Accelerated Lecture 7: Writing Functions

Harris Coding Camp

Summer 2022

Functions

```
# example of a function
circle_area <- function(r) {
   pi * r ^ 2
}
```

- Why do we want to write our own functions?
- What is function?
- How do we write functions in practice?

What is the code doing?

```
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(a)))
```

```
## # A tibble: 100 x 4
##         a     b     c     d
##     <dbl> <dbl> <dbl> <dbl>
##  1 0.539 0.977 0.449 1.21
##  2 0.580 0.308 0.425 0.905
##  3 0.755 0.479 0.732 0.489
##  4 0.501 0.510 0.632 1.60
##  5 0.867 0.494 0.610 1.15
##  6 0.574 0.778 0.327 1.23
##  7 0.240 0.711 0.934 0.557
##  8 0.807 0.541 0     0.789
##  9 0.869 0.481 0.489 0.530
## 10 0.596 0.436 0.242 1.18
## # ... with 90 more rows
```

What is the code doing?

```
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 0.539 0.977 0.449 0.582
##  2 0.580 0.308 0.425 0.434
##  3 0.755 0.479 0.732 0.234
##  4 0.501 0.510 0.632 0.765
##  5 0.867 0.494 0.610 0.553
##  6 0.574 0.778 0.327 0.588
##  7 0.240 0.711 0.934 0.267
##  8 0.807 0.541 0     0.378
##  9 0.869 0.481 0.489 0.254
```

Why write functions?

Functions

- encapsulate logic
- communicate what a chunk of code does

and help us

- re-use code we've put effort into
- avoid copy-paste headaches

Coder folk wisdom: **D**o not **R**epeat **Y**ourself

"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)."

- Grolemund and Wickham (chapter 26)

Function anatomy: name, arguments and body

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

# function call
function_name(x, y)
```

1. **function name**
   - specify function name with <-
2. **function arguments** (sometimes called "inputs")
   - can be any R object: vectors, data frames, lists, functions, etc.
3. **function body**
   - code that does stuff (typically with the inputs)

In "function call", we set `argument_1 = x` and `argument_2 = y`

- The function returns `do_this(x, y)`

Function anatomy: example

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

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

rescale_01(c(.1, 1.1, .6))
```

```
## [1] 0.0 1.0 0.5
```

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

- We do not explicitly call `return()`

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

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

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

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

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

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

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

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

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

What part of the code is changing?

Or what aspects of the code do you want to change?

```
print("Hello Jasmin!")
print("Hello Joan!")
print("Hello Andrew!")
# and so on...
```

- Make this an **argument**

Writing a function: parameterize the code

```
# 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:

```
# 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

Use **names** that actively tell the user what the code does

- We recommend `verb()` or `verb_thing()`
  - **good**: `filter()`, `rescale_01()` or `compare_prices()`
  - **bad**: `prices()`, `calc()`, or `fun1()`

```
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

You hear there's something called the law of large numbers.[1]

You're curious to see if it works through simulations.

So you start calculating the mean of i.i.d. samples with increasing sample sizes.

[1] You will learn about this in Stats I

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

```
rnorm(1, mean = 5, sd = 10)
```

```
## [1] 3.024819
```

```
rnorm(3, mean = 5, sd = 10)
```

```
## [1]  0.9497615 -3.1374033  6.0598231
```

```
rnorm(6, mean = 5, sd = 10)
```

```
## [1]   4.4095936  -0.2806806 -18.9663948 -13.4286629   5.1492043 -20.
```

You calculate the *mean* of i.i.d. samples with increasing sample sizes.

```
mean(rnorm(1, mean = 5, sd = 10))
```

```
## [1] 16.59023
```

```
mean(rnorm(3, mean = 5, sd = 10))
```

```
## [1] 5.065391
```

```
mean(rnorm(6, mean = 5, sd = 10))
```

```
## [1] 7.211749
```

```
# et cetera
```

Second example: calculating the mean of a sample

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

- Call it `n`.
  - You could call it anything ... `sample_size`, `jerry` etc.

The **body** is almost identical to the code you already wrote.

```
calc_sample_mean <- function(n) {
  mean(rnorm(n, mean = 5, sd = 10))
}
```

Commenting functions with clear names

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

```
calc_sample_mean <- function(sample_size) {

  our_sample <- rnorm(sample_size, mean = 5, sd = 10)
  sample_mean <- mean(our_sample)  # probably overkill

  sample_mean

}
```

The last line of code run is returned by default.

