Week 14: *Final Review*

EMSE 4571: Intro to Programming for Analytics

John Paul Helveston

May 04, 2023

Format: 2-part exam

Part 1

- Hand-written exam (like midterm).
- You may use a single 8.5 × 11 sheet of notes.
- No calculators, no books, no computers, no phones, no internet.

Part 2 (data analysis)

- Need your laptop (make sure it's charged!).
- Can only start Part 2 after turning in Part 1.
- You may use RStudio, the course website, and chatGPT.

What's on the final?

Comprehensive, except for Webscraping & Monte Carlo

Part 1

- 10 True / False questions.
- 4 Short answer questions.
- Hand-write one function and test function.

Part 2

- Read in a dataset.
- Answer questions about the data (using tidyverse tools).
- Make a visualization about the data.
- Bonus: Scrape a website.

Zero tolerance policy on cheating

Reasons to not cheat:

- Evidence of working with another person on the final results in a 0 for all individuals involved (and I'll push for class failure too).
- It's sooooooo easy to tell if you cheated (trust me, I'll know).
- I'm letting you use chatGPT for part 2!
- I'm a pretty soft grader anyway (you'll get 50% for just *trying*!).

Things to review

- Lecture slides, especially practice puzzles covered in class)
- Previous quizzes
- Memorize syntax for:
    - operators (e.g. mod `%%` and integer division `%/%`)
    - "number choping"
    - if / else statements
    - loops
    - functions
    - test functions
    - dplyr functions (`select`, `filter`, `mutate`, `arrange`, `group_by`, `summarise`)
    - How to use ggplot

Week 14: *Final Review*

1. Programming
2. Data Analytics

Week 14: *Final Review*

1. Programming
2. Data Analytics

Basics

Operators: Relational (=, <, >, <=, >=) and Logical (&, |, !)

```
x <- FALSE
y <- FALSE
z <- TRUE
```

a Write a logical statement that compares the objects x, y, and z and returns TRUE

b) Fill in **relational** operators to make this statement return TRUE:

```
! (x __ y) & ! (z __ y)
```

c) Fill in **logical** operators to make this statement return FALSE:

```
! (x __ y) | (z __ y)
```

Numeric Data

Doubles:

```r
typeof(3.14)
```

```
#> [1] "double"
```

"Integers":

```r
typeof(3)
```

```
#> [1] "double"
```

Actual Integers

Check if a number is an "integer":

```r
n <- 3
is.integer(n) # Doesn't work!
```

```
#> [1] FALSE
```

```r
n == as.integer(n) # Compare n to a converted version of itself
```

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

Logical Data

TRUE or FALSE

```r
x <- 1
y <- 2
```

```r
x > y # Is x greater than y?
```

```
#> [1] FALSE
```

```r
x == y
```

```
#> [1] FALSE
```

Tricky data type stuff

Logicals become numbers when doing math

```
TRUE + 1 # TRUE becomes 1
```

```
#> [1] 2
```

```
FALSE + 1 # FALSE becomes 0
```

```
#> [1] 1
```

Be careful of accidental strings

```
typeof("3.14")
```

```
#> [1] "character"
```

```
typeof("TRUE")
```

```
#> [1] "character"
```

Integer division: %/%

Integer division drops the remainder

```
4 / 3 # Regular division
```

```
#> [1] 1.333333
```

```
4 %/% 3 # Integer division
```

```
#> [1] 1
```

Integer division: %/%

Integer division drops the remainder

What will this return?

```r
4 %/% 4
```

```
#> [1] 1
```

What will this return?

```r
4 %/% 5
```

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

Modulus operator: %%

Modulus returns the remainder *after* doing integer division

```
5 %% 3
```

```
#> [1] 2
```

```
3.1415 %% 3
```

```
#> [1] 0.1415
```

Modulus operator: `%%`

Modulus returns the remainder *after* doing integer division

What will this return?

```r
4 %% 4
```

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

What will this return?

```r
4 %% 5
```

```
#> [1] 4
```

Number "chopping" with 10s (only works with `n > 0`)

The mod operator (`%%`) "chops" a number and returns everything to the *right*

```
123456 %% 1
```

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

```
123456 %% 10
```

```
#> [1] 6
```

```
123456 %% 100
```

```
#> [1] 56
```

Integer division (`%/%`) "chops" a number and returns everything to the *left*

```
123456 %/% 1
```

```
#> [1] 123456
```

```
123456 %/% 10
```

```
#> [1] 12345
```

```
123456 %/% 100
```

```
#> [1] 1234
```

Functions

Basic function syntax

```r
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Basic function syntax

In English:

> "`functionName` is a `function` of `arguments` that does..."

```
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Basic function syntax

Example:

> "`squareRoot` is a `function` of `n` that...returns the square root of `n`"

```r
squareRoot <- function(n) {
    return(n^0.5)
}
```

```r
squareRoot(64)
```

```
#> [1] 8
```

Test function "syntax"

Function:

```r
functionName <- function(arguments) {
    # Do stuff here
    return(something)
}
```

Test function:

```r
test_functionName <- function() {
    cat("Testing functionName()...")
    # Put test cases here
    cat("Passed!\n")
}
```

Writing test cases with `stopifnot()`

`stopifnot()` stops the function if whatever is inside the `()` is not `TRUE`.

Function:

```
isEven <- function(n) {
    return((n %% 2) == 0)
}
```

- `isEven(1)` should be `FALSE`
- `isEven(2)` should be `TRUE`
- `isEven(-7)` should be `FALSE`

Test function:

```
test_isEven <- function() {
    cat("Testing isEven()...")
    stopifnot(isEven(1) == FALSE)
    stopifnot(isEven(2) == TRUE)
    stopifnot(isEven(-7) == FALSE)
    cat("Passed!\n")
}
```

When testing *numbers*, use `almostEqual()`

Rounding errors can cause headaches:

```
x <- 0.1 + 0.2
x
```

```
#> [1] 0.3
```

```
x == 0.3
```

```
#> [1] FALSE
```

```
print(x, digits = 20)
```

```
#> [1] 0.30000000000000004441
```

Define a function that checks if two values are *almost* the same:

```
almostEqual <- function(n1, n2, threshold
= 0.00001) {
    return(abs(n1 - n2) <= threshold)
}
```

```
x <- 0.1 + 0.2
almostEqual(x, 0.3)
```

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

Make sure you know how to write `almostEqual()`

```
almostEqual <- function(n1, n2, threshold = 0.00001) {
    return(abs(n1 - n2) <= threshold)
}
```

Conditionals

Use `if` statements to filter function inputs

Example: Write the function `isEvenNumber(n)` that returns `TRUE` if `n` is an even number and `FALSE` otherwise. **If `n` is not a number, the function should return `FALSE`.**

```r
isEvenNumber <- function(n) {
    return((n %% 2) == 0)
}
```

```r
isEvenNumber(2)
```

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

```r
isEvenNumber("not_a_number")
```

```
#> Error in n%%2: non-numeric
argument to binary operator
```

```r
isEvenNumber <- function(n) {
    if (! is.numeric(n)) { return(FALSE) }
    return((n %% 2) == 0)
}
```

```r
isEvenNumber(2)
```

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

```r
isEvenNumber("not_a_number")
```

```
#> [1] FALSE
```

Loops

Use `for` loops when the number of iterations is ***known***.

1. Build the sequence
2. Iterate over it

```
for (i in 1:5) { # Define the sequence
    cat(i, '\n')
}
```

```
#> 1
#> 2
#> 3
#> 4
#> 5
```

Use `while` loops when the number of iterations is ***unknown***.

1. Define stopping condition
2. Manually increase condition

```
i <- 1
while (i <= 5) { # Define stopping condition
    cat(i, '\n')
    i <- i + 1 # Increase condition
}
```

```
#> 1
#> 2
#> 3
#> 4
#> 5
```

Search for something in a sequence

Example: count the **even** numbers in sequence: `1, (2), 3, (4), 5`

`for` loop

```
count <- 0 # Initialize count
for (i in seq(5)) {
    if (i %% 2 == 0) {
        count <- count + 1 # Update
    }
}
```

```
count
```

```
#> [1] 2
```

`while` loop

```
count <- 0 # Initialize count
i <- 1
while (i <= 5) {
    if (i %% 2 == 0) {
        count <- count + 1 # Update
    }
    i <- i + 1
}
```

```
count
```

```
#> [1] 2
```

Vectors

The universal vector generator: c()

Numeric vectors

```r
x <- c(1, 2, 3)
x
```

```
#> [1] 1 2 3
```

Character vectors

```r
y <- c('a', 'b', 'c')
y
```

```
#> [1] "a" "b" "c"
```

Logical vectors

```r
z <- c(TRUE, FALSE, TRUE)
z
```

```
#> [1]  TRUE FALSE  TRUE
```

Elements in vectors must be the same type

Type hierarchy:

- `character` > `numeric` > `logical`
- `double` > `integer`

Coverts to characters:

```
c(1, "foo", TRUE)
```

```
#> [1] "1"    "foo"  "TRUE"
```

Coverts to numbers:

```
c(7, TRUE, FALSE)
```

```
#> [1] 7 1 0
```

Coverts to double:

```
c(1L, 2, pi)
```

```
#> [1] 1.000000 2.000000
3.141593
```

Most functions operate on vector *elements*

```
x <- c(3.14, 7, 10, 15)
```

```
round(x)
```

```
#> [1]  3  7 10 15
```

```
isEven <- function(n) {
    return((n %% 2) == 0)
}
```

```
isEven(x)
```

```
#> [1] FALSE FALSE  TRUE FALSE
```

"Summary" functions **return one value**

```
x <- c(3.14, 7, 10, 15)
```

```
length(x)
```

```
#> [1] 4
```

```
min(x)
```

```
#> [1] 3.14
```

```
sum(x)
```

```
#> [1] 35.14
```

```
max(x)
```

```
#> [1] 15
```

```
prod(x)
```

```
#> [1] 3297
```

```
mean(x)
```

```
#> [1] 8.785
```

Use brackets [ ] to get elements from a vector

```
x <- seq(1, 10)
```

Indices start at 1:

```
x[1] # Returns the first element
```

```
#> [1] 1
```

```
x[3] # Returns the third element
```

```
#> [1] 3
```

```
x[length(x)] # Returns the last element
```

```
#> [1] 10
```

Slicing with a vector of indices:

```
x[1:3]  # Returns the first three elements
```

```
#> [1] 1 2 3
```

```
x[c(2, 7)] # Returns the 2nd and 7th elements
```

```
#> [1] 2 7
```

Use negative integers to *remove* elements

```r
x <- seq(1, 10)
```

```r
x[-1] # Drops the first element
```

```
#> [1]  2  3  4  5  6  7  8  9 10
```

```r
x[-1:-3] # Drops the first three elements
```

```
#> [1]  4  5  6  7  8  9 10
```

```r
x[-c(2, 7)] # Drops the 2nd and 7th elements
```

```
#> [1]  1  3  4  5  6  8  9 10
```

```r
x[-length(x)] # Drops the last element
```

```
#> [1] 1 2 3 4 5 6 7 8 9
```

Slicing with logical indices

```
x <- seq(1, 20, 3)
x
```

```
#> [1]  1  4  7 10 13 16 19
```

```
x > 10 # Create a logical vector based on some condition
```

```
#> [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

Slice `x` with logical vector - only `TRUE` elements will be returned:

```
x[x > 10]
```

```
#> [1] 13 16 19
```

Comparing vectors

Check if 2 vectors are the same:

```r
x <- c(1, 2, 3)
y <- c(1, 2, 3)
```

```r
x == y
```

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

Comparing vectors with `all()` and `any()`

`all()`: Check if *all* elements are the same

```
x <- c(1, 2, 3)
y <- c(1, 2, 3)
all(x == y)
```

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

