Coding Lab: Functions

Ari Anisfeld

Fall 2020

Functions

```
# example of a function
circle_area <- function(r) {

   pi * r ^ 2

  }
```

- What are functions and why do we want to use them?
- How do we write functions in practice?
- What are some solutions to avoid frustrating code?

Motivation

"You should consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)"

- *Hadley Wickham, R for Data Science*

Instead of repeating code . . .

```
data %>%
  mutate(a = (a - min(a)) / (max(a) - min(a)),
         b = (b - min(b)) / (max(b) - min(b)),
         c = (c - min(c)) / (max(c) - min(c)),
         d = (d - min(d)) / (max(d) - min(d)))
```

```
## # A tibble: 100 x 4
##          a     b     c     d
##      <dbl> <dbl> <dbl> <dbl>
##  1 0.0924 0.155 0.531 0.172
##  2 0.732  0.400 0.688 0.835
##  3 0.493  0.797 0.146 0.479
##  4 0.478  0.556 0.631 0.244
##  5 0.514  0.423 0.523 0.534
##  6 0.681  0.766 0.295 0.663
##  7 0.835  0.366 0.840 0.225
##  8 0.370  0.180 0.607 0.462
##  9 0.227  0.636 0.938 0.237
## 10 0.845  0.488 0.981 0.626
```

Write a function

```
rescale_01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}



data %>%
  mutate(a = rescale_01(a),
         b = rescale_01(b),
         c = rescale_01(c),
         d = rescale_01(d))
```

```
## # A tibble: 100 x 4
##          a     b     c     d
##      <dbl> <dbl> <dbl> <dbl>
##  1 0.0924 0.155 0.531 0.172
##  2 0.732  0.400 0.688 0.835
##  3 0.493  0.797 0.146 0.479
##  4 0.478  0.556 0.631 0.244
##  5 0.514  0.423 0.523 0.534
```

Function anatomy

The anatomy of a function is as follows:

```
function_name <- function(arguments) {
  do_this(arguments)
}
```

A function consists of

1. Function arguments[1]
2. Function body

We can assign the function to a name like any other object in R.

[1] Tech detail: R refers to these as `formals`.

Function anatomy: example

- **arguments**: `x`
- **body**: `(x - min(x)) / (max(x) - min(x))`
- assign to **name**: `rescale_01`

```
rescale_01 <- function(x) {
  (x - min(x)) / (max(x) - min(x))
}
```

Note that we don't need to explicitly call `return()`

- the last line of the code will be the value returned by the function.

Writing a function: printing output

You start writing code to say Hello to all of your friends.

- You notice it's getting repetitive. . . . time for a function

```
print("Hello Jasmin!")
```

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

```
print("Hello Joan!")
```

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

```
print("Hello Andrew!")
```

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

```
# and so on...
```

Writing a function: parameterize the code

Start with the **body**.

Ask: What part of the code is changing?

- Make this an **argument**

Writing a function: parameterize the code

Start with the **body**.

Rewrite the code to accommodate the parameterization

```
# print("Hello Jasmin!") becomes ...

name <- "Jasmin"

print(paste0("Hello ", name, "!"))
```

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

Check several potential inputs to avoid future headaches

Writing a function: add the structure

```r
# name <- "Jasmin"
# print(paste0("Hello ", name, "!"))

function(name) {
  print(paste0("Hello ", name, "!"))
}
```

Writing a function: assign to a name

Try to use **names** that actively tell the user what the code does

- We recommend verb_thing()
  - **good** `calc_size()` or `compare_prices()`
  - **bad** `prices()`, `calc()`, or `fun1()`.

```
# name <- "Jasmin"
# print(paste0("Hello ", name, "!"))

say_hello_to <- function(name) {
  print(paste0("Hello ", name, "!"))
}
```

Simple example: printing output

Test out different inputs!

```
say_hello_to("Jasmin")
```

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

```
say_hello_to("Joan")
```

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

```
say_hello_to(name = "Andrew")
```

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

```
# Cool this function is vectorized!
say_hello_to(c("Jasmin", "Joan", "Andrew"))
```

```
## [1] "Hello Jasmin!" "Hello Joan!"   "Hello Andrew!"
```

Question: does `name` exist in my R environment after I run this function? Why or why not?

