Classwork 13A: Writing Your Own Functions in R

In this classwork, you'll learn how to write your own functions in R. So far, we've learned about a lot of great functions that make data analysis easy. For example, all these are functions: `sum()`, `mean()`, `ggplot()`, `lm()`, `filter()`, `summarize()`.

Functions take arguments as inputs, perform an action in the body, and produce some output at the end. If you're having trouble using a function, it's probably because there's something about the function's arguments, body, or output that you're not understanding. Reading the help docs can often give you the clarity you need about those things.

Run this code to get started:

```r
library(tidyverse)
library(gapminder)
```

This is the syntax to define a function in R:

```r
# my_function <- function(arg1, arg2, arg3) {
#     body
#     return(output)
# }
```

`my_function` is the name of the function. You can call your functions (almost) anything you'd like. Variable names can have letters, numbers, periods, and underscores. It's good practice to avoid using common names for your functions. That is, if you plan on using `dplyr::filter`, avoid creating functions called `filter` to avoid unnecessary confusion. And if you plan on using `c()`, don't name other things `c`.

`arg1`, `arg2`, and `arg3` are the function arguments. You can have any number of arguments, including zero.

`body` defines the action that the function takes.

`output` is the output of the function.

For example: this is a function that adds two numbers.

```r
plus <- function(a, b) {
  return(a + b)
}
```

1. Does `plus()` work on numbers? Fill in the blanks to make the code return TRUE.

```r
# plus(__, __) == 3
```

2. Does `plus()` work on vectors?

```r
# plus(__, __) == c(4, 0, 1)
```

3. Write a function that takes one argument as the input and returns its square.

So for example, 'square(6)' will return 36. You can name the argument anything you want (x, y, arg1, etc), as long as you're consistent.

```r
# square <- function(__) {
#   __
# }
```

4. Verify that `square()` works on numbers.

```r
# square(__) == 16
```

5. Verify that `square()` works on vectors.

```r
# square(__) == c(1, 4, 9)
```

6. Verify that `square()` works on variables inside of tibbles using `mutate`.

Take `gapminder` and add a new variable `gdpPercap_squared` that is the square of the gdpPercap variable, using your function `square()`.

```r
# __
```

If you aren't explicit about what you want the output of your function to be by using a `return()` statement, R will automatically return the last thing it evaluated inside the function body. So our `plus()` function could also be defined as:

```r
plus <- function(a, b) {
  a + b
}

plus(2, 3)
```

```
## [1] 5
```

Note that the function below will also output a + b, because a + b was the last thing the function evaluated:

```r
plus <- function(a, b) {
  a - b
  a + b
}

plus(2, 3)
```

```
## [1] 5
```

So in order to get your function to output multiple values, you need to output something like a vector or tibble.

```r
plus_minus <- function(a, b) {
  c(a + b, a - b)
}

plus_minus(5, 2)
```

```
## [1] 7 3
```

7. Your turn: write a function called `generate_weekdays` that takes zero arguments and returns all the days of the week: "Monday", "Tuesday", etc.

Make sure to call the function by running `generate_weekdays()` to verify that it outputs what you're expecting.

```
# generate_weekdays <- function() {
#   __
# }

# generate_weekdays()
```

Classwork 13B: Ramsey RESET Test as a Function in R

In this classwork, you'll conduct the Ramsey RESET test for functional misspecification for our `students` dataset and a model that's linear in variables. Then you'll write a function `ramsey` which will be able to take *any* model and dataset and run the Ramsey RESET test on it.

Run this code to get started:

```r
library(tidyverse)
library(gapminder)
```

```r
students <- read_csv("https://raw.githubusercontent.com/cobriant/students_dataset/main/students.csv")
```

1. Take `students` and estimate the model $final\_grade = \beta_0 + \beta_1 sex + \beta_3 failures + \beta_4 romantic + \beta_6 absences + u$. Interpret the estimates for the coefficients.

