Defining Functions

Augustus Pendleton

Why write functions in R?

As biologists, writing functions seems like computer science-cy stuff that we don't need to touch. However, writing functions can make our code more readable, more reproducible, and altogether more flexible. There are also times in R where defining a function is required for specific analyses or tasks. Luckily, writing functions in R is fairly approachable. Let's go ahead and define our first functions.

Functions describe *what to do* to a specified *argument*

A function is like a recipe we wrote, which takes defined ingredients, performs some task on them, and returns an output. We call these ingredients **arguments**. Below, I demonstrate the standard syntax for defining a function:

```
times_two <- function(x){
  x * 2
}
```

Now, we've defined a function called `times_two`. Defining a function is the same as with any object in R - we use the assignment operator `<-`. Next, we use the (function) `function`, and in parentheses, we define the *arguments* that function takes. In this case, the function takes 1 argument, called x. Finally, in curly brackets, we define what the function *does* to its argument. In this case, it takes x, and multiplies it by two. We can see this happen below

```r
times_two(2)
```

```
[1] 4
```

```r
times_two(x = 4)
```

```
[1] 8
```

```r
times_two(y = 9)
```

```
Error in times_two(y = 9): unused argument (y = 9)
```

Note, it's optional whether we specify the name of the argument when we call the function (`x=4`); if we don't, R assumes we are referring to x. However, if we explicitly refer to an argument that that function doesn't know (`y`), we will get an error.

Functions can be defined with as many arguments as you want:

```r
divide <- function(number1, number2){
  number1 / number2
}
```

Now I've defined a function call `divide`, which takes two arguments (`number1` and `number2`) and divides `number1` by `number2`. We can check that it does this:

```r
divide(12,3)
```

```
[1] 4
```

```r
divide(20,4)
```

```
[1] 5
```

```r
divide(4, 20)
```

```
[1] 0.2
```

By default, R assumes you provide arguments in the same order you defined them. We can provide arguments in a different order by using their names:

```r
divide(number2 = 3, number1 = 12)
```

```
[1] 4
```

When is defining a function useful?

Okay, at this point, I hope you are as delighted as I am about how fun it is to define your own functions. However, the above examples might feel a little silly - the functions for adding or dividing numbers are already defined in R - why would we ever define our own? Generally, it's helpful to define our own functions for specific tasks that are relevant to us, but that we need to do multiple times. Let's work through an example, to see what I'm talking about.

Analyzing multiple Taylor Swift datasets

In our data folder, we have multiple csv sheets, each corresponding to a Taylor Swift album. Let's take a look at some of these files.

```r
list.files("data")
```

```
[1] "evermore.csv"
[2] "fearless_taylors_version.csv"
[3] "fearless.csv"
[4] "folklore.csv"
[5] "lover.csv"
[6] "midnights.csv"
[7] "red_taylors_version.csv"
[8] "red.csv"
```

```
 [9] "reputation.csv"
[10] "speak_now.csv"
[11] "taylor_swift.csv"
[12] "the_taylor_swift_holiday_collection.csv"
[13] "x1989.csv"
```

```r
read.csv("data/evermore.csv")
```

```
   album_release               track_name danceability energy loudness   tempo
1     2020-12-11                   willow        0.392  0.574   -9.195  81.112
2     2020-12-11       champagne problems        0.462  0.240  -12.077 171.319
3     2020-12-11                gold rush        0.512  0.462  -10.491 112.050
4     2020-12-11    'tis the damn season        0.575  0.434   -8.193 145.916
5     2020-12-11              tolerate it        0.316  0.361  -10.381  74.952
6     2020-12-11       no body, no crime        0.546  0.613   -7.589  79.015
7     2020-12-11                happiness        0.559  0.334  -10.733 122.079
8     2020-12-11                 dorothea        0.605  0.488   -8.322 119.966
9     2020-12-11             coney island        0.537  0.537  -11.266 107.895
10    2020-12-11                      ivy        0.515  0.545   -9.277  88.856
11    2020-12-11           cowboy like me        0.604  0.517   -9.014 127.967
12    2020-12-11         long story short        0.546  0.730   -7.704 157.895
13    2020-12-11                 marjorie        0.535  0.561  -11.609  96.103
14    2020-12-11                  closure        0.689  0.704  -10.813 151.884
15    2020-12-11                 evermore        0.390  0.270  -10.673 125.177
16    2020-12-11 right where you left me        0.581  0.619   -6.524 137.915
17    2020-12-11           it's time to go        0.592  0.410  -12.426 151.923
   duration_ms
1       214707
2       244000
```

```
3        185320
4        229840
5        245440
6        215627
7        315147
8        225880
9        275320
10       260440
11       275040
12       215920
13       257773
14       180653
15       304107
16       245027
17       254640
```

```r
read.csv("data/x1989.csv")
```

