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 0.833  0.246 0.328 0.455
##  2 0.211  0.393 0.470 0.539
##  3 0.315  0.593 0.235 0.472
##  4 0.424  0.257 0.607 0.364
##  5 0.638  0.411 0.407 0.209
##  6 0.336  0.265 0.285 0.633
##  7 0.773  0.400 0.500 0.730
##  8 0.0770 0.531 0.167 0.563
##  9 0.464  0.352 0.768 0.528
## 10 0.455  0.629 0.547 0.287
```

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 0.833  0.246 0.328 0.455
##   2 0.211  0.393 0.470 0.539
##   3 0.315  0.593 0.235 0.472
##   4 0.424  0.257 0.607 0.364
##   5 0.638  0.411 0.407 0.209
```

Function anatomy

The anatomy of a function is as follows:

```r
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`

```r
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

```r
# 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!

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

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

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

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

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

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

```r
# 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.

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

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

```r
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.

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

```
## [1] 0.9743667
```

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

```
## [1] -0.6290661
```

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

```
## [1] -0.009555868
```

```r
# et cetera
```

Second example: calculating the mean of a sample

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

```r
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.

```r
calc_sample_mean(1)
```

```
## [1] 0.04058937
```

```r
calc_sample_mean(1000)
```

```
## [1] -0.03409345
```

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

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

```
## [1] -0.2300791
```

How to deal with unvectorized functions

If we don't want to change our function, but we want to use it to deal with vectors, then we have a couple options: Here we are going to use the function rowwise

```
#creating a vector to test our function
sample_tibble <- tibble(sample_sizes = c(1, 3, 10, 30))

#using rowwise groups the data by row, allowing calc_sampl
sample_tibble %>%
  rowwise() %>%
  mutate(sample_means = calc_sample_mean(sample_sizes))
```

```
## # A tibble: 4 x 2
## # Rowwise:
##   sample_sizes sample_means
##          <dbl>        <dbl>
## 1            1      -1.54
## 2            3      -0.251
## 3           10       0.0151
```

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

```r
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.

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

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

  mean(sample)
}
```

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

```
## [1] 0.080253
```

Setting defaults

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

```
## [1] -1.317715
```

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

```
## [1] 5.818235
```

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

```
## [1] 5.577494
```

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

Probability distributions

R has built-in functions for working with distributions.

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

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 examples.

We should be familar with r functions

- rnorm(): random sampling

```
rnorm(1)
```

```
## [1] 0.1669768
```

```
rnorm(5)
```

```
## [1] -0.1132515 -1.8828934 -0.2025573 -0.1816280 -0.53511
```

```
rnorm(30)
```

```
##  [1]  1.1588340 -1.1655278  2.2723098 -0.2096508 -0.5072
##  [7]  0.1494255  0.9268971  0.6766631 -0.4712107  0.8556
## [13] -0.6097615  0.2945506  1.3716269  1.8340736 -1.4239
## [19]  1.9016704 -0.2865639 -0.1807973 -1.0397804 -1.0332
## [25] -0.9445835 -0.5347266 -0.2358799  0.1373871 -1.4559
```

What are p and q?

pnorm returns the probability we observe a value less than or equal to some value q.

```r
pnorm(1.96)
```

```
## [1] 0.9750021
```

```r
pnorm(0)
```

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

qnorm returns the inverse of pnorm. Plug in the probability and get the cutoff.