2. Using `ggplot` and intuition, which explanatory variables might have a nonlinear relationship with `final_grade`? That is, do you think it makes sense to include any squared terms or interactions in the model above?

3. In your own words, explain how the Ramsey RESET test works for functional misspecification.

4. Take `students` and conduct the first step in the Ramsey RESET test: take the linear model from question 1 and add a column `yhat` to `students` that is the fitted values from that regression.

```r
# students %>%
#   mutate(yhat = __)
```

5. Take your answer to question 4 and use it to estimate the new model: $final\_grade = \beta_0 + \beta_1 sex + \beta_2 failures + \beta_3 romantic + \beta_4 absences + \beta_5 yhat^2 + u$. Conduct a hypothesis test on β_5 by looking at the p-value from `broom::tidy()`. What are the results from the Ramsey test?

```r
# students %>%
#   mutate(yhat = __) %>%
#   lm(__ + I(yhat^2), data = .) %>%
#   broom::tidy() %>%
#   slice_tail(n = 1) %>%
#   select(p.value) %>%
#   mutate(nonlinear_detected = __ < .05)
```

6. Use your code from question 5 to write a function to conduct the Ramsey test on any model and dataset.

```
# ramsey <- function(dataset, linear_model) {
#   __ %>%
#     mutate(yhat = __) %>%
#     lm(paste0(__, " + I(yhat^2)"), data = .) %>%
#     broom::tidy() %>%
#     slice_tail(n = 1) %>%
#     select(p.value) %>%
#     mutate(nonlinear_detected = __ < .05)
# }
```

Make sure you can call your function like this:

```
# students %>%
#   ramsey("final_grade ~ sex + failures + romantic + absences")
```

And also like this:

```
# gapminder %>%
#   ramsey("lifeExp ~ gdpPercap + year")
```

Classwork 14A: Intro to `map(.x, .f)`

In this classwork, I'll introduce you to the function `map()`. It will be useful to run monte-carlo experiments starting in part B of this classwork.

```r
library(tidyverse)
```

Vectorized Functions

In CW13A, we learned that we could define our own custom functions. Here I define a function called `pct_change` that takes two arguments: an "old" value and a "new" value, and it returns the percentage change between them.

```r
pct_change <- function(old, new) {
  (new - old) / old
}
```

1. Verify that `pct_change` works on values.

```r
# pct_change(__, __) == 1
```

2. Verify that `pct_change` works on vectors.

```r
# pct_change(__, __) == c(0, 1, 2)
```

How is it that `pct_change` works on vectors? Well, `pct_change` is defined only using subtraction and division, which both work element-wise on vectors. So since you can plug vectors directly into the computation that's done in the function body without an issue, `pct_change` works on vectors without an issue:

```r
(c(4, 5, 6) - c(1, 2, 3)) / c(1, 2, 3)
```

```
## [1] 3.0 1.5 1.0
```

Lots of the functions we've been using are "vectorized" (they work on vectors): `sum()`, `mean()`, `max()`, etc. But some functions may not work on vectors so smoothly, or in the way you want. For example: consider `rnorm()`: it generates `n` random numbers from a normal distribution with any mean and standard deviation (by default, mean 0 and sd 1).

```r
# ?rnorm
```

3. Use `rnorm()` to generate 10 random numbers from the normal distribution with mean 0 and sd 1, that is, N(0, 1).

```r
# rnorm(__, __, __)
```

Suppose you need to generate 10 random numbers from N(0, 1), then 10 random numbers from N(0, 2), then 10 random numbers from N(0, 3), then 10 random numbers from N(0, 4), all the way up to 10 random numbers from N(0, 100).

