Final Exam Review

EMSE 4574: Intro to Programming for Analytics

John Paul Helveston

December 10, 2020

Download the `p4a-final-review.zip` file for class today

Link in `slack/classroom`

Things to review

- All lecture slides
  (esp. exercises covered in class)
- All HWs and quizzes
- Memorize syntax
  (functions, test functions, tidyverse)
- Practice your weaknesses!
  (where did you struggle on a quiz or hw?)

Things **not** on the final

- Python
- Monte Carlo Methods

Format

- Similar to midterm (I'll email you a PDF)
- You will have **100** minutes (1 hr, 40 min)
- You'll be working in a templated .Rmd file

Final Exam Review

1. R basics

2. Data wrangling

3. Data visualization

Final Exam Review

1. R basics
2. Data wrangling
3. Data visualization

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

```r
typeof(3L)
```

```
#> [1] "integer"
```

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
```

```
sum(c(TRUE, FALSE, TRUE))
```

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

Be careful of accidental strings

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

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

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

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

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`"

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

```
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:

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

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

Test function:

```r
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()`

```r
addTwo <- function(n1, n2) {
    return(n1 + n2)
}
```

A **bad** test function

```r
test_addTwo <- function() {
    cat("Testing addTwo()...")
    stopifnot(addTwo(0.1, 0.2) == 0.3)
    cat("Passed!\n")
}

test_addTwo()
```

```
#> Testing addTwo()...
```

A **good** test function

```r
test_addTwo <- function() {
    cat("Testing addTwo()...")
    stopifnot(almostEqual(
      addTwo(0.1, 0.2), 0.3))
    cat("Passed!\n")
}

test_addTwo()
```

```
#> Testing addTwo()...Passed!
```

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

1. Build the sequence
2. Iterate over it

```r
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

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

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

The universal vector generator: `c()`

Numeric vectors

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

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

Character vectors

```r
y <- c('one', 'two',
'three')
y
```

```
#> [1] "one"   "two"
"three"
```

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*

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

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

Works on custom functions too:

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

isEven(x)
```

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

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

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

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

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

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

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

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

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

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

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

Final Exam Review

1. R basics

2. Data wrangling

3. Data visualization

Steps to importing external data files

1. Open your .RProj file! (not the .R file)

2. Use `here()` to create the file paths

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

```
#> [1] "/Users/jhelvy/gh/0gw/P4A/p4a_planning/class/16-final-
review/data/data.csv"
```

3. Use `read_csv()` to import the data

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

Previewing all variables

```
glimpse(msleep)
```

```
#> Rows: 83
#> Columns: 11
#> $ name         <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Great…
#> $ genus        <chr> "Acinonyx", "Aotus", "Aplodontia", "Blarina", "Bos…
#> $ vore         <chr> "carni", "omni", "herbi", "omni", "herbi", "herbi"…
#> $ order        <chr> "Carnivora", "Primates", "Rodentia", "Soricomorpha…
#> $ conservation <chr> "lc", NA, "nt", "lc", "domesticated", NA, "vu", NA…
#> $ sleep_total  <dbl> 12.1, 17.0, 14.4, 14.9, 4.0, 14.4, 8.7, 7.0, 10.1,…
#> $ sleep_rem    <dbl> NA, 1.8, 2.4, 2.3, 0.7, 2.2, 1.4, NA, 2.9, NA, 0.6…
#> $ sleep_cycle  <dbl> NA, NA, NA, 0.1333333, 0.6666667, 0.7666667, 0.383…
#> $ awake        <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, …
#> $ brainwt      <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.0…
#> $ bodywt       <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.49…
```

Previewing *first* 6 rows

```
head(msleep)
```

```
#> # A tibble: 6 x 11
#>   name  genus vore  order conservation sleep_total sleep_rem sleep_cycle
#>   <chr> <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl>
#> 1 Chee… Acin… carni Carn… lc                  12.1      NA        NA
#> 2 Owl … Aotus omni  Prim… <NA>                17         1.8      NA
#> 3 Moun… Aplo… herbi Rode… nt                  14.4       2.4      NA
#> 4 Grea… Blar… omni  Sori… lc                  14.9       2.3       0.133
#> 5 Cow   Bos   herbi Arti… domesticated         4         0.7       0.667
#> 6 Thre… Brad… herbi Pilo… <NA>                14.4       2.2       0.767
#> # … with 3 more variables: awake <dbl>, brainwt <dbl>, bodywt <dbl>
```

Previewing *last* 6 rows