	album_release	track_name	danceability	energy	loudness
1	2014-10-27	Welcome To New York	0.789	0.634	-4.762
2	2014-10-27	Blank Space	0.760	0.703	-5.412
3	2014-10-27	Style	0.588	0.791	-5.595
4	2014-10-27	Out Of The Woods	0.553	0.841	-6.937
5	2014-10-27	All You Had To Do Was Stay	0.605	0.725	-5.729
6	2014-10-27	Shake It Off	0.647	0.800	-5.384
7	2014-10-27	I Wish You Would	0.653	0.893	-5.966
8	2014-10-27	Bad Blood	0.646	0.794	-6.104
9	2014-10-27	Wildest Dreams	0.550	0.688	-7.416
10	2014-10-27	How You Get The Girl	0.765	0.656	-6.112

```
11    2014-10-27                This Love        0.481  0.435   -8.795
12    2014-10-27            I Know Places        0.602  0.755   -4.991
13    2014-10-27                    Clean        0.815  0.377   -7.754
14    2014-10-27               Wonderland        0.422  0.692   -5.447
15    2014-10-27           You Are In Love        0.474  0.480   -8.894
16    2014-10-27           New Romantics        0.633  0.889   -5.870
     tempo duration_ms
1  116.992      212600
2   95.997      231827
3   94.933      231000
4   92.008      235800
5   96.970      193293
6  160.078      219200
7  118.035      207440
8  170.216      211933
9  139.997      220440
10 119.997      247533
11 143.950      250093
12 159.965      195707
13 103.970      271000
14 184.014      245560
15 170.109      267107
16 121.956      230467
```

It looks like our files all have the same data in them. Our goal is to find the longest song on every album. I might do it using code that looks like this:

```
evermore_data <- read.csv("data/evermore.csv") # Read in data
```

```
durations <- evermore_data[["duration_ms"]] # Get durations as a vector

index_of_longest <- which.max(durations) # Find the index of the largest duration

evermore_data$track_name[index_of_longest] # Select the corresponding track name
```

```
[1] "happiness"
```

Then, if I wanted to do this for each album, I could copy and paste that code 13 times, changing "evermore" to each album name in every one.

What are the downsides to this? I would have a very long script with many repetitions. This makes the chance of typos much higher, and makes troubleshooting code more difficult. Also, if I decide I want to change something (perhaps find the shortest song instead of the longest), I need to change that 13 times!

We can simplify this process using by defining a function that does these tasks for us.

```
longest_song <- function(album_path){
  album_data <- read.csv(album_path) # Read in data

  durations <- album_data[["duration_ms"]] # Get durations as a vector

  index_of_longest <- which.max(durations) # Find the index of the largest duration

  longest_song <- album_data$track_name[index_of_longest] # Select the corresponding track

  return(longest_song)

}
```

Notice that now I also used a function called `return` to define exactly which value the function should output. By default, a function in R will just return whatever was the last value called in the function. However, it is good practice to be explicit about what the function outputs. Let's use ouf function now

```r
longest_song("data/evermore.csv")
```

```
[1] "happiness"
```

```r
longest_song("data/red.csv")
```

```
[1] "All Too Well"
```

```r
longest_song("data/x1989.csv")
```

```
[1] "Clean"
```

