User-Defined Functions and Loops

Functions

Up to this point you've used a multitude of built-in functions; that is, functions that are either already installed and loaded with base R (e.g., `mean()`), or else can be loaded into R with a package, like **tidyverse** (e.g., `ggplot()`). One of the most powerful R skills is the ability to create custom, user-specific functions. Not every function is developed with your specific needs in mind. So being able to create your own functions is a powerful skill because you won't be dependent on other people who developed functions with objectives that may not correspond to your own.

Why create your own functions? For one thing, as you practice more and more you'll find yourself copy and pasting code over and over and over, which is a laborious and inefficient way to complete a task. Creating a function to automate this copy-and-paste process will expedite the task and also helps to avoid mistakes that naturally result when you copy and paste multiple times. There's an informal rule that says once you've copy and pasted a block of code more than twice, then you should think about creating a function.

The generic syntax of user-defined functions is as follows:

```
# Generic syntax for user-defined functions
function_name <- function(arg1, arg2, ...) {
                     body_of_function
}
```

As an example, let's create a function that multiplies two numbers. We need to assign it a name, say, `multiply`, and then put 2 arbitrary but distinct symbols for the function's arguments, say, `x` and `y`. The body of the function, what we want the function to do, will then be `x*y`. Altogether then, we'll define the function `multiply()` by:

```
# Defining the multiply function
multiply <- function(x, y) {
                x*y
}
```

To use it we just insert 2 numbers in place of `x` and `y` in `multiply()`. For example, the product of `2` and `-5`:

```
# Multipying 2 and -5 with the multiply function
multiply(2, -5)
```

More complicated functions require more than one line of code in the body of the function. Consider two very similar functions: one that multiplies 2 numbers and then adds those 2 numbers and another function that adds 2 numbers and then multiplies those 2 numbers. This is how these 2 functions are defined:

```
multiply_add <- function(x, y){
                        x*y
                        x + y
}

add_multiply <- function(x, y){
                        x + y
                        x*y
}
```

Before evaluating the functions with specific `x` and `y` values, do you think the functions equivalent (i.e., return the same output for the same arguments)? Or are they different functions?

The functions `multiply_add()` and `add_multiply()` are different functions as verified with `x = 4` and `y = -7`:

```
multiply_add(4, -7)
add_multiply(4, -7)
```

Note that `multiply_add()` returned the sum of `4` and `-7`, while `add_multiply()` returned the product of `4` and `-7`. Custom functions execute the last line of code in the body of the function, and all code above that last line is ignored. If we wanted `multiply_add()` to instead return the product of `x` and `y`, then we need to insert the `return()` function at the end of body, while also assigning names to the sum and product of `x` and `y`, as shown here:

```
multiply_add <- function(x, y){
                          mult <- x*y
                          add <- x + y
                          return(mult)
}
```

Now, note that we get product of `4` and `-7` even though the command adding `x` and `y` is below the command multiplying `x` and `y`:

```
multiply_add(4, -7)
```

It's also worth pointing out that, since I redefined `multiply_add()` while retaining its name, the more recent definition with the `return()` function replaces the original `multiply_add()` function.

One thing I find myself doing a lot is constructing plots with `ggplot()`. As you're aware by now, `ggplot()` code blocks can easily get lengthy. To reduce the amount of code you can automate the building blocks of most plots by creating a function. The function below, which I've named `gg_fcn`, only requires you to insert 3 arguments: The data frame, the x-variable and the y-variable. All other plot components (its geometry, style add-ons, and so on) can be introduced as in the past by extending the command with a `+` between plot components. Here's the function:

```
gg_fcn <- function(data, x_var, y_var){
  my_ggplot <- ggplot(data = data, mapping = aes(x = x_var, y = y_var))
  return(my_ggplot)
}
```

As an example, create a scatterplot with a regression line using `mpg` data frame:

```
gg_fcn(mpg, mpg$displ, mpg$hwy) +
  geom_smooth() +
  geom_point(mapping = aes(color= class)) +
  labs(x = "Engine Displacement (in Liters)",
       y = "Highway Miles per Gallon")
```

```
## Error in ggplot(data = data, mapping = aes(x = x_var, y = y_var)): could not find function "ggplot"
```

Notice that, when using the user-defined function, we do have to specify the data frame and column using `$` notation for the x- and y-variables. Still, `gg_fcn()` does save time, especially if you're frequently generating plots. And you can of course further customize it to suit your specific needs.