Technical aside: `typeof(your_function)`

Like other R objects functions have types.

Primative functions are of type "builtin"

```
typeof(`+`)
```

```
## [1] "builtin"
```

```
typeof(sum)
```

```
## [1] "builtin"
```

Technical aside: `typeof(your_function)`

Like other R objects functions have types.

User defined functions, functions loaded with packages and many base R functions are type "closure":

```
typeof(say_hello_to)
```

```
## [1] "closure"
```

```
typeof(mean)
```

```
## [1] "closure"
```

Technical aside: `typeof(your_function)`

This is background knowledge that might help you understand an error.

For example, you thought you assigned a number to the name "c" and want to calculate ratio.

```
ratio <- 1 / c
```

```
Error in 1/c : non-numeric argument to binary operator
```

```
as.integer(c)
```

```
Error in as.integer(c) :
  cannot coerce type 'builtin' to vector of type 'integer'
```

"builtin" or "closure" in this situation let you know your working with a function!

Second example: calculating the mean of a sample

Your stats prof asks you to simulate a central limit theorem, by calculating the mean of samples from the standard normal distribution with increasing sample sizes.

```
mean(rnorm(1))
```

```
## [1] 0.09486633
```

```
mean(rnorm(3))
```

```
## [1] -0.3880646
```

```
mean(rnorm(30))
```

```
## [1] 0.1469901
```

```
# et cetera
```

Second example: calculating the mean of a sample

The number is changing, so it becomes the **argument**.

```
calc_sample_mean <- function(sample_size) {

      mean(rnorm(sample_size))

  }
```

- The number is the sample size, so I call it `sample_size`. `n` would also be appropriate.
- The **body** code is otherwise identical to the code you already wrote.

Second example: calculating the mean of a sample

For added clarity you can unnest your code and assign the intermediate results to meaningful names.

```
calc_sample_mean <- function(sample_size) {

  random_sample <- rnorm(sample_size)

  sample_mean <- mean(random_sample)

  return(sample_mean)
  }
```

`return()` explicitly tells R what the function will return.

- The last line of code run is returned by default.

Second example: calculating the mean of a sample

If the function can be fit on one line, then you can write it without the curly brackets like so:

```
calc_sample_mean <- function(n) mean(rnorm(n))
```

Some settings call for anonymous functions, where the function has no name.

```
function(n) mean(rnorm(n))
```

```
## function(n) mean(rnorm(n))
```

Always test your code

Try to foresee the kind of input you expect to use.

```
calc_sample_mean(1)
```

```
## [1] -0.7721886
```

```
calc_sample_mean(1000)
```

```
## [1] -0.007513187
```

We see below that this function is not vectorized. We might hope to get 3 sample means out but only get 1

```
# read ?rnorm to understand how rnorm
# inteprets vector input.
calc_sample_mean(c(1, 3, 30))
```

```
## [1] -0.3832335
```

Adding additional arguments

If we want to be able to adjust the details of how our function runs we can add arguments

- typically, we put "data" arguments first
- and then "detail" arguments after

```
calc_sample_mean <- function(sample_size,
                                        our_mean,
                                        our_sd) {

  sample  <- rnorm(sample_size,
                   mean = our_mean,
                   sd = our_sd)

  mean(sample)
}
```

Setting defaults

We usually set default values for "detail" arguments.

```
calc_sample_mean <- function(sample_size,
                                          our_mean = 0,
                                          our_sd = 1) {

  sample  <- rnorm(sample_size,
                   mean = our_mean,
                   sd = our_sd)

  mean(sample)
}
```

```
# uses the defults
calc_sample_mean(sample_size = 10)
```

```
## [1] -0.2030251
```

Setting defaults

```
# we can change one or two defaults.
# You can refer by name, or use position
calc_sample_mean(10, our_sd = 2)
```

```
## [1] -0.0548051
```

```
calc_sample_mean(10, our_mean = 6)
```

```
## [1] 6.348498
```

```
calc_sample_mean(10, 6, 2)
```

```
## [1] 5.870006
```

