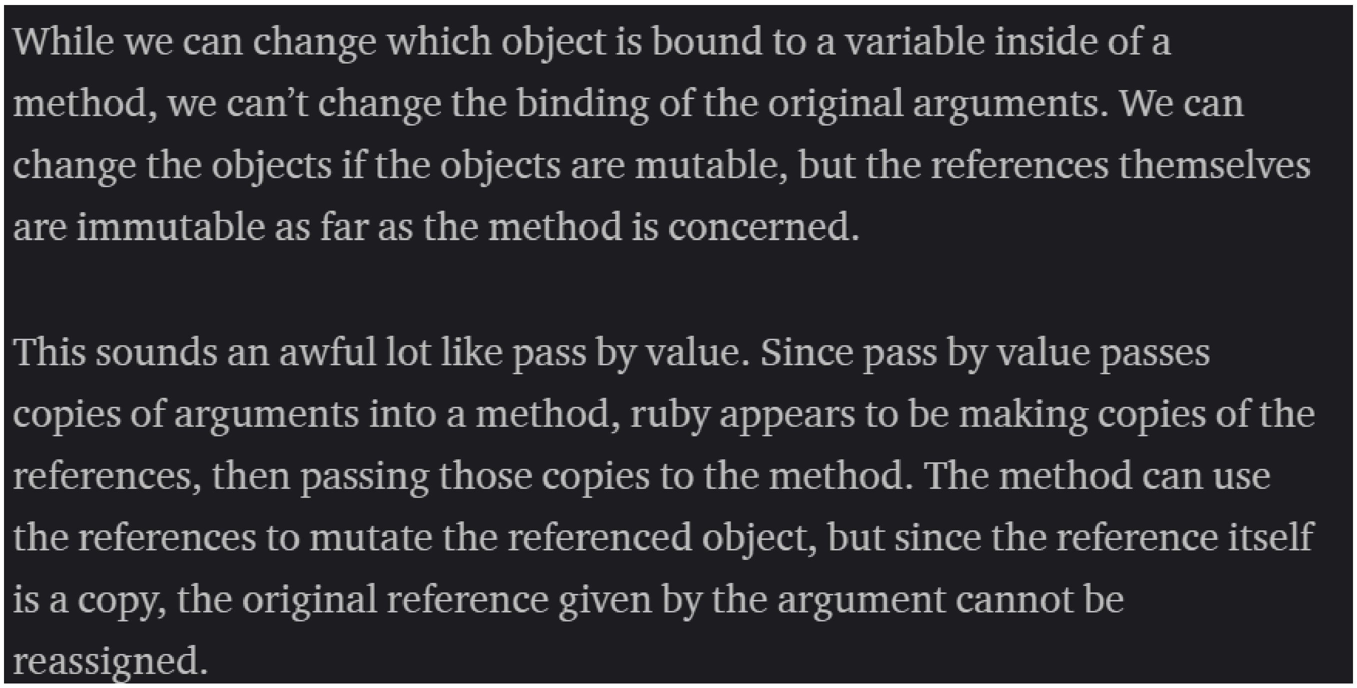
What is pass by value?

A copy of the object is created and it is the copy that gets passed around. Since it is only a copy, it is impossible to change the original object.
Passing around immutable objects in ruby acts a lot like pass by value.
Ruby appears to be pass by value with respect to immutable objects

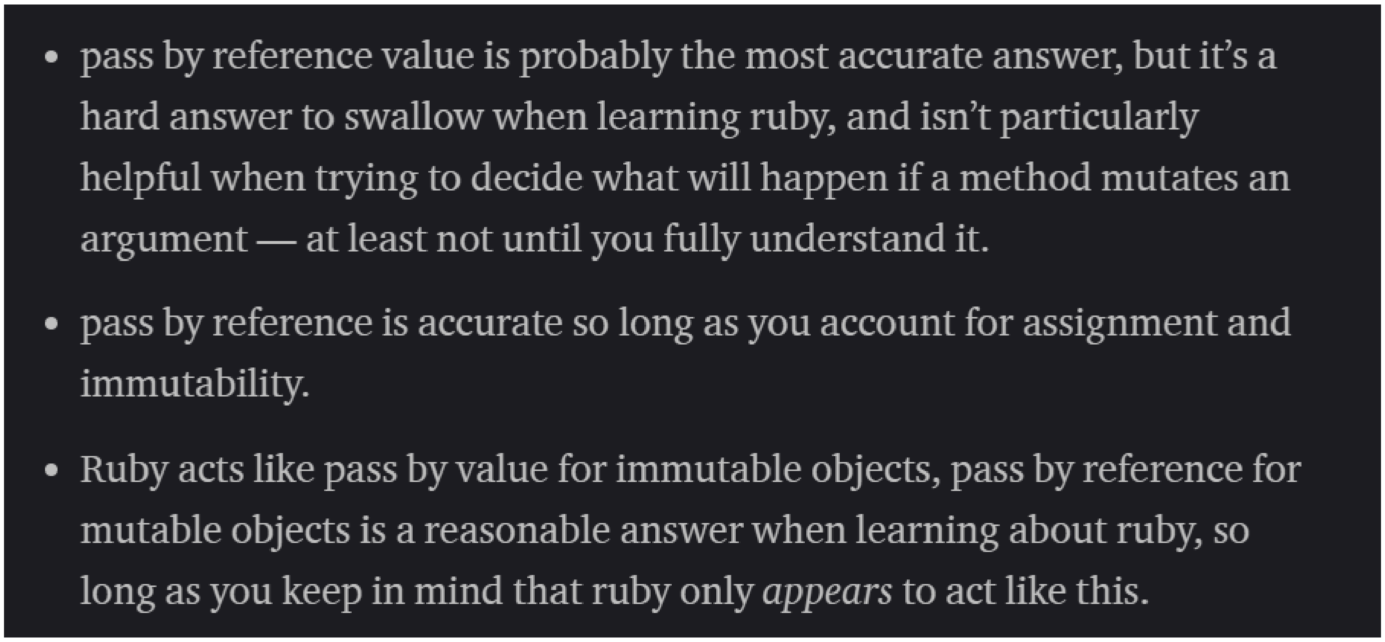
What is pass by reference?

A reference to an object is passed around. This creates an alias between the argument and the original object. Both the argument and the object refer to the same location in memory


don’t contain objects; they are merely references to objects. Even if we pass a literal to a method, ruby will first convert that literal to an object, then, internally, create a reference to the object. You can think of such literal references as anonymous — unnamed — references.


While we can change which object is bound to a variable inside of a method, we can’t change the binding of the original arguments. We can change the objects if the objects are mutable, but the references themselves are immutable as far as the method is concerned.

This sounds an awful lot like pass by value. Since pass by value passes copies of arguments into a method, ruby appears to be making copies of the references, then passing those copies to the method. The method can use the references to mutate the referenced object, but since the reference itself is a copy, the original reference given by the argument cannot be reassigned.


- pass by reference value is probably the most accurate answer, but it’s a hard answer to swallow when learning ruby, and isn’t particularly helpful when trying to decide what will happen if a method mutates an argument — at least not until you fully understand it.
- pass by reference is accurate so long as you account for assignment and immutability.
- Ruby acts like pass by value for immutable objects, pass by reference for mutable objects is a reasonable answer when learning about ruby, so long as you keep in mind that ruby only *appears* to act like this.

# Coding Tips

Monday, 11 October 2021 8:15 pm

# Coding Tips

## Learning through dramatic experience

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.

## Naming Things

Choose descriptive variable and method names. Dont save on characters.

## Variable Names

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 that variable name.

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.

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

## Mutating Constants

Dont Mutate Constants.

> CONSTANTS should be immutable.

## Methods

Its good to extract code to a method. However, the method should do only one thing and its responsibility should be very limited.

### Guidlines for writing good methods:

- Dont display something to the output and return a meaningful value. Returning a value shouldnt be the intent of the method.

- Decide if the method should return a value with no side effects or perform side effects with no return value. The method name should reflect whether it has side effects or not

- 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.

> **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.**

Name your methods approprately i.e capture the intent of the method in the method name. And a method shoud do only one simple thing i.e. mutate or return a meaningful value or print something

# Coding Tips 2

Use new lines to organize code.

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

### Different parts of a program
variable initialization
user input and validation
using the variable

## Should a method return or display?

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

"side effects": either displaying something to the output or mutating an object.

Pay attention to method's side effects vs. return values

if a method has both side effects and a meaningful return value, it's a red flag.

## Naming methods

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

> 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 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

Don't mutate a collection while iterating through it.

You can, however, mutate the individual elements within that collection, just not the collection itself.

## 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 essentially prevents you from accessing the outer scope variable from an inner scope.

```ruby
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:

```ruby
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.

PEDAC Videos

Sunday, 8 August 2021 8:55 pm

https://dkq85ftleqhzg.cloudfront.net/videos/pedac/pedac_video_1_understand_the_problem.mp4

8:50

https://dkq85ftleqhzg.cloudfront.net/videos/pedac/pedac_video_2_examples_and_test_cases.mp4

8:52

https://dkq85ftleqhzg.cloudfront.net/videos/pedac/pedac_video_3_data_structures.mp4

8:53

https://dkq85ftleqhzg.cloudfront.net/videos/pedac/pedac_video_4_algorithms.mp4

8:55

https://dkq85ftleqhzg.cloudfront.net/videos/pedac/pedac_video_5a_code_ruby.mp4

From <https://app.slack.com/client/T0YKP5Z9T/D021391962X>

Collections Basics

Sunday, 22 August 2021 7:37 pm

What are Collections

Collections e.g Strings, Arrays and Hashes are made up of individual elements/objects

In order to be able to work with collections, we need to understand:

- their structure
- how to reference individual elements.
- how to assign individual elements.

How to reference Individual Elements of Collections

Strings

What is the Structure of Strings:

Strings use an integer based index that represents each character in the string. The index starts from 0 and increments by one.

How to reference individual characters of strings:

How to reference multiple characters of strings:

Arrays

What is the Structure of Arrays:

Arrays are: lists of elements/objects that are ordered by an integer based index.

- lists of objects
- ordered by
- a zero/integer based index

How to Reference an element in an Array:

A specific element in an array can be referenced using it's index .
Use the element reference method or the slice method:
Array#[] is alternative syntax for the Array#slice method.
Element Reference — Returns the element at index,

How to reference multiple elements in an array:

In both strings and arrays element reference is done in the same way. What is one difference?

Hashes

What is the Structure of a Hash:

Hashes are a collection which:

- stores elements by associated keys
- Entries are called key-value pairs

- keys are unique
- keys and values can be any type of object

What are the different ways of creating hashes?

Hash::[] - Hash[key,value,...] -> new_hash
Hash[[[key,value],...]] -> new_hash, takes an array of key-value pairs

Hash::new - Hash.new -> new_empty_hash, (defualt value = nil)
Hash.new(obj)-> new_empty_hash, (default value = obj)
Hash.new{|hash,key| block} -> new_hash
In this form, when accessing keys that do not exist in the hash, the block is called with the hash obj and the key and should return the default value. We can also store the value in the hash if required.

What is a default value in a hash? How can default values be set?

How do you refer to values in a hash using their keys?

Can two values have the same key?

Can values in a hash be duplicated?

What is the difference between using Array#fetch method and the Array#[]

Array#[] returns the obj at the index provided or nil if the provided index is out of bounds for the array. The problem is that we cant be sure if the nil obj returned is an actual obj stored in the array or is it the return value for an out of bounds index.
This is where the Array#fetch comes in. If passed in a single argument that is an integer, it return the obj at that integer index. And if the given index is out of bounds, it throws an IndexError.
fetch(index) → obj
fetch(index, default) → obj
fetch(index) {|index| block} → obj
- Tries to return the element at position index,
- but throws an IndexError exception if the referenced index lies outside of the array bounds.
- This error can be prevented by supplying a second argument, which will act as a default value.
- if a block is given it will only be executed when an invalid index is referenced.