```
tail(msleep)
```

```
#> # A tibble: 6 x 11
#>   name  genus vore  order conservation sleep_total sleep_rem sleep_cycle
#>   <chr> <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl>
#> 1 Tenr… Tenr… omni  Afro… <NA>                15.6       2.3      NA
#> 2 Tree… Tupa… omni  Scan… <NA>                 8.9       2.6       0.233
#> 3 Bott… Turs… carni Ceta… <NA>                 5.2      NA        NA
#> 4 Genet Gene… carni Carn… <NA>                 6.3       1.3      NA
#> 5 Arct… Vulp… carni Carn… <NA>                12.5      NA        NA
#> 6 Red … Vulp… carni Carn… <NA>                 9.8       2.4       0.35
#> # … with 3 more variables: awake <dbl>, brainwt <dbl>, bodywt <dbl>
```

Slicing data frames: `$` and `[row, col]`

Data frame *columns* are vectors:

```
msleep$name
```

```
#>  [1] "Cheetah"                        "Owl monkey"
#>  [3] "Mountain beaver"                "Greater short-tailed shrew"
#>  [5] "Cow"                            "Three-toed sloth"
#>  [7] "Northern fur seal"              "Vesper mouse"
#>  [9] "Dog"                            "Roe deer"
#> [11] "Goat"                           "Guinea pig"
#> [13] "Grivet"                         "Chinchilla"
#> [15] "Star-nosed mole"                "African giant pouched rat"
#> [17] "Lesser short-tailed shrew"      "Long-nosed armadillo"
#> [19] "Tree hyrax"                     "North American Opossum"
#> [21] "Asian elephant"                 "Big brown bat"
#> [23] "Horse"                          "Donkey"
#> [25] "European hedgehog"              "Patas monkey"
#> [27] "Western american chipmunk"      "Domestic cat"
#> [29] "Galago"                         "Giraffe"
```

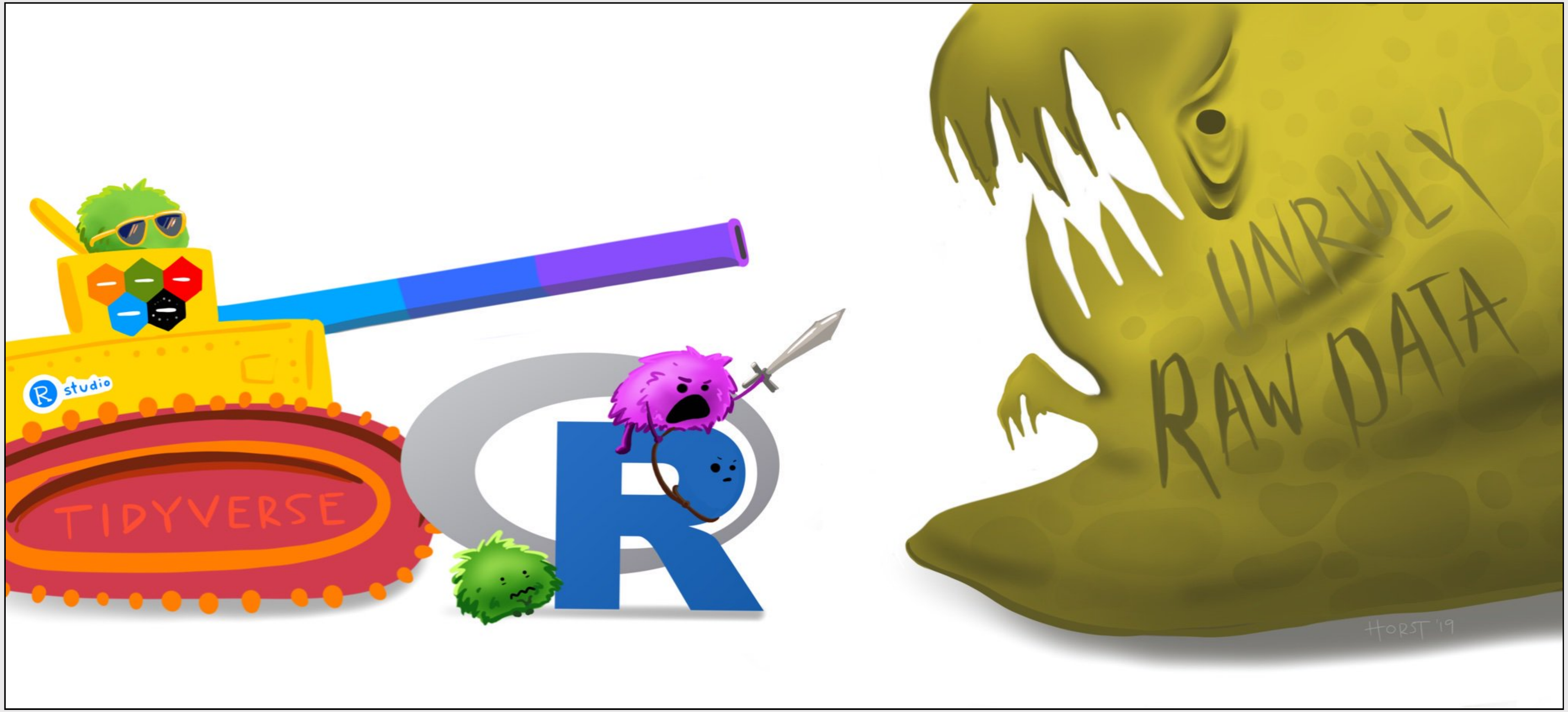
Slicing data frames: `$` and `[row, col]`

Data frame *rows* are observations:

