1. Basic Building Blocks

Solutions to Swirl's R Programming Exercises

06-19-2022

Acknowledgements: R Language Concepts and code questions (with minor modifications) are used here from the swirl package. https://www.r-project.org/nosvn/pandoc/swirl.html

Important note: We don't require to use library(swirl) and swirl() here because we are not going to run the R script in RStudio Console.

In this lesson, we will explore some basic building blocks of the R programming language. In its simplest form, R can be used as an interactive calculator. Type 5 + 7 and press Enter.

```r
5 + 7
```

```
## [1] 12
```

R simply prints the result of 12 by default. However, R is a programming language and often the reason we use a programming language as opposed to a calculator is to automate some process or avoid unnecessary repetition.

In this case, we may want to use our result from above in a second calculation. Instead of retyping 5 + 7 every time we need it, we can just create a new variable that stores the result.

The way you assign a value to a variable in R is by using the assignment operator, which is just a 'less than' symbol followed by a 'minus' sign. It looks like this: <-

Think of the assignment operator as an arrow. You are assigning the value on the right side of the arrow to the variable name on the left side of the arrow. To assign the result of 5 + 7 to a new variable called x, you type x <- 5 + 7. This can be read as 'x gets 5 plus 7'. Give it a try now.

```r
x <- 5 + 7
```

You'll notice that R did not print the result of 12 this time. When you use the assignment operator, R assumes that you don't want to see the result immediately, but rather that you intend to use the result for something else later on.

To view the contents of the variable x, just type x and press Enter. Try it now.

```r
x
```

```
## [1] 12
```

Now, store the result of x - 3 in a new variable called y.

```r
y <- x - 3
```

What is the value of y? Type y to find out.

```
y
```

```
## [1] 9
```

Now, let's create a small collection of numbers called a vector. Any object that contains data is called a data structure and numeric vectors are the simplest type of data structure in R. In fact, even a single number is considered a vector of length one.

The easiest way to create a vector is with the *c()* function, which stands for 'concatenate' or 'combine'. To create a vector containing the numbers 1.1, 9, and 3.14, type c(1.1, 9, 3.14). Try it now and store the result in a variable called z.

```
z <- c(1.1, 9, 3.14)
```

Anytime you have questions about a particular function, you can access R's built-in help files via the `?` command. For example, if you want more information on the *c()* function, type *?c* without the parentheses that normally follow a function name. Give it a try.

```
?c
```

```
## starting httpd help server ... done
```

Type z to view its contents. Notice that there are no commas separating the values in the output.

```
z
```

```
## [1] 1.10 9.00 3.14
```

You can combine vectors to make a new vector. Create a new vector that contains z, 555, then z again in that order. Don't assign this vector to a new variable, cso that we can just see the result immediately.

```
c(z, 555, z)
```

```
## [1]   1.10   9.00   3.14 555.00   1.10   9.00   3.14
```

Numeric vectors can be used in arithmetic expressions. Type the following to see what happens: z * 2 + 100.

```
z * 2 + 100
```

```
## [1] 102.20 118.00 106.28
```

First, R multiplied each of the three elements in z by 2. Then it added 100 to each element to get the result you see above.

Other common arithmetic operators are `+`, `-`, `/`, and ^ (where x^2 means 'x squared'). To take the square root, use the sqrt() function and to take the absolute value, use the abs() function.

Take the square root of z - 1 and assign it to a new variable called my_sqrt.

```r
my_sqrt <- sqrt(z -1)
```

Before we view the contents of the my_sqrt variable, what do you think it contains?

1: a vector of length 0 (i.e. an empty vector)

2: a vector of length 3

3: a single number (i.e a vector of length 1)

Think about how R handled the other 'vectorized' operations: element-by-element.

Print the contents of my_sqrt.

```r
my_sqrt
```

```
## [1] 0.3162278 2.8284271 1.4628739
```

As you may have guessed, R first subtracted 1 from each element of z, then took the square root of each element. This leaves you with a vector of the same length as the original vector z.

Now, create a new variable called my_div that gets the value of z divided by my_sqrt.

```r
my_div <- z/my_sqrt
```

Which statement do you think is true?

1: my_div is a single number (i.e a vector of length 1)

2: The first element of my_div is equal to the first element of z divided by the first element of my_sqrt, and so on…

3: my_div is undefined

Selection: 2

You are quite good my friend!

Go ahead and print the contents of my_div.

```r
my_div
```

```
## [1] 3.478505 3.181981 2.146460
```

Arithmetic operation (`+`, `-`, `*`, etc.) element-by-element If the vectors are of different lengths, R 'recycles' the shorter vector until it is the same length as the longer vector.

When we did z * 2 + 100 in our earlier example, z was a vector of length 3, but technically 2 and 100 are each vectors of length 1.

Behind the scenes, R is 'recycling' the 2 to make a vector of 2s and the 100 to make a vector of 100s. In other words, when you ask R to compute z * 2 + 100, what it really computes is this: z * c(2, 2, 2) + c(100, 100, 100).

To see another example of how this vector 'recycling' works, try adding c(1, 2, 3, 4) and c(0, 10). Don't worry about saving the result in a new variable.

```
c(1, 2, 3, 4) + c(0, 10)
```

```
## [1]  1 12  3 14
```

If the length of the shorter vector does not divide evenly into the length of the longer vector, R will still apply the 'recycling' method, but will throw a warning to let you know something fishy might be going on.

Try c(1, 2, 3, 4) + c(0, 10, 100) for an example.

```
c(1, 2, 3, 4) + c(0, 10, 100)
```

```
## Warning in c(1, 2, 3, 4) + c(0, 10, 100): longer object length is not a multiple
## of shorter object length
```

```
## [1]   1  12 103   4
```

Before concluding this lesson, I'd like to show you a couple of time-saving tricks.

Earlier in the lesson, you computed z * 2 + 100. Let's pretend that you made a mistake and that you meant to add 1000 instead of 100. You could either re-type the expression, or…

In many programming environments, the up arrow will cycle through previous commands. Try hitting the up arrow on your keyboard until you get to this command (z * 2 + 100), then change 100 to 1000 and hit Enter. If the up arrow doesn't work for you, just type the corrected command.

```
c(1, 2, 3, 4) + c(0, 10, 1000)
```

```
## Warning in c(1, 2, 3, 4) + c(0, 10, 1000): longer object length is not a
## multiple of shorter object length
```

```
## [1]    1   12 1003    4
```

If your environment does not support the up arrow feature, then just type the corrected command to move on.

The following five functions are probably swirl-specific - not sure! When you are at the R prompt (>):

– Typing skip() allows you to skip the current question.

– Typing play() lets you experiment with R on your own; swirl will ignore what you do…

– UNTIL you type nxt() which will regain swirl's attention.

– Typing bye() causes swirl to exit. Your progress will be saved.

– Typing main() returns you to swirl's main menu.

– Typing info() displays these options again.

In many programming environments, the up arrow will cycle through previous commands. Try hitting the up arrow on your keyboard until you get to this command (z * 2 + 100), then change 100 to 1000 and hit Enter. If the up arrow doesn't work for you, just type the corrected command.