### Negative Indices

### Invalid Hash Keys

## Conversion

### Array to strings

### Strings to arrays

### Hashes to arrays

### Arrays to Hashes

## Element Assignment

Selection and Transformation

Monday, 23 August 2021 11:47 am

```
What is Selection?
     Selection is picking certain elements of a collection based on a
     criterion.
What is Transformation?
     Transformation is manipulating all the elements of a collection?
```

# Looping

Monday, 23 August 2021 12:17 pm

# Looping

## What are loops used for?

## Example of a loop

Incrementing each element in array of integers by 1:

```ruby
arr = [1,2,3,4,5] # Init a local variable arr and assign it an array of integers
counter = 0 # init a local variable counter and assign it to an integer 0, this will used as the index to access elements in the array

# We invoke the Kernel#loop method and pass in a block,
loop do
  arr[counter] += 1 # Using the Array#[]+=, Element Reassignment method, we increment the integer at index counter by 1
  counter += 1 # Then we increment the counter by 1
  break if counter == arr.size # check if the value assigned to counter has reached the size of the array which is 1 more than the last index in the given array, if it is we break from the loop
end
```
## Controlling a Loop

To control a loop so that it executes only a certain number of times that is predetermined by us, we need to keep track of how many iterations have been performed. To do that we use a variable counter and increment counter by 1 each time an iteration is performed.

```ruby
counter = 0

loop do
  puts "hello"
  counter += 1
  break if counter == 5
end

```

### Break Placement

Also, if an if condition is appended to the break statement, then the break will be executed when that condition becomes true

The placement of break in your block for loop, changes how your loop will be executed.

If placed at the beginning of your loop then that mimics a while loop, where the condition to break from the loop is checked at the beginning of each iteration.

If placed at the end, this will mimic the do/while loop, in which case the loop will be executed atleast once, the condition to break will checked at the end of the first iteration and all subsequent iterations.

### Next

`next` is used in loops to skip the rest of the current iteration, and start the next one.

Similar to break, when next is executed, any code after it will be ignored.

## Iterating over collections

### Strings

```ruby
alphabet = ('a'..'z').to_a.join("")
counter = 0

loop do
  break if counter == alphabet.size
  puts alphabet[counter]
  counter += 1
end
```

Instead of using a specific condition in the if modifier to break out of the loop, its preferrable to use a more general condition. Because if some how the specific condition in the if modifier is surpassed, it would result in an infinite loop.

### Arrays

```ruby
colors = %w(green blue purple orange)

counter = 0

loop do
  break if counter == colors.size

  puts "I am the color #{colors[counter]}"

  counter += 1
end
```

### Hashes

Hashes use key-value pairs instead of zero based index to store elements. This means that each value in a hash is associated with a specific key.

Since hash keys can be any type of object a simple counter variable won't allow us to fetch the values.

```ruby
number_of_pets = {
  'dogs' => 2,
  'cats' => 4,
  'fish' => 1
}

pets = number_of_pets.keys
counter = 0

loop do
  break if counter == number_of_pets.size # Use Hash#size
  current_pet = pets[counter] #

  current_pet_number = number_of_pets[current_pet]

  puts "I have #{current_pet_number} #{current_pet}"

  counter += 1
end
```

Iterating over hashes is a two step process:
- Use a counter to iterate over keys of the hash---> use the counter to access the keys
- Use the key to fetch the value for that key

## Conclusion

Looping over Collections comprises of four basic elements:

- a loop
- a counter
- a way to retrieve the current value
- a way to exit the loop

Sorting

Saturday, 28 August 2021 8:54 pm

What is Sorting?
    Sorting is setting the order of the items in a collection according to a certain criterion
Which methods are used in Ruby for sorting?
    \`sort\` and \`sort_by\`

Understand the way in which sort applies criterion in order to return an ordered collection.

Sorting is carried out by *comparing* the items in a collection with each other, and ordering them based on the result of that comparison. *Comparison is at the heart of how sorting works*.

How does the <=> operator work?
    This method performs *comparison* between two objects of the same type and returns a `-1`, `0`, or `1`, depending on whether the first object is less than, equal to, or greater than the second object; if the two objects cannot be compared then `nil` is returned.
    The return value of the `<=>` method is used by `sort` to determine the order in which to place the items.

How does ruby know how to order an array of integers of strings using sort?
    Ruby uses the \`<=>\` operator to sort elements in arrays. Any object in a collection that we want to sort must implement a \`<=>\` comparison method.
    If `<=>` returns `nil` to `sort` then it throws an argument error.

What are two things you need to know if want to sort a collection that contains a particular type of objects (strings or integers)
- Does that object implement the <=> comparison method?
- What is the specific implementation of <=> comparison method for that object type?

How does Ruby understand the concepts like greater than less and equal to in the case of strings?
    The order of strings is determined by the character's position in the ASCII table. It is this ASCII character order that determines the result when we compare one ASCII character to another using the String <=> method.

How can you determine a string's ASCII position?
    By using the String #ord method.

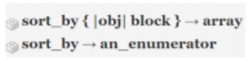
- Uppercase letters come before lowercase ones
- Digits and most punctuation come before letters
- The extended ASCII table containing accented and other characters. This comes after the main ASCII table.

How does String#<=> compare multi character strings?
    `String#<=>` compares multi-character strings character by character, so the strings beginning with `'a'` will come before those beginning with `'b'`; if both characters are the same then the next characters in the strings are compared, and so on.

    If the strings are of different lengths, and the strings are equal when compared up to the shortest length, then the longer string is considered greater than the shorter one.

How does Array#<=> method work?
    Each object in each array is compared (using the <=> operator).
    Arrays are compared in an “element-wise” manner; the first element of ary is compared with the first one of other_ary using the <=> operator, then each of the second elements, etc… As soon as the result of any such comparison is non zero (i.e. the two corresponding elements are not equal), that result is returned for the whole array comparison.
    If all the elements are equal, then the result is based on a comparison of the array lengths. Thus, two arrays are “equal” according to Array#<=> if, and only if, they have the same length and the value of each element is equal to the value of the corresponding element in the other array.

The Enumerable #sort_by method



sort_by { |obj| block } → array
sort_by → an_enumerator

How does the sort_by method work?
    Sort_by takes a block. The block is invoked for each element of the Array/ Hash and the return values of the block for each element are compared using <=>. The array or hash is sorted based on return value of the block for each element.

The sort Method

The sort method is found in which classes?
    Array and Enumerable, so we can use it for hashes as well.
The sort_by method is found in which clasess?
    Arrays and Enumerable, so we can use it on hashes.
How does the sort method work?
    The Array#sort method returns a new array in which the elements are sorted in ascending order. It takes an optional block which takes two arguments. The block must return-1,0 or 1. The sort method uses the comparison method also called the spaceship operator <=> to do comparisons. The return value of <=> is used by sort to determine the order in which to place the items in the new array. If <=> returns nil to sort then it throws an argument error.

Nested Collections

Wednesday, 1 September 2021 3:35 pm

How are copies of objects made? What is a shallow copy?

Ruby provides two methods that let us copy an object, including collections: `dup` and `clone`. Both of these methods create a *shallow copy* of an object. This means that only the object that the method is called on is copied. If the object contains other objects - like a nested array - then those objects will be *shared*, not copied. This has major impact to nested collections.

What does `freeze` actually freeze?

freeze only freezes the object it's called on. If the object it's called on contains other objects, those objects won't be frozen. For example, if we have a nested array the nested objects can still be modified after calling freeze.

What is the main difference between clone and dup?

What does `freeze` actually freeze? `freeze` only freezes the object it's called on. If the object it's called on contains other objects, those objects won't be frozen. For example, if we have a nested array the nested objects can still be modified after calling `freeze`.

Working With Blocks

Wednesday, 1 September 2021 6:44 pm

```
[[1, 2], [3, 4]].each do |arr|
  puts arr.first
end
# 1
# 3
# => [[1, 2], [3, 4]]
```

The Array#each method is being called on the **multi-dimensional array** [[1, 2], [3, 4]]. Each inner array is **passed to the block in turn** and **assigned to the local variable** arr.
The Array#first method is called on arr and returns the object at index 0 of the current array - in this case the integers 1 and 3, respectively. The puts method then **outputs a string representation of the integer**. puts returns nil and, since this is t**he last evaluated statement within the block**, the return value of the block is therefore nil. each doesn't do anything with this returned value though, and since the return value of each is the calling object - in this case the nested array [[1, 2], [3, 4]] - this is what is ultimately returned.

- What is the type of action being performed (method call, block, conditional, etc..)?
- What is the object that action is being performed on?
- What is the side-effect of that action (e.g. output or destructive action)?
- What is the return value of that action?
- Is the return value used by whatever instigated the action?

```
my_arr = [[18, 7], [3, 12]].each do |arr|
  arr.each do |num|
    if num > 5
      puts num
    end
  end
end
```

Action	Object	Side Effect	Return Value	Is the Return Value used
Variable Assignment	Outer array	none	Outer array	no
Method call each	Outer array	none	Same outer array	Yes used by variable assignment to my_arr
Block Execution	Each Sub array	none	The sub array	no
Method call each	Each sub array	none	Sub array	Yes used to determine the return value of outer block
Block execution	Each element of each sub array	none	nil	no
comparison	Element of the sub-array in current iteration	none	Boolean	Yes, By if the conditional
If conditional	Return value of comparison	none	nil	Yes, used to determine the return value of the inner block
Method Call puts	Each element	Outputs string representation of num	nil	no

```
[[1, 2], [3, 4]].map do |arr|
  puts arr.first
  arr.first
End
```

Action	Object	Side Effect	Return Value	Is the Return Value used
Method call (map)	The outer array	None	A new Array [1,3]	No
Block Execution	Each sub array	None	1 and 3	Yes used by map for transformation
Method call first	Each inner array	none	1 and 3	Yes by puts
Method call puts	Element at index 0 of each sub array	Outputs the string representation of 1 and 3	Nil	no
Method call first	Each inner array	none	1 and 3	It is the return value of the block

```
[[1, 2], [3, 4]].map do |arr|
  arr.map do |num|
    num * 2
  end
End
```

Action	Object	Side Effect	Return Value	Is the Return Value used

The Array#map method is being invoked on the multi dimensional array [[1,2], [3,4]]. The Array#map method is passed a block. This block is invoked once for each sub array of the outer array and each sub array is passed to the block in turn and assigned to the block local variable arr. The Array#map method returns a new array containing the return value of the block for each block execution/ iteration.

For each iteration of the outer block, the Array#map method is invoked on the subarray in the current iteration of the outer block. This Array#map invocation is passed a block which is invoked once for each element in the current sub array. The return value of the the Array#map method invocation on lines 2 to 4, is a new array containing the return value of the inner block for each iteration of this block.

The return value of the inner block for each iteration is the return value of the expression `num*2` for the current element of the current sub array, since it is the last expression evaluated in the inner block.

The inner Array#map invocation returns the arrays [2,4] on the first iteration of the outer block and [6,8] on the second iteration. The outer Array#map invocation also return a new array containing the return values of the inner Array#map method invocation. So the return value of the outer map method is [[2,4], []6,8]

Map returns a new array containing values transformed based on the block's return value.

Tuesday, 19 October 2021 3:10 pm

Variables and Local Variable Scope

Thursday, 16 September 2021 4:36 pm

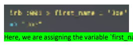
How local variables interact with method invocations with blocks and method definitions

What is a variable?

Variables are used to store data to be referenced and manipulated in a program.
They are also used to label data with a descriptive name so our programs can be understood with more clearly.
Variable can be thought of as containers that hold information.
Their sole purpose is to label and store data in memory.
Variables act as pointers to a place or address space in memory.

Variable Assignment Definition

What is variable assignment?
Variable assignment is essentially setting a variable to point towards an object in memory.



```
irb :001 > first_name = 'Joe'
=> "Joe"
```

Here, we are assigning the variable `first_name` to the string object `Joe`.

What does the gets method do? Which class does it belong to?
The `gets` method waits for an input from the user through the keyboard and keeps waiting until the `enter` key is pressed, at which point it returns the characters entered as a string object.