As a first attempt, try putting the vector 1:100 into the `sd` argument:

```r
rnorm(n = 10, mean = 0, sd = 1:100)
```

```
##  [1]  0.3336260  0.7557768 -5.2914044  6.7720983 -2.0948916 -4.2542528
##  [7] -0.1438212 13.9721384  2.0626713 10.2940250
```

What happens? R generates one random number from N(0, 1), one random number from N(0, 2), one random number from N(0, 3), all the way up to one random number from N(0, 10). It stops at 10 because `n = 10`. `rnorm` is vectorized, but not in the way that helps us solve this problem.

Second attempt: you could copy-paste 100 times, changing the sd each time:

```r
rnorm(n = 10, mean = 0, sd = 1)
rnorm(n = 10, mean = 0, sd = 2)
rnorm(n = 10, mean = 0, sd = 3)
rnorm(n = 10, mean = 0, sd = 4)
#etc.
```

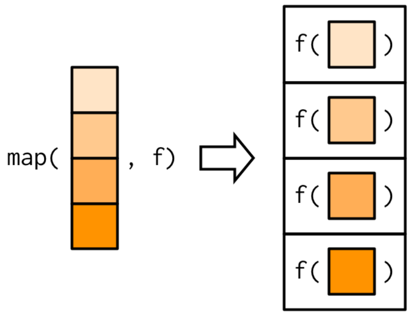
But this method is annoying for you to do, it contains a lot of extra code for a reader to have to read, and it doesn't use the computer as the powerful tool it is. And if you make a mistake with all that copy-pasting, it's hard to catch.

A better solution: use `map(.x, .f)`.

`map(.x, .f)` is from the package `purrr`, which is the last tidyverse package we'll talk about in this course. `purrr` is named the way it is because it helps "make your functions purrr" like a well oiled machine (helps make your functions vectorized in the way you want). And that's exactly what we needed help with in the example above: `rnorm()` wasn't purrring for us!

`map(.x, .f)` does one simple thing: It applies the function `.f` to each element of the vector `.x`.

`map(.x, .f)` will return a list of the same length as the inputs `.x`. Check out the diagram that explains



visually how to use `map`:

The name `map` refers to how "map" is used in math: a mapping is an operation that associates each element of a set with elements in a different set. The `.x` are the inputs, and the `.f` is the function to apply to get to the set of outputs.

So how do we use `map(.x, .f)` to solve our problem?

Write out the beginning of the copy-paste solution:

```r
rnorm(n = 10, mean = 0, sd = 1)
rnorm(n = 10, mean = 0, sd = 2)
rnorm(n = 10, mean = 0, sd = 3)
```

What should be the vector of inputs `.x`? It's whatever needs to **change** in the copy-paste version. In that code, everything stays the same except for the `sd` needs to change: it needs to go from 1 to 100. So `.x` should be the vector `1:100`.

What's the function `.f` we'll apply to every element of `.x`? It's a function that should take a standard deviation and output 10 random normal numbers with a mean of zero and that custom standard deviation:

```r
# function(standard_dev) {
#   rnorm(n = 10, mean = 0, sd = standard_dev)
# }
```

```r
map(
  .x = 1:100,
  .f = function(standard_dev) {
    rnorm(n = 10, mean = 0, sd = standard_dev)
  }
)
```

4. Use `map(.x, .f)` to generate 10 random normals with a mean of 1, 10 random normals with a mean of 2, 10 random normals with a mean of 3, all the way up to 10 random normals with a mean of 100.

They should all have a standard deviation of 1.

```r
# map(
#   .x = __,
#   .f = __
# )
```

One more example: if you wanted to take a vector and multiply each element by 3, this is the best way:

```r
3 * 1:10
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30
```

But for the sake of practice, in the next problem, you'll use `map(.x, .f)` to do the same task.

5. Use `map(.x, .f)` to take the vector 1:10 and apply a function which multiplies each element by 3 in order to get the vector (3, 6, 9, 12, . . .).

