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

ARRAYS

The map method iterates over an array applying a block to each element of the array and returns a new array with those results. The irb session below shows how to use map to get the square of all numbers in an array. The collectmethod is an alias to map - they do the same thing.

You'll notice that after performing these methods there is no change to the initial array. These methods are not destructive (i.e., they don't mutate the caller). How do you know which methods mutate the caller and which ones don't? You have to use the methods and pay attention to the output in irb; that is, you have to memorize or know through using it.

The delete\_at method can be helpful if you'd like to eliminate the value at a certain index from your array. You'll want to be careful with this one, because it modifies your array destructively. Once you call this method, you are changing your array permanently.

As a side note, sometimes you will know the value that you want to delete, but not the index. In these situations you will want to use the delete method. Thedelete method permanently deletes all instances of the provided value from the array.

Another way to remember these methods: use each for iteration and map for transformation.

Files:

Always close open files. Open files continue to occupy space in memory.

* r: read-only (starts at beginning of file)
* w: write-only (if the file exists, **overwrites** everything in the file)
* w+: read and write (if the file exists, **overwrites** everything in the file)
* a+: read-write (if file exists, starts at end of file. Otherwise creates a new file). Suitable for updating files.

Ruby will automatically close a file if open method is called with a block.

Ruby Style

When you define or initialize a method, variable or file, always use snake\_case.

When you want to represent a value that will not change, you define a constant variable and denote it in ALLCAPS

When working with do . . . end blocks, prefer {} when the entire code fits on one line

Name classes in CamelCase

* no spaces and capitalize every word

Pass by Value and Pass by Reference

https://launchschool.com/blog/object-passing-in-ruby

Pass by value:

With pass by value, a copy of an object is created, and it is that copy that gets passed around. Since it is merely a copy, it is impossible to change the original object; any attempt to change the copy just changes the copy and leaves the original object unchanged.

Ruby appears to be pass by value, at least with respect to immutable values.

Pass by reference:

By contrast, with pass by reference, a *reference* to an object is passed around. This establishes an alias between the argument and the original object: both the argument and object refer to the same location in memory. If you modify the argument’s state, you also modify the original object.

Ruby appears to use pass by reference when passing mutable objects

Note, however,

You may be ready to ask “But, we can’t modify immutable objects! Isn’t that what pass by reference is all about?” The key here is that pass by reference isn’t limited to mutating methods. A non-mutating method can use pass by reference as well, so pass by reference can be used with immutable objects. There may be a reference passed, but the reference isn’t a guarantee that the object can be modified.

Pass by Reference Value

Given all of this, it’s not uncommon to just say that ruby is *pass by reference value*, *pass by reference of the value*, or *pass by value of the reference*. It’s all a little muddy, but the 3 terms mean essentially the same thing: ruby passes around copies of the references. In short, ruby is neither pass by value nor pass by reference, but instead employs a third strategy that blends the two strategies.

So, is that our final answer to the question of whether ruby is pass by reference or pass by value? It’s neither? Yes. Well, maybe not entirely; there are actually three answers to the question of what object passing strategy ruby uses:

* **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 modifies 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.

Re-read this for more discussion: https://launchschool.com/blog/object-passing-in-ruby