Local Variable Scope Definition

What does a variable's scope mean?
A variable's scope determines where in a program it is available for use. In other words, it determines where in a program it can be referenced.
How is a variable's scope determined?
A variable's scope is determined by where it is initialized or created.
How is a variable's scope defined in Ruby?
In Ruby it is defined by method definitions or by a block?

Variables and Method Definitions

How does a method definition determine a local variable's scope?
Method definitions have self-contained scope with respect to local variables.
They create their own scope which is separate from the scope of the rest of the program.
That means that you cannot reference or modify any local variables that are not initialized or created inside the method's body. It also means that any local variables initialized inside the method's body are also not available outside the method definition.
A method definition has no notion of "outer" or "inner" scope

Variables and Blocks.

What is a block?
A block is a piece of code that follows a method's invocation and is delimited by either curly breaces `{ }` or `do..end`.
How does a block determine a local variable's scope?
Blocks create a new scope for local variables. The scope created by a block following a method invocation can be thought of as an inner scope for local variables. Nested block will create nested scopes.
Inner scope can access local variables initialized in an outer scope . But outer scope cannot access any local variables initialized in an in inner scope.
How do peer blocks interact with local variables?
Peer blocks cannot reference variables initialized in other blocks.
How do nested blocks interact with local variables?
Nested blocks follow the same rules for inner and outer scoped variables.

# Variables and Scope

##
## `b = a` assigning a variable to reference another variable basically means pointing the first variable to the same address space in memory as the second variable.

## If you call a method that mutates the caller, it will change the value in that address space and any variables also pointing there will be affected.

## Reassignment including += does not mutate the caller / variable; instead it points/ binds the variable to a new object.

## numbers in ruby are immutable.

## Some operations mutate the address space in memory but pthers change the variable to point to a different address space in memory.

## When we are passing in arguments to a method, we are essentially assigning a variable to another variable

# Method Definitions Scoping Rules:

## A variable's scope determines where in a program a variable is available for use.

## A variable's scope is defined by where the variable is initialized.

## In ruby variable scope is defined by a block.

## Inner scope can access variables initialized in an outer scope. But not vice versa.

## We can use the same variable inside method definitions as those outside the method definition, why? Because method definitions are self contained with respect to local variables.

## LVINVO, So the variables inside have no ability to be seen outside.

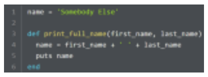
## LVONVI, So the top level variables are not available inside a method definition.

## Assignment never changes the value of an object; instead, it creates a new object, and then stores a reference to that object in the variable on the left.

## assignment to a variable (an object) never mutates the object that is referenced.

## Shadowing occurs when a block argument hides a local variable that is defined outside the block.

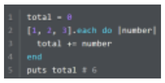
Local Variable Scope for Method Definitions Examples:



```
1  name = 'Somebody Else'
2
3  def print_full_name(first_name, last_name)
4    name = first_name + ' ' + last_name
5    puts name
6  end
```

In the above code, we can't use or change the name variable from line 1 from inside the print_full_name method. We can, however, create and use a different name variable that is locally scoped to the method. That is why lines 3 and 4 work without changing the value of name from line 1.

Local Variable Scope for Blocks Example:



```
total = 0
[1, 2, 3].each do |number|
  total += number
end
puts total # 6
```

the block can access and modify variables that are defined outside of the block. Thus, blocks can access and modify total. However, any variables initialized inside the block (such as number) can't be accessed outside the block.

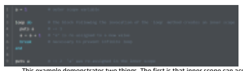
What if a local variable and a method were to share the same name?

Ruby will first search for the local variable, and if it is not found, then Ruby tries to find a method with the given name. If neither local variable nor method is found, then a NameError message will be thrown. To remove some of the ambiguity in a situation like this, we can indicate that we want to call the method by including a set of empty argument parentheses with the method invocation.

Scoping Rules for Constants

In procedural style programming, constants behave like globals.
Constants are said to have lexical scope,

Local Variable interaction with blocks

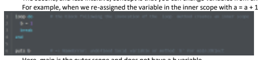


```
a = 1           # [illegible]

loop do         # [illegible]
  puts a        # [illegible]
  a = a + 1     # [illegible]
  break         # [illegible]
end

puts a          # [illegible]
```

This example demonstrates two things. The first is that inner scope can access outer scope variables.
The second, and less intuitive, concept is that you can change variables from an inner scope and have that change affect the outer scope.
For example, when we re-assigned the variable in the inner scope with a = a + 1, that reassignment was visible in the outer scope.

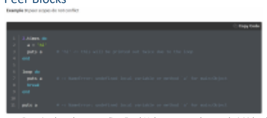


```
loop do         # [illegible]
  b = 1
  break
end

puts b          # [illegible]
```

Here, main is the outer scope and does not have a b variable.
Remember that where a variable is initialized determines its scope.
In the above example, b is initialized in an inner scope.

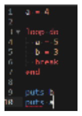
Peer Blocks



[illegible]

```
1  3.times do
2    a = 'hi'
3    puts a      # [illegible]
4  end
5
6  loop do
7    puts a      # [illegible]
8    break
9  end
10
11 puts a        # [illegible]
```

Executing the code puts a on lines 7 and 11 throws an error because the initial a = 'hi' is scoped within the block of code t hat follows the times method invocation.
Peer blocks cannot reference variables initialized in other blocks. This means that we could use the same variable name a in the block of code that follows the loop method invocation.
However, it's not the same variable as in the first block.



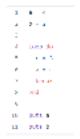
```ruby
a = 4

loop do
  a = 5
  b = 3
  break
end

puts b
puts a
```

The code outputs 5 and raises an error, undefined local variable or method b .

here we have two scopes. An inner scope which is defined by the do..end block and outer scope which includes everything else. Now comes the important part -> local variables that are initialized in an inner scope CAN'T be accessed in the outer scope, but local variables that are initialized in the outer scope CAN be accessed in an inner scope.



```ruby
1   a = 4
2   [illegible] = a
3
4   loop do
5     c = 3
6     a = c
7     break
8   end
9
10  puts a
11  puts b
```

On line 1 we are initializing a local variable a with in outer scope.
On line 4 we call the loop method and pass the do..end block to it as an argument. Inside of this block we initialize new local variable c which has this block as its scope.
On line 6 we reassign the local variable a so that it points now to the same object that the local variable c is referencing. Since we are still in the inner scope defined by the block, local variable c is accessible
On line 10 we call the puts method and pass the local variable a as an argument and this code outputs the value of variable a which is 3 now since we reassigned it inside of the block.

Method definition and Method Invocation

Thursday, 16 September 2021 4:40 pm

Method Invocations and blocks

Technically any method can be called with a block, but the block is only executed if the method is defined in a particular way.

A block is part of the method invocation. In fact, method invocation followed by curly braces or do..end is the way in which we define a block in Ruby.

Essentially the block acts as an argument to the method. In the same way that a local variable can be passed as an argument to a method at invocation, when a method is called with a block, the block acts as an argument to that method.

The way that an argument is used, whether it is a method parameter or a block, depends on how the method is defined.

Blocks and methods can interact with each other; the level of that interaction is set by the method definition and then used at method invocation.

Method Definition and Method Invocation with a block

Method definitions cannot directly access local variables initialized outside of the method definition, nor can local variables initialized outside of the method definition be reassigned from within it. A block can access local variables initialized outside of the block and can reassign those variables. We already know that methods can access local variables passed in as arguments, and now we have seen that methods can access local variables through interaction with blocks.

Method definitions set a certain scope for any local variables in terms of the parameters that the method definition has, what it does with those parameters, and also how it interacts (if at all) with a block.
Method Invocation then uses the scope set by the method definition

If the method is defined to use a block, then the scope of the block can provide additional flexibility in terms of how the method invocation interacts with its surroundings.

Summary

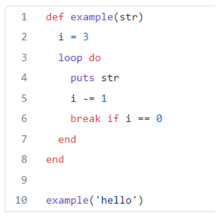
- The `def..end` construction in Ruby is method definition
- Referencing a method name, either of an existing method or subsequent to definition, is method invocation
- Method invocation followed by `{..}` or `do..end` defines a block; the block is *part of* the method invocation
- Method definition *sets* a scope for local variables in terms of parameters and interaction with blocks
- Method invocation *uses* the scope set by the method definition

implicit return value of method invocations and blocks

Method Invocations return the evaluated result of the last expression in the method definition that is executed unless an explicit return comes before it.

Ruby methods ALWAYS return the evaluated result of the last line of the expression unless an explicit return comes before it.

Method Definition



```
1   def example(str)
2     i = 3
3     loop do
4       puts str
5       i -= 1
6       break if i == 0
7     end
8   end
9
10  example('hello')
```

-
- On lines 1–8 we are defining the method example which takes 1 parameter. On line 10 we are calling the method example and passing in the string hello as an argument to it.
- Methods are defined with parameters, but they are called with arguments.
- On line 2 of this code we are initializing the local variable i and assigning to it an integer with value 3
- On line 3 we are calling the method loop (yes, loop is actually a method of the Kernel module) and passing in the do..end block as an argument. So the block here is passed to a method call as an argument
- And also on line 4 we are actually calling the method puts and passing in local variable str to it as an argument
- On line 5 the local variable i is reassigned. -= is actually reassignment and it is Ruby's syntactical sugar for i = i — 1 . And while we are talking about syntactical sugars, that code is also one, since — is not an operator but a method and that code can also be written as i = i.-(1) . So inside of this code we are actually, reassigning the local variable i to the return value of a method call Integer#- on a local variable i with integer 1 passed to it as an argument.
- On line 6 we are breaking out of the loop by using the keyword break if the value of the object that local variable i is referencing is equal to 0.
- On line 10 we are calling the method example and passing in string hello as an argument.
- Finally, but not less important, this code outputs string hello 3 times and returns nil. That is very important to distinguish. The last evaluated expression is returned since we don't have an explicit return inside of the method definition. That last evaluated expressionis break if i == 0 in this case, which returns nil.

Variable Shadowing

Wednesday, 22 September 2021 5:40 pm

Variable Shadowing Definition

This is the concept called **variable shadowing** and it happens when name of the block parameter is the same as the name of the local variable which was initialized outside of the block.
Variable shadowing prevents access to outer scoped variables having the same name as the block parameter.

Variable shadowing also prevents us from making changes to the outer scoped variable.

disregards the outer scoped local variable.

How to prevent variable shadowing:

Varaible shadowing can be prevented by using a different name for the block parameter

```
1   a = 4
2   b = 2
3
4   2.times do |a|
5     a = 5
6     puts a
7   end
8
9   puts a
10  puts b
```

On line 1 we are initializing the local variable a and setting its value to 4.
On line 2 we are initializing the local variable b and setting its value to 2.
On line 4 we are calling the times method on integer 2 and passing in the do..end block as an argument with one parameter a .
What is happening on line 5 ? We are reassigning the local variable a that was initialized in the outer scope right? Not really…

Variables as pointers

Thursday, 16 September 2021 4:39 pm

Variables as pointers

In Ruby, variables act as pointers to an address space in memory

The variable does not contain the actual object. Instead it contains a reference to a particular area in the computer's memory that actually contains the object.

variables are pointers to physical space in memory.

variables are essentially labels we create to refer to some physical memory address in your computer.

Some operations mutate the address space, while others simply make the variable point to a different address space.

If you call a method that mutates the caller, it will change the value in that object's address space, and any other variables that also point to that object will be affected.

```ruby
a = "hi there"
b = a
a = "not here"

Puts b
```

On line `1` the local variable `a` is initialized to the string object `"hi there"` Then, on line `2` we initialize the local variable `b` to the same string object that `a` points to. On line `3` we re-assign `a` to a different string object `"not here"`. So now `a` and `b` are no longer pointing to the same object. `b` still points the initial string object it was initialized to.

```ruby
a = "hi there"
b = a
a << ", Bob"

Puts b
```

On line `1` the local variable `a` is initialized to the string object `"hi there"` Then, on line `2` we initialize the local variable `b` to the same string object that `a` points to. On line `3` we are invoking the method `String#<<` on the string object referenced by `a`. This method appends a new string object `" ,Bob"` to the string object referenced by `a`. This is a mutating method. This mutation actually changes the value of the object referenced by `a` in the address space in memory and also returns the same initial string object but only mutated. So `a` and `b` still point to the same string object they were initialized to but whose value or state has now changed.