Wow - much nice! We've taken what would have been 12 lines of repetitive code and reduced it to 3. This is already a huge improvement. While not the focus of today's lesson, defining functions becomes especially useful when we combine them with iterative functions like `map`, which apply a function across a list of values. I demonstrate this below:

```r
files <- list.files("data", full.names = TRUE)

purrr::map_chr(files, longest_song)
```

```
 [1] "happiness"
 [2] "Untouchable (Taylor's Version)"
 [3] "Untouchable"
 [4] "betty"
 [5] "Daylight"
 [6] "Would've, Could've, Should've"
 [7] "All Too Well (10 Minute Version) [Taylor's Version] [From The Vault]"
```

```
 [8] "All Too Well"
 [9] "End Game"
[10] "Dear John"
[11] "Tied Together With A Smile"
[12] "Christmas Must Be Something More"
[13] "Clean"
```

This automatically applied our `longest_song` function to all thirteen data files that we have. Now, the process of reading in each file and calculating the longest song has been reduced from ~53 lines of code to just 2.

Functions help make us more flexible

THe other benefit of a function is that you can make it more flexible, so that you can use it for multiple different tasks. Below, I re-write our `longest_song` function so that I can find the song with the maximum over any variable in the dataset (e.g. the most loud, the most long).

```
most_x_song <- function(album_path, var_of_interest){
  album_data <- read.csv(album_path) # Read in data

  values <- album_data[[var_of_interest]] # Get durations as a vector

  index_of_most <- which.max(values) # Find the index of the largest duration

  most_song <- album_data$track_name[index_of_most] # Select the corresponding track name

  return(most_song)

}
```

Now, I can find the loudest, longest, or most danceable song for each album

```r
most_x_song("data/evermore.csv", "loudness")
```

```
[1] "right where you left me"
```

```r
most_x_song("data/evermore.csv", "tempo")
```

```
[1] "champagne problems"
```

We can similarly use map functions to quickly apply this function across all of our files.

```r
purrr::map_chr(files, most_x_song, var_of_interest = "loudness")
```

```
 [1] "right where you left me"
 [2] "Tell Me Why (Taylor's Version)"
 [3] "Tell Me Why"
 [4] "the 1"
 [5] "ME!"
 [6] "Would've, Could've, Should've"
 [7] "I Bet You Think About Me (Taylor's Version) [From The Vault]"
 [8] "Red"
 [9] "This Is Why We Can't Have Nice Things"
[10] "Haunted"
[11] "Picture To Burn"
[12] "Christmas Must Be Something More"
[13] "Welcome To New York"
```

```r
purrr::map_chr(files, most_x_song, var_of_interest = "energy")
```

```
 [1] "long story short"
 [2] "Tell Me Why (Taylor's Version)"
 [3] "Tell Me Why"
```

```
 [4] "mad woman"
 [5] "ME!"
 [6] "Would've, Could've, Should've"
 [7] "Holy Ground (Taylor's Version)"
 [8] "Red (Original Demo Recording)"
 [9] "This Is Why We Can't Have Nice Things"
[10] "Haunted"
[11] "I'm Only Me When I'm With You"
[12] "Christmas Must Be Something More"
[13] "I Wish You Would"
```