```
x <- c(1, 2, 3)
y <- c(-1, 2, 3)
all(x == y)
```

```
#> [1] FALSE
```

`any()`: Check if *any* elements are the same

```
x <- c(1, 2, 3)
y <- c(1, 2, 3)
any(x == y)
```

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

```
x <- c(1, 2, 3)
y <- c(-1, 2, 3)
any(x == y)
```

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

Strings

Case conversion & substrings

Function	Description
`str_to_lower()`	converts string to lower case
`str_to_upper()`	converts string to upper case
`str_to_title()`	converts string to title case
`str_length()`	number of characters
`str_sub()`	extracts substrings
`str_locate()`	returns indices of substrings
`str_dup()`	duplicates characters

Quick practice:

05:00

Create this string object:

```
x <- 'thisIsGoodPractice'
```

Then use **stringr** functions to transform `x` into the following strings:

- `'thisIsGood'`
- `'practice'`
- `'GOOD'`
- `'thisthisthis'`
- `'GOODGOODGOOD'`

Hint: You'll need these:

- `str_to_lower()`
- `str_to_upper()`
- `str_locate()`
- `str_sub()`
- `str_dup()`

Padding, splitting, & merging

Function	Description
`str_trim()`	removes leading and trailing whitespace
`str_pad()`	pads a string
`paste()`	string concatenation
`str_split()`	split a string into a vector

Quick practice:

05:00

Create the following objects:

```
x <- 'this_is_good_practice'
y <- c('hello', 'world')
```

Use `stringr` functions to transform `x` and `y` into the following:

- `"hello world"`
- `"***hello world***"`
- `c("this", "is", "good", "practice")`
- `"this is good practice"`
- `"hello world, this is good practice"`

Hint: You'll need these:

- `str_trim()`
- `str_pad()`
- `paste()`
- `str_split()`

Detecting & replacing

Function	Description
`str_sort()`	sort a string alphabetically
`str_order()`	get the order of a sorted string
`str_detect()`	match a string in another string
`str_replace()`	replace a string in another string

Quick practice:

05:00

```
fruit[1:5]
```

```
#> [1] "apple"        "apricot"      "avocado"      "banana"       "bell pepper"
```

Use `stringr` functions to answer the following questions about the `fruit` vector:

1. How many fruit have the string `"rr"` in it?
2. Which fruit end with string `"fruit"`?
3. Which fruit contain more than one `"o"` character?

Hint: You'll need to use `str_detect()` and `str_count()`

Week 14: *Final Review*

1. Programming
2. Data Analytics

Data Frame Basics

Columns: *Vectors* of values (must be same data type)

```
beatles
```

```
#> # A tibble: 4 × 5
#>   firstName lastName  instrument yearOfBirth deceased
#>   <chr>     <chr>     <chr>            <dbl> <lgl>
#> 1 John      Lennon    guitar            1940 TRUE
#> 2 Paul      McCartney bass              1942 FALSE
#> 3 Ringo     Starr     drums             1940 FALSE
#> 4 George    Harrison  guitar            1943 TRUE
```

Extract a column using `$`

```
beatles$firstName
```

```
#> [1] "John"   "Paul"   "Ringo"  "George"
```

Create new variables with the $ symbol

Add the hometown of the bandmembers:

```
beatles$hometown <- 'Liverpool'
beatles
```

```
#> # A tibble: 4 × 6
#>   firstName lastName  instrument yearOfBirth deceased hometown
#>   <chr>     <chr>     <chr>            <dbl> <lgl>    <chr>
#> 1 John      Lennon    guitar            1940 TRUE     Liverpool
#> 2 Paul      McCartney bass              1942 FALSE    Liverpool
#> 3 Ringo     Starr     drums             1940 FALSE    Liverpool
#> 4 George    Harrison  guitar            1943 TRUE     Liverpool
```

Rows: Information about individual observations

Information about *John Lennon* is in the first row:

```
beatles[1,]
```

```
#> # A tibble: 1 × 6
#>   firstName lastName instrument yearOfBirth deceased hometown
#>   <chr>     <chr>    <chr>            <dbl> <lgl>    <chr>
#> 1 John      Lennon   guitar            1940 TRUE     Liverpool
```

Information about *Paul McCartney* is in the second row:

```
beatles[2,]
```

```
#> # A tibble: 1 × 6
#>   firstName lastName  instrument yearOfBirth deceased hometown
#>   <chr>     <chr>     <chr>            <dbl> <lgl>    <chr>
#> 1 Paul      McCartney bass              1942 FALSE    Liverpool
```

Access elements by index: `DF[row, column]`

General form for indexing elements:

```
DF[row, column]
```

Select the element in row 1, column 2:

```
beatles[1, 2]
```

```
#> # A tibble: 1 × 1
#>   lastName
#>   <chr>
#> 1 Lennon
```

Select the elements in rows 1 & 2 and columns 2 & 3:

```
beatles[c(1, 2), c(2, 3)]
```

```
#> # A tibble: 2 × 2
#>   lastName  instrument
#>   <chr>     <chr>
#> 1 Lennon    guitar
#> 2 McCartney bass
```

Steps to importing external data files

1. Create a path to the data

```r
library(here)
pathToData <- here('data', 'data.csv')
pathToData
```

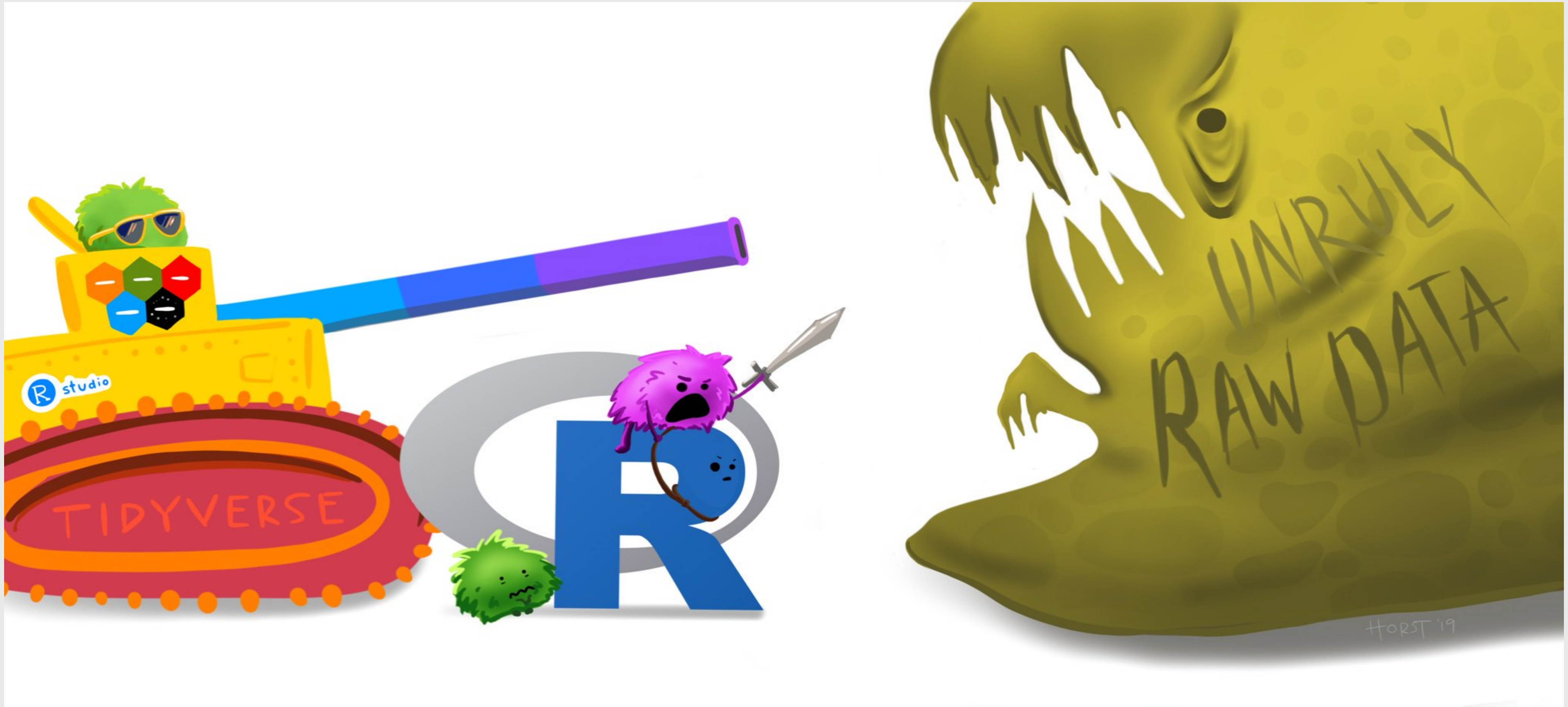
```
#> [1] "/Users/jhelvy/gh/teaching/P4A/2023-Spring/class/14-final-review/data/data.csv"
```

2. Import the data

```r
library(readr)
df <- read_csv(pathToData)
```

Data Wrangling

The tidyverse: `stringr` + `dplyr` + `readr` + `ggplot2` + ...

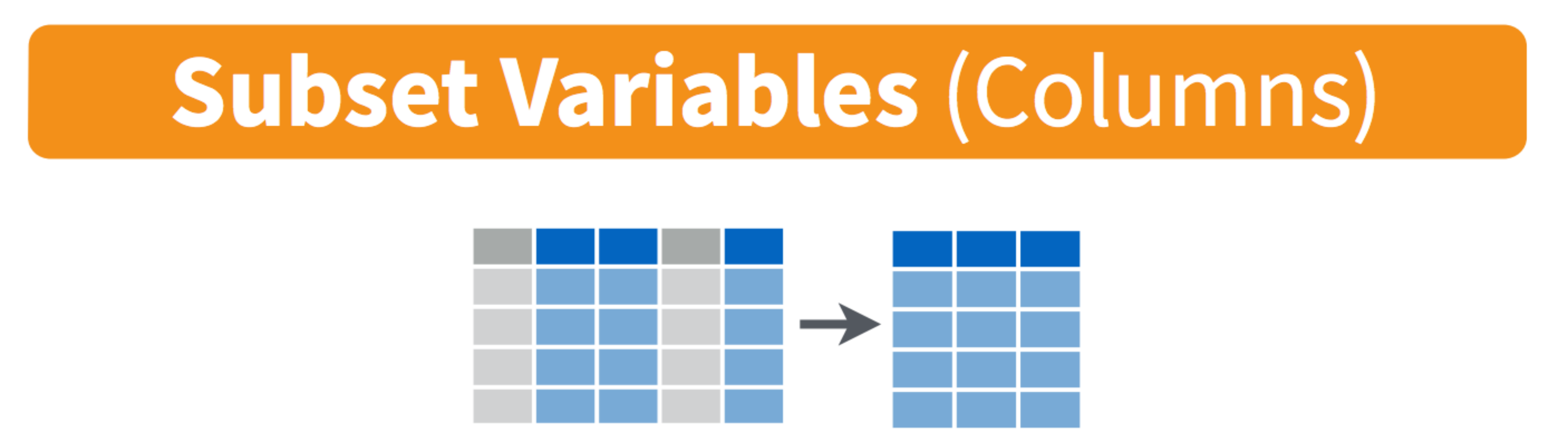


Art by Allison Horst

Know how to use these functions!

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `arrange()`: sort data frame
- `mutate()`: create new columns by using information from other columns
- `group_by()`: group data to perform grouped operations
- `summarize()`: create summary statistics (usually on grouped data)
- `count()`: count discrete rows

Select columns with `select()`

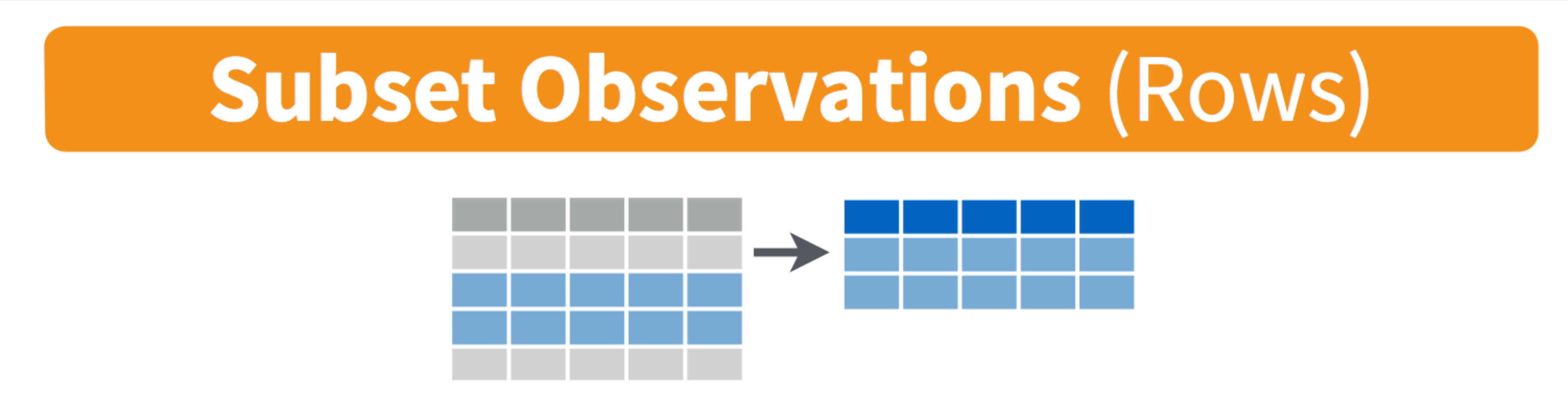


Select columns with `select()`