Puts vs Return

Thursday, 16 September 2021 4:39 pm

When we call the `puts` method, we're telling Ruby to print something to the console.
However, `puts` does not return what is printed to the screen. Instead it returns the object `nil`.

`puts` is a method in the `Kernel` module in the Ruby Core library. It takes one or multiple arguments and prints the string representation of the arguments to the console. It invokes the object's `to_s` method to convert it to a string if it's not a string. It also prints a new line after each object. In case of an array, it prints each element on a separate line.

However, `return` is a keyword in Ruby that is used to explicitly return or exit from a method . And what is passed to `return` as an argument is returned back to the line where the method was called. It does not output or print anything to the console. Although an explicit return` keyword is not required to return from a method. In Ruby, all methods invocations return the evaluated result of the last expression that is executed in the method definition. Unless the keyword `return` is used before the last expression. In which case the last line is not executed and the method returns immediately to the line it was invoked on.

On the other hand, `puts` does not return what is output. Instead it always returns `nil`

Expressions **do** something, but they also **return** something. The value returned is not necessarily the action that was performed.

Truthiness & False vs. nil

Thursday, 16 September 2021 4:40 pm

What is Truthiness?

Truthiness means that ruby considers more than the `true` object to be truthy.

What do "truthy" and "falsey" mean?

A non-boolean object that evaluates as true in a boolean context (for example in an if conditional) is known as "truthy," and a non-boolean object that evaluates as false in a boolean context is known as "falsey.
This does not mean that they are equal to the `true` or `false` objects. Only that they are considered as true or false. In Ruby only the objects `false` and `nil` are falsey. All other objects are truthy.

The object `nil` is not equal to the boolean object `false` but it evaluates as false in a boolean context e.g. when used in an if conditional or with logical operators like `&&` and `||`.

Arrays and Their Methods

Thursday, 16 September 2021 4:38 pm

Array#each

```
[1, 2, 3, 4].each { |num| puts num }
```

We are invoking the `Array#each` method on the array `[1,2,3,4]`. The `{}` or `do..end` next to the `each` method invocation is a block which is passed to the `each` method invocation as an argument.

The `each` method iterates over the array object `[1,2,3,4]` and executes the code within the block once on each iteration passing in the current element from the array to the block as an argument where it is assigned to the block parameter `num` which is a block local variable.

On each iteration, within the block, the `Kernel#puts` method is called and the object currently referenced by `num` is passed to it as an argument. The `puts` method prints the string representation of the current element to the console and returns `nil`. When the iterations are complete, the `each` method returns the original calling array. The `Array#each` method does not consider or use the return value of the block.

What does the Array#each method do?

The `Array#each` method iterates over the calling array object. It takes a block as an argument and invokes the block on each iteration passing each element of the array to the block as an argument where its assigned to the block local variable or block parameter.

When the block is invoked the code within the block is executed with the block parameter evaluating to the current element in the iteration. The `Array#each` method does not consider or use the return value of the block. When the iterations are complete the `each` method returns the original calling array.

Array#any?

```
[1, 2, 3].any? do |num|
  num > 2
end
# => true
```

We are invoking the `Enumerable#any?` method on the array object referenced by `ary`. The `do..end` following the `any?` method invocation is a block that is passed to the `any?` method call as an argument.

The `any?` method iterates over the calling array object `[1,2,3]` and executes the code within the block on each iteration passing in the current element as an argument to the block where it is assigned to the block parameter `num` which is a block local variable.

Within the block we are invoking the `Integer#>` method on the current integer object referenced by `num` and passing in the integer object `2` as an argument. `num > 2` will always return a boolean object either `true` or `false`. It compares the calling object to the argument and returns `true` if current integer referenced by `num` is greater than `2` and `false` otherwise.

The return value of the block for each iteration will be the evaluated result of the `num>2` since it is the last expression evaluated in the block.

The `any?` method returns `true` if the block returns a truthy object i.e. any object other than `false` or `nil`, for any of elements of the array passed to the block. Once a truthy object is returned by the block for any of the elements in the array, the `any?` method stops iterating and returns `true`. If, however, the block returns `false` or `nil` for all the elements in the array, `any?` will return `false`.

Array#select

```
[1, 2, 3].select do |num|
  num.odd?
End
```

The above example returns the array object `[1, 3]` but doesn't output anything to the console.

On line `1` we are invoking the `Array#select` method on the array `[1,2,3]`. The `{ }` or `do..end` following the `select` method invocation is a block which is passed to the `select` method invocation as an argument.

The `select` method iterates over the array object `[1,2,3]` and executes the code within the block once on each iteration passing in the current element from the array to the block as an argument where it is assigned to the block parameter `num` which is a block local variable. Inside the block, we are invoking the `Integer#odd?` method on the current element that `num` references. `odd?` returns the boolean object `true` if the object referenced by `num` is odd and `false` if the object is even.

`select` performs selection based on the truthiness of the block's return value.
Since the last expression evaluated in the block is `num.odd?`, the return value of this expression will be the return value of the block on each iteration. `select` evaluates the return value of the block on each iteration and if it is truthy then the current element is selected and stored in a new array. If it is falsey then it is not selected.

`select` returns a new array containing those elements from the calling array `[1,2,3]` for which the block's return value is truthy. In the above example the block returns `true` for the integers `1` and `3`. So `select` will store these objects in a new array. This new array will be the returned object of the `select` method invocation on line `1`.

Array#all?

`all?` functions in a similar way to `any?`. It also looks at the truthiness of the block's return value, but the method only returns `true` if the block's return value in **every** iteration is truthy (that is, not `false` or `nil`).

Array#each_with_index?

```
[1, 2, 3].each_with_index do |num, index|
  puts "The index of #{num} is #{index}."
end

# The index of 1 is 0.
# The index of 2 is 1.
# The index of 3 is 2.
# => [1, 2, 3]
```

Array#map

```
[1, 2, 3].map do |num|
  num * 2
end
```

The above example returns the array object `[2,4,6]` but doesn't output anything to the console.

On line `1` we are invoking the `Array#map` method on the array `[1,2,3]`. The `{ }` or `do..end` following the `map` method invocation is a block which is passed to the `map` method invocation as an argument.

The `map` method iterates over the array object `[1,2,3]` and executes the code within the block once on each iteration passing in the current element from the array to the block as an argument where it is assigned to the block parameter `num` which is a block local variable.

Inside the block, we are invoking the `Integer#*?` method on the current element that `num` references and passing in the integer `2` as an argument. `*` returns the result of multiplying the current element of the array with the integer `2`. Since this is the last expression evaluated in the block, the returned object by `num*2` is the return value of the block for each iteration.

`map` uses the return value of the block on each iteration to perform transformation of the array. On each iteration, the block's return value is stored in a new array and once all iterations are complete this new array is returned by the `map` method invocation.

Array#each_with_object

```
{ a: "ant", b: "bear", c: "cat" }.each_with_object([]) do |pair, array|
  array << pair.last
end
# => ["ant", "bear", "cat"]
```

We are invoking the `Enumerable#each_with_object` method on the hash object `{ a: "ant", b: "bear", c: "cat" }` and passing in an empty array object as an argument. The `do..end` following the `each_with_object` method invocation is a block that is passed to the it as argument as well.

The `each_with_object` method iterates over the calling hash object `{ a: "ant", b: "bear", c: "cat" } and executes the code within the block on each iteration passing in the current key -value pair and the empty array object as arguments to the block where they are assigned to the block parameters `pair` and `array` respectively. The key-value pairs are assigned as an array of the current key and associated value with that key e.g. `[:a, "ant"].

Within the block, we are invoking the `Array#<<` method on the array object referenced by `array` and passing in the the return value of `pair.last` method invocation as argument to the `Array#<<` method call. The `pair.last` returns the last element of each key -value pair passed to the block. And the `Array#<<` method appends the current value to the end of array object referenced by `array`. So essentially, we are storing the values for each key -value pair from the original hash in the array `array`.

The `each_with_object` method will return the array object referenced by the `array` which is the same object that was passed to it as an argument.

Hashes and their Methods

Thursday, 16 September 2021 4:38 pm

Hash#each

```
Hsh = {a: 1, b: 2, c: 3}
hsh.each { |key, value| puts "#{key}, #{value}"}
```

On line `1` we are assigning the hash object `{a: 1, b: 2, c: 3}` to the local variable `hsh`.

On line `2` we are invoking the `Hash#each` method on the hash object referenced by `hsh` . The `{ }` or `do..end` next to the `each` method invocation is a block which is passed to the `each` method invocation as an argument.

The `each` method iterates over the hash object `{a: 1, b: 2, c: 3}` and executes the code within the block once on each iteration passing in the current key-value pair from `hsh` to the block as arguments where the current key is assigned to the first block parameter `key` and the current value is assigned to the second block parameter `value` which are both block local variables.

On each iteration, within the block, the `Kernel``#puts` method is called and the string object `"#{key}, #{value}"` is passed to it as an argument. The `#{key}` and `#{value}` are string interpolation which concatenates the surrounding string with the string representation of the returned objects of the ruby expressions inside the `#{ }`.

`key` and `value` return the current key and value objects referenced by these local variables. The `puts` method prints the string `"#{key}, #{value}"` to the console and returns `nil`. When the iterations are complete, that is, when the code in the block has been executed once for each key-value pair in `hsh`, the `each` method returns the original calling hash object referenced by `hsh` . The `Hash#each` method does not consider or use the return value of the block.

Hash#each_with_index

```
{ a: "ant", b: "bear", c: "cat" }.each_with_index do |pair, index|
  puts "The index of #{pair} is #{index}."
end

# The index of [:a, "ant"] is 0.
# The index of [:b, "bear"] is 1.
# The index of [:c, "cat"] is 2.
# => { :a => "ant", :b => "bear", :c => "cat" }
```

Hash#select

```
Hsh = {:a=>1, :b=>2, :c=>3}
hsh.select {|key, value| value >= 2}
 => {:b=>2, :c=>3}
```

On line `2` we are invoking the `Hash#select` method on the hash object referenced by `hsh` . The `{ }` or `do..end` following the `select` method invocation is a block which is passed to the `select` method invocation as an argument.

The `select` method iterates over the hash object `{a: 1, b: 2, c: 3}` and executes the code within the block once on each iteration passing in the current key-value pair from `hsh` to the block as arguments where the current key is assigned to the first block parameter `key` and the current value is assigned to the second block parameter `value` which are both block local variables.

Inside the block, we are invoking the `Integer#>=` method on the current object that `value` references. `value >= 2` returns the boolean object `true` for the integers `2` and `3` and `false` for `1`. Since `value >= 2` is the last expression evaluated in the block, the block's return value on each iteration shall be either `true` or `false`.

`select` performs selection based on the truthiness of the block's return value for each key -value pair passed into the block. If the block's return value is truthy, that key -value pair is selected and stored in a new hash; if it is falsey then that key-value pair is not selected. The `select` method invocation will return a new hash containing those key-value pairs for which the block's return value is truthy. In this case those key value pairs are `:b=>2`, `:c=>3`. So new hash will be returned containing these two key -value pairs.

Hash#map

```
{a: 1, b: 2, c: 3}.map do |key, value|
 [key,  value * 2]
end
```

The above example returns the array object `[:a, 2], [:b, 4], [:c, 6]` but doesn't output anything to the console.

On line `1` we are invoking the `Hash#map` method on the hash object `{a: 1, b: 2, c: 3`. The `{ }` or `do..end` following the `map` method invocation is a block which is passed to the `map` method invocation as an argument.