Hint: the copy-paste version would be:

```r
3 * 1
3 * 2
3 * 3
# etc
```

What changes? That's what needs to go in your `.x`. Your `.f` should be a function that takes an element of `.x` and multiplies that element by 3.

```r
# map(
#   .x = __,
#   .f = function(__) {
#     __
#   }
# )
```

Classwork 14b: Ramsey RESET Simulation with `map(.x, .f)`

In CW13b, you wrote a function to take any model and dataset and to conduct the Ramsey RESET test for functional misspecification on it. In this classwork, you will use that function to run an experiment to learn a little more about when you can expect the test to work well and when you might not.

```r
library(tidyverse)
library(gapminder)
students <- read_csv("https://raw.githubusercontent.com/cobriant/students_dataset/main/students.csv")
```

1. Define your function `ramsey`.

```r
# ramsey <- function(__) {
#   __
# }
```

2. Generate a (fake) dataset with 3 variables: `x`, `z`, and `y` and 100 observations.

`x` and `z` should be independent random variables that are random uniform between 0 and 10. `y` should be `1 + .4 * x + .3 * z - .04 * x^2 - .04 * z^2 + .03 * x * z + u`, where `u` is N(0, 1). Don't give the tibble a name. Since it has random elements, each time you run the code, the values will be different. That's the behavior we want.

```r
#?runif

# tibble(
#   x = runif(__),
#   z = runif(__),
#   y = 1 + .4 * x + .3 * z - .04 * x^2 - .04 * z^2 + .03 * x * z + rnorm(n = 100)
# )
```

3. Take your tibble from question 2 and visualize a scatterplot of `x` and `y`.

Also estimate the misspecified model `y ~ x + z`. Running the code multiple times, does it seem like the coefficients on x and z are unbiased?

```r
# tibble(
#   x = runif(__),
#   z = runif(__),
#   y = 1 + .4 * x + .3 * z - .04 * x^2 - .04 * z^2 + .03 * x * z + rnorm(n = 100)
# ) %>%
#   __
```

4. Use your function `ramsey` to conduct the Ramsey RESET test on your tibble.

Running the code several times, does it seem to successfully detect the presence of possible nonlinear relationships most of the time?

```r
# tibble(
#   x = __,
#   z = __,
#   y = __ + rnorm(n = 100)
# ) %>%
#   ramsey("y ~ x + z")
```

In statistics, a type 1 error is a false positive: if you reject the null when it's actually true in the population, you have a type 1 error. A type 2 error is a false negative: if you fail to reject the null when it's actually false, you have a type 2 error.

5. If `ramsey` outputs a p.value greater than .05 in the previous question, has a type 1 error occurred, has a type 2 error occurred, or has no error occurred?

6. Next, we'll use `map(.x, .f)` to do a simulation where we run the code in question 4 100 times and count the number of times the Ramsey test rejects the null hypothesis.

We want to use map to run the function `.f` 100 times and output the results of each iteration as a tibble, so we'll use `map_dfr` (dfr stands for data frame in rows). `.x` will be the vector `1:100`, just making sure we run the function 100 times. `.f` should begin with `function(...)`: the dot-dot-dot lets that function accept the element of `.x` as an argument, but it throws it away instead of being unpacked and used inside the function body.

```
# map_dfr(
#   1:100,
#   function(...) {
#     tibble(
#       x = __,
#       z = __,
#       y = __ + rnorm(n = 100)
#     ) %>%
#       ramsey("y ~ x + z")
#   }
# ) %>%
#   count(__)
```

7. Experiment with the data generating process to explore when you can expect the Ramsey test to be more or less reliable.

Is the Ramsey test more or less reliable for smaller sample sizes? Is the Ramsey test more or less reliable when var(u) is larger? Create an example where the Ramsey test works at least 90% of the time. ## Extra Credit: instead of using `count()` at the end of the experiment, plot p-values from each iteration using geom_density. Is the Ramsey test more or less reliable for smaller sample sizes? Compare multiple sample sizes in one plot and use `fill = n`. Draw a vertical line for the p-value of .05.