Writing functions

Being able to group by and summarize data is great. But so far all we know how to do is use canned functions - ones that come with base R or one of the packages we've covered. We'll need to write our own functions eventually.

Functions in R are defined almost the same as variables. The general syntax looks like this:

```
name <- function(reqd_arg, optional_arg=42) {
    # do stuff
    return(result)
}
```

Let's create a function that adds two numbers together as an example.

```
adder <- function(num1, num2) {
    result <- num1 + num2
    return(result)
}
```

We can now use our function just like any other.

```
adder(5, 6)
```

```
## [1] 11
```

We also have the ability to specify optional arguments. Optional arguments are just ones where we've given it a default. In this case, we'll make our `adder` function just add 10 if a second number is not specified. Notice that we've also eliminated saving the `results` variable and do everything in one line.

```
adder <- function(num1, num2=10) {
    return(num1 + num2)
}
adder(5)
```

```
## [1] 15
```

Exercise - Writing our own functions:

Write a function that converts feet to meters. 1 foot equals 0.3048 meter.

Exercise - Applying our own functions:

The `airports` table from `nycflights13` is using feet for altitude instead of meters. Add a column `alt_meters` to correct this mistake. You'll need to use your function from the last example.

Conditional expressions

Sometimes we need to have functions do things differently if a certain condition is met. For this, we use if/else statements. `if` executes a block of code if some condition was met.

```
number <- 5
if (number > 4) {
    print('number was greater than 4!')
}
```

```
## [1] "number was greater than 4!"
```

`else` statements are executed if a statement is not met.

```
number <- 3
if (number > 4) {
    print('number was greater than 4!')
} else {
    print('number was not greater than 4')
}
```

```
## [1] "number was not greater than 4"
```

We can add an `else if` statemet to check a second condition.

```
number <- 3
if (number > 4) {
    print('number was greater than 4!')
} else if (number == 4) {
    print('number was equal to 4')
} else {
    print('number was not greater than 4')
}
```

```
## [1] "number was not greater than 4"
```

These else/if statements are useful when writing functions.

Running functions on stuff besides dataframes

The `dplyr` package is very cool, but what if we want to perform analyses on stuff besides dataframes (like vectors and matrices!)? We'll need to use the `purrr` package (also bundled with `tidyverse`).

Though we could use a for loop like this, there is a more efficient way of doing things:

```
for (i in 1:10) {
    print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

`purrr` provides a set of functions to provide map/reduce-style functionality. "Map" means to apply a function to every piece of a dataset. "Reduce" means to calculate some kind of summary statistic on a large group of data.

Let's demonstrate with several examples using `map()`. The function requires two arguments, something to iterate over, and a function to apply to each piece.

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------------------------------------

## filter(): dplyr, stats
## lag():    dplyr, stats

# iterate over 1:10, apply sqrt() to each number
mapped <- map(1:10, sqrt)
mapped

## [[1]]
## [1] 1
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
##
## [[5]]
## [1] 2.236068
##
## [[6]]
## [1] 2.44949
##
## [[7]]
## [1] 2.645751
##
## [[8]]
## [1] 2.828427
##
## [[9]]
## [1] 3
##
## [[10]]
## [1] 3.162278
```

Notice how we get back a weird datastructure as a result. `map` returns a list by default. Lists are a special datastructure that can contain any type or size of element.

```
example_list <- list(1, "a", c(5, 9 , 10), TRUE)
```

One note on indexing lists: to extract individual elements we need to use two square brackets (`[[]]`) instead of one (`[]`). Using single brackets just returns a one-element list.

```
example_list[1]

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

example_list[[1]]

## [1] 1
```

Exercise - Retrieving nested values:

Retrieve the 10 in `example_list` via indexing. You'll need to use multiple sets of square brackets (and index twice)

The advantage of using `map()` is that it can operate on lists as if it were a vector, which is normally hard to do. In this case, we'll retrieve what type of data is contained in each element. Just as with other tidyverse functions, we can still use the pipe as well!

```
example_list %>% map(class)
```

```
## [[1]]
## [1] "numeric"
##
## [[2]]
## [1] "character"
##
## [[3]]
## [1] "numeric"
##
## [[4]]
## [1] "logical"
```

However, we usually don't want a list back as output. There are a number of extra map functions that let us specify what type of output we want.

```
# return a numerical vector
1:10 %>% map_dbl(sqrt)
```

```
##  [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
##  [8] 2.828427 3.000000 3.162278
```

```
# return a character vector
1:10 %>% map_chr(sqrt)
```

```
##  [1] "1.000000" "1.414214" "1.732051" "2.000000" "2.236068" "2.449490"
##  [7] "2.645751" "2.828427" "3.000000" "3.162278"
```

Note that `purrr`'s map functions will complain if you ask to get output where type coercion would result in loss of data. This is a useful feature called type-safety.

```
1:10 %>% map_int(sqrt)
```

```
Error: Can't coerce element 1 from a double to a integer
Execution halted
```

That error prevents this kind of stuff from happening (without you knowing about it!).

```
as.integer(sqrt(7))
```

```
## [1] 2
```

Anonymous (lambda) functions

Sometimes it can be a bit of a pain to define a function only to use it once. In these scenarios, we can use what's called an anonymous function - we never give the function a name, using it immediately.

Here is an example. In this case we want to get the standard error of the mean (SEM), but it is not already defined in R. We'll use an anonymous function to define our SEM function in-line.

```
data <- list(1:4, 10:8, 50:60)
data %>% map(function(var) {
    return(sd(var) / sqrt(length(var)))
})
```

```
## [[1]]
## [1] 0.6454972
##
## [[2]]
## [1] 0.5773503
```

```
##
## [[3]]
## [1] 1
```

`purrr` also provides a shortcut to define an anonymous function. We can use `~` to replace the `function` keyword, and `.x` to replace the variable name. Using this shortcut might look like this:

```
data %>% map(~sd(.x) / sqrt(length(.x)))
```

```
## [[1]]
## [1] 0.6454972
##
## [[2]]
## [1] 0.5773503
##
## [[3]]
## [1] 1
```

Exercise - Iterating through rows of a dataframe:

The `starwars` dataset has a list of Star Wars characters. However, some of the columns are a little funny - they are lists!

```
class(starwars$films)
```

```
## [1] "list"
```

Use `map_int()` and an anonymous function to determin how many films each character appeared in (from the `starwars$films` column).

Operating on matrices

When it comes to working with rows/columns of matrices, the base R function `apply()` function is still superior to everything else.

```
mat <- matrix(1:50, nrow=5, ncol=10)
# operate on rows with margin=1
apply(mat, 1, sum)
```

```
## [1] 235 245 255 265 275
```

```
# operate on columns with margin=2
apply(mat, 2, sum)
```

```
##  [1]  15  40  65  90 115 140 165 190 215 240
```

Next section