The `map` method iterates over the hash object `{a: 1, b: 2, c: 3}` and executes the code within the block once on each iteration passing in the current key-value pair from the hash to the block as arguments where they are assigned to the block parameters `key` and `value` which are block local variables. For each keyvalue pair the key is assigned to the first parameter `key` and the value is assigned to the second parameter `value`.

Inside the block, we are creating a new array on each iteration using the Array. The first element is the object referenced by `key` and the second element is the object returned by by `value`.

`map` uses the return value of the block on each iteration to perform transformation of the array. On each iteration, the block's return value is stored in a new array and this new array is returned by the `map` method invocation.

Strings and their Methods

Thursday, 16 September 2021 4:39 pm

String interpolation

The `#{key}` and `#{value}` are string interpolation which concatenates the surrounding string with the string representation of the returned objects of the ruby expressions inside the `#{ }`.

Sorting

Thursday, 16 September 2021 4:42 pm

Array#sort

Sorting of elements of an array is done by comparing the elements in the array with each other, and ordering them based on the result of that comparison.

`Array#sort` uses the `<=>` operator to sort elements in arrays. Any object in a collection that we want to sort must implement a `<=>` comparison method. This method performs comparison between two objects of the same type and returns a `-1`,`0`, or `1`, depending on whether the first object is less than, equal to, or greater than the second object respectively; if the two objects cannot be compared then `nil` is returned.

The return value of the `<=>` method is used by `sort` to determine the order in which to place the items.

String#<=>

The order of strings is determined by the character's position in the ASCII table. It is this ASCII character order that determines the result when we compare one ASCII character to another using the String <=> method.

- Uppercase letters come before lowercase ones
- Digits and most punctuation come before letters
- The extended ASCII table containing accented and other characters. This comes after the main ASCII table.

Array#<=>

Each object in each array is compared (using the <=> operator).
Arrays are compared in an "element-wise" manner; the first element of ary is compared with the first one of other_ary using the <=> operator, then each of the second elements, etc... As soon as the result of any such comparison is non zero (i.e. the two corresponding elements are not equal), that result is returned for the whole array comparison.
If all the elements are equal, then the result is based on a comparison of the array lengths. Thus, two arrays are "equal" according to Array#<=> if, and only if, they have the same length and the value of each element is equal to the value of the corresponding element in the other array.

Array#sort_by

Variable References and Mutability

Saturday, 24 July 2021 8:12 pm

Objects and Variables

What is an object?

An object is a bit of data that has some sort of state or value and associated behavior.

How are objects assigned to variables?

objects can be assigned to variables.

```
`greeting = "Hello"`
=> "Hello"
```

What does the above code tell ruby?

In ruby, how would you describe the relationship between greeting and the string object "Hello"?

`greeting` references or points to the string object.

`greeting` is bound to the String object.

What is the act of doing this greeting = "Hello" known as?

String object represented by the literal 'Hello' is assigned to the variable `greeting`

How is the variable greeting able to reference the string object "Hello"?

It does so by storing the object id of the string object "Hello"

What is the #object_id method used for?

Every object in Ruby has a unique object id, and that object id can be retrieved simply by calling `#object_id` on the object in question. Even literals, such as numbers, booleans, `nil`, and Strings have object ids

Summary

- Ruby variables are merely references to objects in memory; that is, a variable is merely a name for some object.
- Multiple variables can reference the same object, so mutating an object using a given variable name will be reflected in every other variable that is bound to that object.
- We've also learned that assignment to a variable merely changes the binding; the object the variable originally referenced is not mutated. Instead, a different object is bound to the variable.
- We've also learned that certain object types, primarily numbers and Booleans but also some other types, are immutable in Ruby— unchanging; many other objects are mutable— changeable.
- If you attempt to change an immutable object, you won't succeed— at best, you can create a new object, and bind a variable to that object with assignment.
- Mutable objects, however, can be mutated without creating new objects.
- Finally, we've learned a bit about what pass by value and pass by reference mean. We've established a mental model that says that Ruby is pass by value for immutable objects, pass by reference otherwise.
- This model isn't perfect, but it can be used to help determine whether the object associated bound to an argument will be mutated.

Variable Reassignment

What happens if you assign a variable to another variable?

It causes the second variable to reference the same object as the first variable. Both variables will store the same object id. Since both variables are associated with the same object, using either variable to mutate the object is reflected in the other variable.

What does reassignment to a variable do to that variable? And what doesn't it do?

Reassignment to a variable doesn't mutate the object referenced by that variable;

Reassignment binds a different object to the variable.

What does reassignment do to the original object?

The original object is merely disconnected from the variable. It is no longer accessible if there are no other variables pointing to it.

Mutable and Immutable Objects

What is Mutability and Immutability?

In Ruby which object types are immutable?

`+=`, `*=` and other operators like these, what do these do?

How does a class establish itself as immutable?

By not providing any methods that mutate the object.

What is one characteristic of Mutable Objects?

They are objects of a class that permits changes to the object's state in some way.

What if mutation is permitted by setter methods, is it still a mutable object?

What is a setter method?

What is the signature of setter methods?

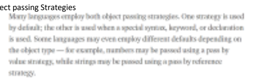
What is the array element setter method?

What concept does this code demonstrate?

What concept does this picture demonstrate?

When we assign to `a[1]` what are we actually doing?

Object passing Strategies



Many languages employ both object passing strategies. One strategy is used by default; the other is used when a special syntax, keyword, or declaration is used. Some languages may even employ different defaults depending on the object type — for example, numbers may be passed using a pass by value strategy, while strings may be passed using a pass by reference strategy.

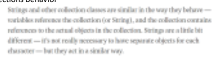
Object Passing - Introduction

When we pass an object as an argument to a method, the method can mutate the object or leave it unchanged. What does this ability to mutate an object passed as an argument depend on?

What does the pass by value object passing strategy mean?

What does the pass by reference object passing strategy mean?

Collections Behavior



Strings and other collection classes are similar in the way they behave — variables reference the collection (or String), and the collection contains references to the actual objects in the collection. Strings are a little bit different — it's not really necessary to have separate objects for each character — but they act in a similar way.

Initial Mental Model for Object Passing Strategy in Ruby

Immutable Objects: What Object passing Strategy does ruby employ for immutable Objects?

Mutable Object: What Object passing Strategy does ruby employ for immutable Objects?

Ruby Object's Mutating and Non Mutating Methods

Monday, 26 July 2021 8:56 pm

Mutating and Non Mutating Methods

```
What are non-mutating methods?
What are mutating methods?
What is important when deciding whether a method is mutating or non-mutating?
What are the different types of mutating method?

Are there any methods that are mutating for immutable Objects?
```

Assignment

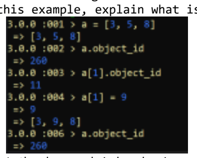
```
Is assignment mutating? =

What's happening in the below code?
What's happening in the below code example?
What does assignment do?
What does assignment do to the original object  referenced by the variable?
What do *=, +=, -=, /=, %= do?
What does the following code do?
```

Re Assignment

Mutating Methods

```
What is a mutating method?
    A method is said to be mutating with respect to an argument or it calling object if it
    mutates its value in the process. If the variable references the same object before and
    after the method is called and changes the state of the object, then the method is said to
    be mutating with respect to that object.
Is indexed assignment mutating? Which objects use indexed assignment?
    Index assignment i.e. #[]= is mutating with respect to its calling object.
    It is used by strings, arrays and hashes. It doesn’t change the binding of the object.
Why is indexed assignment mutating while simple assignment is not?
    Indexed assignment is a method that a class must supply if it needs indexed assignment.
In this example, explain what is happening to ary?
```



```
3.0.0 :001 > a = [3, 5, 8]
 => [3, 5, 8]
3.0.0 :002 > a.object_id
 => 260
3.0.0 :003 > a[1].object_id
 => 11
3.0.0 :004 > a[1] = 9
 => 9
 => [3, 9, 8]
3.0.0 :006 > a.object_id
 => 260
```

In the above code indexed assignment was used to mutate the array `a`. As we can see that
The object id of `a` remains the same but the state of the object it references has changed.
This means that the []= method mutated the array. However, the object referenced by a[1] has changed. Although the object referenced by the second array element was changed i.e. reassigned to a different object, but `a` still references the same array object but with a different state.

Concatenation is Mutating

What ways are there for String and Array Concatenation? Which are mutating?
#<< , concat and += are used for string and array concatenation.
#<< and #concat are mutating for both strings and arrays. While += is non mutating for both. += only reassigns the target array to a new array.

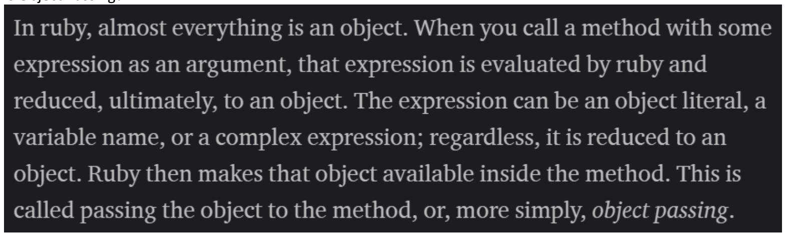
Setters are Mutating

What is the difference between indexed assignment and setter methods?
Indexed assignment and setter methods both use the something= syntax.
While indexed assignment replaces elements of a collection, setter alter the state of the object.

Object Passing in Ruby - Pass by Value or Pass by Reference

Saturday, 31 July 2021 6:22 pm

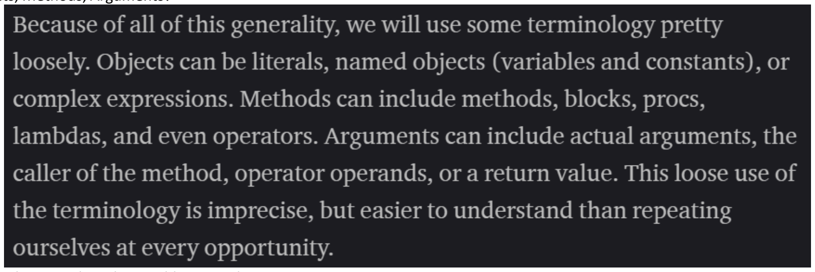
What is Object Passing?



In ruby, almost everything is an object. When you call a method with some expression as an argument, that expression is evaluated by ruby and reduced, ultimately, to an object. The expression can be an object literal, a variable name, or a complex expression; regardless, it is reduced to an object. Ruby then makes that object available inside the method. This is called passing the object to the method, or, more simply, *object passing*.

In ruby, almost everything is an object. When you call a method with some expression as an argument, that expression is evaluated by ruby and reduced, ultimately, to an object. The expression can be an object literal, a variable name, or a complex expression; regardless, it is reduced to an object. Ruby then makes that object available inside the method. This is called passing the object to the method, or, more simply, object passing.

Objects, Methods, Arguments?

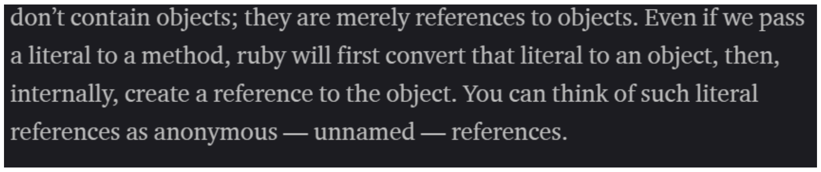


Because of all of this generality, we will use some terminology pretty loosely. Objects can be literals, named objects (variables and constants), or complex expressions. Methods can include methods, blocks, procs, lambdas, and even operators. Arguments can include actual arguments, the caller of the method, operator operands, or a return value. This loose use of the terminology is imprecise, but easier to understand than repeating ourselves at every opportunity.