Setting defaults

This won't work though:

```
calc_sample_mean(our_mean = 5)
```

```
Error in rnorm(sample_size, mean = our_mean, sd = our_sd) :
  argument "sample_size" is missing, with no default
```

Key points

- Write functions when you are using a set of operations repeatedly
- Functions consist of arguments and a body and are usually assigned to a name.
- Functions are for humans
  - pick names for the function and arguments that are clear and consistent
- Debug your code as much as you can as you write it.
  - if you want to use your code with `mutate()` test the code with vectors

For more: See Functions Chapter in R for Data Science

Additional material

Functions in functions

We can pass functions as arguments to other functions. Before:

```
calc_sample_mean <- function(sample_size,
                                        our_mean = 0,
                                        our_sd = 1) {
  sample_mean <- mean(rnorm(sample_size,
                            mean = our_mean,
                            sd = our_sd))

  sample_mean
}
```

Functions in functions

We can pass functions as arguments to other functions. After:

```
summarize_sample <- function(sample_size,
                                        our_mean = 0,
                                        our_sd = 1,
                                        summary_fxn = mean) {
  summary_stat <- summary_fxn(rnorm(sample_size,
                                    mean = our_mean,
                                    sd = our_sd))

  summary_stat
}
```

Functions in functions

```
calc_sample_mean(sample_size = 10,
                 our_mean = 0,
                 our_sd = 1)
```

```
## [1] -0.2553635
```

```
summarize_sample(sample_size = 10,
                 our_mean = 0,
                 our_sd = 1,
                 summary_fxn = max)
```

```
## [1] 1.744373
```

`calc_sample_mean()` is now probably the wrong name for this function - we should call it `summarize_sample()` or something like that.

Detour: probability distributions

R has built-in functions for working with distributions.

	example	what it does?
d	`dnorm(x)`	returns pdf value at x
p	`pnorm(q)`	returns CDF value at q
q	`qnorm(p)`	returns inverse CDF (the quantile) for a given probability
r	`rnorm(n)`	generates a random sample of size n

Probability distributions you are familiar with are likely built-in to R.

For example, the binomial distribution has `dbinom()`, `pbinom()`, `qbinom()`, `rbinom()`. The t distribution has `dt()`, `pt()`, `qt()`, `rt()`, etc.

Read this tutorial for more information.

Detour: probability distributions

- `dnorm()`: density function, the PDF evaluated at X.

```
dnorm(0)
```

```
## [1] 0.3989423
```

```
dnorm(1)
```

```
## [1] 0.2419707
```

```
dnorm(-1)
```

```
## [1] 0.2419707
```

Detour: probability distributions

Detour: probability distributions

- `pnorm()`: cumulative distribution function, the CDF evaluated at X.

```
pnorm(0)
```

```
## [1] 0.5
```

```
pnorm(1)
```

```
## [1] 0.8413447
```

```
pnorm(-1)
```

```
## [1] 0.1586553
```

Detour: probability distributions

Detour: probability distributions

- `qnorm()`: quantile function, the inverse CDF evaluated at a quantile.

```
qnorm(c(0.05, 0.95))
```

```
## [1] -1.644854  1.644854
```

```
qnorm(c(0.025, 0.975))
```

```
## [1] -1.959964  1.959964
```

```
pnorm(qnorm(c(0.025, 0.975)))
```

```
## [1] 0.025 0.975
```

Detour: probability distributions

- `rnorm()`: random sampling

```
rnorm(1)
```

```
## [1] 0.7244536
```

```
rnorm(5)
```

```
## [1] -1.4225062 -0.5960598  0.3353115  0.1752422 -1.2480
```

```
rnorm(30)
```

```
##  [1]  0.7412269020  0.2417359429 -0.1755121315 -0.2699
## [6] -0.3039516771 -0.7319315023 -0.9072302122 -0.1074
## [11]  0.6332928702  0.0033477103 -0.5941185630  0.4490
## [16]  0.0151888586 -0.9255065013  0.4645399559  1.0179
## [21]  0.3655853978  0.9052002039 -3.0692947194 -1.1294
## [26]  0.8857969353  0.0054698593  0.4577974162  0.8974
```