(For) Loops

Compared to other programming languages, loops do not figure into the regular work flow in R. Nevertheless, knowing how to use iteration techniques is not just fundamental to learning how to program; it reveals the

logic behind the code, an understanding of which, will extend into other, more complex realms.

One way to use a loop is to simulate a distribution. You're already familiar with the binomial distribution (recall Buzz, Doris and flashing buttons). We can use the binomial distribution to examine the distribution of births since babies are born as either boys or girls,.[1]

The probability a baby is born a girl is approximately 48.8%, and this probability doesn't vary much across the world. To simplify this example, let's suppose 500 babies are born in a hospital in a given year. How many do you expect to be girls?

We can simulate the number of baby girls born out of 500 births using the `rbinom()` function:

```
# Simulate number of female births out of 500
rbinom(n = 1, size = 500, prob = 0.488)
```

```
## [1] 238
```

Note that this is analogous to flipping a coin 500 times, each toss having a 48.8% chance of landing heads.

This shows what could happen in 500 births; it's a particular number between 0 and 500, inclusive, randomly generated from a binomial distribution. To get a sense of the *distribution* of what could happen, let's simulate the process 1000 times.

Because we're simulating data and want to obtain the same results, let's first set a seed:

```
# Set seed for reproducible results
set.seed(12345)
```

There are 3 steps to create the desired distribution of female births:

1. Determine the number the simulations to perform.
    - We already decided this will be 1000, which we'll code as `n_sims <- 1000`. Referring back to the coin toss example, this would be like flipping a coin 500 times and recording the number of heads, then flipping a coin 500 times and recording the number of heads, then . . . and so on, until you've done it 1000 times. We'll obtain 1000 observations to form the distribution.
2. Create a vector to store the simulated data.
    - This can be done using the `vector()` function. Its two arguments are the type of data to be stored and the length of the vector. In this example, we're dealing with real numbers (or integers if you wish), so put `"double"` as the first argument. The second argument of `vector()`, its length, is 1000 and so we can use `n_sims`.
3. Create a for loop to iterate
    - This is the hard part.

Putting these 3 parts together, we have:

```
# Number of simulations
n_sims <- 1000

# Create vector to store simulated data; specify type and length of vector
n_girls <- vector("double", n_sims)

# Loop over number of MORE MORE MORE
for (i in 1:n_sims) {
  n_girls[[i]] <- rbinom(n = 1, size = 500, prob = 0.488)
}
```

[1] Remember: Binomial just means two names (e.g., "heads" or "tails," "success" or "failure"). In the present example, the two names are "boy" and "girl."

More concretely, `n_girls[1]` is the first randomly drawn number of female births out of 500 births, `n_girls[2]` is the second randomly drawn number of female births out of 500 births, . . ., and `n_girls[1000]` is the thousandth randomly drawn number of female births out of 500 births, each of these 1000 draws having probability 0.488.

You can probably guess determine the mean number of girl births out of 500 should (in theory) be if the probability of a girl birth is 48.8%. In the Console, run `0.488*500` to verify it's 244 girl births per 500 births. In fact, for any binomial distribution the mean is the number of draws (the `size` in `rbinom()`) multiplied by the probability of drawing a success. Let's see how the mean of the simulated data compares:

```
# Compute mean of simulated data
mean(n_girls)
```

```
## [1] 243.776
```

Get a feel for the simulated distribution by creating a histogram:

```
# View ggplot histogram of simulated data
ggplot(data = data.frame(n_girls),
       mapping = aes(x = n_girls)) +
  geom_histogram(binwidth = 1,
                 color = "black",
                 fill = "lightblue")
```

```
## Error in ggplot(data = data.frame(n_girls), mapping = aes(x = n_girls)): could not find function "gg
```

The 1000 simulations capture the uncertainty that was present but veiled when we performed just one simulation. We could, of course, increase the number of simulations to gain a better approximation to the theoretical distribution. But, as shown above, if we're interested in the expected value of female births out of 500, then 1000 simulations is enough to determine the approximate value.

We simulated the distribution using for loop to get practice with loops. But all this can be done more directly using the `rbinom()` function. For example, to find another simulated mean, run:

```
# Find mean of same simulated binomial distribution
mean(rbinom(n = n_sims, size = 500, prob = 0.488))
```