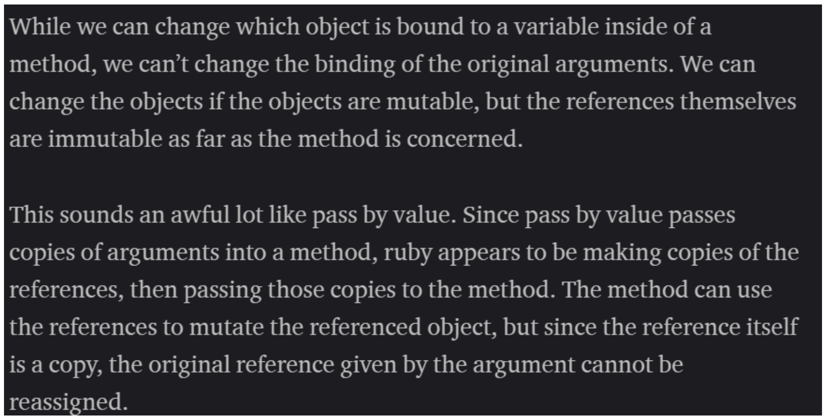
Because of all of this generality, we will use some terminology pretty loosely. Objects can be literals, named objects (variables and constants), or complex expressions. Methods can include methods, blocks, procs, lambdas, and even operators. Arguments can include actual arguments, the caller of the method, operator operands, or a return value. This loose use of the terminology is imprecise, but easier to understand than repeating ourselves at every opportunity.

Objects == literals, variables, complex expressions
Methods == methods, blocks, procs, lamdas and operators
Arguments == arguments, the caller of the method, operator operands or a return value.

What is the evaluation strategy used by Ruby?
Ruby uses strict evaluation strategy. According to this strategy, every expression is evaluated and converted to an object before it is passed along to a method.
What are the two most common strict evaluation strategies?
Pass by value and pass by reference. Collectively known as object passing strategies.
What is pass by value?
A copy of the object is created and it is the copy that gets passed around. Since it is only a copy, it is impossible to change the original object.
Passing around immutable objects in ruby acts a lot like pass by value.
Ruby appears to be pass by value with respect to immutable objects
What is pass by reference?
A reference to an object is passed around. This creates an alias between the argument and the original object. Both the argument and the object refer to the same location in memory

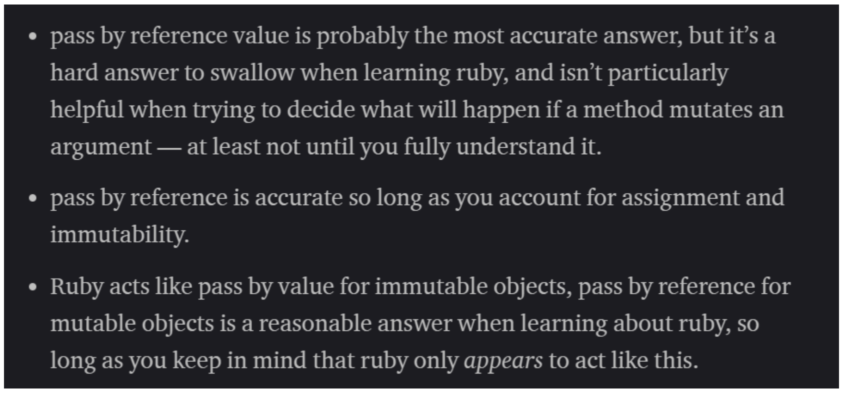


don't contain objects; they are merely references to objects. Even if we pass a literal to a method, ruby will first convert that literal to an object, then, internally, create a reference to the object. You can think of such literal references as anonymous — unnamed — references.



While we can change which object is bound to a variable inside of a method, we can't change the binding of the original arguments. We can change the objects if the objects are mutable, but the references themselves are immutable as far as the method is concerned.

This sounds an awful lot like pass by value. Since pass by value passes copies of arguments into a method, ruby appears to be making copies of the references, then passing those copies to the method. The method can use the references to mutate the referenced object, but since the reference itself is a copy, the original reference given by the argument cannot be reassigned.



- pass by reference value is probably the most accurate answer, but it's a hard answer to swallow when learning ruby, and isn't particularly helpful when trying to decide what will happen if a method mutates an argument — at least not until you fully understand it.
- pass by reference is accurate so long as you account for assignment and immutability.
- Ruby acts like pass by value for immutable objects, pass by reference for mutable objects is a reasonable answer when learning about ruby, so long as you keep in mind that ruby only *appears* to act like this.

Order of Precedence

Thursday, 14 October 2021 8:17 pm

Nested Datastructures

Friday, 15 October 2021 2:18 pm

Working with Blocks

Friday, 15 October 2021 2:18 pm

- What is the type of action being performed (method call, block, conditional, etc..)?
- What is the object that action is being performed on?
- What is the side-effect of that action (e.g. output or destructive action)?
- What is the return value of that action?
- Is the return value used by whatever instigated the action?

Written Assessment - Preparation Plan

Friday, 24 September 2021 5:31 pm

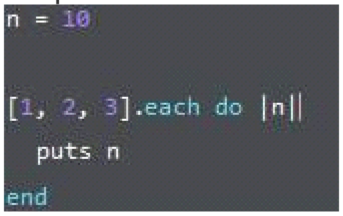
1. Make notes for all the topics from the study guide -> Go through Ruby book, lessons and everything
2. Revise these notes until you know them my heart and understand those concepts in depth
3. For each topic come up with examples to explain those concepts
4. Find questions related to those concepts from ruby basics, lesson 3, lesson 4 and lesson 5
5. Go through the quiz questions again. Make questions for assessment from them
6. Dry run. Time yourself. If you can get 8 done in an hour or less, you should be in good shape.
7. Make questions out of the

Written Assessment Study Guide - Riaz

Thursday, 5 August 2021 11:42 am

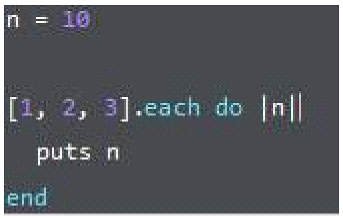
- Part 1: Study Guide for Test
    - Specific Topics of Interest

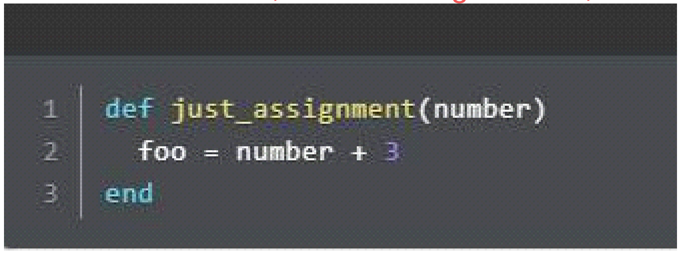
Be able to explain clearly the following topics:

- local variable scope
    - Determines where in a program a variable is available for use
    - Defined by where the variable is initialized or created
    - In Ruby, variable scope is defined by a *block*
    - A block is a piece of code following a method invocation, usually delimited by either curly braces {} or do/end
- especially how local variables interact with method invocations with blocks and method definitions
    - Visual mental model:
        - **Inner scope can access variables initialized in an outer scope, but not vice versa.**
        - **Local variables initialized outside of a block delimited by do...end are like people who live in the suburbs. They go into the city (the block delimited by do...end) to work. Therefore, those local variables are accessible outside of the block.**
        - **Local variables initialized inside the block however, are like people who live in the city (the block delimited by do..end). People in the city do not work in the suburbs. Therefore, you will raise an error if you try to use a local variable originally initialized within the block delimited by do..end outside of the block.**
- especially how local variables interact with method definitions
    - Fortress
    - A method definition--you can't break into it EXCEPT when you pass it as an argument to it.
- Variable shadowing
    - Def: two local variables in the inner scope with the same name
    - **variable shadowing** occurs when a variable declared within the inner scope of a block delimited by do..end has the same name as a variable declared in an outer scope.
    - **variable shadowing(Wikipedia definition)** occurs when a variable declared within a certain **scope** (decision block, method, or inner class) has the same name as a variable declared in an outer scope.
    - But what if we had a variable named n in the outer scope? We know that the inner scope has access to the outer scope, so we'd essentially have two local variables in the inner scope with the same name. When that happens, it's called *variable shadowing*, and it prevents access to the outer scope local variable
    - The puts n will use the block parameter n and disregard the outer scoped local variable. Variable shadowing also prevents us from making changes to the outer scoped n:
    - 

```
n = 10

[1, 2, 3].each do |n|
  puts n
end
```

- 
  ```
  n = 10

  [1, 2, 3].each do |n||
    puts n
  end
  ```
- Srdjan's Variable Shadow Definition:
  - This is the concept called **variable shadowing** and it happens when parameter name of the block is the same as the name of the local variable which was initialized outside of the block.
  - Prioritize own name
  - naming the same as outerscope, making it so block can't access outer scope variable
  -
- how passing an object into a method definition can or cannot permanently change the object
  - Refer to Christian Larwood's study guide
- working with collections (Array, Hash, String),
- and popular collection methods (each, map, select, etc). Review the two lessons on these topics thoroughly.
  - - #select returns a new array based on the block's return value. If the block's return value evaluates to true, then the element is selected
  - - #map returns a new array based on the blocks return value. Each element is transformed based on the return value (block parameter does not have to be used)
  - - #each returns the original array the method was called on. The blocks return value does not influence what the method each returns
- **variables as pointers**
  - See articles
  - Variables point a specific place in memory
- **puts vs return**
  - See articles
  - puts is outputs a string to the screen and returns nil
  - Return is the evaluated result of the executed code; the last expression in a method is returned; has chaining abilities;
  - 
    ```
    def just_assignment(number)
      foo = number + 3
    end
    ```
    - The value of just_assignment(2) is going to be 5 because the assignment **expression evaluates to** 5, therefore that's what's returned.

- false vs nil and the idea of "truthiness"
  - In Ruby: true is true
    - false is falsy
    - Nil is falsy
    - All other objects are truthy
- So what are mutating methods in Ruby?
  - Shortest answer would be, that those are methods which are changing the value of a calling object.
- method definition and method invocation
      - Method Definition and Method Invocation
  - When discussing methods, particularly in terms of how blocks and methods interact with local variables, we want you to explain this in terms of method definition and method invocation. You can review **this assignment** for an outline of the mental model to use.
    - Methods are defined using the keywords def and end
      - Has its own variable scope
      - Returns the last evaluated expression
    - Methods are called/invoked by writing out the method name;
      - Some methods require block parameters, others don't
  - implicit return value of method invocations and blocks
    - Implicit return value of method invocations
      - Is simply the last evaluated expression of the invoked method
    - Implicit return value of method blocks
      - Is simply the last evaluated expression of the block
        - The block is defined when it is called on the method

- how the Array#sort method works
  - Utilizes the ⇔ operator?
    - Value to the left -1, 0, 1
    - uses the return value where in the Array to place the element;
    - also know that if b < = > a, it would be descending order
  - Review docs
  - Returns a new array created by sorting self.
  - Comparisons for the sort will be done using the <=> operator or using an optional code block.

- The block must implement a comparison between a and b and return an integer less than 0 when b follows a, 0 when a and b are equivalent, or an integer greater than 0 when a follows b.
- The sort method uses the spaceship operator and compares one by one, adjacent objects.
  - It determines if the first object is less than, equal to, or greater than the second number.
  - If the first number is less than the second, returns -1
  - Equal → returns 0
  - First number greater than second, returns 1
  - Based on return values, flips numbers/strings until they are from least to greatest or earliest in the alphabet to latest
  - http://cs.armstrong.edu/liang/animation/web/BubbleSortNew.html

Truthiness

In the assessment we want you to be very clear about the distinction between *truthy* and the boolean true (and similarly the distinction between *falsey* and the boolean false).

In Ruby, every value apart from false and nil, *evaluates to true* in a boolean context. We can therefore say that in Ruby, every value apart from false and nil is *truthy*; we can also say that false and nil are *falsey*. This is **not** the same as saying every value apart from false and nil **is** true, or **is equal to** true. These may seem like subtle distinctions but they are important ones.

To sum up:

- Use "evaluates to true" or "is truthy" when discussing an expression that evaluates to true in a boolean context
- Do not use "is true" or "is equal to true" unless specifically discussing the boolean true

Variable References and Object Mutability Articles

We wrote a series of blog posts that thoroughly cover variable references and object mutability:

- **Variable References and Mutability of Ruby Objects**
- **Mutating and Non-Mutating Methods in Ruby**
- **Object Passing in Ruby - Pass by Reference or Pass by Value**

with the method definition the parameter is being set up to be initialized,

string = 'Hello'

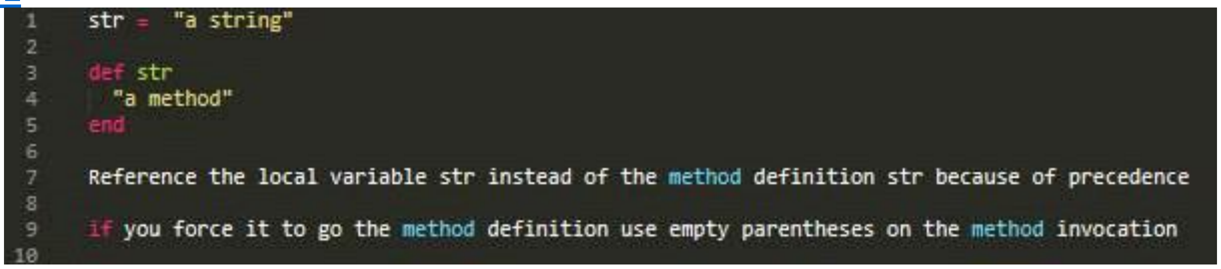
- method(string)

- The local variablestring is passed to method as an argument. At method invocation, the method local variable param is initialized to the object referenced by string. Both string and param now point to the same object, 'Hello'.

Spend some time reviewing these articles to ensure that you have a good understanding of these topics.

- Assessment Prep Videos

We did a Beginning Ruby series that will serve as great review for this test. Specifically, the sessions that are relevant to this assessment are:

- **Part 1**
  - 