Anonymous Functions

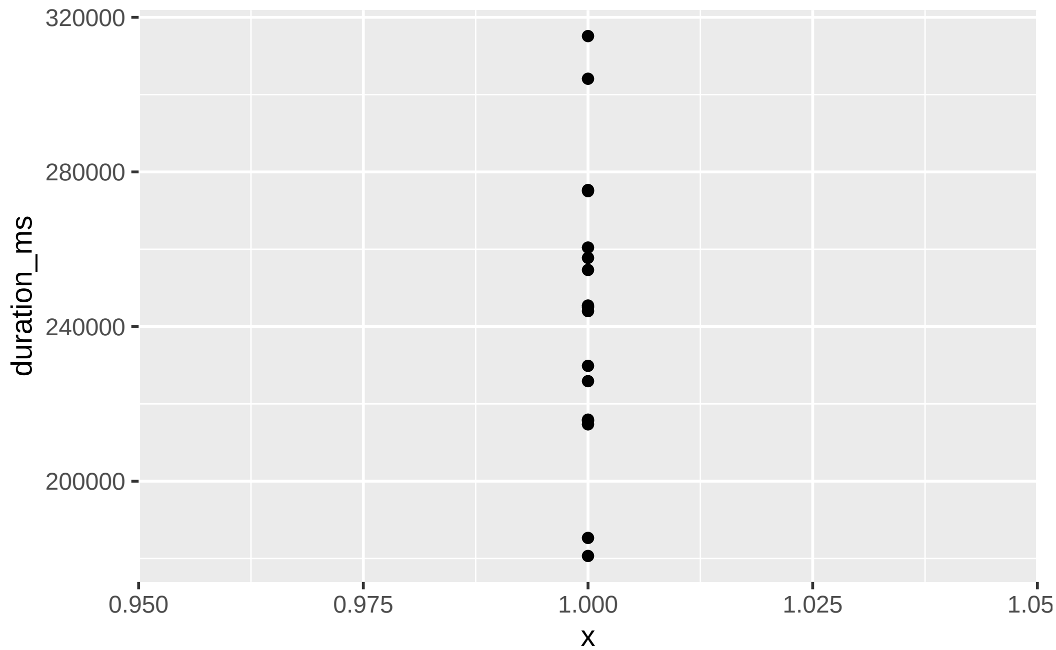
The last thing we'll cover today is the concept of anonymous functions. These are, as the name implies, functions that don't have a name - they are anonymous. I use anonymous functions a LOT, and typically in two places:

- Inside of map, to define a new function
- When using ggplot, to calculate a summary statistic

We'll use the evermore data to discuss the second point. First, let's load ggplot and make an example plot:

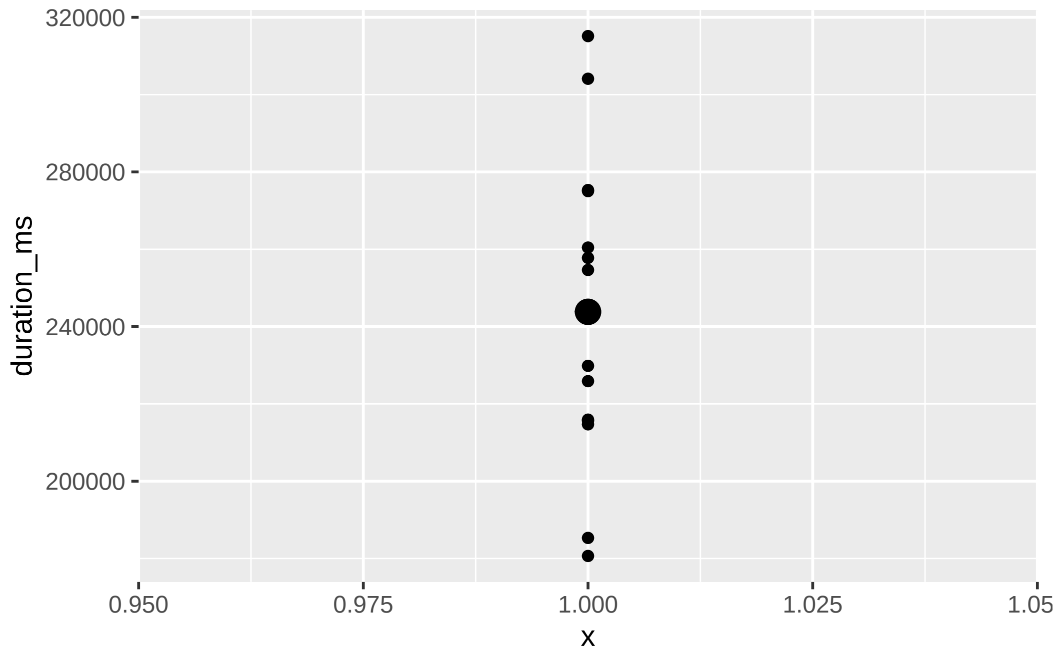
```r
library(ggplot2)

ggplot(data = evermore_data, aes(x = 1, y = duration_ms)) +
  geom_point()
```