```
calc_sample_mean <- function(n) {
  our_sample <- rnorm(n, mean = 5, sd = 10)
  mean(our_sample)
}

set.seed(1)
calc_sample_mean(4)
```

```
## [1] 5.792104
```

You can specify what to `return()`

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

```
calc_sample_mean <- function(n) {
  our_sample <- rnorm(n, mean = 5, sd = 10)
  return(mean(our_sample))
}

set.seed(1)
calc_sample_mean(4)
```

```
## [1] 5.792104
```

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

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

- Avoid this.

```
calc_sample_mean <- function(n) {
  # last line of code is an assignment!
  sample_mean <- mean(rnorm(n, mean = 5, sd = 10))
}

# looks like nothing happened
calc_sample_mean(10)
# but we can capture the output with an assignment
x <- calc_sample_mean(10)
x
```

```
## [1] 8.209093
```

One-liners and anonymous functions

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

```
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, mean = 5, sd = 10))
```

Always test your code

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

```
calc_sample_mean(1)
```

```
## [1] 0.6198257
```

```
calc_sample_mean(3)
```

```
## [1] -0.5608922
```

```
calc_sample_mean(30)
```

```
## [1] 0.1162788
```

```
# what is the output if we plug in 0?
calc_sample_mean(0)
```

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

Try to foresee the kind of input you expect to use

```
calc_sample_mean(c(1, 3, 30))
```

```
## [1] 0.9454276
```

Hmmm . . . We hoped to get 3 sample means out but only got 1

Debug tool: Add temporary `print()` statements

```
calc_sample_mean <- function(n) {

  our_sample <- rnorm(n, mean = 5, sd = 10)
  print("our_sample:")
  print(our_sample)

  mean(our_sample)
}

set.seed(1)
calc_sample_mean(c(1, 3, 30))
```

```
## [1] "our_sample:"
## [1] -1.264538  6.836433 -3.356286

## [1] 0.7385363
```

Debug tool: Put suspicious code into the console

```
rnorm(c(1, 3, 30))
```

```
## [1]  1.5952808  0.3295078 -0.8204684
```

```
rnorm(c(-1, 0, 3, 12))
```

```
## [1]  0.4874291  0.7383247  0.5757814 -0.3053884
```

Uh-oh. `rnorm()` is not vectorized!

- "If length(n) $>$ 1, the length is taken to be the number required." see ?rnorm()

How to deal with unvectorized functions

If we want to vector input with length > 1

- Use loops (tomorrow)
- Use purrr::map() or apply() family of functions
- In a data context use `group_by(row_number())`

```
# create a tibble to test our function
sample_tibble <- tibble(sample_sizes = c(1, 3, 6))

# split the data by row
# apply the function to each row
# combine
sample_tibble |>
  group_by(row_number()) |>
  mutate(sample_means = calc_sample_mean(sample_sizes))
```

rowwise() is short-hand for group_by(row_number())

```
tibble(sample_sizes = c(1, 3, 6)) |>
  rowwise() |>
  mutate(sample_means = calc_sample_mean(sample_sizes))
```

```
## [1] "our_sample:"
## [1] 20.11781
## [1] "our_sample:"
## [1]   8.898432  -1.212406 -17.146999
## [1] "our_sample:"
## [1] 16.249309  4.550664  4.838097 14.438362 13.212212 10.939013
```

```
## # A tibble: 3 x 2
## # Rowwise:
##   sample_sizes sample_means
##          <dbl>        <dbl>
## 1            1         20.1
## 2            3         -3.15
## 3            6         10.7
```

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
}
```

Help a user out.

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

```
calc_sample_mean <- function(n) {

  if (length(n) > 1) {
    print("Warning: n should be length == 1")
  }


  our_sample <- rnorm(n, mean = 5, sd = 10)
  mean(our_sample)
}

calc_sample_mean(c(2,3,1))
```

```
## [1] "Warning: n should be length == 1"
```

```
## [1] 10.91893
```

- NB: `warning()` makes warnings and `stop()` throws errors.

Helping a user out.

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

```
calc_sample_mean <- function(n) {

  if (length(n) > 1) {
    stop("n must be length == 1")
  }

  our_sample <- rnorm(n, mean = 5, sd = 10)
  mean(our_sample)
}

calc_sample_mean(c(2,3,1))
```

Error in calc_sample_mean(c(2, 3, 1)) : n must be length == 1

- Can replace `if` with `stopifnot(length(n) ==. 1)`

Making functions work for you

1. Use clear names, even for objects inside the functions
2. Anticipate inputs and test them
3. Debug using print() statements and the console
4. Functions return the last line of code

- `return()` unnecessary
- avoid assignment <- in last line

Some data use -99 or -98 to represent missing-ness

1. Write a function that takes a vector and replaces negative values with `NA`.

```
replace_neg(c(-98, 1, 2, 1))
```

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

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

What if we don't always want `mean=5` and `sd=10`?