```
str =  "a string"

def str
  "a method"
end

Reference the local variable str instead of the method definition str because of precedence

if you force it to go the method definition use empty parentheses on the method invocation

```

- **Part 2**
- **Part 3**

Pay careful attention to the way the instructor explains concepts, and use that vocabulary to describe code on the test.

Several Launch School students have blogged about their assessment experiences:

Srđan prepared a four-part **blog series** in which he reviews the core concepts covered in course RB101 that should be mastered prior to taking the assessment.

Written Assessment Practice Questions

Thursday, 5 August 2021 11:41 am

Suggested response format (based on feedback from other students & Srdjan's blog post):

- **What does the code output? What are the return values?**
- **Answer the why behind the output/return:**
  - Focus only on the lines of code that deliver the output and return values.
- **Summarize what the problem demonstrates and why:** This problem demonstrates the concept of local variable scope/etc…
  - This can be at the beginning or end of your answer - personal preference.

Using Markdown: use `backticks` (Markdown formatting) to highlight variable names, methods, and lines you are referring to: On `line 1` we initialize the local variable…

Always aim to answer: What does the following code output and return? Why? What concept does it demonstrate?

https://docs.google.com/document/d/16XteFXEm3lFbcavrXDZs45rNEc1iBxSYC8e4pLhT0Rw/edit#

From <https://app.slack.com/client/T0YKP5Z9T/D021391962X>

Additional Practice Problems:

1. Collection Methods from Lesson 4
2. Ruby Basics: Variable Scope
3. Ruby Basics: Return

Local Variable Scope

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 'Hello'
b = a
a = 'Goodbye'
```

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 4

loop do
  a = 5
  b = 3
  break
end

puts a
puts b
```

Example 3

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 4
b = 2

loop do
  c = 3
  a = c
  break
end

puts a
puts b
```

Example 4

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def example(str)
  i = 3
  loop do
    puts str
    i -= 1
    break if i == 0
  end
end

example('hello')
```

Example 5

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def greetings(str)
  puts str
  puts "Goodbye"
end

word = "Hello"

greetings(word)
```

Problem link

Example 6

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4]

counter = 0
sum = 0

loop do
  sum += arr[counter]
  counter += 1
  break if counter == arr.size
end

puts "Your total is #{sum}"
```

Example 7

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 'Bob'

5.times do |x|
  a = 'Bill'
end

p a
```

Example 8

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
animal = "dog"

loop do |_|
  animal = "cat"
  var = "ball"
  break
end

puts animal
puts var
```

Variable Shadowing

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 4
b = 2

2.times do |a|
  a = 5
  puts a
end

puts a
puts b
```

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
n = 10

1.times do |n|
  n = 11
end

puts n
```

Example 3

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
animal = "dog"

loop do |animal|
  animal = "cat"
  break
end

puts animal
```

Object Passing/Variables As Pointers

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = "hi there"
b = a
a = "not here"
```

What are a and b?

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = "hi there"
b = a
a << ", Bob"
```

What are a and b?

Example 3

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = [1, 2, 3, 3]
b = a
c = a.uniq
```

What are a, b, and c? What if the last line was `c = a.uniq!`?

Example 4

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def test(b)
  b.map {|letter| "I like the letter: #{letter}"}
end

a = ['a', 'b', 'c']
test(a)
```

What is `a`? What if we called `map!` instead of `map`?

Example 5

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = 5.2
b = 7.3

a = b

b += 1.1
```

What is `a` and `b`? Why?

Example 6

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def test(str)
  str  += '!'
  str.downcase!
end

test_str = 'Written Assessment'"
test(test_str)

puts test_str
```

Link to explanation of examples below

Example 7

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def plus(x, y)
  x = x + y
```

```
end

a = 3
b = plus(a, 2)

puts a
puts b
```

Example 8

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def increment(x)
  x << 'b'
end

y = 'a'
increment(y)

puts y
```

Example 9

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def change_name(name)
  name = 'bob'      # does this reassignment change the object outside the method?
end

name = 'jim'
change_name(name)
puts name
```

Example 10

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def cap(str)
  str.capitalize!   # does this affect the object outside the method?
end

name = "jim"
cap(name)
puts name
```

Example 11

What is `arr`? Why? What concept does it demonstrate?

```
a = [1, 3]
b = [2]
arr = [a, b]
arr

a[1] = 5
arr
```

Example 12

Link to example

```
arr1 = ["a", "b", "c"]
arr2 = arr1.dup
arr2.map! do |char|
  char.upcase
end

puts arr1
puts arr2
```

Object Mutability/Mutating Methods

Here's the article with some of the below examples

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def fix(value)
  value.upcase!
  value.concat('!')
  value
end

s = 'hello'
t = fix(s)
```

What values do `s` and `t` have? Why?

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def fix(value)
  value = value.upcase
  value.concat('!')
end

s = 'hello'
t = fix(s)
```

What values do `s` and `t` have? Why?

Example 3

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def fix(value)
  value << 'xyz'
  value = value.upcase
  value.concat('!')
end
```

```
s = 'hello'
t = fix(s)
```

What values do `s` and `t` have? Why?

Example 4

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def fix(value)
  value = value.upcase!
  value.concat('!')
end

s = 'hello'
t = fix(s)
```

What values do `s` and `t` have? Why?

Example 5

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def fix(value)
value[1] = 'x'
value
end

s = 'abc'
t = fix(s)
```

What values do `s` and `t` have? Why?

Example 6

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def a_method(string)
  string << ' world'
end

a = 'hello'
a_method(a)

p a
```

Example 7

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
num = 3

num = 2 * num
```

Example 8

What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
a = %w(a b c)
a[1] = '-'
p a
```

Example 9

Link to page with #9 & #10

```ruby
def add_name(arr, name)
  arr = arr + [name]
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names
```

Example 10

```ruby
def add_name(arr, name)
  arr = arr << name
end

names = ['bob', 'kim']
add_name(names, 'jim')
puts names
```

Each, Map, Select

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
array = [1, 2, 3, 4, 5]

array.select do |num|
  puts num if num.odd?
end
```

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

arr.select { |n| n.odd? }
```

Example 3

What does the following code return? What does it output? Why? What concept does it demonstrate?

```ruby
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.select do |n|
  n + 1
end
p new_array
```

Example 4

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.select do |n|
  n + 1
  puts n
end
p new_array
```

Example 5

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
words = %w(jump trip laugh run talk)

new_array = words.map do |word|
  word.start_with?("t")
end

p new_array
```

Example 6

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

arr.each { |n| puts n }
```

Example 7

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

incremented = arr.map do |n|
            n + 1
            end
p incremented
```

Example 8

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.map do |n|
  n > 1
end
p new_array
```

Example 9

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

new_array = arr.map do |n|
  n > 1
  puts n
end
p new_array
```

Example 10

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = "hello"

[1, 2, 3].map { |num| a }
```

Example 11

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
[1, 2, 3].each do |num|
  puts num
end
```

Other Collection Methods

Link to all examples below

Example 1

```
[1, 2, 3].any? do |num|
  num > 2
end
```

Example 2

```
{ a: "ant", b: "bear", c: "cat" }.any? do |key, value|
  value.size > 4
end
```

Example 3

```
[1, 2, 3].all? do |num|
  num > 2
end
```

Example 4

```
{ a: "ant", b: "bear", c: "cat" }.all? do |key, value|
  value.length >= 3
```

```
end
```

Example 5

```
[1, 2, 3].each_with_index do |num, index|
  puts "The index of #{num} is #{index}."
end
```

Example 6

```
{ a: "ant", b: "bear", c: "cat" }.each_with_object([]) do |pair, array|
  array << pair.last
end
```

Example 7

```
{ a: "ant", b: "bear", c: "cat" }.each_with_object({}) do |(key, value), hash|
  hash[value] = key
end
```

Example 8

```
odd, even = [1, 2, 3].partition do |num|
  num.odd?
end

p odd
p even
```

Truthiness

Example 1

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
a = "Hello"

if a
  puts "Hello is truthy"
else
  puts "Hello is falsey"
end
```

Example 2

What does the following code return? What does it output? Why? What concept does it demonstrate?

```
def test
  puts "written assessment"
end

var = test
```

```ruby
if var
  puts "written assessment"
else
  puts "interview"
end
```

Written Test Study Guide- Launch School

Thursday, 16 September 2021 4:35 pm

Part 1: Study Guide for Test

Assessment RB109 will test your knowledge of courses LS100 and RB101, which has a huge surface area in that it covers the Ruby programming language broadly. It will not cover Object Oriented Programming.

Specific Topics of Interest

You should be able to clearly explain the following topics:

- local variable scope, especially how local variables interact with method invocations with blocks and method definitions
- mutating vs non-mutating methods, pass-by-reference vs pass-by-value
- working with collections (Array, Hash, String), and popular collection methods (each, map, select, etc). Review the two lessons on these topics thoroughly.
- variables as pointers
- puts vs return
- false vs nil and the idea of "truthiness"
- method definition and method invocation
- implicit return value of method invocations and blocks
- how the `Array#sort` method works

How to Answer the Assessment Questions

The questions in this assessment will typically be testing your knowledge and understanding on more than one level.

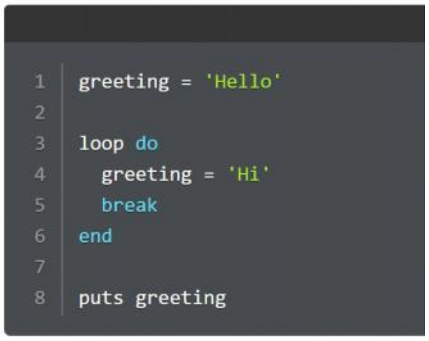
- On one level the question is testing your ability to parse code and to describe it with precision, or testing your knowledge of some specific syntactical aspect or language-specific feature of the Ruby programming language.
- On another level, the question is checking your understanding of some deeper, underlying principle; this might be some more fundamental aspect of the Ruby language, or a non-language-specific concept that applies to programming more generally.

When answering the questions, you should:

- Explain your reasoning with reference to specific lines in the program. You can use line numbers to refer to specific lines of code where necessary.
- Answer with extreme precision. For example, say "method definition" or "method invocation" as opposed to just "method" (see the section on 'Precision of Language' below for more on this).
- Highlight any specific syntactical conventions or technical observations where relevant.
- Identify the key fundamental concept or concepts being demonstrated in the question.

Example

Examine the code example below. The last line outputs the String `'Hi'` rather than the String `'Hello'`. Explain what is happening here and identify the underlying principle that this demonstrates.