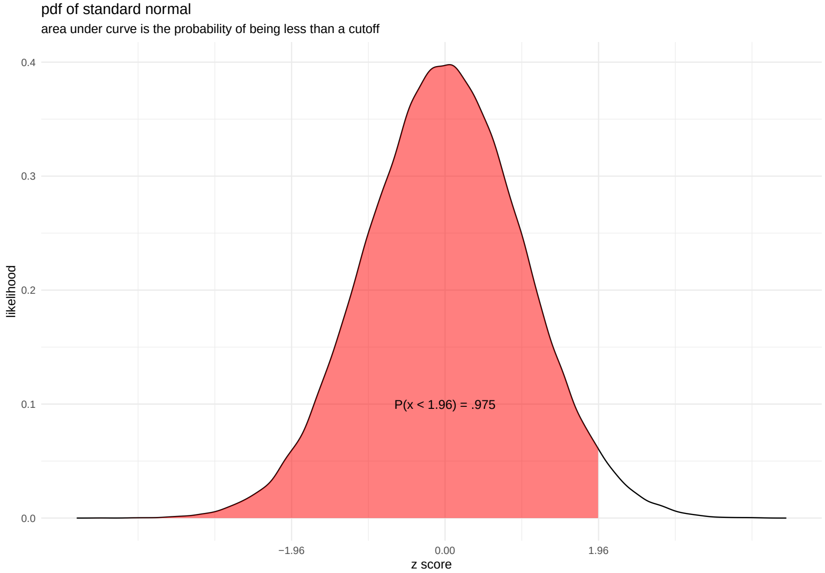
```r
qnorm(.975)
```

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

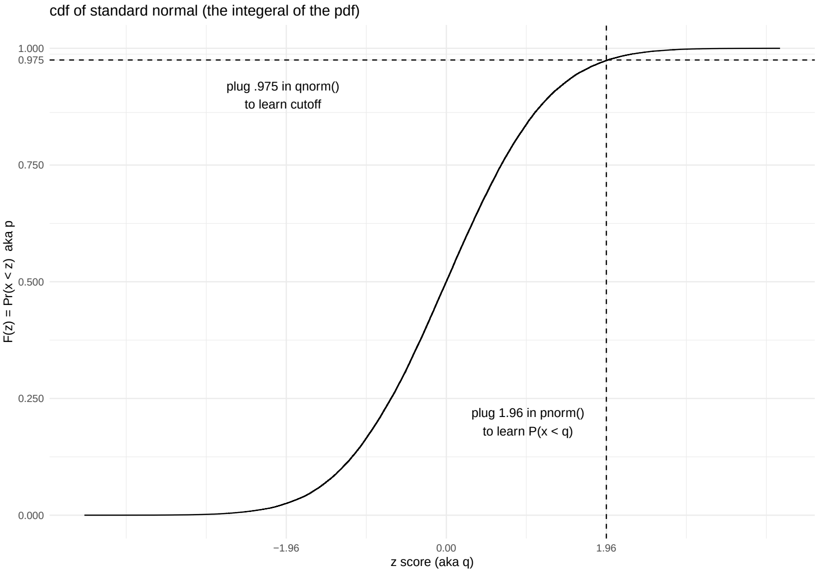
```r
qnorm(.5)
```

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

What are p and q?



What are p and q?



What is d?

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

```r
dnorm(0)
```

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

```r
dnorm(1)
```

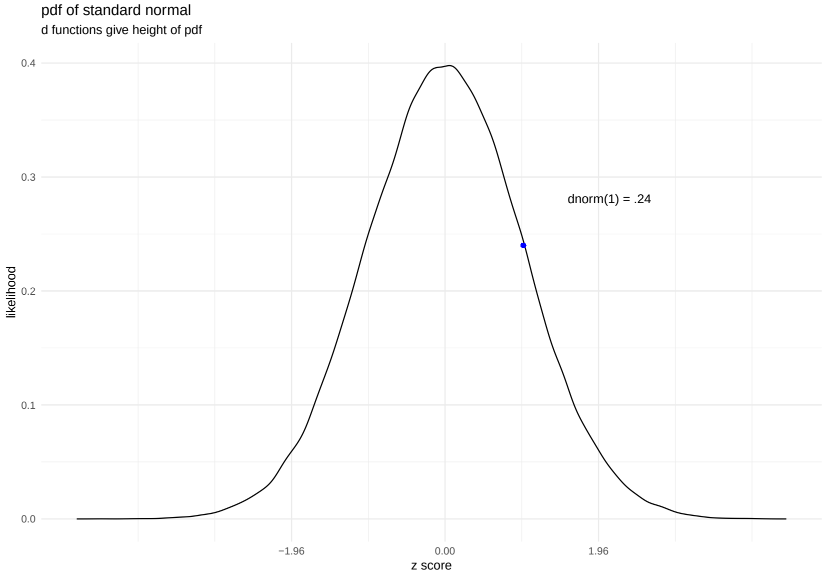
```
## [1] 0.2419707
```

```r
dnorm(-1)
```

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

What is d?

`dnorm` gives the height of the distribution function. Sometimes this is called a likelihood.



Functions in functions

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

```r
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

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

```
## [1] -0.1303855
```

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

```
## [1] 1.19347
```

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