101 Practice Problems: Easy 1

```ruby
numbers = [1,2,2,3]
numbers.uniq

puts numbers

# 1
# 2
# 2
# 3
```

`numbers.uniq` returned a new array, but didn't modify the caller. So, we get the original back.

Question 1

```
Describe the difference between ! and ? in Ruby. And explain what would happen in the following scenarios:

what is != and where should you use it?
put ! before something, like !user_name
put ! after something, like words.uniq!
put ? before something
put ? after something
put !! before something, like !!user_name
```

Question 2

!= means "not equal" and can be used in comparisons
Putting a ! before something changes the boolean associated with it (makes it the opposite)
Putting an ! after something doesn't necessarily mean anything; it depends on the method being called
Putting a ? after something doesn't necessarily mean anything; it depends on the method being called
Putting a !! before something turns an object into its boolean equivalent

Question 3

```ruby
# Replace the word "important" with "urgent" in this string:

advice = "Few things in life are as important as house training your pet dinosaur."
```

`advice.gsub!(“important”, “urgent”)`

Question 4

What do the following method calls do (assume we reset numbers to the original array between method calls)?

```
numbers.delete_at(1)
numbers.delete(1)
```

`.delete_at()` deletes an item at the specified place (modifies the caller) and returns the value of that item
`.delete()` deletes an item that matches the argument (modifies the caller) and returns the value of that item

Question 5

Programmatically determine if 42 lies between 10 and 100.

```
numbers = []
(10..100).each do |num|
  numbers << num
  end

numbers.include?(42)

# LS's solution: (10..100).cover?(42)
```

Question 6

Starting with the string:

```
famous_words = "seven years ago..."
```

show two different ways to put the expected "Four score and " in front of it.

```
"Four score and " + famous_words
```

```
"Four score and " << famous_words
```

Question 7

Fun with gsub:

```
def add_eight(number)
  number + 8
end

```

```ruby
number = 2

how_deep = "number"
5.times { how_deep.gsub!("number", "add_eight(number)") }

p how_deep
```

This gives us a string that looks like a "recursive" method call:

```ruby
"add_eight(add_eight(add_eight(add_eight(add_eight(number)))))"
```

What is the result if we take advantage of Ruby's Kernel#eval method to have it execute this string as if it were a "recursive" method call?

```ruby
eval(how_deep)
```

ANSWER: 42. Not sure why.

Question 8

If we build an array like this:

```ruby
flintstones = ["Fred", "Wilma"]
flintstones << ["Barney", "Betty"]
flintstones << ["BamBam", "Pebbles"]
```

We will end up with this "nested" array:

```ruby
["Fred", "Wilma", ["Barney", "Betty"], ["BamBam", "Pebbles"]]
```

Make this into an un-nested array.

```ruby
flintstones.flatten!
```

Question 9

Given the hash below:

```ruby
flintstones = { "Fred" => 0, "Wilma" => 1, "Barney" => 2, "Betty" => 3, "BamBam" => 4,
"Pebbles" => 5 }
```

Turn this into an array containing only two elements: Barney's name and Barney's number.

```ruby
array = flintstones.assoc("Barney")
```