Accelerated Lecture 7: Writing Functions

Harris Coding Camp

Summer 2022

Functions

```r
# 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

- Grolemund and Wickham chapter 19:
  - "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)."
- Darin Christenson refers to the programming mantra **DRY**
  - "Do not Repeat Yourself (DRY)"
  - "Functions enable you to perform multiple tasks (that are similar to one another) without copying the same code over and over"

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 0.366  0.557 0.0804 0.324
##  2 0.0913 0.865 0.589  0.381
##  3 0.678  0.171 0.436  0.472
##  4 0.235  0.179 0.520  0
##  5 0.574  0.391 0.699  0.504
##  6 0.624  0.559 0.525  0.497
##  7 0.148  0.686 0.0764 0.280
##  8 0.119  0.818 0.240  0.491
##  9 0.583  0.565 0.0382 0.376
## 10 0.726  0.373 0.333  0.0405
## # ... with 90 more rows
```

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 0.366  0.557 0.0804 0.324
##  2 0.0913 0.865 0.589  0.381
##  3 0.678  0.171 0.436  0.472
##  4 0.235  0.179 0.520  0
##  5 0.574  0.391 0.699  0.504
##  6 0.624  0.559 0.525  0.497
##  7 0.148  0.686 0.0764 0.280
##  8 0.119  0.818 0.240  0.491
##  9 0.583  0.565 0.0382 0.376
```

Function anatomy

The anatomy of a function is as follows:

```r
function_name <- function(argument_1, argument_2) {
  do_this(argument_1, argument_2)
}
```

Three components of a function:

1. **function name**
   - specify function name before the assignment operator `<-`
2. **function arguments** (sometimes called “inputs” or “arguments”)
   - Inputs that the function takes
     - can be vectors, data frames, strings, etc.
   - In above hypothetical code, the function took two inputs `argument_1`,`argument_2`
   - In “function call,” you specify values to assign to these function arguments
3. **function body**
   - What the function does to the inputs

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))
}
```

The last line of the code will be the value returned by the function!

- We do not explicitly call `return()`

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**.

Q: What part of the code is changing? Or what aspects of the code do you want to change?

- Make this an **argument**

Writing a function: parameterize the code

Start with the **body**.

(Re)write 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

Now let's add the **structure** to formally define the new function:

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

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

- **arguments**: `name`
- **body**: `print(paste0("Hello ", name, "!"))`
- assign to **name**: not yet…

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()`.

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

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

- **arguments**: `name`
- **body**: `print(paste0("Hello ", name, "!"))`
- assign to **name**: `say_hello_to`

First example: printing output

Test out different inputs!

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

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

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

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

```
say_hello_to("Andrew")
```

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

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

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

Second example: calculating the mean of a sample

Your stats prof asks you to simulate the law of large numbers, by calculating the mean of i.i.d. samples with increasing sample sizes.[1]

[1] You will learn about this in Stats I

Recall `rnorm(n)` it generates a random sample of size n

We want to: calculate the mean of i.i.d. samples with increasing sample sizes.

```
rnorm(1)
```

```
## [1] -1.479624
```

```
rnorm(5)
```

```
## [1]  0.11986690 -2.00903745  0.06560656  0.07299062  1.62590634
```

```
rnorm(30)
```

```
##  [1] -0.91400914  0.47681536 -0.77609866  1.28553245 -0.11925843 -0.
##  [7] -0.60135499  0.25866997  0.28448158 -0.68979493 -1.02385411 -1.
## [13]  0.10809473 -2.54813543  0.08043742 -0.36098802  1.19379476 -2.
## [19]  1.25423888  0.81454710  0.74803835 -0.51388576  0.64985375  0.
## [25]  2.04689252 -0.40550303  1.07816616 -0.04297627 -1.60562631 -0.
```

Too much copy paste

We want to: calculate the *mean* of i.i.d. samples with increasing sample sizes.

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

```
## [1] -1.128639
```

```
mean(rnorm(5))
```

```
## [1] 0.2547068
```

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

```
## [1] 0.04544075
```

```
# et cetera
```

Second example: calculating the mean of a sample

The sample size is changing, so it becomes the **argument**:

```
calc_sample_mean <- function(sample_size) {
  mean(rnorm(sample_size))
}
```

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

Function at work

We want to: calculate the *mean* of i.i.d. samples with increasing sample sizes.