```
msleep[1,]
```

```
#> # A tibble: 1 x 11
#>   name  genus vore  order conservation sleep_total sleep_rem sleep_cycle
#>   <chr> <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl>
#> 1 Chee… Acin… carni Carn… lc                  12.1        NA          NA
#> # … with 3 more variables: awake <dbl>, brainwt <dbl>, bodywt <dbl>
```

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



Art by Allison Horst

Know the main `dplyr` verbs

- `select()`: subset columns
- `filter()`: subset rows on conditions
- `arrange()`: sort results
- `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 how many rows (like `nrow()` by group)

Select columns with `select()`

```
birds %>%
    select(state, airport)
```

```
#> # A tibble: 56,978 x 2
#>    state airport
#>    <chr> <chr>
#>  1 FL    MIAMI INTL
#>  2 IN    INDIANAPOLIS INTL ARPT
#>  3 N/A   UNKNOWN
#>  4 N/A   UNKNOWN
#>  5 N/A   UNKNOWN
#>  6 FL    MIAMI INTL
#>  7 FL    ORLANDO INTL
#>  8 N/A   UNKNOWN
#>  9 N/A   UNKNOWN
#> 10 FL    FORT LAUDERDALE/HOLLYWOOD
INTL
#> # … with 56,968 more rows
```

Search by column name:
`starts_with()`, `ends_with()`, `contains()`

```
birds %>%
    select(starts_with('incident'))
```

```
#> # A tibble: 56,978 x 3
#>    incident_date
incident_month incident_year
#>    <dttm>
<dbl>         <dbl>
#>  1 2018-12-31 00:00:00
12          2018
#>  2 2018-12-29 00:00:00
12          2018
#>  3 2018-12-29 00:00:00
12          2018
```

Sort rows with `arrange()`

What is the highest recorded incident?

```
birds %>%
    select(height) %>%
    arrange(desc(height))
```

```
#> # A tibble: 56,978 x 1
#>     height
#>      <dbl>
#>  1   25000
#>  2   25000
#>  3   24300
#>  4   23000
#>  5   22000
#>  6   21000
#>  7   20000
#>  8   20000
#>  9   20000
#> 10   20000
```

Select rows with `filter()`

Common task: drop missing values

```
birds %>%
    select(height)
```

```
#> # A tibble: 56,978 x 1
#>    height
#>     <dbl>
#>  1    700
#>  2      0
#>  3     NA
#>  4     NA
#>  5     NA
#>  6     NA
#>  7    600
#>  8     NA
#>  9     NA
#> 10      0
#> # … with 56,968 more rows
```

```
birds %>%
    filter(! is.na(height)) %>%
    select(height)
```

```
#> # A tibble: 38,940 x 1
#>    height
#>     <dbl>
#>  1    700
#>  2      0
#>  3    600
#>  4      0
#>  5      0
#>  6      0
#>  7    500
#>  8    100
#>  9      0
#> 10   1000
```

Drop NA for mean(), min(), max()

What is the mean incident height?