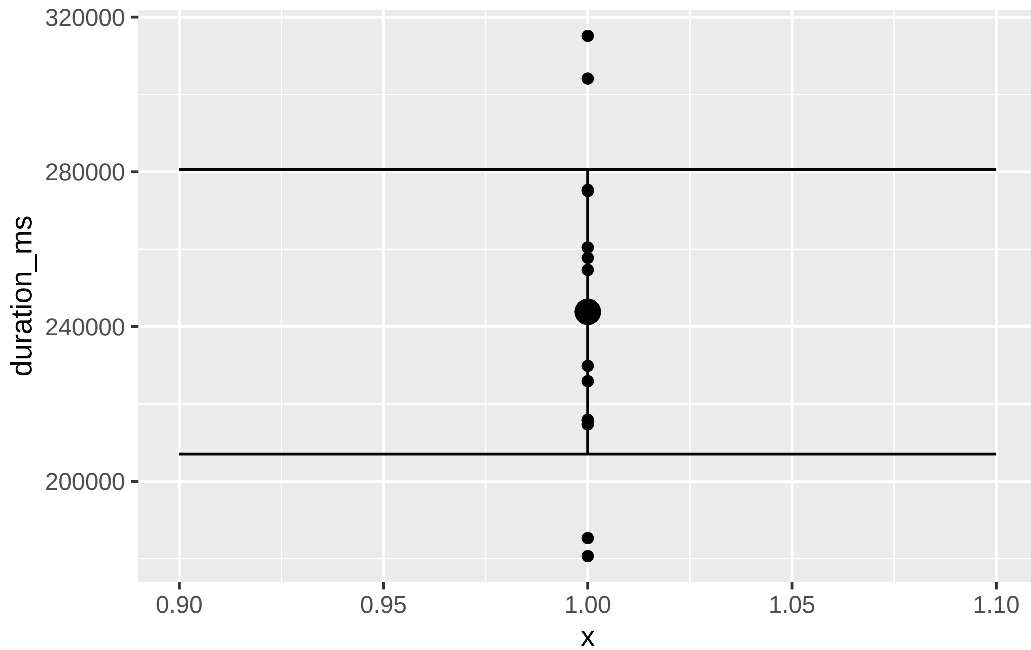
Here, each point is a song, and on the y axis, we have its duration. I think it would be helpful to add a larger point, representing the mean duration. We can do this with a function call `stat_summary`

```
ggplot(data = evermore_data, aes(x = 1, y = duration_ms)) +
  geom_point() +
  stat_summary(geom = "point", fun = mean, size = 4)
```



In stat_summary we provided a function (`fun`) called `mean` to calculate the position of our new geom (`point`). I also change the size so that we can notice that it's the mean. All good - but what if we want to add error bars? We can again use stat_summary, but this time we'll define a function *within it* that calculates the upper and lower bounds of the errorbar

```
ggplot(data = evermore_data, aes(x = 1, y = duration_ms)) +
  geom_point() +
  stat_summary(geom = "point", fun = mean, size = 4) +
  stat_summary(geom = "errorbar", width = 0.2,
               fun.min = function(x)mean(x) - sd(x),
               fun.max = function(x)mean(x) + sd(x))
```



Here, we define a function which takes the mean of the points and either subtracts or adds the standard deviation of the points. We could define this function outside and give it a name (something like `upper_error` and `lower_error` and pass that to stat_summary instead. However, since we're just using it here, we provide a function without giving it a name - hence, an anonymous function.

While it may seem silly off the bat, anonymous functions are very popular, and as you google questions and use others code, you'll keep running into them. As such, it's worth while to understand what they're doing.