- This seems like the same amount of copy paste as before. (We'll improve this soon.)
- We're still better off. If we change our mind about something or find a bug, we only have to fix it once!

```r
calc_sample_mean(1)
```

```
## [1] 0.7133265
```

```r
calc_sample_mean(5)
```

```
## [1] -0.6764853
```

```r
calc_sample_mean(30)
```

```
## [1] -0.004813473
```

Commenting functions with clear names

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

```r
calc_sample_mean <- function(sample_size) {

  our_sample <- rnorm(sample_size)
  sample_mean <- mean(our_sample) # <- probably overkill

  sample_mean

  }
```

Using `return()`

The last line of code run is returned by default.

- Occassionally you'll want to specify what to return

```
calc_sample_mean <- function(sample_size) {
  return(mean(rnorm(sample_size)))
}
```

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

- Style guide says only use `return()` to break out of a function early.

Capturing assigned output

If the last line is an assignment there is no *visible* output

- Avoid this.

```r
calc_sample_mean <- function(sample_size) {
  sample_mean <- mean(rnorm(sample_size))
}

# looks like nothing happened
calc_sample_mean(1)
# but we can capture the output with an assignment
x <- calc_sample_mean(1e6)
x
```

```
## [1] 0.0008858516
```

One-liners and anonymous functions

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

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

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

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

Always test your code

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

```r
calc_sample_mean(0)
```

```
## [1] NaN
```

```r
calc_sample_mean(1e6)
```

```
## [1] 0.0005480699
```

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

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

```
## [1] 0.02079634
```

How to deal with unvectorized functions

If we don't want to change our function, but we want to use on vectors, then we have a couple options

- Here we are going to use the function `rowwise()`:

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

# rowwise groups the data by row
# then our function is applied to each "group"
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      1.60
## 2            3      0.114
## 3           10     -0.449
## 4           30      0.00551
```

Complicating the matter.

The Stats professor now calls for different paramaterizations of the normal distribution.

- They want you to re-run your analysis with normals with different means and variances!

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

If there's a "natural" default, we can 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 defaults
calc_sample_mean(sample_size = 10)
```

```
## [1] -0.09742854
```

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] 1.392886
```

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

```
## [1] 6.11375
```

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

```
## [1] 4.886894
```

Setting defaults

This won't work though:

- the most important argument is missing!

```
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
```

Another complication

Now your curious about extremes. What happens to the max as we increase our sample size?

Functions as arguments to other functions

Before

```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)
}
```

Functions as arguments to other functions

After

```
summarize_sample <- function(sample_size,
                             our_mean = 0,
                             our_sd = 1,
                             summary_func = mean) {

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

  summary_func(sample)
}
```

Use a new descriptive name: `summarize_sample()`

- `calc_sample_mean()` would be a misleading name

Functions in functions, in action

```r
# mean(rnorm(10, mean = 0, sd = 1))
summarize_sample(sample_size = 10) # mean is default!
```

```
## [1] -0.1459511
```

```r
# max(rnorm(10, mean = 0, sd = 1))
summarize_sample(sample_size = 10, summary_func = max)
```

```
## [1] 0.8476351
```

```r
# min(rnorm(10, mean = 0, sd = 1))
summarize_sample(sample_size = 10, summary_func = min)
```

```
## [1] -1.85666
```

You try

1. What will happen if we execute the code below? Explain?

```
new.function <- function(a, b) {
   print(a)
   print(b)
}

new.function(6)
```

You try

2. Write a function that takes a vector and replaces negative values with `NA`. (Some data use -99 or similar numbers to represent missing-ness.)

```
replace_neg(c(-19, 1, 2, 1))
```

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

3. Write examples where you use `replace_neg()` with columns in a tibble.
4. Does your function work on non-numeric inputs?

Aside: Wait . . . what?

```
# +a < 0  => TRUE
# -b < 0  => TRUE
# c < 0   => FALSE
# d < 0   => FALSE

replace_neg(c("+a", "-b", "c", "d"))
```

```
## [1] NA  NA  "c" "d"
```

Conditional operators work with characters!

- They test alphabetical order
- And treat special chars and punctuation as negative!
- So, in a real sense, you are greater than `R`

```
"you" > "R"
```

```
## [1] TRUE
```

Review: Conditional execution

`if` statements allow you to conditionally execute certain blocks of code depending on whether a condition is satisfied

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Suppose you don't like this behavior

We can test whether the input is numeric.