```
spicegirls %>%
    select(firstName, lastName)
```

```
#> # A tibble: 5 × 2
#>   firstName lastName
#>   <chr>     <chr>
#> 1 Melanie   Brown
#> 2 Melanie   Chisholm
#> 3 Emma      Bunton
#> 4 Geri      Halliwell
#> 5 Victoria  Beckham
```

Select rows with `filter()`



Select rows with `filter()`

Example: Filter the band members born after 1974

```
spicegirls %>%
    filter(yearOfBirth > 1974)
```

```
#> # A tibble: 2 × 5
#>   firstName lastName spice yearOfBirth deceased
#>   <chr>     <chr>    <chr>       <dbl> <lgl>
#> 1 Melanie   Brown    Scary        1975 FALSE
#> 2 Emma      Bunton   Baby         1976 FALSE
```

Removing missing values

Drop all rows where `variable` is `NA`

```
data %>%
    filter(!is.na(variable))
```

Don't make this common mistake!

Wrong!

```
data %>%
  filter(data, condition)
```

Right!

```
data %>%
  filter(condition)
```

Or:

```
filter(data, condition)
```

Create new variables with `mutate()`

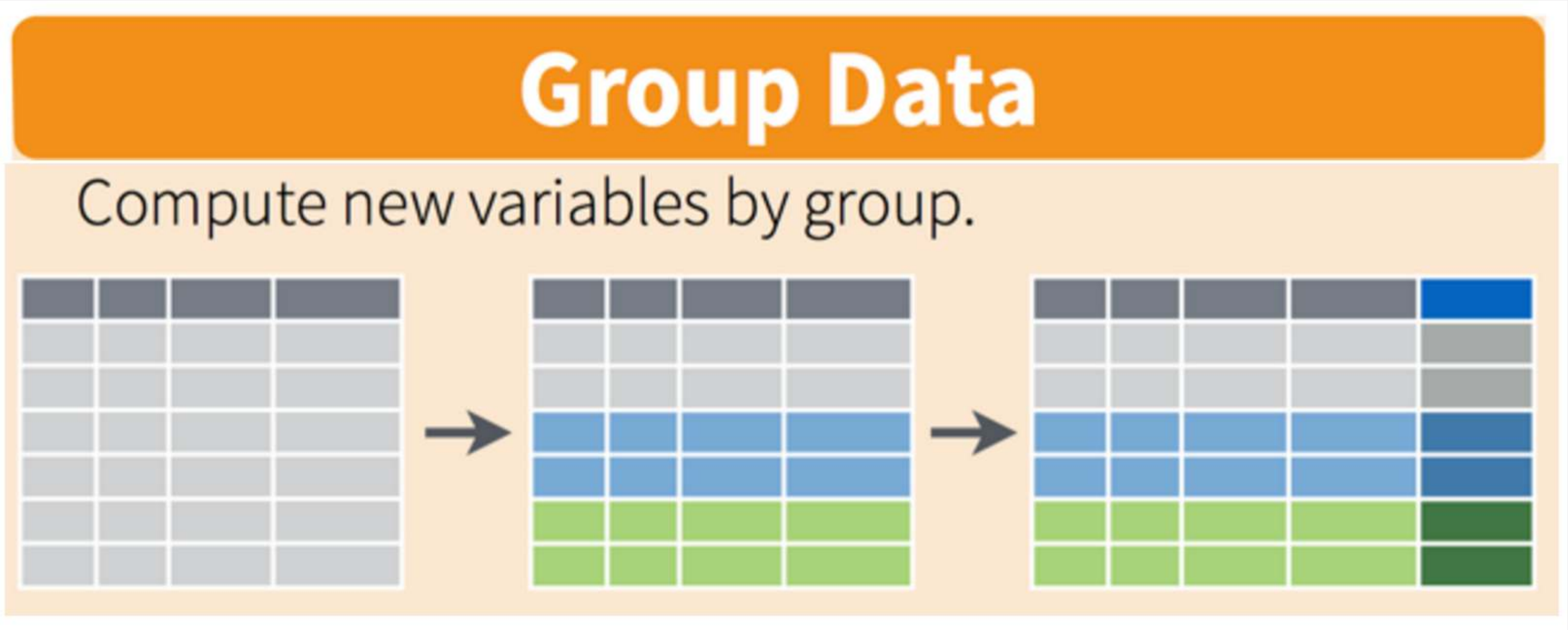
Create new variables with mutate()

Example: Use the yearOfBirth variable to compute the age of each band member

```
spicegirls %>%
    mutate(age = 2022 - yearOfBirth)
```

```
#> # A tibble: 5 × 6
#>   firstName lastName  spice  yearOfBirth deceased   age
#>   <chr>     <chr>     <chr>        <dbl> <lgl>    <dbl>
#> 1 Melanie   Brown     Scary         1975 FALSE       47
#> 2 Melanie   Chisholm  Sporty        1974 FALSE       48
#> 3 Emma      Bunton    Baby          1976 FALSE       46
#> 4 Geri      Halliwell Ginger        1972 FALSE       50
#> 5 Victoria  Beckham   Posh          1974 FALSE       48
```

Split-apply-combine with `group_by`



Compute values by group with `group_by`

Compute the mean band member age for **each band**

```
bands %>%
    mutate(
        age = 2020 - yearOfBirth,
        mean_age = mean(age)) # This is the mean across both bands
```

```
#> # A tibble: 9 × 8
#>   firstName lastName  yearOfBirth deceased band       instrument   age mean_age
#>   <chr>     <chr>           <dbl> <lgl>    <chr>      <chr>      <dbl>    <dbl>
#> 1 Melanie   Brown            1975 FALSE    spicegirls <NA>          45     60.4
#> 2 Melanie   Chisholm         1974 FALSE    spicegirls <NA>          46     60.4
#> 3 Emma      Bunton           1976 FALSE    spicegirls <NA>          44     60.4
#> 4 Geri      Halliwell        1972 FALSE    spicegirls <NA>          48     60.4
#> 5 Victoria  Beckham          1974 FALSE    spicegirls <NA>          46     60.4
#> 6 John      Lennon           1940 TRUE     <NA>       guitar        80     60.4
#> 7 Paul      McCartney        1942 FALSE    <NA>       bass          78     60.4
#> 8 Ringo     Starr            1940 FALSE    <NA>       drums         80     60.4
#> 9 George    Harrison         1943 TRUE     <NA>       guitar        77     60.4
```

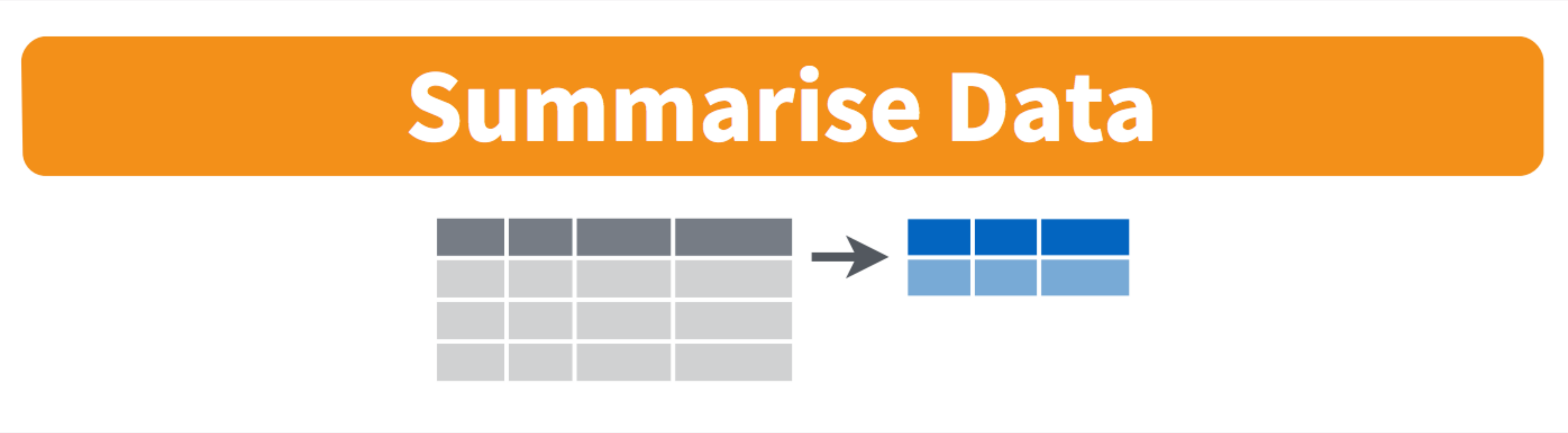
Compute values by group with `group_by`

Compute the mean band member age for each band

```r
bands %>%
    mutate(age = 2020 - yearOfBirth) %>%
    group_by(band) %>% # Everything after this will be done each band
    mutate(mean_age = mean(age))
```

```
#> # A tibble: 9 × 8
#> # Groups:   band [2]
#>   firstName lastName  yearOfBirth deceased band       instrument   age mean_age
#>   <chr>     <chr>           <dbl> <lgl>    <chr>      <chr>      <dbl>    <dbl>
#> 1 Melanie   Brown            1975 FALSE    spicegirls <NA>          45     45.8
#> 2 Melanie   Chisholm         1974 FALSE    spicegirls <NA>          46     45.8
#> 3 Emma      Bunton           1976 FALSE    spicegirls <NA>          44     45.8
#> 4 Geri      Halliwell        1972 FALSE    spicegirls <NA>          48     45.8
#> 5 Victoria  Beckham          1974 FALSE    spicegirls <NA>          46     45.8
#> 6 John      Lennon           1940 TRUE     <NA>       guitar        80     78.8
#> 7 Paul      McCartney        1942 FALSE    <NA>       bass          78     78.8
#> 8 Ringo     Starr            1940 FALSE    <NA>       drums         80     78.8
#> 9 George    Harrison         1943 TRUE     <NA>       guitar        77     78.8
```

Summarize data frames with `summarise()`



Summarize data frames with `summarise()`

Compute the mean band member age for **each band**

```
bands %>%
    mutate(age = 2020 - yearOfBirth) %>%
    group_by(band) %>%
    summarise(mean_age = mean(age)) # Drops all variables except for group
```