We can add additional arguments!

- typically, 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

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] -0.1468848
```

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] -0.03143666
```

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

```
## [1] 6.118949
```

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

```
## [1] 6.871897
```

Setting defaults

This won't work though:

- the most important argument is missing!

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

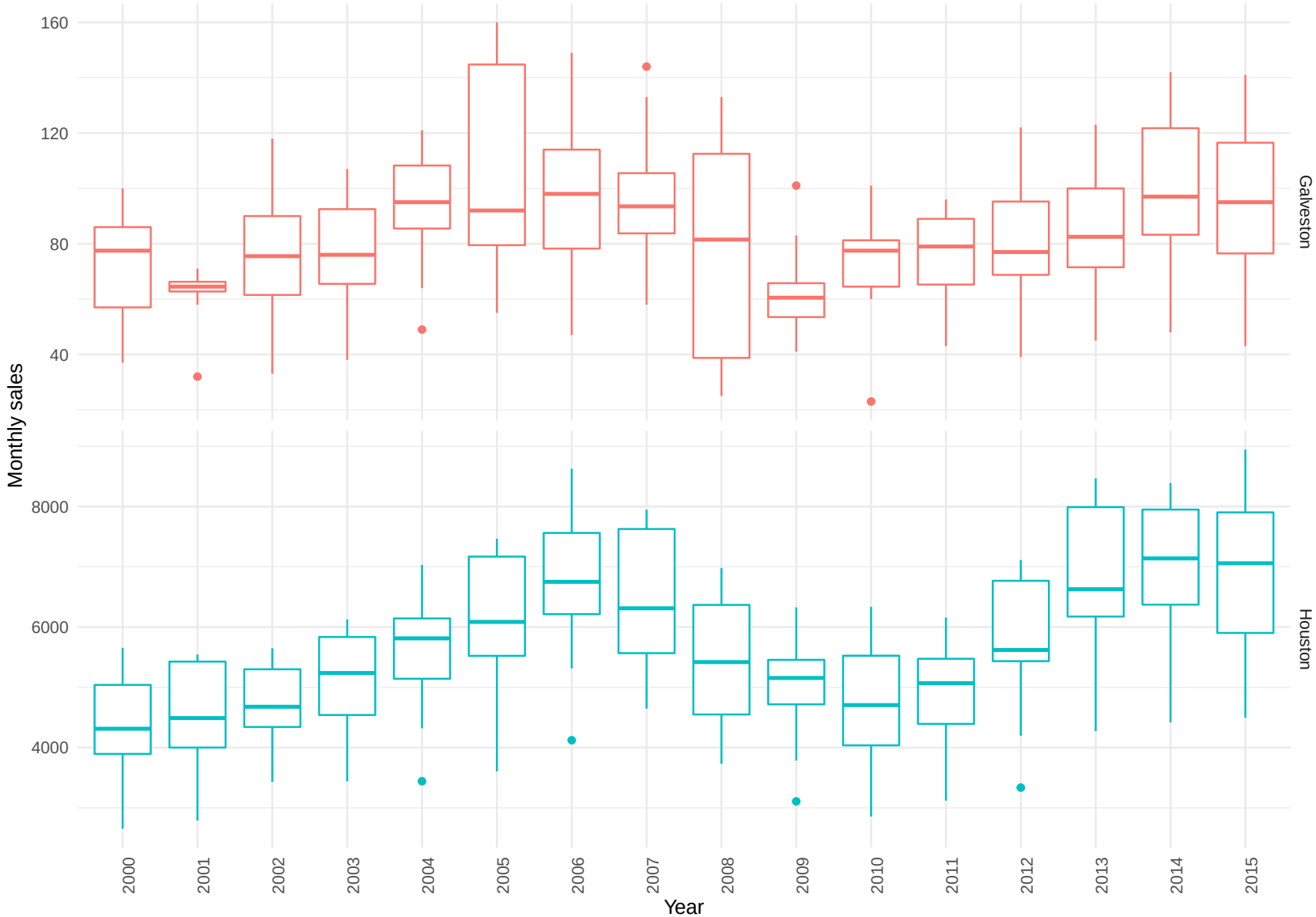
```
Error in rnorm(sample_size, mean = our_mean, sd = our_sd) :
  argument "sample_size" is missing, with no default
```

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

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

new.function(6)
```

Using functions for data visualization

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



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(vars(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") ...
```

How would you parameterize this to take arbitrary cities?

This code

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

becomes ...