```
birds %>%
    summarise(meanHeight = mean(height))
```

```
#> # A tibble: 1 x 1
#>   meanHeight
#>        <dbl>
#> 1         NA
```

```
birds %>%
    filter(! is.na(height)) %>%
    summarise(meanHeight = mean(height))
```

```
#> # A tibble: 1 x 1
#>   meanHeight
#>        <dbl>
#> 1       984.
```

Create new variables with `mutate()`

How many years ago did each attack occur?

```
bears %>%
    mutate(event_age = 2020 - year) %>%
    select(year, event_age)
```

```
#> # A tibble: 166 x 2
#>     year event_age
#>    <dbl>     <dbl>
#>  1  1901       119
#>  2  1901       119
#>  3  1901       119
#>  4  1906       114
#>  5  1908       112
#>  6  1916       104
#>  7  1922        98
#>  8  1929        91
#>  9  1929        91
#> 10  1929        91
```

mutate() vs. summarise()

What is the mean age of bear attack victims?

```
bears %>%
    filter(!is.na(age)) %>%
    mutate(mean_age = mean(age)) %>%
    select(age, mean_age)
```

```
#> # A tibble: 164 x 2
#>        age mean_age
#>      <dbl>    <dbl>
#>  1       3     36.1
#>  2       5     36.1
#>  3       7     36.1
#>  4      18     36.1
#>  5       1     36.1
#>  6      61     36.1
#>  7      60     36.1
#>  8       9     36.1
#>  9      52     36.1
```

```
bears %>%
    filter(!is.na(age)) %>%
    summarise(mean_age = mean(age))
```

```
#> # A tibble: 1 x 1
#>   mean_age
#>      <dbl>
#> 1     36.1
```

Grouped calculations

What is the mean age of bear attack victims **by gender**?

```
bears %>%
    filter(!is.na(age)) %>%
    group_by(gender) %>%
    mutate(mean_age = mean(age)) %>%
    select(age, mean_age)
```

```
#> # A tibble: 164 x 3
#> # Groups:   gender [3]
#>    gender   age mean_age
#>    <chr>  <dbl>    <dbl>
#>  1 female     3     31.2
#>  2 male       5     38.2
#>  3 male       7     38.2
#>  4 male      18     38.2
#>  5 <NA>       1      1
#>  6 male      61     38.2
#>  7 male      60     38.2
```

```
bears %>%
    filter(!is.na(age)) %>%
    group_by(gender) %>%
    summarise(mean_age = mean(age))
```

```
#> # A tibble: 3 x 2
#>   gender mean_age
#>   <chr>     <dbl>
#> 1 female     31.2
#> 2 male       38.2
#> 3 <NA>        1
```

Use `ifelse()` for conditionals

```
birds %>%
    mutate(speciesUnknown = ifelse(
      str_detect(str_to_lower(species), "unknown"), 1, 0)) %>%
    select(species, speciesUnknown)
```

```
#> # A tibble: 56,978 x 2
#>    species                 speciesUnknown
#>    <chr>                            <dbl>
#>  1 Unknown bird - large                 1
#>  2 Owls                                 0
#>  3 Short-eared owl                      0
#>  4 Southern lapwing                     0
#>  5 Lesser scaup                         0
#>  6 Unknown bird                         1
#>  7 Unknown bird - small                 1
#>  8 Eastern meadowlark                   0
#>  9 Red-winged blackbird                 0
#> 10 Cattle egret                         0
#> # … with 56,968 more rows
```

Count observations with `count()`

How many victims were killed by grizzly bears?

Method 1: `group_by()` + `summarise()`

```r
bears %>%
    group_by(grizzly) %>%
    summarise(n = n())
```

```
#> # A tibble: 2 x 2
#>   grizzly     n
#>     <dbl> <int>
#> 1       0   139
#> 2       1    27
```

Method 2: `count()`

