# Array/Hash Defaults

## Arrays of Arrays

Here's a very common problem that everyone will run into. Let's say we want to make an array of arrays:

[4] pry(main)> arr\_of\_arrs = Array.new(3, [])

=> [[], [], []]

[5] pry(main)> arr\_of\_arrs[0] << "a"

=> ["a"]

[6] pry(main)> arr\_of\_arrs

=> [["a"], ["a"], ["a"]]

Wait, what happened? We added "a" to the first array, but all of them were modified?

The reason is that only **two** arrays are created in the example: (1) arr\_of\_arrs, (2) the single empty array passed into the Array constructor ([]). arr\_of\_arrs stores three **references** to the same empty array.

Thus, when you access the array at position 0 and mutate it, you're mutating the same array referenced by position at one and two.

The way to solve this problem is like so:

[7] pry(main)> arr\_of\_arrs = Array.new(3) { Array.new }

=> [[], [], []]

[8] pry(main)> arr\_of\_arrs[0] << "a"

=> ["a"]

[9] pry(main)> arr\_of\_arrs

=> [["a"], [], []]

Here, instead of passing a reference to a single empty array (which would be stored at three locations), we've passed a block. The block will be run to produce a value to store for each position in the array. The block constructs a new array each time it is run.

I wrote Array.new in the block to make it clear that a new array is constructed each time the block is executed, but you could equivalently write [] in the block.

## Mutable vs Immutable

Okay, we've seen that arrays store references to objects. We've seen a naive problem where we stored several references to the same object. Since all references refer to the same underlying object, a mutation through one reference (arr\_of\_arrs[0] << "a") is also visible through another reference (arr\_of\_arrs[1] == ["a"]).

You may have previously written code like this:

arr2 = Array.new(3, 1)

arr2[0] += 1

arr2[0] == 2 # true

arr2[1] == 1 # true

arr2[2] == 1 # true

Does this contradict what we've just discussed about references and mutations? Why isn't the change visible at position 1?

Let's unpack arr2[0] += 1. This is Ruby shorthand for:

arr2[0] = arr2[0] + 1

Let's further break this into steps:

1. First, fetch the number at position 0 (which is 1).
2. Next, add one to this number. **This creates a new number object**. The + operation **does not** mutate the original object.
3. Finally, assign a reference to the new object (2) to position 0 of arr.

The trick is that we **never mutate** any number. We produce a new one and reset arr2[0] to refer to the new object. That's why none of the other indices are affected.

The Integer and Float classes are called **immutable**. None of their methods modify data inside the Integer/Float; they produce new values instead. nil is another example.

## Hash default values

Providing a default value for a Hash has the same issues as with an Array:

[1] pry(main)> cats = Hash.new([])

=> {}

[2] pry(main)> cats["Devon"]

=> []

[3] pry(main)> cats

=> {}

Providing an argument to Hash.new merely changes what is returned when we look up a key that isn't present in the hash. Note how this doesn't assign a value to "Devon" through mere access of the key. To do that, we can do something like:

[4] pry(main)> cats["Devon"] += ["Earl"]

=> ["Earl"]

[5] pry(main)> cats

=> {"Devon"=>["Earl"]}

[6] pry(main)> cats["Devon"] += ["Breakfast"]

=> ["Earl", "Breakfast"]

[7] pry(main)> cats

=> {"Devon"=>["Earl", "Breakfast"]}

Better. cats["Devon"] += ["Earl"] means cats["Devon"] = cats["Devon"] + ["Earl"]. This constructs a new array and stores it for key "Devon"

But what about this?

[7] pry(main)> cats = Hash.new([])

=> {}

[8] pry(main)> cats["John"] << "Kiki"

=> ["Kiki"]

[9] pry(main)> cats

=> {}

[10] pry(main)> cats["Raul"]

=> ["Kiki"]

Let's think through what's happening here. On line 8, we try to get a value for cats["John"]. "John" is not a key in the hash, so the default (an empty array) is returned. We then mutate the default value by adding "Kiki" to it.

We never set a value for "John" though, so this is not stored in the Hash (see the result of line 9).

Later, when we try to access some other non-present key ("Raul"), the default value is returned again. But since we mutated the value by shovelling "Kiki" in, this is no longer empty. This is bad, because we never meant for "Raul" to own "Kiki".

We can start to fix the problem as before:

[11] pry(main)> cats2 = Hash.new() { [] }

=> {}

[12] pry(main)> cats2["Devon"] << "Breakfast"

=> ["Breakfast"]

[13] pry(main)> cats2["George"]

=> []

Hash will use the block to produce a new default value each time. Modifying the value won't have an affect on looking up other non-existent keys, since we create a new value each time, instead of reusing a single default object.

But we have the other problem again: we're still not setting a value.

[16] pry(main)> cats2

=> {}

Let's fix this:

[17] pry(main)> cats3 = Hash.new { |h, k| h[k] = [] }

=> {}

[18] pry(main)> cats3["Devon"]

=> []

[19] pry(main)> cats3

=> {"Devon"=>[]}

[20] pry(main)> cats3["John"] << "Kiki"

=> ["Kiki"]

[21] pry(main)> cats3

=> {"Devon"=>[], "John"=>["Kiki"]}

Here we've modified the block to take two arguments: when Hash needs a default value, it will pass itself (h) and the key (k). The block will not only create an empty array, but also assign it to the hash.

You can see one somewhat funny side-effect when we look up "Devon"; even when we just want to lookup a value, if it is not present we'll incur the side-effect of mutating the hash (the key "Devon" got added).