```r
replace_neg <- function(x) {
  if (is.numeric(x)) {
    return(ifelse(x < 0, NA, x))
  }
}

out <- replace_neg(c("-a", "b"))
out
```

```
## NULL
```

```r
replace_neg(c(-3, 4))
```

```
## [1] NA  4
```

Test the function in the context you are interested in!

```r
# oops ... we probably don't want to lose char!
tibble(char = c("-a", "+b", "c"),
       num = c(1, 2, -3 )) %>%
  mutate(char = replace_neg(char))
```

```
## # A tibble: 3 x 1
##     num
##   <dbl>
## 1     1
## 2     2
## 3    -3
```

Fixing the bug

```r
replace_neg <- function(x) {
  if (!is.numeric(x)) {
      return(x)
  }
  ifelse(x < 0, NA, x)
}
```

```r
# better!
tibble(char = c("-a", "+b", "c"),
       num  = c(1, 2, -3)) %>%
  mutate(char = replace_neg(char),
         num  = replace_neg(num),)
```

```
## # A tibble: 3 x 2
##   char    num
##   <chr> <dbl>
## 1 -a        1
## 2 +b        2
```

Helping a user out.

We may want to warn the user that they did something funny.

```
replace_neg <- function(x) {
  if (!is.numeric(x)) {
    print("Non-numeric input to replace_neg returning x")
    return(x)
  }

  ifelse(x < 0, NA, x)
}

replace_neg(c("-a"))
```

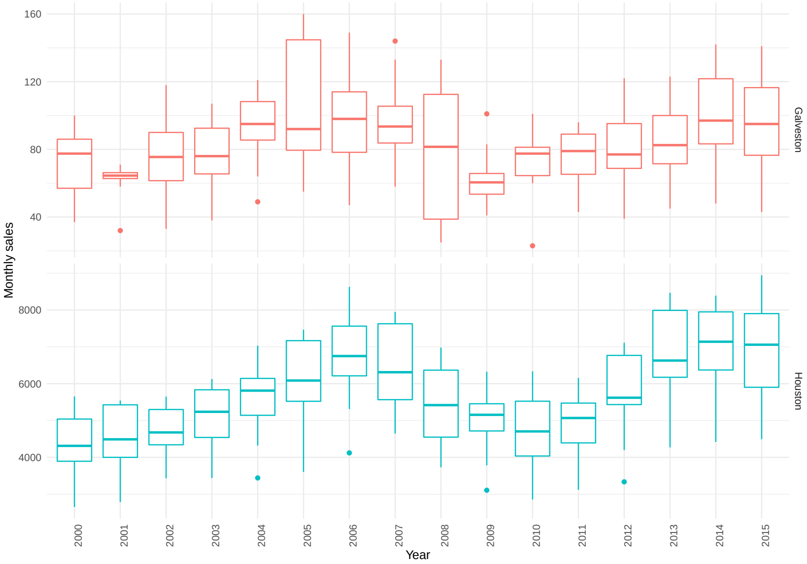
```
## [1] "Non-numeric input to replace_neg returning x"
```

```
## [1] "-a"
```

- NB: `message()` makes messages and `stop()` throws errors.

Using functions for data visualization

You're exploring data and want to make this plot for a variety of city pairings.



Using functions for data visualization

Suppose we want another set of plots with Austin and San Antonio.

- Copy-paste adjust and then … you decide you want to tinker with the plot more! Now you have to do it twice.