```r
bears %>%
    count(grizzly)
```

```
#> # A tibble: 2 x 2
#>   grizzly     n
#>     <dbl> <int>
#> 1       0   139
#> 2       1    27
```

Final Exam Review

1. R basics

2. Data wrangling

3. Data visualization

Making plot layers with ggplot2

1. The data

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

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

Layer 1: The data

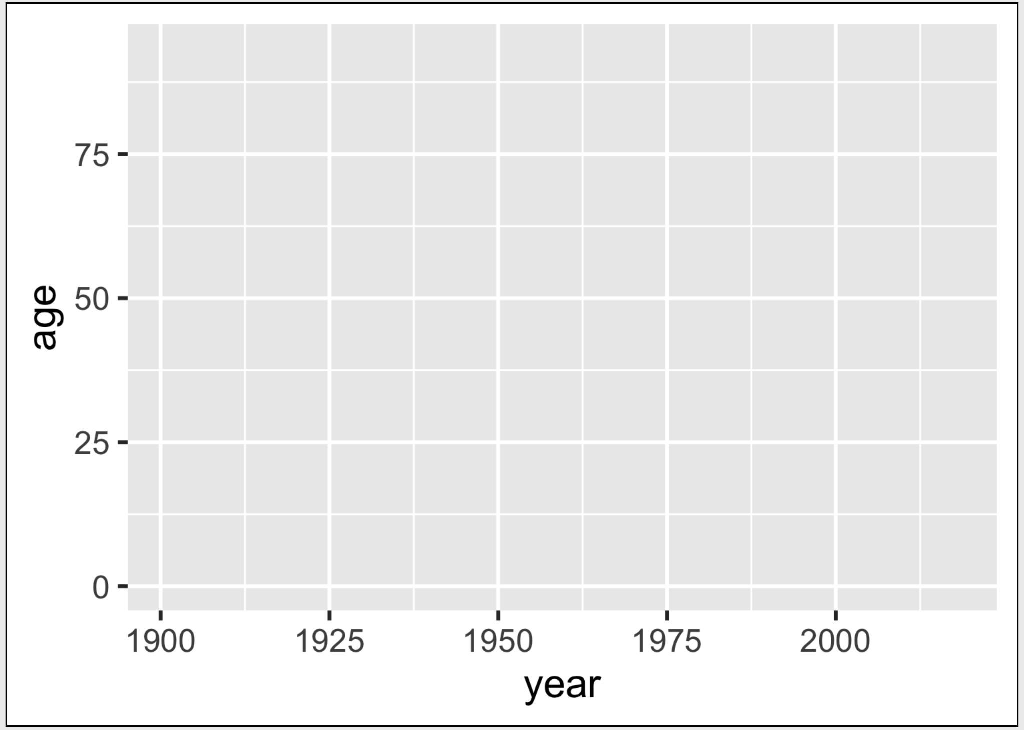
The `ggplot()` function initializes the plot with whatever data you're using