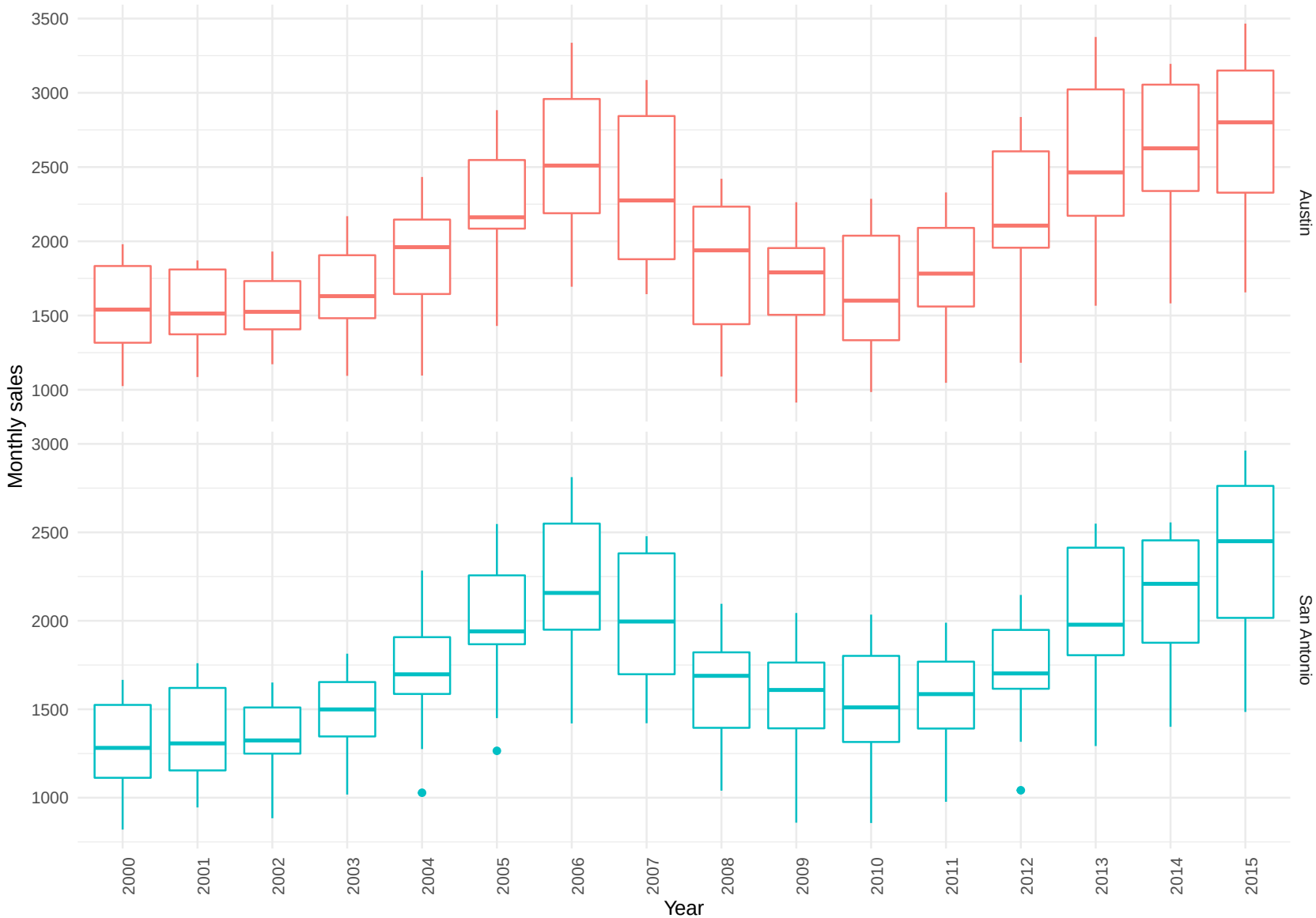
```
sales_box_plot <- function(city1, city2) {
  txhousing |>
     filter(city == city1 | city == city2) ...

# OR

sales_box_plot <- function(cities) {
  txhousing |>
    filter(city %in% cities) |> ...
```

A function for our plot

```
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(vars(city), scales = "free_y") +
    theme(axis.text.x = element_text(angle = 90))
}
```

Function magic!