```
txhousing %>%
  filter(city == "Houston" | city == "Galveston")) %>%
  ggplot(aes(x = as_factor(year),
             y = sales,
             color = city)) +
  geom_boxplot(show.legend = FALSE) +
  labs(color = NULL,  y = "Monthly sales", x = "Year") +
  theme_minimal() +
  facet_grid(city~., scales = "free_y") +
  theme(axis.text.x = element_text(angle = 90))
```

How would you parameterize this to take arbitrary cities?

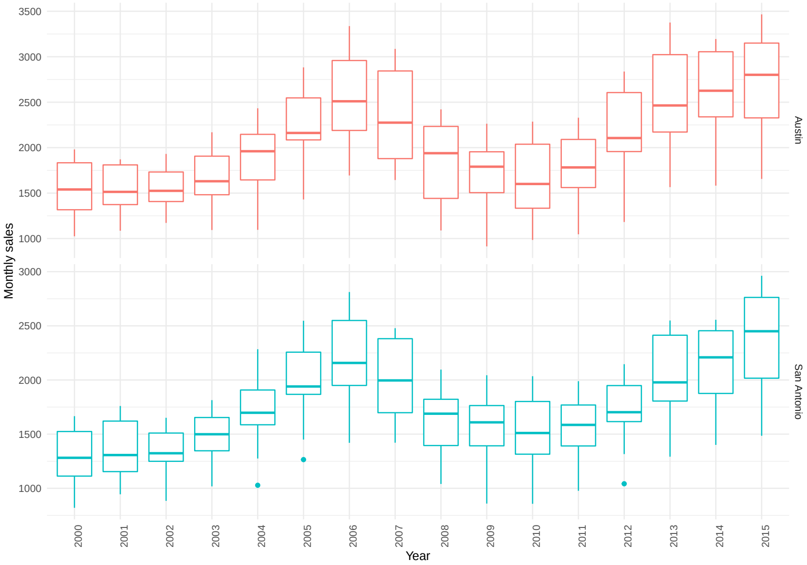
```
# zooming in
txhousing %>%
  filter(city == "Houston" | city == "Galveston") ...
```

A function for our plot

```r
sales_box_plot <- function(cities) {
  txhousing %>%
    filter(city %in% cities) %>%
    ggplot(aes(x = as_factor(year),
               y = sales,
               color = city)) +
    geom_boxplot(show.legend = FALSE) +
    labs(color = NULL,  y = "Monthly sales", x = "Year") +
    theme_minimal() +
    facet_grid(city~., scales = "free_y") +
    theme(axis.text.x = element_text(angle = 90))
}
```

Voila!

```
sales_box_plot(c("Austin", "San Antonio"))
```



Suppose you want y too . . .

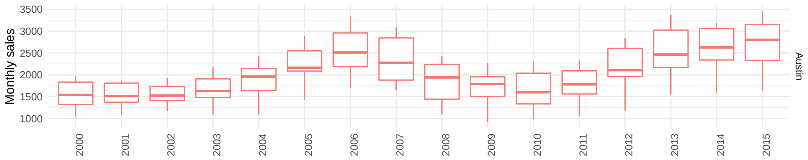
{{...}} embrace arguments that refer to columns in your data set.

- This is “sugar” that tells R to look within the data for the variable.

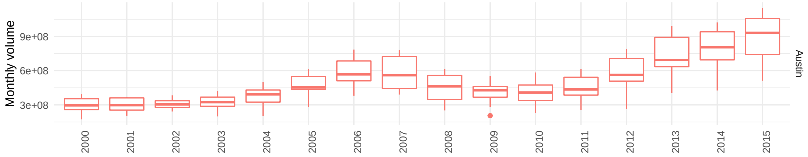
```
housing_box_plot <- function(cities, y, ylab) {
  txhousing %>%
    filter(city %in% cities) %>%
    ggplot(aes(x = as_factor(year),
               y = {{ y }},
               color = city)) +
    geom_boxplot(show.legend = FALSE) +
    labs(color = NULL,  y = ylab, x = NULL) +
    theme_minimal() +
    facet_grid(city~., scales = "free_y") +
    theme(axis.text.x = element_text(angle = 90))
}
```

Voila

```r
housing_box_plot(c("Austin"), sales, "Monthly sales")
```



```r
housing_box_plot(c("Austin"), volume, "Monthly volume")
```



Complicating `calc_sample_mean` even further.

Now you're getting really curious and want to see if these ideas hold with different distributions!

Recall:

```
calc_sample_mean
```

```
## function(sample_size, our_mean = 0, our_sd = 1) {
##
##   sample <- rnorm(sample_size,
##                   mean = our_mean,
##                   sd = our_sd)
##   mean(sample)
## }
```

One approach – make new functions for each distribution

```r
calc_sample_mean_t <- function(sample_size, our_df) {
  sample <- rt(sample_size, our_df)
  mean(sample)
}

calc_sample_mean_chisq <- function(sample_size, our_df) {
  sample <- rchisq(sample_size, our_df)
  mean(sample)
}

