Section 1

Foundations of Statistical Inference (PLSC503)

2019-01-24

Contents

Warm-up Exercises with Swirl

When you get to section, start with some warm-up exercises. Run the following in your console:

```
install.packages("swirl")
library("swirl")
```

You should see a message starting with "Hi!" – go ahead and follow those directions, choosing the following options when prompted:

- "1: R Programming: The basics of programming in R"
- "1: Basic Building Blocks"

When you get past 30%, raise your hand so we know where you are at. When we are done, save and exit by entering `bye()`.

Packages and Functions

Today, we will learn about functions, which are the main building blocks in R. Functions are a way to package code. And packages (e.g. Swirl) are an extension of that. Like apps on a smart phone, packages are a way for people to share different tools that they have built. In other words, it is a way for everyone to benefit by collaboration.

Most of the time, you will use packages in the following way. From Hadley Wickham, R Packages:

- You install them from CRAN with `install.packages("x")`.
- You use them in R with `library("x")`.
- You get help on them with `package?x` and `help(package = "x")`.

Once you have installed a package, just make sure to include `library("x")` in each R script.

Functions (User-Defined Functions)

What is a Function?

Quiz: What's a function? (In the context of yesterday's class.)

$$X : \Omega \to \mathbb{R}.$$

There are a few ways functions can be formally defined in math and computer science, but for our purposes, understand that functions take an **input** and returns an **output**. (Unlike the condition seen often in math that a function must map each input to exactly one output value, we can define functions in R to take some input values and perhaps return multiple output values, but that is a technically that is not important for us to understand functions today.)

The Guessing Game

Let us imagine a game where player A thinks of a number, and player B takes guesses. We will first code each guess and response as one round (or iteration, or loop). Later, we will code the sequences of guesses.

In Basic Code

Let us define x_a to be the value defined by player A. 87 is player A's lucky number (this is arbitrary).

```r
x_a <- 87
```

For player B's guess, let's take a random draw from the uniform distribution.

```r
runif(n = 1, min = 0, max = 100)
```

```
## [1] 15.05317
```

We can take another draw.

```r
runif(n = 1, min = 0, max = 100)
```

```
## [1] 21.32851
```

Now let's store this.

```r
set.seed(080)
guess_decimal <- runif(n = 1, min = 0, max = 100)
guess_decimal
```

```
## [1] 43.96122
```

(Question: What does `set.seed()` above do? Why do we need it?)

Next, let's round to the nearest integer. We will store this as guess_b.

```r
guess_b <- as.integer(guess_decimal)
guess_b
```

```
## [1] 43
```

How would you test whether the two values, `x` and `guess`, are equal?

```r
x_a == guess_b
```

```
## [1] FALSE
```

As a Function

The structure of functions

The basic structure of functions is:

```
myfunction <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

From Quick-R online. See link for more: Quick-R by DataCamp.

Defining our function for the guessing game

We will convert the basic code from earlier into a function. A function lets us define variables (in this case, `x` and `guess`) within any argument. In in this case, this lets us guess different numbers without having to re-write the code over and over again (which may seem alright for simple code like this, but increases the potential of error and headaches when you start working with more complicated code).

Let's name the function `guessing`.

```r
guessing <- function(x, guess) {
  # Store the result of the logical operator in `match`
  match <- x == guess

  # Store the result of the ifelse test in `out`
  out <- ifelse(test = match == TRUE, yes = "Correct!", no = "Wrong!")

  # Print `out`
  print(out)
}
```

Let's try our new function! Let's use values 1 and 2 to start.

```r
guessing(x = 1, guess = 2)
```

```
## [1] "Wrong!"
```

As a Loop

Making 100 guesses

Let's say we have 100 values we want to test for the guessing game.

Let's store our 100 guesses into a new object, `guesses`. We will use `sample` to draw 100 values from the vector of integers from 1 to 100:

```r
set.seed(080)
vec <- c(1:100)
guesses <- sample(x = vec, size = 100)
guesses
```

```
##   [1]  44  56  77   8  59   1  64  96  55  68  33  19  45  74  36  17  27
##  [18]  13 100  75  37  89   9  38   6  10  67  90  80  23  52  35  84  21
##  [35]  29  81  58  78  20  63  65  22  15  28  82  99  62  88  39  71  54
##  [52]  14  48  66  46  76  42  61   3  86  95   4  24  91  31  97   5  30
##  [69]  18  41  93  32  92  26   2  11  73  53  72   7  16  47  34  94  98
##  [86]  69  51  83  85  87  70  12  79  43  60  40  25  57  50  49
```

How can we go about testing all of these values? We could paste the function and define values for each `guess` variable… but that seems like a lot. Surely there must be a better way!?, you ask. Indeed, there are several! First, let's try using a "for" loop.

The structure of a "for" loop

The idea of a "for" loop is to repeat code over a vector of values.

```
for (value in sequence) {
  statement
}
```

A simple example: For each of the values in `vector_example`, we apply the arguments defined within the curly braces:

```r
vector_example <- c(10, 20, 30)

# For each of the values in `vector_example`,
for (val in vector_example) {
  # Add 1 and store in the object `out`
  # (This defines a new `out` for each loop)
  out <- val + 1

  # Print the value of `out` (for each loop)
  print(out)
}
```

```
## [1] 11
## [1] 21
## [1] 31
```

Defining our loop for the guessing game

The variable in our loop will be the values stored in the vector `guesses`. Each loop will take one value, $guesses_i$. There are n values we want to guess, so we define i over 1 to n, i.e. the first loop will ask: does $guesses_1$, the first value in `guesses`, equal x, the correct answer?, and so on, until $guesses_100$.

Now we can run the loop:

```r
# Define n
n <- length(guesses)

# Define x value chosen by player a
x_a <- 87

# The loop
for (i in 1:n) {
  # Define `guess` as the i_th value from the `guesses` vector
  guess <- guesses[i]

  # Store the result of the logical operator test into the object `match`
  match <- x_a == guess

  # Store the ifelse statement result into `out`
  out <- ifelse(test = match == TRUE, yes = "Correct!", no = "Wrong!")
  # Print `out`
  print(out)
}
```

```
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Correct!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
```

Since we have already defined the function, we can also just loop it over the function `guessing`:

```r
# Define x value chosen by player a
x_a <- 87

for (i in 1:n){
  guessing(x = x_a, guess = guesses[i])
}
```

```
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Correct!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
## [1] "Wrong!"
```

If interested, here is a fancier version: Guessing game example from rexamples.com.