```
#> # A tibble: 2 × 2
#>   band       mean_age
#>   <chr>         <dbl>
#> 1 spicegirls     45.8
#> 2 <NA>           78.8
```

If you only want a quick count, use `count()`

These do the same thing:

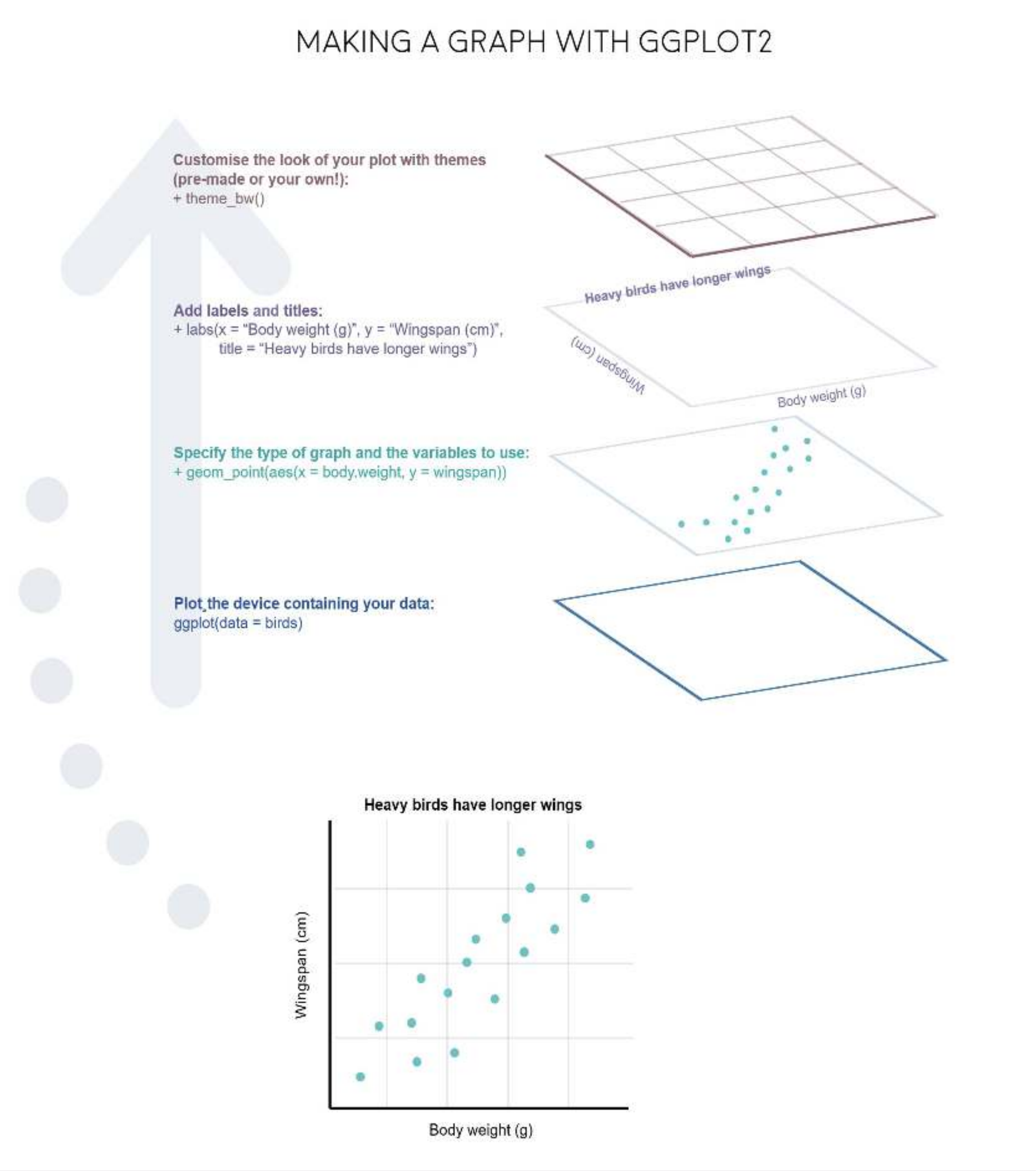
```r
bands %>%
    group_by(band) %>%
    summarise(n = n())
```

```
#> # A tibble: 2 × 2
#>   band           n
#>   <chr>      <int>
#> 1 spicegirls     5
#> 2 <NA>           4
```

```r
bands %>%
    count(band)
```

```
#> # A tibble: 2 × 2
#>   band           n
#>   <chr>      <int>
#> 1 spicegirls     5
#> 2 <NA>           4
```

Data Visualization



"Grammar of Graphics"

Concept developed by Leland Wilkinson (1999)

ggplot2 package developed by Hadley Wickham (2005)

Making plot layers with ggplot2

1. The data (we'll use `bears`)

2. The aesthetic mapping (what goes on the axes?)

3. The geometries (points? bars? etc.)

Layer 1: The data

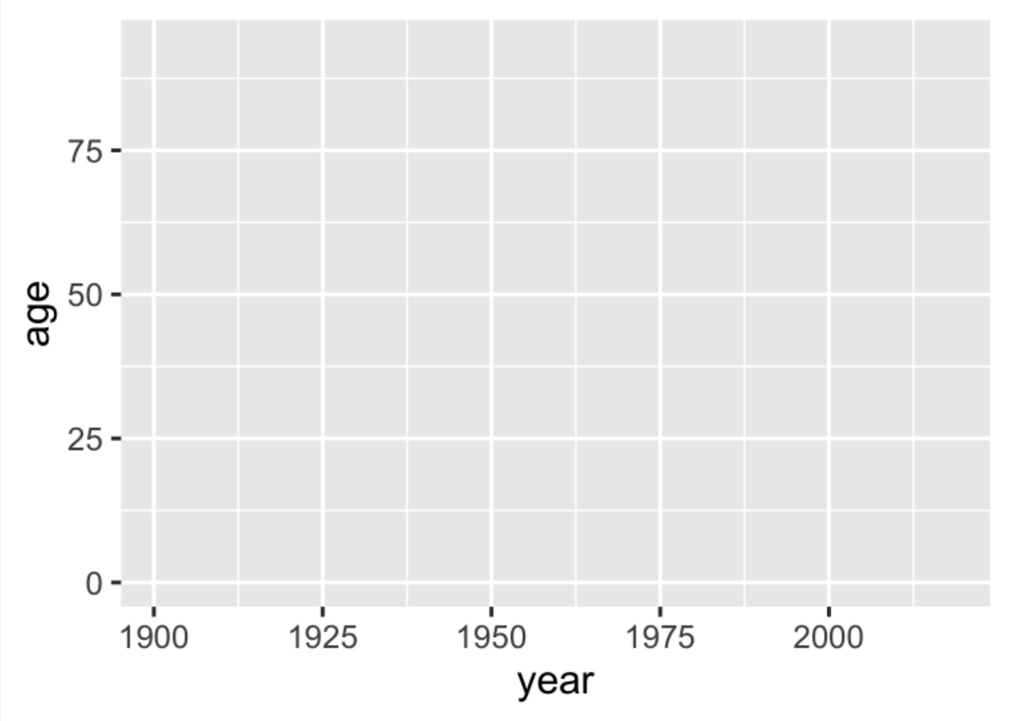
The `ggplot()` function initializes the plot with whatever data you're using

```
bears %>%
  ggplot()
```

Layer 2: The aesthetic mapping

The `aes()` function determines which variables will be *mapped* to the geometries (e.g. the axes)

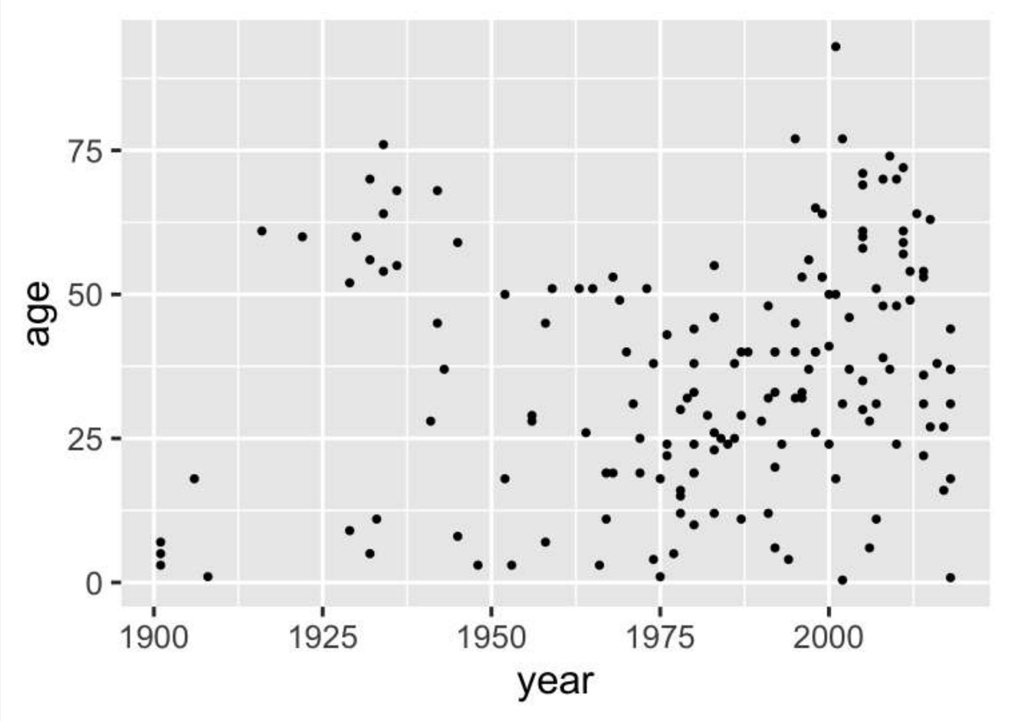
```
bears %>%
  ggplot(aes(x = year, y = age))
```



Layer 3: The geometries

Use + to add geometries (e.g. points)

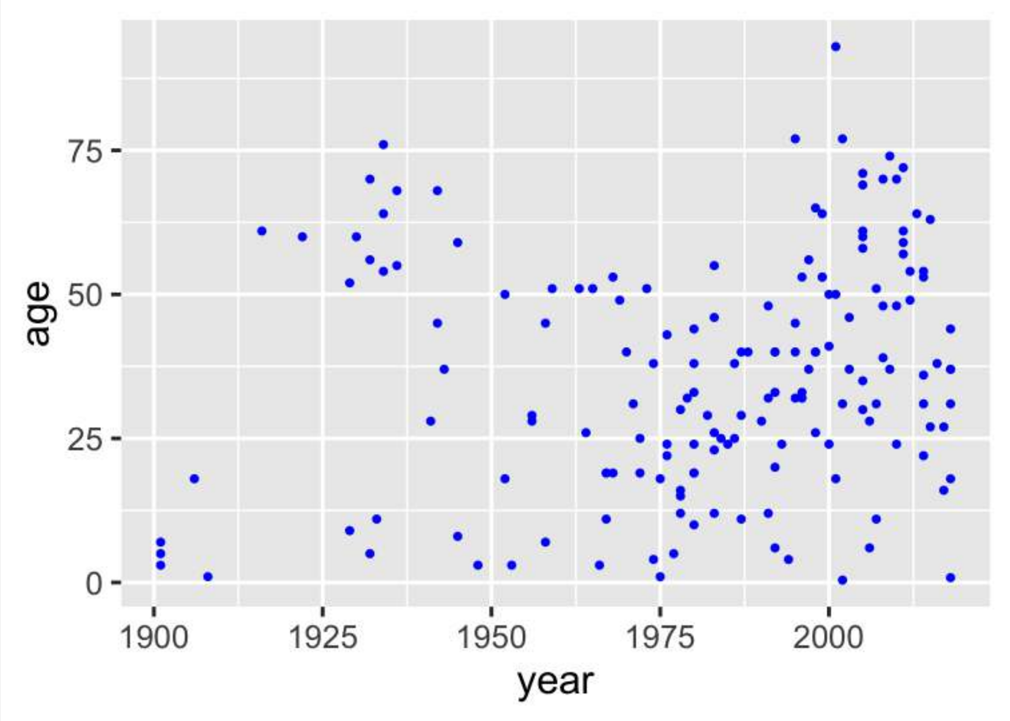
```
bears %>%
  ggplot(aes(x = year, y = age)) +
  geom_point()
```



Scatterplots with geom_point()

Change the color of all points:

```
bears %>%
  ggplot(aes(x = year, y = age)) +
  geom_point(color = 'blue')
```

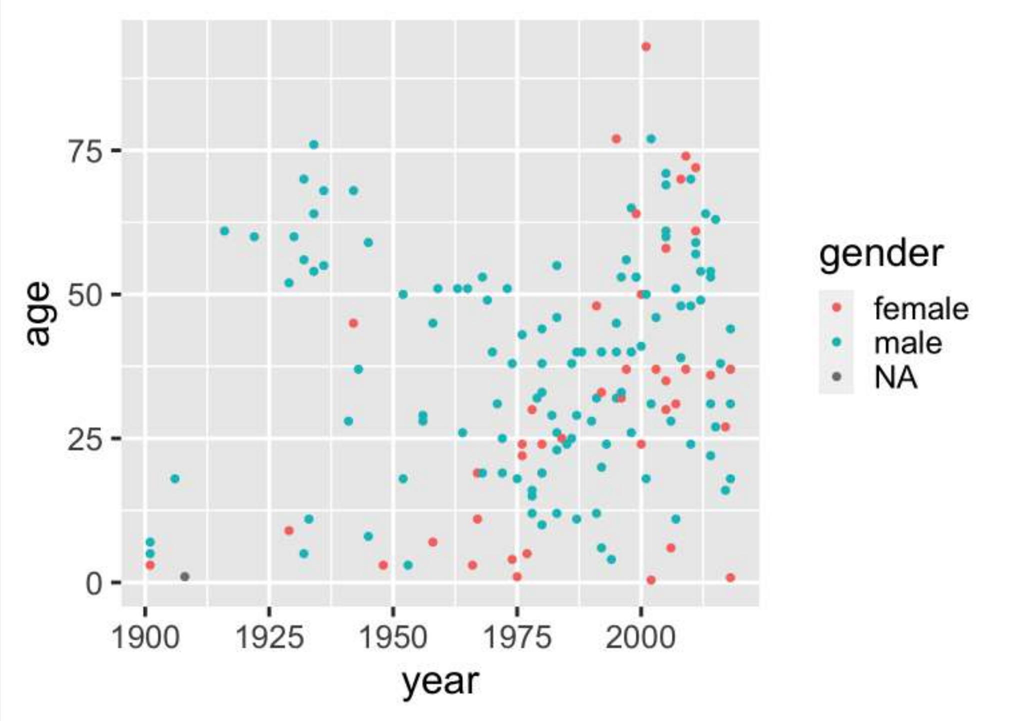


Scatterplots with `geom_point()`

Map the point color to a **variable**:

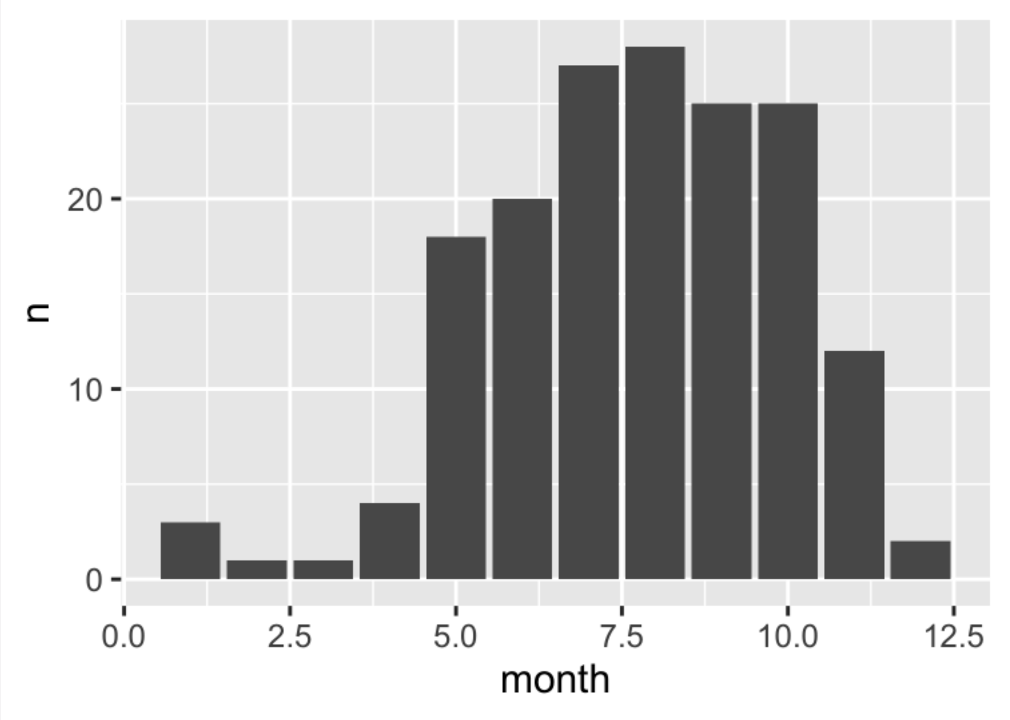
```
bears %>%
  ggplot(aes(x = year, y = age)) +
  geom_point(aes(color = gender))
```

Note that `color = gender` is *inside* `aes()`



Make bar charts with geom_col()

```
bears %>%
  count(month) %>%
  ggplot() +
  geom_col(aes(x = month, y = n))
```

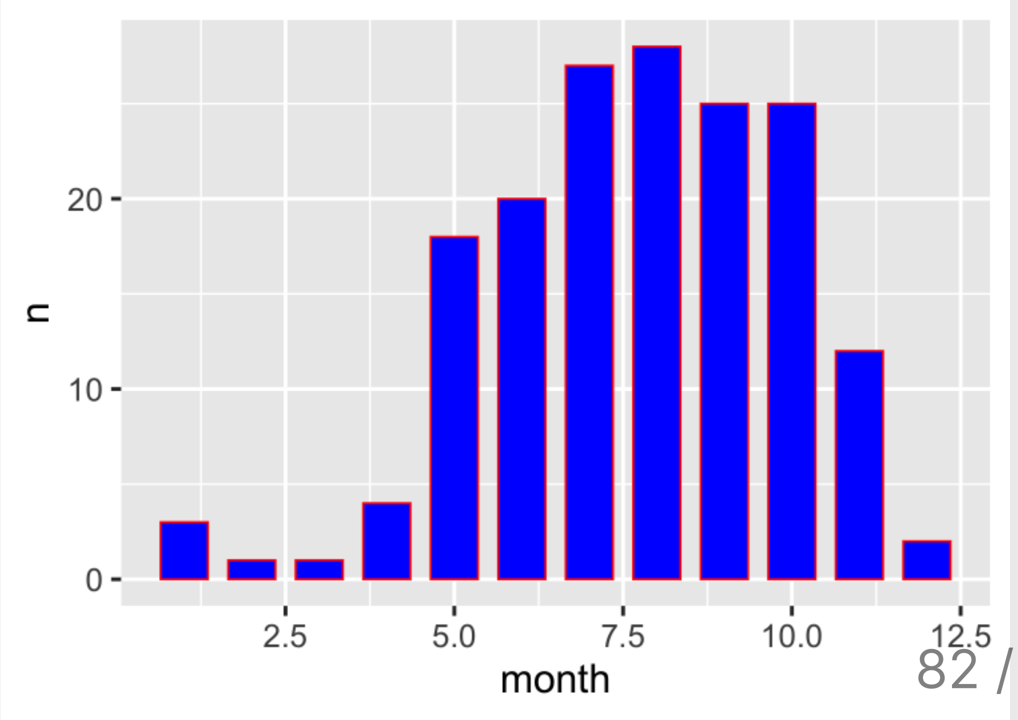


Change bar width: `width`

Change bar color: `fill`

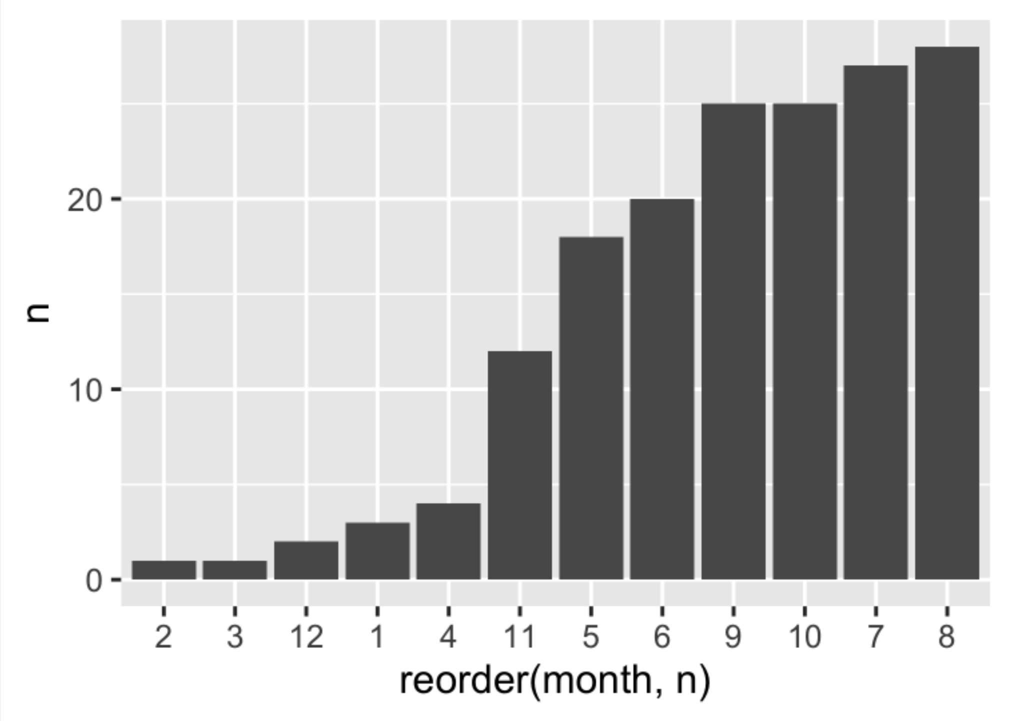
Change bar outline: `color`

```
bears %>%
  count(month) %>%
  ggplot() +
  geom_col(
    mapping = aes(x = month, y = n),
    width = 0.7,
    fill = "blue",
    color = "red"
  )
```



Rearrange bars by reordering the factors

```
bears %>%
  count(month) %>%
  ggplot() +
  geom_col(
    aes(
      x = reorder(month, n),
      y = n
    )
  )
```



Programming with Data

Convert this to a function

Single-use pipeline

```
diamonds %>%
  group_by(color) %>%
  summarise(
    n = n(),
    mean = mean(price),
    sd = sd(price)
  )
```

```
#> # A tibble: 7 × 4
#>   color     n  mean    sd
#>   <ord> <int> <dbl> <dbl>
#> 1 D      6775 3170. 3357.
#> 2 E      9797 3077. 3344.
#> 3 F      9542 3725. 3785.
#> 4 G     11292 3999. 4051.
#> 5 H      8304 4487. 4216.
#> 6 I      5422 5092. 4722.
```

As a function by "embracing" variable 🤗

```
my_summary <- function(df, group, var) {
  df %>%
    group_by({{ group }}) %>%
    summarise(
      n = n(),
      mean = mean({{ var }}),
      sd = sd({{ var }})
    )
}
```

Use it on a different data frame!