# Fun fact: 2^31 -1 is the largest seed in R
set.seed(2147483647)
calc_sample_mean_t(10,  our_df = 5)
```

```
## [1] -0.3284298
```

```r
calc_sample_mean_chisq(10, our_df = 5)
```

```
## [1] 6.111243
```

A sophisticated approach - parameterize the distribution!

- The complication here is each distribution has it's own parameters. `df`, `mean` etc.
- `...` takes arbitrary arguments which you can pass to another function
- Warning: `...` (dot-dot-dot) is a bit challenging to use

```r
calc_sample_mean <- function(sample_size, fn = rnorm, ...) {

  sample <- fn(sample_size, ...)

  mean(sample)
}



set.seed(2147483647)
calc_sample_mean(10, rt, df = 5)
```

```
## [1] -0.3284298
```

```r
calc_sample_mean(10, rchisq, df = 5)
```

```
## [1] 6.111243
```

More examples

... takes arbitrary named arguments which you can pass to another function

```r
# function(sample_size, fn = rnorm, ...)
# sample <- rnorm(10)
calc_sample_mean(10)
```

```
## [1] 0.125801
```

```r
# sample <- rf(4, df1 = 2, df2 = 3)
calc_sample_mean(4, rf, df1 = 2, df2 = 3)
```

```
## [1] 2.897411
```

```r
# sample <- rbeta(9, shape1 = .3, shape2 = 5)
calc_sample_mean(9, rbeta, shape1 = .3, shape2 = 5)
```

```
## [1] 0.02183321
```

in context

```r
tibble(x = c(1, 10, 100, 1000, 1e5)) %>%
  rowwise() %>%
  mutate(normal = calc_sample_mean(x, mean = 4, sd = 6),
         uniform = calc_sample_mean(x, runif, min = 2, max = 6),
         poisson = calc_sample_mean(x, rpois, lambda = 4))
```

```
## # A tibble: 5 x 4
## # Rowwise:
##         x normal uniform poisson
##     <dbl>  <dbl>   <dbl>   <dbl>
## 1       1   6.33    5.90    2   
## 2      10   4.17    3.63    4.4 
## 3     100   3.86    4.16    4.44
## 4    1000   4.09    4.01    3.98
## 5  100000   4.01    4.00    4.00
```

Recap

- 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
- Introduced a few sophisticated ways to work with function arguments!
  - `{{col_name}}` to refer to column names in dplyr context
  - `...` to pass arbitrary arguments to functions.

For more: See Chapter 19 in R for Data Science

Next steps:

Lab:

Today: Writing functions (challenging lab!)

I can encapsulate code into functions, and debug and apply them!

Lecture:

Tomorrow: Loops and iteration.

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] 0.2970824
```

```
rnorm(5)
```

```
## [1] 0.2210584 0.7494140 1.2196237 0.1000417 0.2788310
```

```
rnorm(30)
```

```
##  [1]  0.53273286  0.28082583  0.55823521 -0.05265282 -1.00783232 -0.
##  [7] -0.28801243  1.32804275 -2.53235202 -0.94861018 -0.10179659 -0.
## [13] -1.22921792  0.60717112  0.58380644 -0.07838013 -0.87148769 -1.
## [19]  0.91961159 -1.97193268  0.27856219  0.42130357 -0.74036019  0.
## [25]  0.51806103 -0.44663177 -0.97866049  1.70682308  0.70779553  0.
```

What are p and q?

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

```
pnorm(1.96)
```

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

```
pnorm(0)
```

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

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

```
qnorm(.975)
```

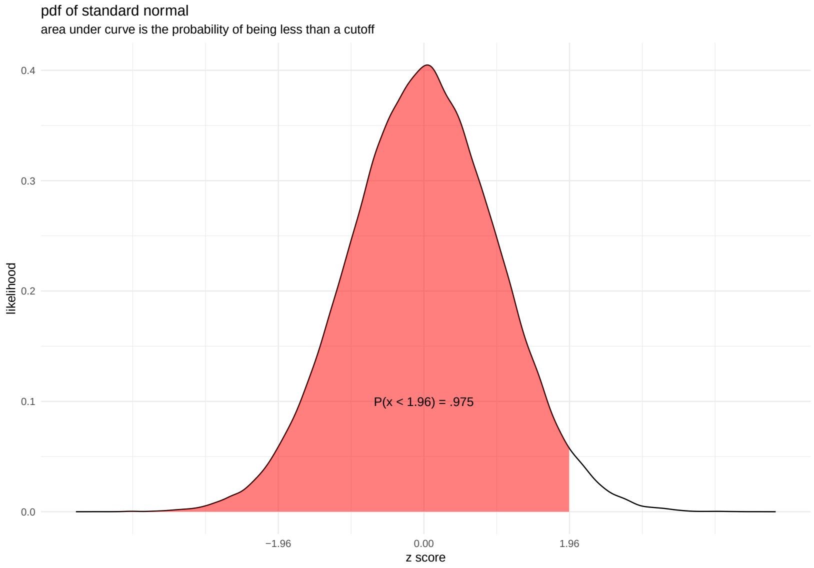
```
## [1] 1.959964
```

```
qnorm(.5)
```