```
ggplot(data = bears)
```

Layer 2: The aesthetic mapping

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

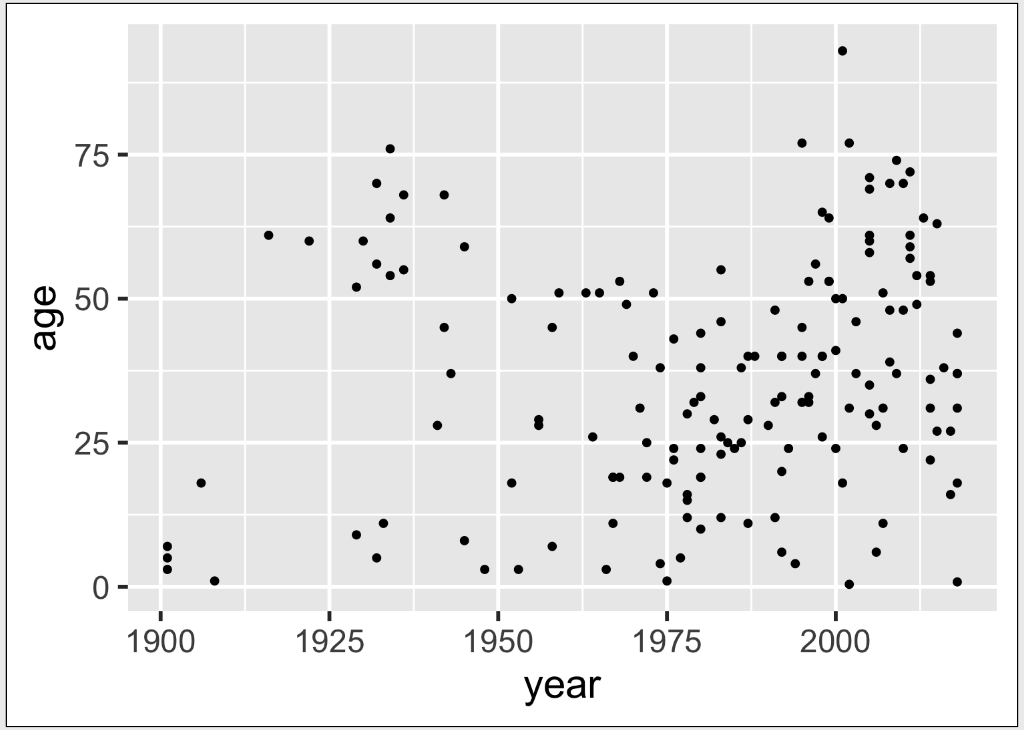
```
ggplot(data = bears,
       aes(x = year, y = age))
```



Layer 3: The geometries

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

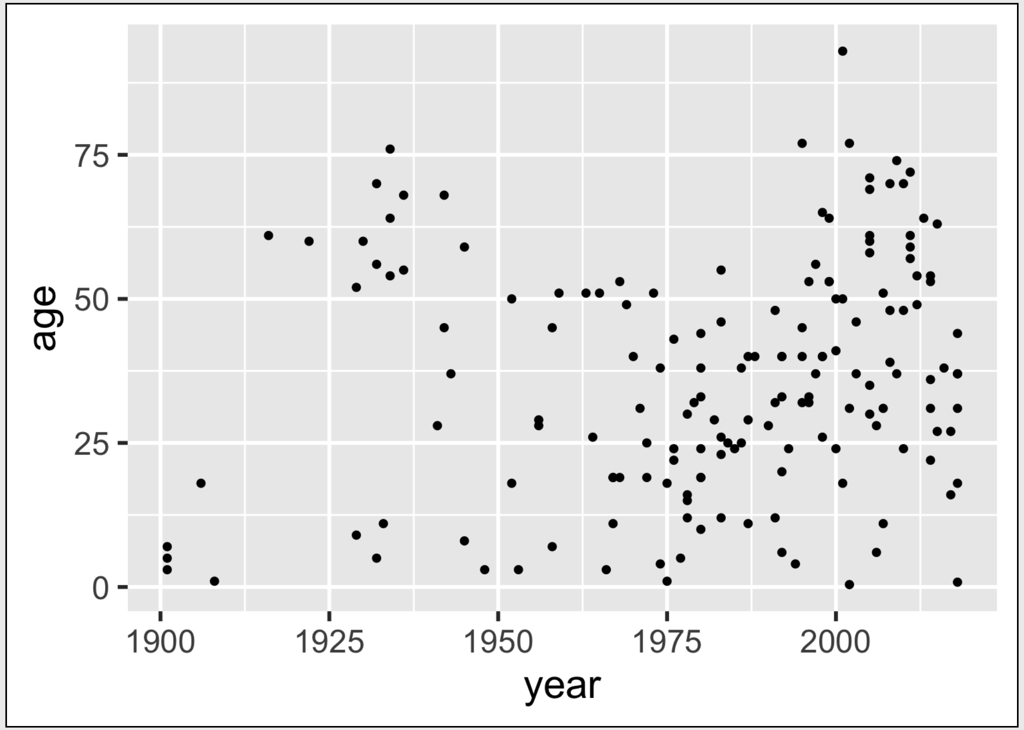
```
ggplot(data = bears,