```r
library(palmerpenguins)

my_summary(penguins, sex, body_mass_g)
```

```
#> # A tibble: 3 × 4
#>   sex        n  mean    sd
#>   <fct>  <int> <dbl> <dbl>
#> 1 female   165 3862.  666.
#> 2 male     168 4546.  788.
#> 3 <NA>      11   NA    NA
```

```r
my_summary(penguins, species,
bill_length_mm)
```

```
#> # A tibble: 3 × 4
#>   species       n  mean    sd
#>   <fct>     <int> <dbl> <dbl>
#> 1 Adelie      152  NA   NA
#> 2 Chinstrap    68  48.8  3.34
#> 3 Gentoo      124  NA   NA
```

Iterating on data with `purrr`

Loaded automatically with `library(tidyverse)`

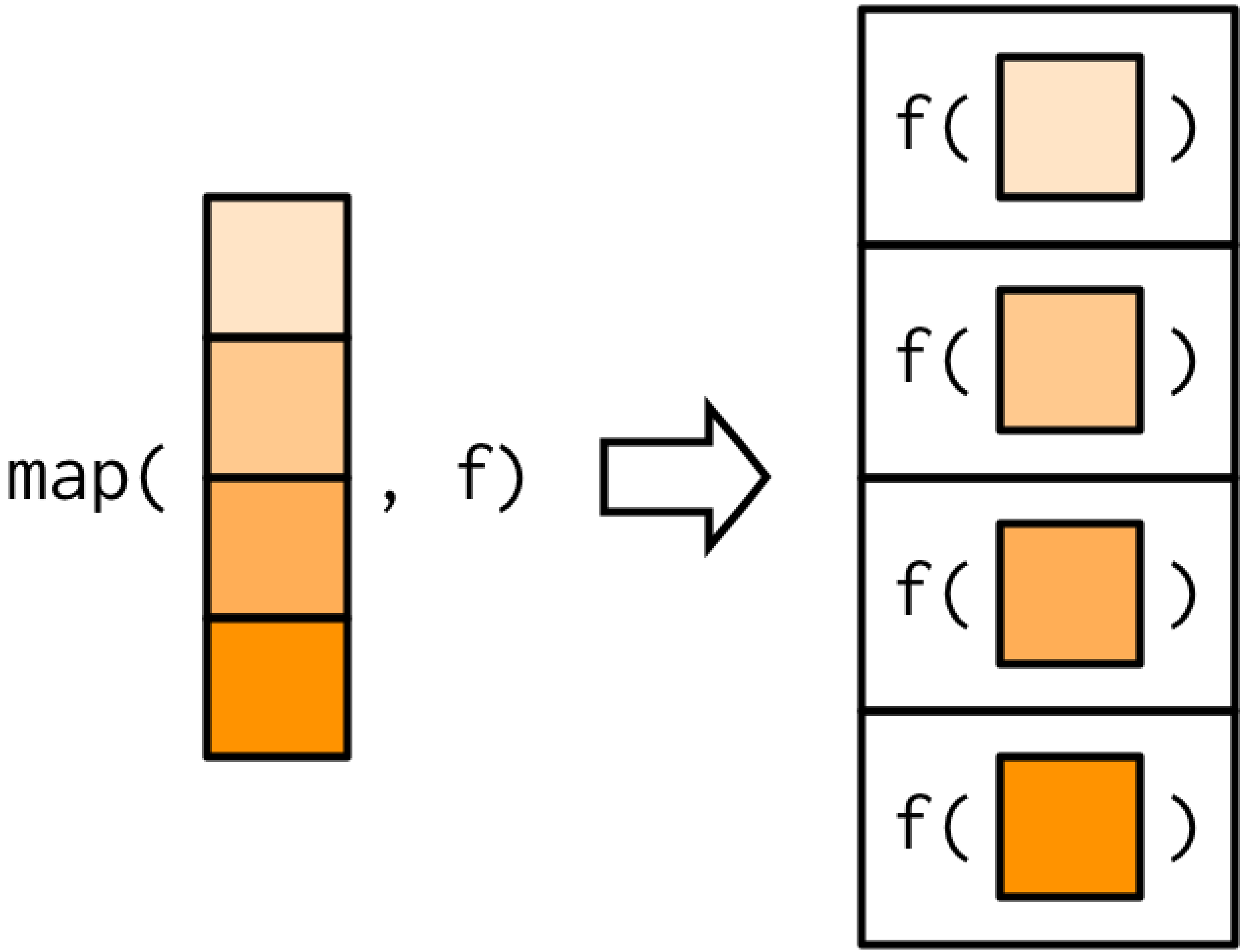
```
purrr::map(x, f, ...)
```

for every element of `x` do `f`


x = minis
f = add_antenna

```
map(minis, add_antenna)
```

for every element of x do f



Some examples

What will this return?

```
# eval: false

map(1:3, \(x) x %% 2 == 0)
```

```
#> [[1]]
#> [1] FALSE
#>
#> [[2]]
#> [1] TRUE
#>
#> [[3]]
#> [1] FALSE
```

```
map(1:3, \(x) x %% 2 == 0)
```

```
#> [[1]]
#> [1] FALSE
#>
```

Some examples

What will this return?

```
# eval: false

sum(map_int(1:3, \(x) x %% 2 == 0))
```

```
#> [1] 1
```

```
sum(map_int(1:3, \(x) x %% 2 == 0))
```

```
#> [1] 1
```

Webscraping

There will be a bonus question on scraping a website

General tips:

`html_element()`:

- Know when to use `html_element()` vs `html_elements()`
- **Warning**: ChatGPT doesn't know `html_element()` - it only knows `html_node()`

`html_table()`:

- If you use `html_table()`, remember it returns a **list** of tables.
- Usually you want the first table, something like this:

```
tables <- html %>% html_table()
df <- tables[[1]]
```

Monte Carlo

Monte Carlo Simulation: *Computing Probability*

General process:

- Run a series of trials.
- In each trial, simulate an event (e.g. a coin toss, a dice roll, etc.).
- Count the number of "successful" trials

$$\frac{\#\ \text{Successful Trials}}{\#\ \text{Total Trials}} = \text{Observed Odds} \simeq \text{Expected Odds}$$

Law of large numbers:
As *N* increases, Observed Odds » Expected Odds

Discrete, **Independent** Events: `sample(replace = TRUE)`

What is the probability of rolling a 6-sided dice 3 times
and getting the sequence 1, 3, 5?

```
library(tidyverse)

dice <- c(1, 2, 3, 4, 5, 6)
N <- 10000

rolls <- tibble(
  roll1 = sample(x = dice, size = N, replace = T),
  roll2 = sample(x = dice, size = N, replace = T),
  roll3 = sample(x = dice, size = N, replace = T)
)

successes <- rolls %>%
  filter(roll1 == 1 & roll2 == 3 & roll3 == 5)

nrow(successes) / N
```

```
#> [1] 0.0054
```

Discrete, **Dependent** Events: `sample(replace = FALSE)`

What are the odds that 3 cards drawn from a 52-card deck will sum to 13?

Repeat the 3-card draw *N* times:

```r
deck <- rep(c(seq(1, 10), 10, 10, 10), 4)

N <- 100000
count <- 0
for (i in 1:N) {
   draw <- sample(x = deck, size = 3, replace = FALSE)
   if (sum(draw) == 13) {
      count <- count + 1
   }
}

count / N # Compute the probability
```

```
#> [1] 0.03642
```

Begin list of all problems solved in class

General function writing

`eggCartons(eggs)`: Write a function that reads in a non-negative number of eggs and prints the number of egg cartons required to hold that many eggs. Each egg carton holds one dozen eggs, and you cannot buy fractional egg cartons.

- eggCartons(0) == 0
- eggCartons(1) == 1
- eggCartons(12) == 1
- eggCartons(25) == 3

`militaryTimeToStandardTime(n)`: Write a function that takes an integer between 0 and 23 (representing the hour in military time), and returns the same hour in standard time.

- militaryTimeToStandardTime(0) == 12
- militaryTimeToStandardTime(3) == 3
- militaryTimeToStandardTime(12) == 12
- militaryTimeToStandardTime(13) == 1
- militaryTimeToStandardTime(23) == 11

Number chopping

`onesDigit(x)`: Write a function that takes an integer and returns its ones digit.

Tests:

- onesDigit(123) == 3
- onesDigit(7890) == 0
- onesDigit(6) == 6
- onesDigit(-54) == 4

`tensDigit(x)`: Write a function that takes an integer and returns its tens digit.

Tests:

- tensDigit(456) == 5
- tensDigit(23) == 2
- tensDigit(1) == 0
- tensDigit(-7890) == 9

Top-down design

Create a function, `isRightTriangle(a, b, c)` that returns `TRUE` if the triangle formed by the lines of length `a`, `b`, and `c` is a right triangle and `FALSE` otherwise. Use the `hypotenuse(a, b)` function in your solution. **Hint**: you may not know which value (`a`, `b`, or `c`) is the hypotenuse.