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



Suppose you want y to change as well . . .

```
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 = "Year") +
    theme_minimal() +
    facet_grid(vars(city), scales = "free_y") +
    theme(axis.text.x = element_text(angle = 90))
}

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

```
Error in FUN(X[[i]], ...) : object 'sales' not found
```

The function doesn't know to look for `sales` in the data

Need to help R distinguish between an object in the global environment

```
sales <- some_object
```

and `sales` column in data

```
txhousing$sales
```

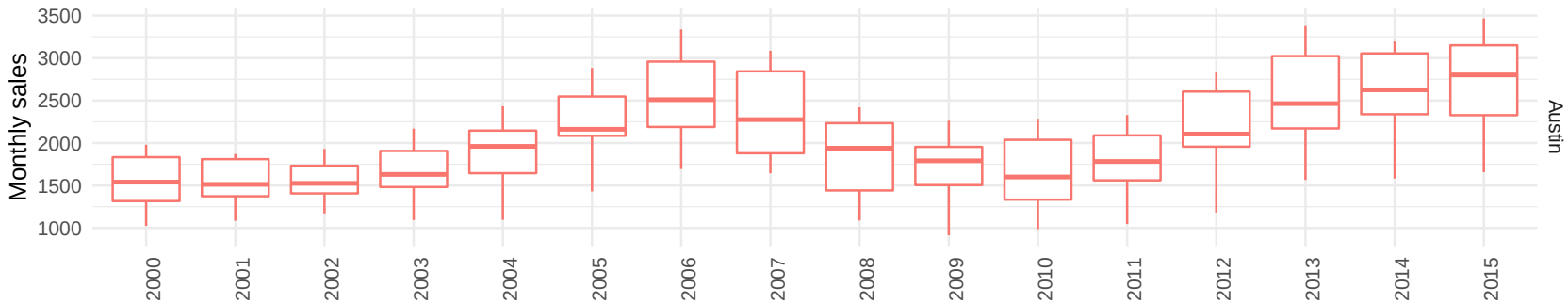
Use {{...}} to refer to columns names passed as function argument

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

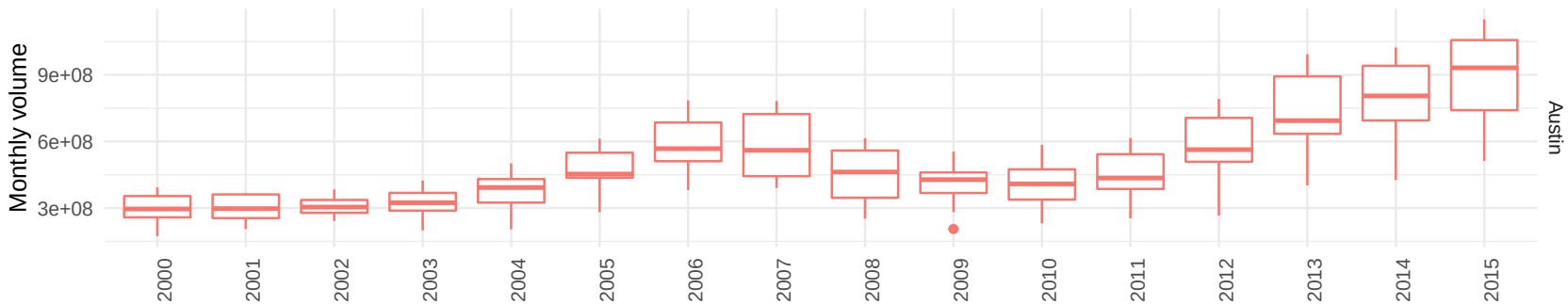
```
housing_box_plot <- function(cities, y, ylab) {
  txhousing |>
    filter(city %in% cities) |>
    ggplot(aes(x = as_factor(year),
               y = {{ y }},
               color = city)) +
    geom_boxplot(show.legend = FALSE) +
    labs(color = NULL,  y = ylab, x = NULL) +
    theme_minimal()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

With {{}}, R knows sales and volume are column names

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




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



Advanced: ... (dot-dot-dot) passes arbitrary inputs

- You've seen `...` as an argument in the help documents.

Advanced: ... (dot-dot-dot) passes arbitrary inputs to a function

...

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

calc_sample_mean(3, mean = 1)
```

```
## [1] 0.8245071
```

```
calc_sample_mean(3, 1)
```

```
## [1] 0.9379499
```

```
calc_sample_mean(3, mean = 1, sd = 3)
```

Compare the code

```r
# explicit --
# users know immediately know what your function takes
# and what the defaults are
calc_sample_mean <-
  function(n, mean = 0, sd = 1) {

  sample <- rnorm(n, mean = mean, sd = sd)

  mean(sample)
}

# implicit --
# user has to dig into rnorm
calc_sample_mean <-
  function(sample_size, ...) {
  sample <- rnorm(sample_size, ...)
  mean(sample)
}
```

We can recreate count() with ...

