**Inject**

Ruby's inject method is perhaps the most flexible when it comes to solving different problems. To master its versatility, let's go through the mechanics of how inject works. To avoid confusion, we'll refer to this method as inject, but it is also known as reduce in other programming languages. Ruby has both an inject and a reduce method. Both are functionally identical.

**Inject with only a block**

The most straightforward way to use inject is to call it on an array and pass in a block. Like many enumerables, inject will iterate through the array, passing in the current element to the block. However, what makes inject very versatile is that the block should accept two params, the accumulator and the current element. inject will maintain an accumulator that will change over time, depending on our block. The block should return what the new accumulator should be after a single iteration. This is hard to visualize, so let's take a look at an example:

[11, 7, 2, 4].inject { |acc, el| acc + el }

Notice that the block params are always in the order of accumulator (acc) followed by element (el). Let's now explore inject in action. Because of how we are calling inject, the initial accumulator will be the first element of the array by default. So our first iteration has the acc set to 11 and el set to 7.

# FIRST ITERATION:

# acc = 11

# el = 7

# new\_acc = 18

[11, 7, 2, 4].inject { |acc, el| acc + el }

Since the block results in 11 + 7, 18 will be assigned to acc in the next iteration. The el simply iterates to the next element of the array:

# SECOND ITERATION:

# acc = 18

# el = 2

# new\_acc = 20

[11, 7, 2, 4].inject { |acc, el| acc + el }

Now the acc is 20 and we have to do one more iteration for the last el:

# acc = 20

# el = 4

# new\_acc = 24

p [11, 7, 2, 4].inject { |acc, el| acc + el } # => 24

Since we are done iterating through all elements, inject will return the final accumulator. We were able to add up all elements of the array using inject! If you're not convinced of how useful inject is, you're probably thinking why don't we use a simpler loop or just use the Array#sum method. inject is an awesome method because of how versatile it is.

Applying our same steps as before, we'll leave it to you to ponder how inject can also find the total product of an array:

p [11, 7, 2, 4].inject { |acc, el| acc \* el } # => 616

Or how about finding the minimum value in an array:

p ([11, 7, 2, 4].inject do |acc, el|

if el < acc

el

else

acc

end

end) # => 2

*Note: You have to wrap the entire inject statement with () because, in Ruby, do ... end blocks have a lower precedence than method invocation argument assignment. (Precedence determines the order in which operations are executed, with higher precedence operations being executed first.) In other words, if you do not explicitly signal with ()s that the whole inject statement should be evaluated and passed as the argument to p, Ruby will interpret the command as p ([11, 7, 2, 4].inject) and then try to attach the do ... end block to the result of the p command. This will cause a no block given error when it tries to execute p ([11, 7, 2, 4].inject).*

Because the result of the block is always reassigned to be the new accumulator, we needed to return the current acc in the event that the el we are iterating through is not smaller than the acc. The else is necessary to avoid our block from resulting in nil.

The key to understanding inject is to remember that the accumulator will be reassigned to the result of the block on every iteration. Because inject performs a simple reassignment to the accumulator, we can design any block to control how the accumulator should change.

**Inject with a default accumulator**

In the last examples we described how the first element of the array will become the initial accumulator and the first iteration technically grabs the second element. We can also use inject by passing in our own initial accumulator. In this scenario, the acc will be our own value and the first el will be the first element of the array. Let's make 100 our initial accumulator:

# FIRST ITERATION:

# acc = 100

# el = 11

# new\_acc = 111

[11, 7, 2, 4].inject(100) { |acc, el| acc + el }

# SECOND ITERATION

# acc = 111

# el = 7

# new\_acc = 118

[11, 7, 2, 4].inject(100) { |acc, el| acc + el }

# THIRD ITERATION

# acc = 118

# el = 2

# new\_acc = 120

[11, 7, 2, 4].inject(100) { |acc, el| acc + el }

# FOURTH ITERATION

# acc = 120

# el = 4

# new\_acc = 124

p [11, 7, 2, 4].inject(100) { |acc, el| acc + el } # => 124

Nice! Being able to set our own initial accumulator can really open up our possibilities. Here's an inject that sums up all even numbers of an array:

[11, 7, 2, 4].inject(0) do |acc, el|

if el.even?

acc + el

else

acc

end

end # => 6