```
hypotenuse <- function(a, b) {
    return(sqrt(sumOfSquares(a, b)))
}
```

```
sumOfSquares <- function(a, b) {
    return(a^2 + b^2)
}
```

Conditionals (if / else)

`getType(x)`: Write the function `getType(x)` that returns the type of the data (either `integer`, `double`, `character`, or `logical`). Basically, it does the same thing as the `typeof()` function (but you can't use `typeof()` in your solution).

- `getType(3) == "double"`
- `getType(3L) == "integer"`
- `getType("foo") == "character"`
- `getType(TRUE) == "logical"`

Conditionals (if / else)

For each of the following functions, start by writing a test function that tests the function for a variety of values of inputs. Consider cases that you might not expect!

`isFactor(f, n)`: Write the function `isFactor(f, n)` that takes two integer values and returns `TRUE` if `f` is a factor of `n`, and `FALSE` otherwise. Note that every integer is a factor of `0`. Assume `f` and `n` will only be numeric values, e.g. `2` is a factor of `6`.

`isMultiple(m, n)`: Write the function `isMultiple(m, n)` that takes two integer values and returns `TRUE` if `m` is a multiple of `n` and `FALSE` otherwise. Note that `0` is a multiple of every integer other than itself. Hint: You may want to use the `isFactor(f, n)` function you just wrote above. Assume `m` and `n` will only be numeric values.

Conditionals (if / else)

Write the function `getInRange(x, bound1, bound2)` which takes 3 numeric values: `x`, `bound1`, and `bound2` (`bound1` is not necessarily less than `bound2`). If `x` is between the two bounds, just return `x`, but if `x` is less than the lower bound, return the lower bound, or if `x` is greater than the upper bound, return the upper bound. For example:

- `getInRange(1, 3, 5)` returns `3` (the lower bound, since 1 is below [3,5])
- `getInRange(4, 3, 5)` returns `4` (the original value, since 4 is between [3,5])
- `getInRange(6, 3, 5)` returns `5` (the upper bound, since 6 is above [3,5])
- `getInRange(6, 5, 3)` returns `5` (the upper bound, since 6 is above [3,5])

Bonus: Re-write `getInRange(x, bound1, bound2)` without using conditionals

`for` loops

`sumFromMToN(m, n)`: Write a function that sums the total of the integers between `m` and `n`.
Challenge: Try solving this without a loop!

- `sumFromMToN(5, 10) == (5 + 6 + 7 + 8 + 9 + 10)`
- `sumFromMToN(1, 1) == 1`

`sumEveryKthFromMToN(m, n, k)`: Write a function to sum every kth integer from `m` to `n`.

- `sumEveryKthFromMToN(1, 10, 2) == (1 + 3 + 5 + 7 + 9)`
- `sumEveryKthFromMToN(5, 20, 7) == (5 + 12 + 19)`
- `sumEveryKthFromMToN(0, 0, 1) == 0`

`sumOfOddsFromMToN(m, n)`: Write a function that sums every *odd* integer between `m` and `n`.

- `sumOfOddsFromMToN(4, 10) == (5 + 7 + 9)`
- `sumOfOddsFromMToN(5, 9) == (5 + 7 + 9)`

`for` loop with `break` & `next`

`sumOfOddsFromMToNMax(m, n, max)`: Write a function that sums every *odd* integer from `m` to `n` until the sum is less than the value `max`. Your solution should use both `break` and `next` statements.

- `sumOfOddsFromMToNMax(1, 5, 4) == (1 + 3)`
- `sumOfOddsFromMToNMax(1, 5, 3) == (1)`
- `sumOfOddsFromMToNMax(1, 5, 10) == (1 + 3 + 5)`

`while` loops

`isMultipleOf4Or7(n)`: Write a function that returns `TRUE` if `n` is a multiple of 4 or 7 and `FALSE` otherwise.

- `isMultipleOf4Or7(0) == FALSE`
- `isMultipleOf4Or7(1) == FALSE`
- `isMultipleOf4Or7(4) == TRUE`
- `isMultipleOf4Or7(7) == TRUE`
- `isMultipleOf4Or7(28) == TRUE`

`nthMultipleOf4Or7(n)`: Write a function that returns the nth positive integer that is a multiple of either 4 or 7.

- `nthMultipleOf4Or7(1) == 4`
- `nthMultipleOf4Or7(2) == 7`
- `nthMultipleOf4Or7(3) == 8`
- `nthMultipleOf4Or7(4) == 12`
- `nthMultipleOf4Or7(5) == 14`
- `nthMultipleOf4Or7(6) == 16`

Loops / Vectors

`isPrime(n)`: Write a function that takes a non-negative integer, `n`, and returns `TRUE` if it is a prime number and `FALSE` otherwise. Use a loop or vector:

- `isPrime(1) == FALSE`
- `isPrime(2) == TRUE`
- `isPrime(7) == TRUE`
- `isPrime(13) == TRUE`
- `isPrime(14) == FALSE`

`nthPrime(n)`: Write a function that takes a non-negative integer, `n`, and returns the nth prime number, where `nthPrime(1)` returns the first prime number (2). Hint: use a while loop!

- `nthPrime(1) == 2`
- `nthPrime(2) == 3`
- `nthPrime(3) == 5`
- `nthPrime(4) == 7`
- `nthPrime(7) == 17`

Vectors

`reverse(x)`: Write a function that returns the vector in reverse order. You cannot use the `rev()` function.

- `all(reverseVector(c(5, 1, 3)) == c(3, 1, 5))`
- `all(reverseVector(c('a', 'b', 'c')) == c('c', 'b', 'a'))`
- `all(reverseVector(c(FALSE, TRUE, TRUE)) == c(TRUE, TRUE, FALSE))`

`alternatingSum(a)`: Write a function that takes a vector of numbers `a` and returns the alternating sum, where the sign alternates from positive to negative or vice versa.

- `alternatingSum(c(5,3,8,4)) == (5 - 3 + 8 - 4)`
- `alternatingSum(c(1,2,3)) == (1 - 2 + 3)`
- `alternatingSum(c(0,0,0)) == 0`
- `alternatingSum(c(-7,5,3)) == (-7 - 5 + 3)`

Strings

1) `reverseString(s)`: Write a function that returns the string `s` in reverse order.

- `reverseString("aWordWithCaps") == "spaChtiWdroWa"`
- `reverseString("abcde") == "edcba"`
- `reverseString("") == ""`

2) `isPalindrome(s)`: Write a function that returns `TRUE` if the string `s` is a [Palindrome] and `FALSE` otherwise.

- `isPalindrome("abcba") == TRUE`
- `isPalindrome("abcb") == FALSE`
- `isPalindrome("321123") == TRUE`

Strings

1) `sortString(s)`: Write the function `sortString(s)` that takes a string `s` and returns back an alphabetically sorted string.

- `sortString("cba") == "abc"`
- `sortString("abedhg") == "abdegh"`
- `sortString("AbacBc") == "aAbBcc"`

2) `areAnagrams(s1, s2)`: Write the function `areAnagrams(s1, s2)` that takes two strings, `s1` and `s2`, and returns `TRUE` if the strings are anagrams, and `FALSE` otherwise. **Treat lower and upper case as the same letters**.

- `areAnagrams("", "") == TRUE`
- `areAnagrams("aabbccdd", "bbccddee") == FALSE`
- `areAnagrams("TomMarvoloRiddle", "IAmLordVoldemort") == TRUE`

Data Frame Basics

Answer these questions using the `beatles` data frame:

1. Create a new column, `playsGuitar`, which is `TRUE` if the band member plays the guitar and `FALSE` otherwise.
2. Filter the data frame to select only the rows for the band members who have four-letter first names.
3. Create a new column, `fullName`, which contains the band member's first and last name separated by a space (e.g. `"John Lennon"`)

Data Wrangling: `select()` & `filter()`

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `wildlife_impacts.csv` file in the `data` folder.

2) Use the `df` object and the `select()` and `filter()` functions to answer the following questions:

- Create a new data frame, `df_birds`, that contains only the variables (columns) about the species of bird.
- Create a new data frame, `dc`, that contains only the observations (rows) from DC airports.
- Create a new data frame, `dc_birds_known`, that contains only the observations (rows) from DC airports and those where the species of bird is known.
- How many *known* unique species of birds have been involved in accidents at DC airports?

Data Wrangling: `select()` & `filter()` w/Pipes

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `wildlife_impacts.csv` file in the `data` folder.

2) Use the `df` object and `select()`, `filter()`, and `%>%` to answer the following questions:

- Create a new data frame, `dc_dawn`, that contains only the observations (rows) from DC airports that occurred at dawn.
- Create a new data frame, `dc_dawn_birds`, that contains only the observations (rows) from DC airports that occurred at dawn and only the variables (columns) about the species of bird.
- Create a new data frame, `dc_dawn_birds_known`, that contains only the observations (rows) from DC airports that occurred at dawn and only the variables (columns) about the KNOWN species of bird.
- How many *known* unique species of birds have been involved in accidents at DC airports at dawn?

Data Wrangling: `mutate()` & `arrange()`

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `wildlife_impacts.csv` file in the `data` folder.

2) Use the `df` object with `%>%` and `mutate()` to create the following new variables:

- `height_miles`: The `height` variable converted to miles (Hint: there are 5,280 feet in a mile).
- `cost_mil`: Is `TRUE` if the repair costs was greater or equal to $1 million, `FALSE` otherwise.
- `season`: One of four seasons based on the `incident_month` variable:
    - `spring`: March, April, May
    - `summer`: June, July, August
    - `fall`: September, October, November
    - `winter`: December, January, February

Data Wrangling: `group_by()` & `summarise()`

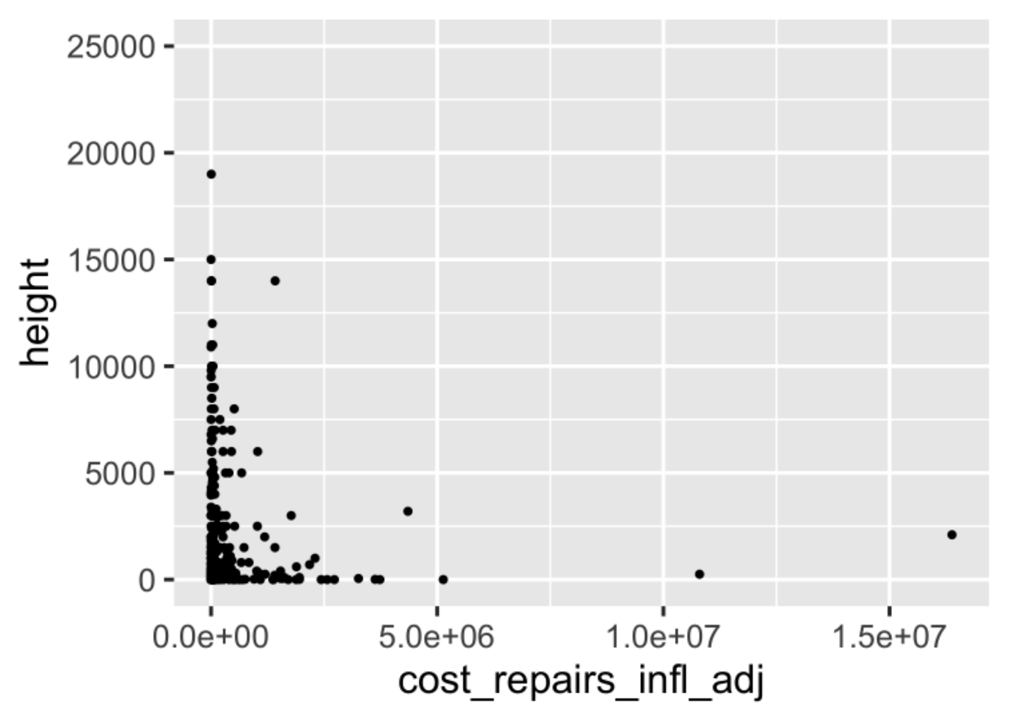
1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `wildlife_impacts.csv` file in the `data` folder.

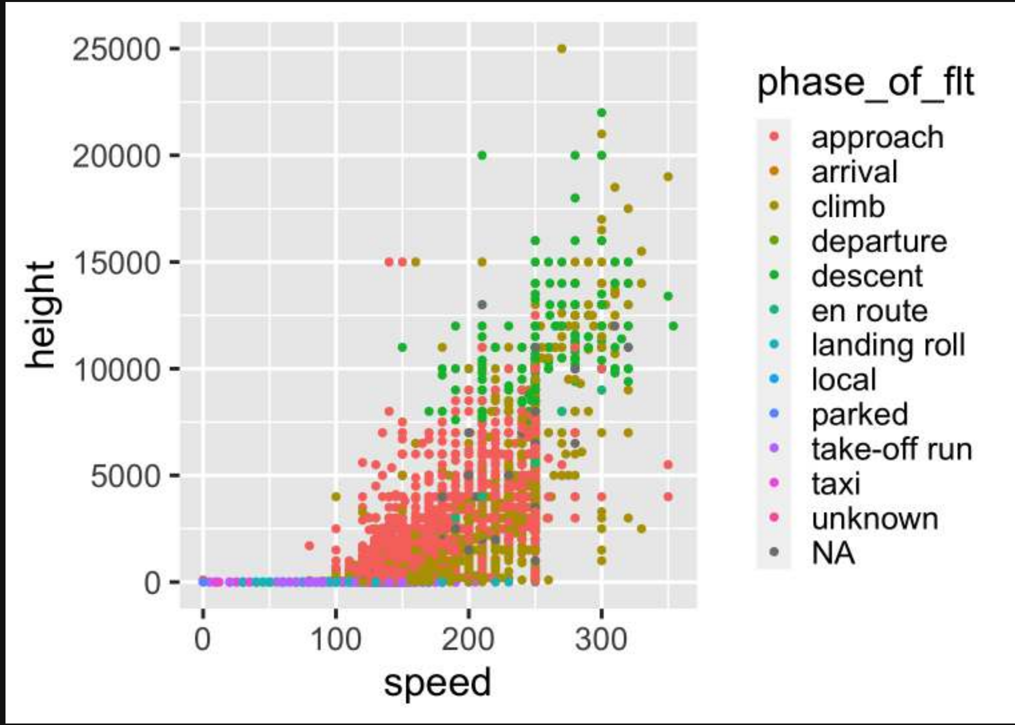
2) Use the `df` object and `group_by()`, `summarise()`, `count()`, and `%>%` to answer the following questions:

- Create a summary data frame that contains the mean `height` for each different time of day.
- Create a summary data frame that contains the maximum `cost_repairs_infl_adj` for each year.
- Which *month* has had the greatest number of reported incidents?
- Which *year* has had the greatest number of reported incidents?

Data Viz: geom_point()

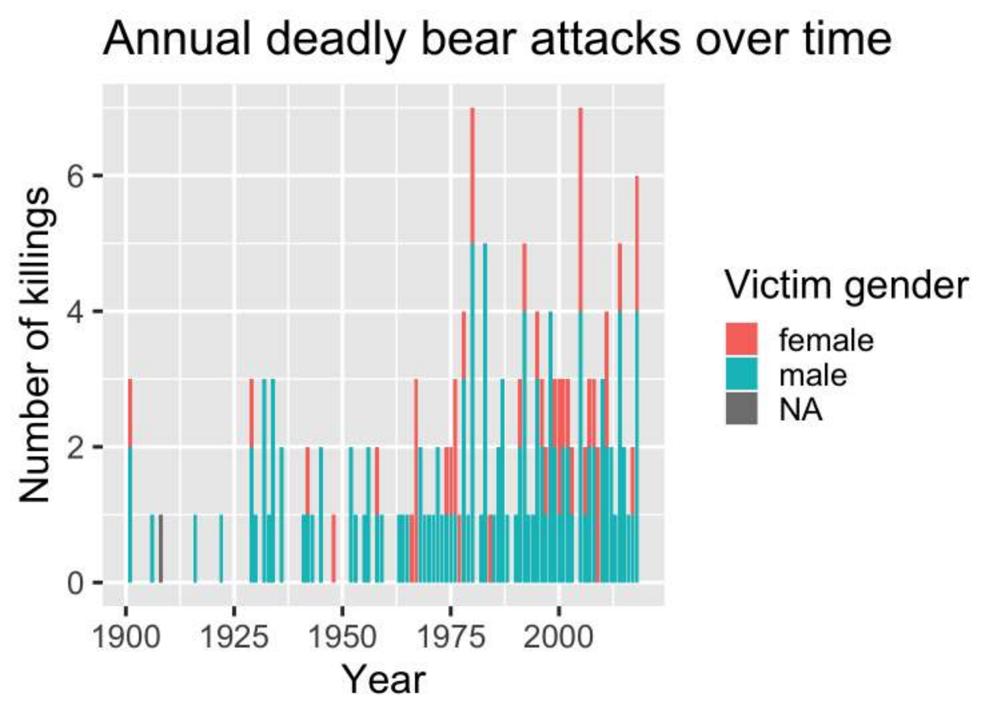
Use the birds data frame to create the following plots

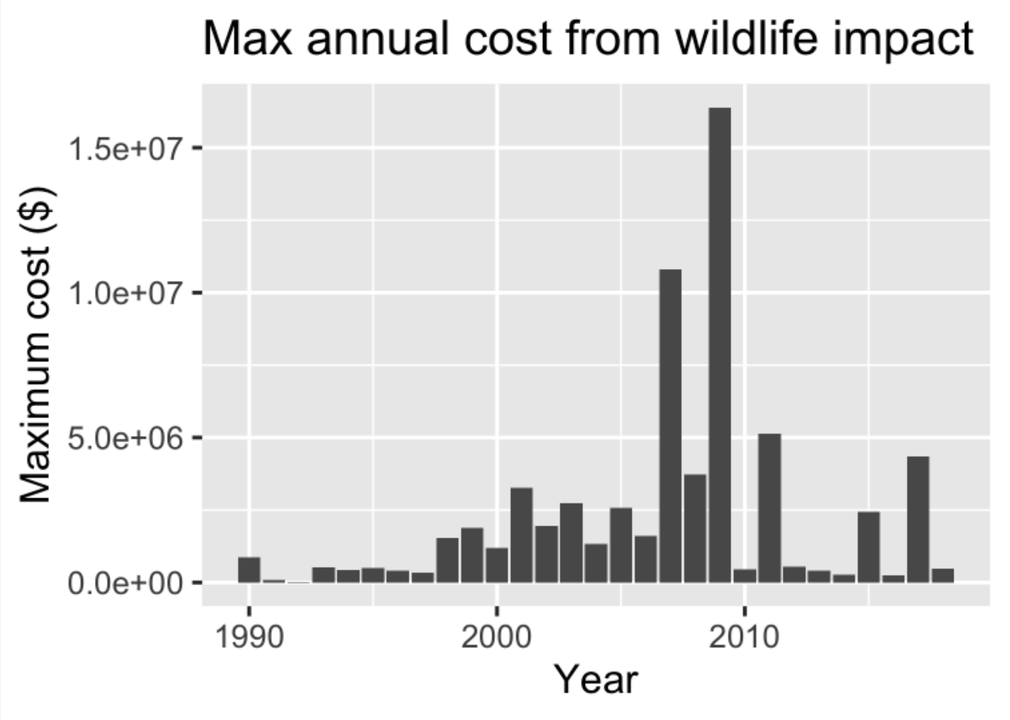




Data Viz: `geom_col()`

Use the `bears` and `birds` data frame to create the following plots





Writing Data Functions 1