```
my_count <- function(data, ...) {

  # ...  Variables to group by

  data |>
    group_by(...) |>
    summarize(n = n()) |>
    ungroup()
}

my_count(midwest, inmetro)
```

```
## # A tibble: 2 x 2
##   inmetro     n
##     <int> <int>
## 1       0   287
## 2       1   150
```

Real code, more sophisticated but same idea!

Recap

- Write functions when you are using a set of operations repeatedly
- Functions consist of arguments and a body and usually 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 Functions Chapter 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.

Appendix

More on working with distributions

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] 0.6107264
```

```
rnorm(5)
```

```
## [1] -0.934097632 -1.253633400  0.291446236 -0.443291873  0.001105352
```

```
rnorm(30)
```

```
##  [1]  0.07434132 -0.58952095 -0.56866873 -0.13517862  1.17808700 -1.
##  [7]  0.59394619  0.33295037  1.06309984 -0.30418392  0.37001881  0.
## [13] -0.54252003  1.20786781  1.16040262  0.70021365  1.58683345  0.
## [19] -1.27659221 -0.57326541 -1.22461261 -0.47340064 -0.62036668  0.
## [25] -0.91092165  0.15802877 -0.65458464  1.76728727  0.71670748  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)
```

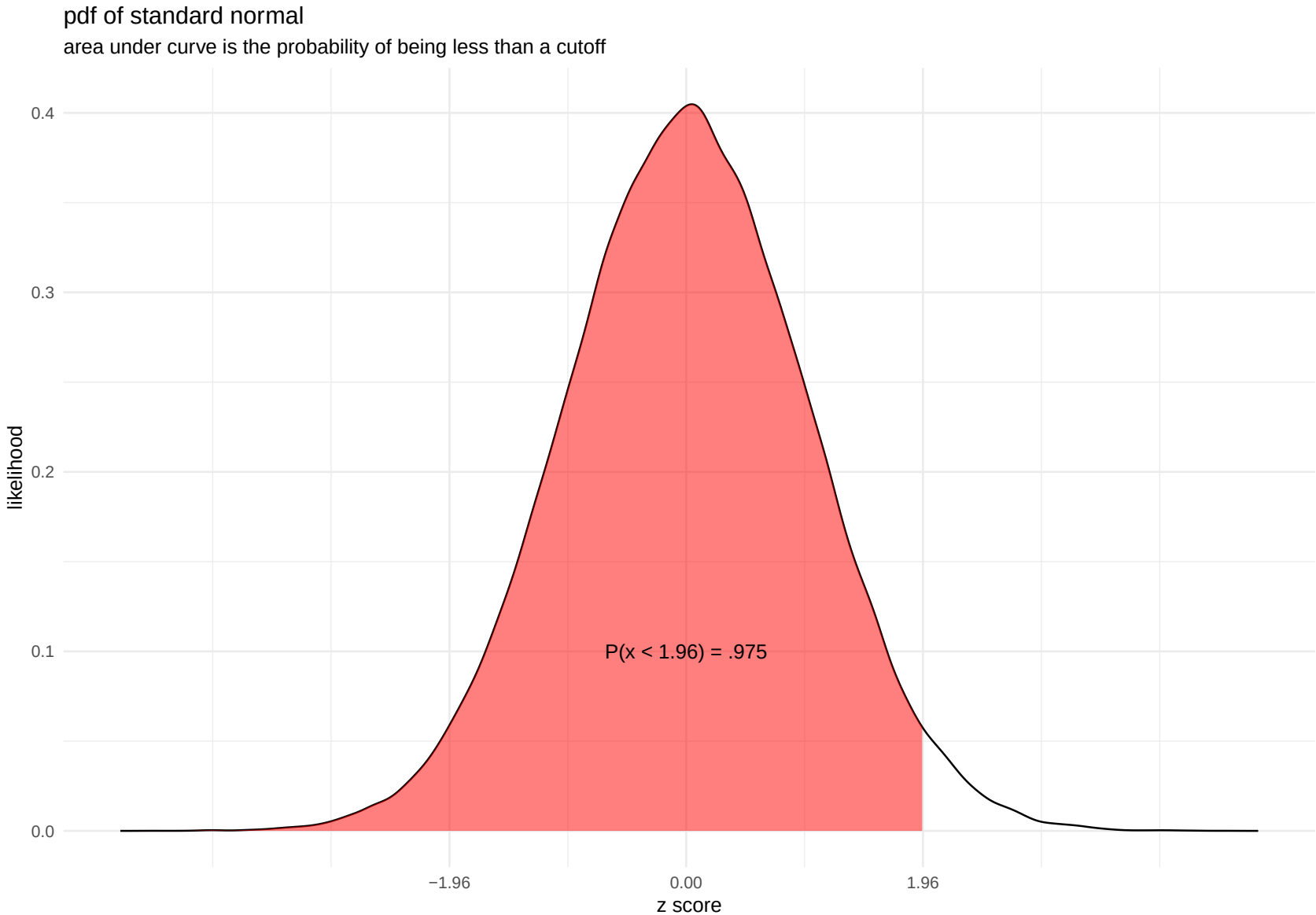
```
## [1] 1.959964
```

```
qnorm(.5)
```