```ruby
greeting = 'Hello'

loop do
  greeting = 'Hi'
  break
end

puts greeting
```

Compare the following possible answers to this question:
A) `greeting` is set to `'Hello'` on line 1. `greeting` is set to `'Hi'` on line 4. Line 8 outputs `greeting`, which is `'Hi'`.
B) The local variable `greeting` is assigned to the String `'Hello'` on line 1. Within the loop, `greeting` is then reassigned to the String `Hi` on line 4. The `puts` method is called on line 8 with the variable `greeting` passed to it as an argument; since `greeting` is now assigned to `'Hi'`, this is what is output.
C) The local variable `greeting` is assigned to the String `'Hello'` on line 1.
The `do..end` alongside the `loop` method invocation on lines 3 to 6 defines a block, within which `greeting` is reassigned to the String `Hi` on line 4. The `puts` method is called on line 8 with the variable `greeting` passed to it as an argument; since `greeting` is now assigned to `'Hi'`, this is what is output.
D) The local variable `greeting` is assigned to the String `'Hello'` on line 1.
The `do..end` alongside the `loop` method invocation on lines 3 to 6 defines a block, within which `greeting` is reassigned to the String `Hi` on line 4. The `puts` method is called on line 8 with the variable `greeting` passed to it as an argument; since `greeting` is now assigned to `'Hi'`, this is what is output. This example demonstrates local variable scoping rules in Ruby; specifically the fact that a local variable initialized outside of a block is accessible inside the block.
While none of these answers is technically incorrect, they all answer the question with varying degrees of detail and precision.

- Answer 'A' describes what is happening in the code example, but does so in a fairly basic way with imprecise language. This wouldn't be a sufficient response to receive full points for any of the questions in the assessment.
- Answer 'B' again describes what is happening, but with much greater precision of language. This would score higher than Answer 'A', but generally wouldn't be sufficient to receive full points for the majority of questions; most questions in the assessment are looking for something more, such as a specific piece of syntactical knowledge and perhaps identification of some fundamental concept.
- Answer 'C', as well as precisely describing the example, identifies an important Ruby syntactical convention that is relevant to the example: the fact that the method invocation combined with `do..end` defines a block in Ruby. For some assessment questions this might be sufficient to receive full points, but many questions will expect you to demonstrate some deeper understanding of the fundamental concept that this illustrates.
- Answer 'D' goes a step further than 'C' by explaining why this is important and the underlying principle that it demonstrates; i.e. the fact that method invocations with blocks in Ruby have particular scoping rules which affect whether or not the local variable can be referenced or reassigned. Based on the way that this particular question is phrased, answer 'D' would be the only answer of the four to receive full points in an actual assessment.

Bullet Points

Many students attempt to use bullet points to answer the questions on the exam. This makes sense in some situations:

- You have a list of explicit reasons why some code does what it does.
- You have a list of pros and cons about.
- You want to provide a list of things.

In short, they work well for **lists**. (Notice that we used a bullet list to list this list of lists!)
However, they don't always work as complete answers for a question. You don't speak in bullet lists; don't write with bullet lists.
To illustrate, consider the following hypothetical explanation of the example code from the previous section:

- Line 1 declares a variable named `greeting` and initializes it as `'Hello'`.
- Line 3 begins a loop that keeps on repeating forever until the code breaks out of the loop.
- Line 4 reassigns `greeting` to `'Hi'`.
- Line 5 breaks out of the loop.
- Line 8 prints the value of `greeting`.
- Variables defined outside a block are accessible inside the block.

This answer is essentially a *laundry list* of statements about the code. Unfortunately, laundry

lists aren't very effective as answers on the assessment. They are difficult to follow, and often leave it to the reader to piece together the logic behind the list.
In the above list, for instance, there's no logical progression that actually explains what is happening. Instead, the student has simply listed a bunch of statements about each line of code, plus one unrelated item that talks about scope. However, a program is not a series of independent lines of code. Code depends on what happened before, and it influences what happens later. There's nothing in the laundry list that connects those individual bits of code together.
From the grader's point of view, this answer is incomplete:

- It doesn't mention what happens when the loop runs.
- it doesn't talk about the fact that `greeting` on line 4 is the same variable as the one shown on lines 1 and 8.
- It doesn't tie the statement about variables to the other statements.

In short, it leaves the grader with the burden of tying your bullet points together in a coherent whole.
These faults can be addressed, to a degree, in a bullet point answer. However, the laundry list approach often leads students to overlook these missing details. Paragraphs make it easier to think about the bigger picture since you're striving for clarity, not a list of everything you can think of.
Some students overcompensate by listing a bunch of facts that aren't really pertinent to the question. For instance, a typical answer might list several additional facts about this code:

- `loop` is a method.
- The words `do` and `end` and everything between them are a block.
- `'Hello'` and `'Hi'` are strings.

This is mostly clutter for the grader. You may also lose some points if the additional details say something that is incorrect.
In general, a clearly written paragraph is easier to understand and grade than a laundry list. While we won't penalize you for using bullet points, it's important to realize that bullet points have weaknesses that are difficult to see when you're writing.

Precision of Language

Most of the questions will require that you explain the code using words. It's important to be able to explain why something happens using precise vocabulary and be able to pinpoint the exact causal mechanism at work. In other words, be precise and don't be vague.
For example, let's take the following piece of code.
Copy Code

```
defa_method
  puts "hello world"end
```

If asked to describe the method, you might be tempted to say *"the results of the method is hello world"*. This isn't wrong, but for a programmer, it's extremely imprecise and doesn't help us understand the method. If you had written that as an answer, you'd score a 5/10 on the question (50% is not a passing score).
The more precise answer would be *"the method outputs the string hello world, and returns nil"*. In programming, we are always concerned with the output and the return value and mutations to objects. We need to speak in those terms, and not use vague words like "results".
When writing answers to the test questions, make sure to be as precise as possible, and use the proper vocabulary. Doing this will help you debug and understand more complex code later in your journey. If your definitions are not precise, you will not be able to lean on them to decompose complicated code. Also, you will likely not be able to pass this assessment.

Some Specifics

As well as requiring a general precision of language in your answers, for the purposes of the assessment there are a few areas where we would like you to refer to certain things in a very clear and fairly specific way; these are outlined below.

Assignments

Consider the following assignment statement:
Copy Code

```
greeting ='Hello'
```

Most of the Launch School material describes this statement as:

> The `greeting` variable is assigned to the string `'Hello'`.

However, there are places where we describe this code as:

> The string `'Hello'` is assigned to the `greeting` variable.

Both of these are acceptable in the assessment. Try to be consistent though -- if you aren't consistent, your meaning may be confused.

Truthiness

In the assessment we want you to be very clear about the distinction between *truthy* and the boolean `true` (and similarly the distinction between *falsey* and the boolean `false`).

In Ruby, every value apart from `false` and `nil`, *evaluates as true* in a boolean context. We can therefore say that in Ruby, every value apart from `false` and `nil` is *truthy*; we can also say that `false` and `nil` are *falsey*. This is **not** the same as saying every value apart from `false` and `nil` **is** `true`, or **is equal to** true. These may seem like subtle distinctions but they are important ones.

Example:

Copy Code

```
a ="Hello"ifa
  puts "Hello is truthy"elseputs "Hello is falsey"end
```

- `a` is `true` and so 'Hello is truthy' is output" would be incorrect
- `a` is equal to `true` and so 'Hello is truthy' is output" would be incorrect
- `a` evaluates as true in the conditional statement and so 'Hello is truthy' is output" would be correct
- `a` is truthy and so 'Hello is truthy' is output" would be correct

To sum up:

- Use "evaluates to true", "evaluates as true", or "is truthy" when discussing an expression that evaluates as true in a boolean context
- Do not use "is `true`" or "is equal to `true`" unless specifically discussing the boolean `true`
- Use "evaluates to false", "evaluates as false", or "is falsy" when discussing an expression that evaluates as false in a boolean context
- Do not use "is `false`" or "is equal to `false`" unless specifically discussing the boolean `false`

Method Definition and Method Invocation

When discussing methods, particularly in terms of how blocks and methods interact with local variables, we want you to explain this in terms of method definition and method invocation. You can review this assignment for an outline of the mental model to use.

Variable References and Object Mutability Articles

We wrote a series of blog posts that thoroughly cover variable references and object mutability:

- Variable References and Mutability of Ruby Objects
- Mutating and Non-Mutating Methods in Ruby
- Object Passing in Ruby - Pass by Reference or Pass by Value

Spend some time reviewing these articles to ensure that you have a good understanding of these topics.

Assessment Prep Videos

We did a Beginning Ruby series that will serve as great review for this test. Specifically, the sessions that are relevant to this assessment are:

- Part 1
- Part 2
- Part 3

Pay careful attention to the way the instructor explains concepts, and use that vocabulary to describe code on the test.

Additional Tips

While working through the assessment questions it is useful to run your code often to check it, so make sure to have either Ruby document/terminal or an online repl prepared beforehand.

Several Launch School students have blogged about their assessment experiences:

- Zach had a rough time making his way through the first two Launch School assessments. In I Failed Programming 101 Three Times his struggle and frustration trying to pass these assessments, and how he eventually embraced the Launch School way.
- Srđan prepared a four-part blog series in which he reviews the core concepts covered in course RB101 that should be mastered prior to taking the assessment.

- Raúl talks about his preparation and experiences as a non-native English speaker in this interesting article.
- Juliette wrote a blog article about her own strategy for Launch School's written assessment preparation.
- Callie's blog article has a wealth of useful information about preparing for both parts of the assessment: the exam and the interview.

All of these articles are well worth your time; don't pass them up!

From <https://launchschool.com/lessons/3ce27abc/assignments/cd8e4629>

Answering Questions

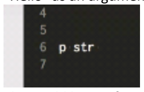
Thursday, 16 September 2021 4:50 pm

When answering the questions, you should:

- Line numbers
- Precision
- syntactical conventions or technical observations
- key fundamental concept demonstrated

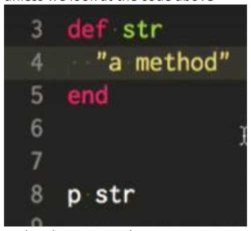
The more precise answer would be *"the method outputs the string hello world, and returns nil"*. In programming, we are always concerned with the output and the return value and mutations to objects. We need to speak in those terms, and not use vague words like "results".

From <https://launchschool.com/lessons/3ce27abc/assignments/cd8e4629>

- Ruby's Syntactical Sugar
    - Puts "Hello"
        - This is a method invocation. We are invoking the method Kernel#puts and passing in the string object "Hello" as an argument
        - 

```ruby
p str
```

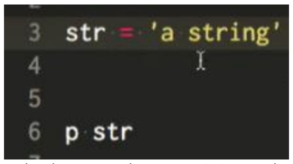
This could be a reference to a local variable str or it could be method invocation. We wont know unless we look at the code above



```ruby
def str
  "a method"
end


p str
```

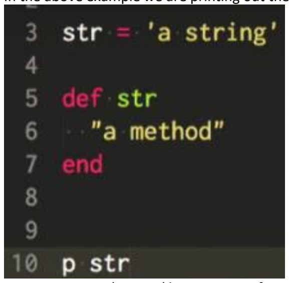
In the above example we are printing out the return value of the method invocation.



```ruby
str = 'a string'


p str
```

In the above example we are printing out the value of the str variable



```ruby
str = 'a string'

def str
  "a method"
end


p str
```

Its going to print 'a string' because str references the local variable str. If you want to invoke the method str instead you have to use parantheses with str.
In Ruby parantheses are optional with method invocation

Check methods >,<,=,<=,>= for all classes.

Srdjan's Tips

Non-Technical Tips

- **Make sure to use markdown**
- **Make sure to divide your answers into paragraphs**
- **Pay close attention to what each question is asking and keep your answers concise**.