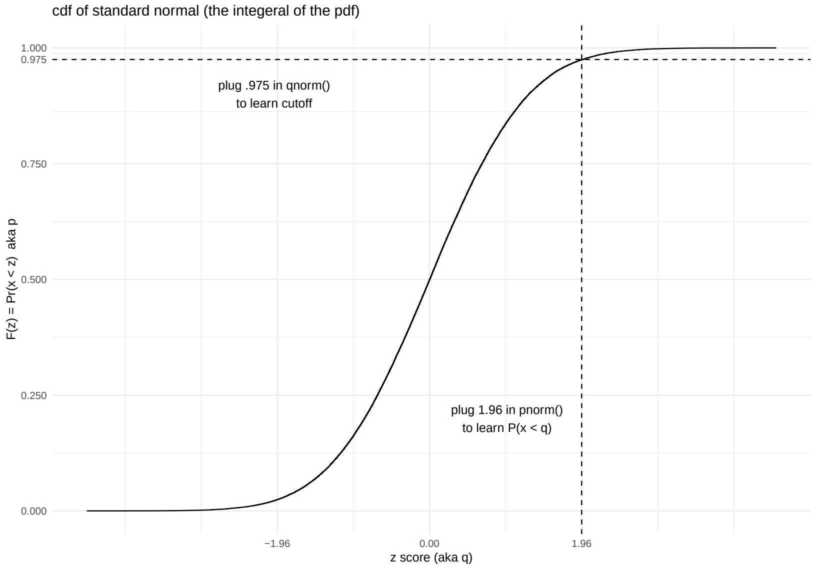
```
## [1] 0
```

This might be easier understood with pictures!

What are p and q?



What are p and q?



What is d?

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

```
dnorm(0)
```

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

```
dnorm(1)
```

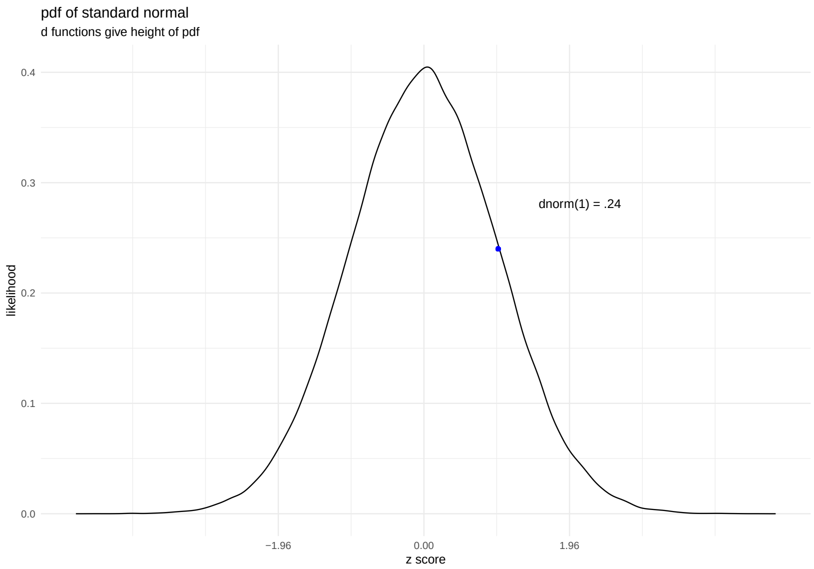
```
## [1] 0.2419707
```

```
dnorm(-1)
```

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

What is d?

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


pdf of standard normal
d functions give height of pdf
likelihood
dnorm(1) = .24
z score