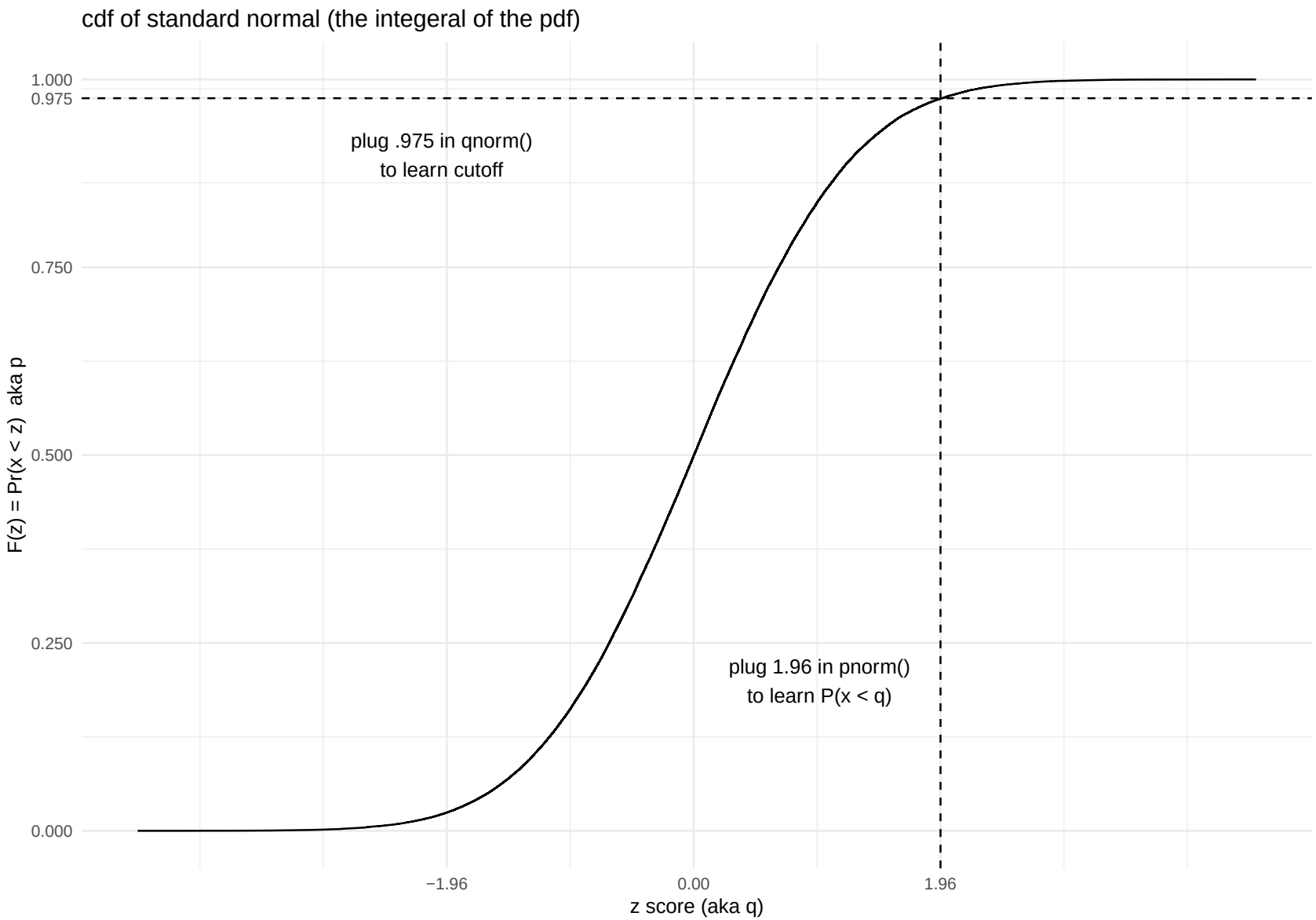
```
## [1] 0
```

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)
```

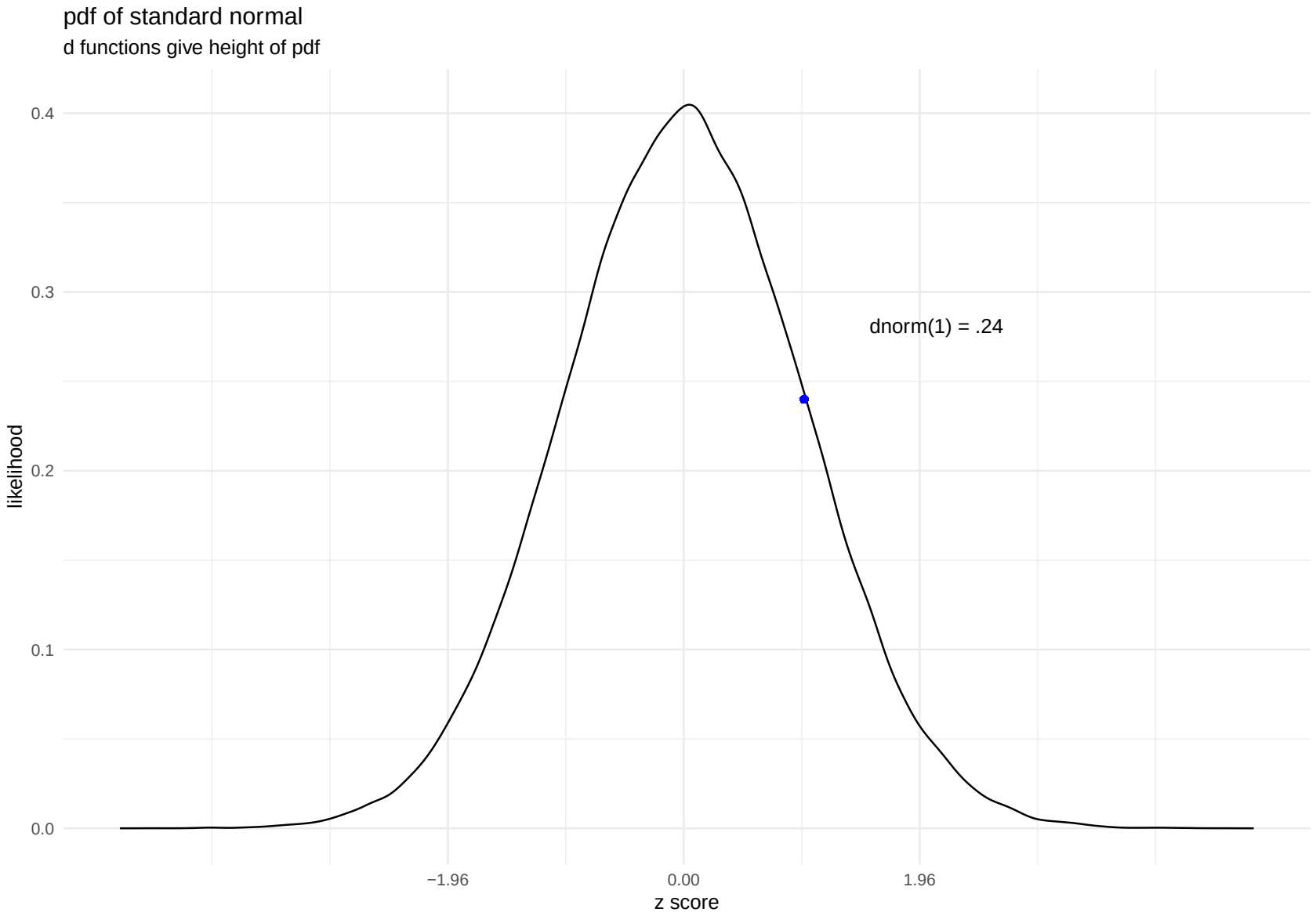
```
## [1] 0.2419707
```

```
dnorm(-1)
```

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

What is d?

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



passing functions to functions

Revisiting `calc_sample_mean`

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

One approach – make new functions for each distribution

```
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
```

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

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

Nothing is stopping you from passing a function as an argument

`rdist` can be any distribution!

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

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

  mean(sample)
}

set.seed(1)
calc_sample_mean(4, rnorm)
```

```
## [1] 0.07921043
```

rdist can be any distribution . . . not yet!

- The complication here is each distribution has it's own parameters. `df`, `mean` etc.

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

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

  mean(sample)
}

calc_sample_mean(4, rf)
```

Error in rdist(sample_size, mean = our_mean, sd = our_sd) : unused arguments (mean = our_mean, sd = our_sd)

A sophisticated approach - parameterize the distribution!

- `...` takes arbitrary arguments which you can pass to another function
- Warning `...` (dot-dot-dot) is a challenge to use

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

  sample <- rdist(sample_size, ...)

  mean(sample)
}
```

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

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

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

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

More examples

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

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

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

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

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

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

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

in a data context

```
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
```