Exploring data 2

Functions

Functions

As you move to larger projects, you will find yourself using the same code a lot.

Examples include:

- Reading in data from a specific type of equipment (air pollution monitor, accelerometer)
- Running a specific type of analysis (e.g., fitting the same model format to many datasets)
- Creating a specific type of plot or map

If you find yourself cutting and pasting a lot, convert the code to a function.

Functions

Advantages of writing functions include:

- Coding is more efficient
- Easier to change your code (if you've cut and paste code and you want to change something, you have to change it everywhere)
- Easier to share code with others

Functions

You can name a function anything you want, as long as you follow the naming rules for all R objects (although try to avoid names of preexisting-existing functions). You then specify any inputs (arguments; separate multiple arguments with commas) and put the code to run in braces. You **define** a function as an R object just like you do with other R objects (`<-`).

Here is the basic structure of "where things go" in an R function definition.

```
## Note: this code will not run
[function name] <- function([any arguments]){
        [code to run]
}
```

Functions

Here is an example of a very basic function. This function takes a number as the input (`number`) and adds 1 to that number. An R function will only return one R object. By default, that object will be the last line of code in the function body.

```r
add_one <- function(number){
        number + 1 # Value returned by the function
}

add_one(number = 1:3)
```

```
## [1] 2 3 4
```

```r
add_one(number = -1)
```

```
## [1] 0
```

Functions

```r
add_one <- function(number){
        number + 1 # Value returned by the function
}
```

- I picked the name of the function (`add_one`) (just like you pick what name you want to use with any R object)
- The only input is a numeric vector. I pick the name I want to use for the vector that is input to the function. I picked `number`.
- Within the code inside the function, the `number` refers to the numeric vector object that the user passed into the function.

Functions

As another example, you could write a small function to fit a specific model to a dataframe you input and return the model object:

```
fit_time_pos_mod <- function(df){
  lm(Tackles ~ Time + Position,
     data = df) # Returns result from this call
}
```

- I picked the name of the function (`fit_time_pos_mod`) (just like you pick what name you want to use with any R object)
- The only input is a dataframe. I pick the name I want to use for the dataframe that is input to the function. I picked `df` (I often use this as a default parameter name for a dataframe).
- Within the code inside the function, the `df` refers to the dataframe object that the user passed into the function.

Functions

Now you can apply that function within a tidy pipeline, for example to fit the model to a specific subset of the data (the top four teams):

```
data(worldcup)
worldcup %>%
  filter(Team %in% c("Spain", "Netherlands",
                     "Uruguay", "Germany")) %>%
  fit_time_pos_mod() %>%
  tidy()
```

```
## # A tibble: 5 x 5
##   term               estimate std.error statistic  p.value
##   <chr>                 <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)         -0.312    1.34       -0.232 8.17e- 1
## 2 Time                 0.0237   0.00268     8.85  3.60e-13
## 3 PositionForward     -3.31     1.49       -2.22  2.95e- 2
## 4 PositionGoalkeeper -12.6      2.66       -4.75  9.77e- 6
## 5 PositionMidfielder   2.23     1.32        1.68  9.68e- 2
```

Functions

- Functions can input any type of R object (for example, vectors, data frames, even other functions and ggplot objects)
- Similarly, functions can output any type of R object
- However, functions can only output one R object. If you have complex things you want to output, a list might be a good choice for the output object type.
- Functions can have "side effects". Examples include printing something or drawing a plot. Any action that a function takes *besides returning an R object* is a "side effect".

Functions—parameter defaults

When defining a function, you can set default values for some of the parameters. For example, in the add_one function, you can set the default value of the number input to 0.

```
add_one <- function(number = 0){
        number + 1 # Value returned by the function
}
```

Now, if someone runs the function without providing a value for number, the function will use 0. If they do provide a value for number, the function will use that instead.

```
add_one()    # Uses 0 for `number`
```

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

```
add_one(number = 3:5)   # Uses 5 for `number`
```

```
## [1] 4 5 6
```

Functions—parameters

You could write a function with no parameters:

```
hello_world <- function(){
  print("Hello world!")
}

hello_world()
```

```
## [1] "Hello world!"
```

However, this will be pretty uncommon as you're first learning to write functions.

Functions—parameters

You can include multiple parameters, some with defaults and some without. For example, you could write a function that inputs two numbers and adds them. If you don't include a second value, 1 will be added as the second number:

```
add_two_numbers <- function(first_number, second_number = 1){
  first_number + second_number
}
```

```
add_two_numbers(first_number = 5:7, second_number = 5)
```

```
## [1] 10 11 12
```

```
add_two_numbers(first_number = 5:7)
```

```
## [1] 6 7 8
```

Functions—the `return` function

You can explicitly specify the value to return from the function (use `return` function).

```
add_one <- function(number = 0){
        new_number <- number + 1
        return(new_number)
}
```

If using `return` helps you think about what's happening with the code in your function, you can use it. However, outside of a few exceptions, you usually won't need to do it.

if / else

In R, the `if` statement evaluates everything in the parentheses and, if that evaluates to `TRUE`, runs everything in the braces. This means that you can trigger code in an `if` statement with a single-value logical vector:

```r
tell_date <- function(){
  cat("Today's date is: ")
  cat(format(Sys.time(), "%b %d, %Y"))

  todays_wday <- lubridate::wday(Sys.time(),
                                 label = TRUE)
  if(todays_wday %in% c("Sat", "Sun")){
    cat("\n")
    cat("It's the weekend!")
  }
}
```

if / else

```
tell_date()
```

```
## Today's date is: Oct 04, 2020
## It's the weekend!
```

if / else

You can add `else if` and `else` statements to tell R what to do if the condition in the `if` statement isn't met.

For example, in the `tell_date` function, we might want to add some code so it will print `"It's almost the weekend!"` on Fridays and how many days until Saturday on other weekdays.

if / else

```r
tell_date <- function(){
  # Print out today's date
  cat("Today's date is: ")
  cat(format(Sys.time(), "%b %d, %Y."), "\n")

  # Add something based on the weekday of today's date
  todays_wday <- lubridate::wday(Sys.time())

  if(todays_wday %in% c(1, 7)){      # What to do on Sat / Sun
    cat("It's the weekend!")
  } else if (todays_wday == c(6)) {  # What to do on Friday
    cat("It's almost the weekend!")
  } else {                           # What to do other days
    cat("It's ", 7 - todays_wday, "days until the weekend.")
  }
}
```

if / else

```r
tell_date()
```

```
## Today's date is: Oct 04, 2020.
## It's the weekend!
```