Puppet Labs
Software Engineer Intern – Installer Team (2018)
Anton Bilbaeno
2/20/18

Programming Challenge #1: Print If Prime

```
#!/usr/bin/ruby

# Puppet Labs
# Software Engineer Intern - Installer Team Application
# 2/19/18
# Programming Challenge Option #1: Range, Sum, Print if Prime
# Description: # Generate a range of sequential numbers from 1 to 100000 (inclusive), and for each
number in the range, add the digits of that number together. Print the sum of those digits if the sum
is a prime number. Otherwise, print nothing.

require 'prime'

def printIfPrime(n)
  if Prime.prime?(n)
    puts n
  end
end

[*1..100000].each { |x| printIfPrime(x.to_s.chars.map(&:to_i).reduce(:+)) }
```

Below is a step by step breakdown of how I arrived at the result above:
1. [*1..100000] is the creation of an array from 1 to 100000, inclusive.
2. ".each" goes through each item in the array
3. "|x|" defines the variable name used when iterating through the array with .each as x
4. "to_s" returns a string representation of the integer x
5. "chars" returns an array of individual character from a string
* At this point we have gone from a single integer into an array of characters.
6. map allows you to go through items in array and perform an operation
7. In this case we're going through every item (indicated by the & symbol) and performing the operation "to_i" which will return the integer representation of our character, so now we have an array of integers instead of an array of characters.
8. "reduce" allows you to take an enumerable and reduce it to a single value, in our case, we're looking to have reduce provide us with the sum of our array of integers
9. Finally, we have to check if this sum of our array of integers is prime or not. Originally I had the one liner print every time for testing and added the check for being a prime number at the end. This required multiple lines. Using a multi-line "each" looks like this:

```
[*1..100000].each do |x|
  sum_of_digits = x.to_s.chars.map(&:to_i).reduce(:+)
  if Prime.prime?(sum_of_digits)
    puts sum_of_digits
  end
end
```

- This result is a little more readable, but potentially uses more memory by storing the result before evaluating if it needs to be printed?
- I'm not sure how to check memory at this point in time, but I did try to time the results to see if either one was quicker. The results varied from 400-600 milliseconds for both implementations, so I couldn't draw any conclusions there. This code is on the next page below.
- Side note: I used the built-in Prime.prime?() function because I'm sure that's going to be much faster than anything I could cobble together at this time.

```
# Attempt at trying to benchmark my code with time… unsuccessful, results too varied
require 'benchmark'
time = Benchmark.realtime do
  [*1..100000].each { |x| printIfPrime(x.to_s.chars.map(&:to_i).reduce(:+)) }
end
puts "Time elapsed #{time*1000} milliseconds"

time = Benchmark.realtime do
  [*1..100000].each do |x|
    sum_of_digits = x.to_s.chars.map(&:to_i).reduce(:+)
    if Prime.prime?(sum_of_digits)
      puts sum_of_digits
    end
  end
end
puts "Time elapsed #{time*1000} milliseconds"
```

Programming Challenge #2: Stopwatch Average

After seeing how easy Ruby makes things with all of the built-in syntax (to_i, to_s, chars, map, reduce, etc.), I looked at the Stopwatch problem and saw that it would be relatively simple to break apart the string, get the integers, and average them. I actually had some fun learning Ruby and figured I might as well give the Stopwatch problem a go. However, then I remembered about having to deal with user input…

```
#!/usr/bin/ruby

# Puppet Labs
# 2/19/18
# Programming Challenge Option #2: Average Stopwatch Time
# Description: A stopwatch records lap times as strings of the form "MM:SS:HS", where MM = minutes, SS
= seconds, and HS = hundredths of a second. Write a program or script that accepts two lap times and
calculates their average, returning the result in the same string format. For example, given the
strings "00:02:20" and "00:04:40" the solution would return "00:03:30".

# This function checks that the user input is in the valid format.
def validTime(string)
  # Check for expected length
  if string.length == 8
    # do nothing, length is valid
  else
    puts "The time that you entered was not in the correct format.\nERROR: incorrect length."
    return false
  end

  # Delimiter in proper position
  valid_delimiter = true if string[2] == ":" and string[5] == ":"
  if string[2] == ":" and string[5] == ":"
    # do nothing, delimiters in correct position
  else
    puts "The time that you entered was not in the correct format.\nERROR: Colons were not in their
expected positions."
    return false
  end

  # Valid integer values
  numbers = string.split(":")
  for i in numbers

    # Remove a preceding zero if there is one
    if i[0] == "0"
      i = i[1]
      #puts "Removed preceeding zero."
    end

    # Check that we didn't convert any other characters to an integer in a somewhat obscure way by
converting the input string to an integer and back to a string again. This takes care of the case where
to_i will convert non integer characters to zero which are then valid integers in the check below.
    if not i.to_i.to_s == i
      puts "The time that you entered was not in the correct format.\nERROR: Invalid non-integer
characters."
      return false
    end

    # Check that the number is between 0 and 99
    # I'm not certain that this needs to occur after we check above that the characters were already
integers.
    i = i.to_i
    if i < 100 and i > -1
      # do nothing, valid integer value
    else
      puts "The time that you entered was not in the correct format.\nERROR: Integer values not between
00 and 99."
      return false
    end
  end

  return true
end
```

```
## Program Start
puts "Welcome to the stopwatch time addition program!"
puts "This program will take two times and average them. Times must be in the format 'MM:SS:HS' where
MM is minutes, SS is seconds, and HS is hundredths of a second. Don't forget your colons!"

# Get Time #1
valid_time = false
until valid_time == true
  puts "\nPlease enter Time #1:"
  t1 = gets.chomp
  valid_time = validTime(t1)
end
puts "Thank you for entering Time #1."

# Get Time #2
valid_time = false
until valid_time == true
  puts "\nPlease enter Time #2:"
  t2 = gets.chomp
  valid_time = validTime(t2)
end
puts "Thank you for entering Time #2."

# Get the integer values between the colons
t1 = t1.split(":").map{ |s| s.to_i }
t2 = t2.split(":").map{ |s| s.to_i }
# Add the values together
total = [t1,t2].transpose.map{ |x| x.reduce(:+) }
# Divide by 2 for the average
average = total.collect{ |t| t / 2 }
# Convert values back to a string
string_total = average.map{ |i| i.to_s }
# Prepend a 0 if single digit integer
average.length.times do |i|
  if average[i] < 10
    string_total[i].insert(0, "0")
  end
end

# Print the result
average = string_total[0] + ":" + string_total[1] + ":" + string_total[2]
puts "\nThe average of Time #1 and Time #2 is:\n#{average}"
```

Notes:

- Originally, the validTime function had three variables which I would set true or false based on the if-statements checking for length, delimiter position, and valid integer values. However, I found that I no longer needed these boolean variables and could simply return false immediately if any of the checks were not passed.
- I think I should have implemented everything in a function and not as much in the main body. Then, running the stopwatch program could be as simple as just calling a single line in the body.
- I had a lot of *fun* dealing with user inputs. I think I have accounted for all cases here. There's only so many buttons on a keyboard. I suppose it would be possible to generate all ASCII characters in an array and feed them into the program. Specifically: re-write the stopwatch program into a function that takes two input strings and iterate through the array of all possible characters for input to the function.