```
my_subset <- function(df, condition, cols)
```

Returns a subset of `df` by filtering the rows based on `condition` and only includes the select `cols`. Example:

```
nycflights13::flights %>%
  my_subset(
    condition = month == 12,
    cols = c("carrier", "flight")
  )
```

```
#> # A tibble: 5 × 2
#>   carrier flight
#>   <chr>    <int>
#> 1 B6         745
#> 2 B6         839
#> 3 US        1895
#> 4 UA        1487
```

```
count_p <- function(df, group)
```

Returns a summary data frame of the count of rows in `df` by `group` as well as the percentage of those counts.

```
nycflights13::flights %>%
  count_p(carrier)
```

```
#> # A tibble: 6 × 3
#>   carrier     n      p
#>   <chr>   <int>  <dbl>
#> 1 UA      58665 0.174
#> 2 B6      54635 0.162
#> 3 EV      54173 0.161
#> 4 DL      48110 0.143
#> 5 AA      32729 0.0972
#> 6 MQ      26397 0.0784
```

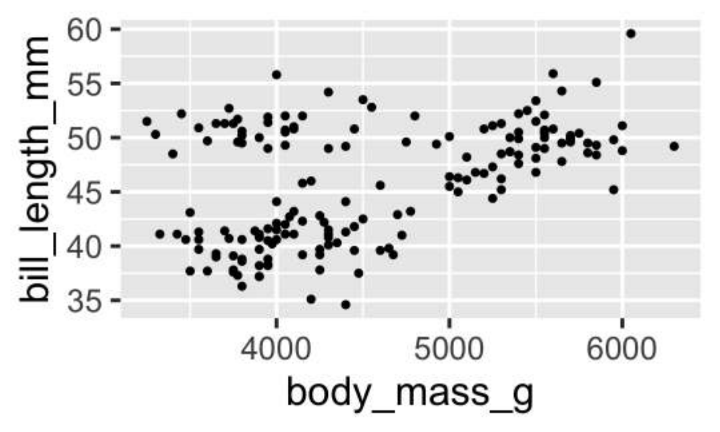
Writing Data Functions 2

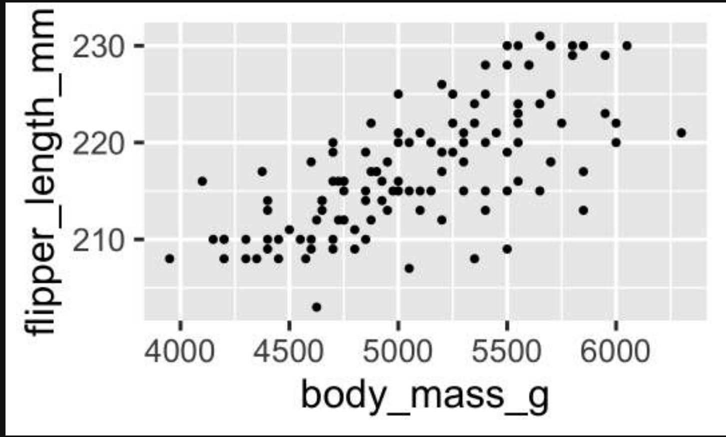
Write the function `filtered_scatter` which plots a scatterplot based on a condition, then use it for the two examples below.

```
filtered_scatter <- function(df, condition, x, y)
```

```
filtered_scatter(
  penguins, sex == "male",
  x = body_mass_g, y = bill_length_mm)
```

```
filtered_scatter(
  penguins, species == "Gentoo",
  x = body_mass_g, y = flipper_length_mm)
```





Monte Carlo: Coins & Dice

Using the `sample()` function, conduct a monte carlo simulation to estimate the answers to these questions:

- If you flipped a coin 3 times in a row, what is the probability that you'll get three "tails" in a row?
- If you rolled 2 dice, what is the probability that you'll get "snake-eyes" (two 1's)?
- If you rolled 2 dice, what is the probability that you'll get an outcome that sums to 8?

Monte Carlo: Cards

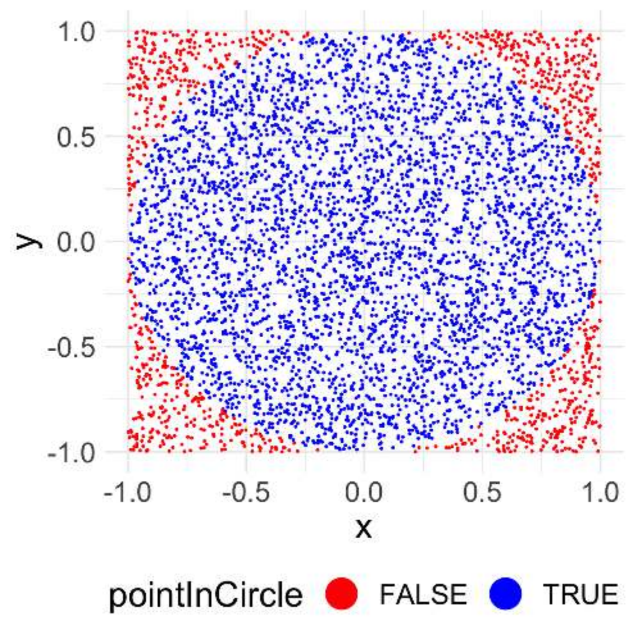
Use the `sample()` function and a monte carlo simulation to estimate the answers to these questions:

- What are the odds that four cards drawn from a 52-card deck will have the same suit?
- What are the odds that five cards drawn from a 52-card deck will sum to a prime number?
- Aces = 1
- Jack = 10
- Queen = 10
- King = 10

Hint: use `isPrime()` to help:

```
isPrime <- function(n) {
    if (n == 2) { return(TRUE) }
    for (i in seq(2, n-1)) {
        if (n %% i == 0) {
            return(FALSE)
        }
    }
    return(TRUE)
}
```

Monte Carlo: Estimate π



$$\pi = 4\left(\frac{A_{circle}}{A_{square}}\right)$$

1. Create a tibble with variables `x` and `y` that each contain 10,000 random points between -1 and 1, representing the (x, y) coordinates to a random point inside a square of side length 2 centered at `(x, y) = (0, 0)`. **Hint**: use `runif()`

2. Create a new column, `radius`, that is equal to the distance to each `(x, y)` point from the center of the square.

3. Create a new column, `pointInCircle`, that is TRUE if the point lies *within* the circle inscribed in the square, and FALSE otherwise.

4. Create the scatterplot on the left (don't worry about the precise colors, dimensions, etc.).

5. Estimate π by multiplying 4 times the ratio of points inside the circle to the total number of points

Monte Carlo: Monte Hall Problem

1. You choose door 1, 2, or 3
2. One door is removed
3. Should you swap doors?

In this simulation, the prize is always behind door #1:

- If you choose door #1, you must KEEP it to win.
- If you choose door #2 or #3, you must SWAP to win.

1) Create the tibble, `choices`, with two variables:

- `door` contains the first door chosen (`1`, `2`, or `3`)
- `swap` contains a logical (`TRUE` or `FALSE`) for whether the contestant swaps doors. **Hint**: use `sample()`

2) Create a new tibble, `wins`, which contains only the rows from `choices` that resulted in a win.

3) Compute the percentage of times the contestant won after swapping doors.