       aes(x = year, y = age)) +
    geom_point()
```



Layer 3: The geometries

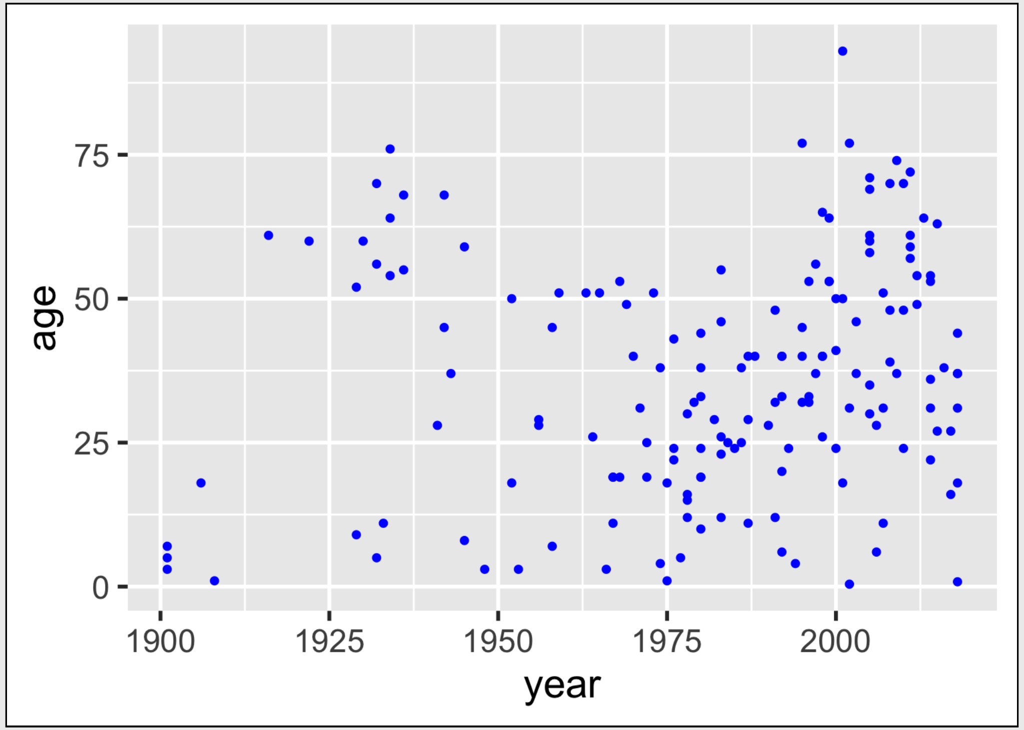
Can also map aesthetics within the geom

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



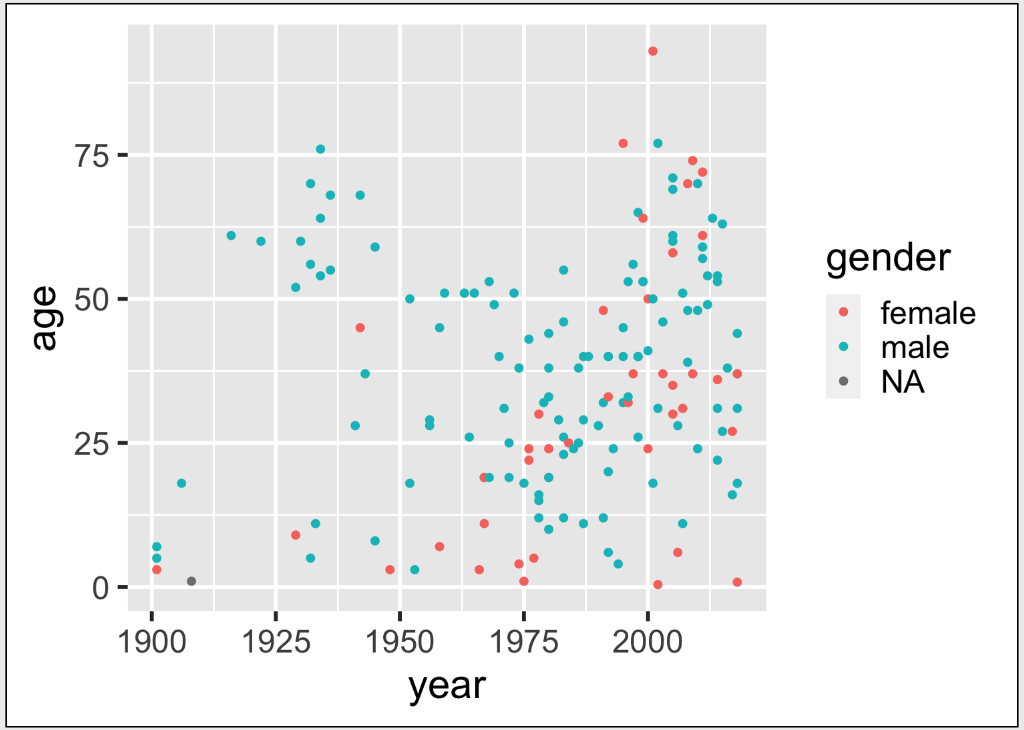
Change the point color

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



Change the point color **based on another variable**:

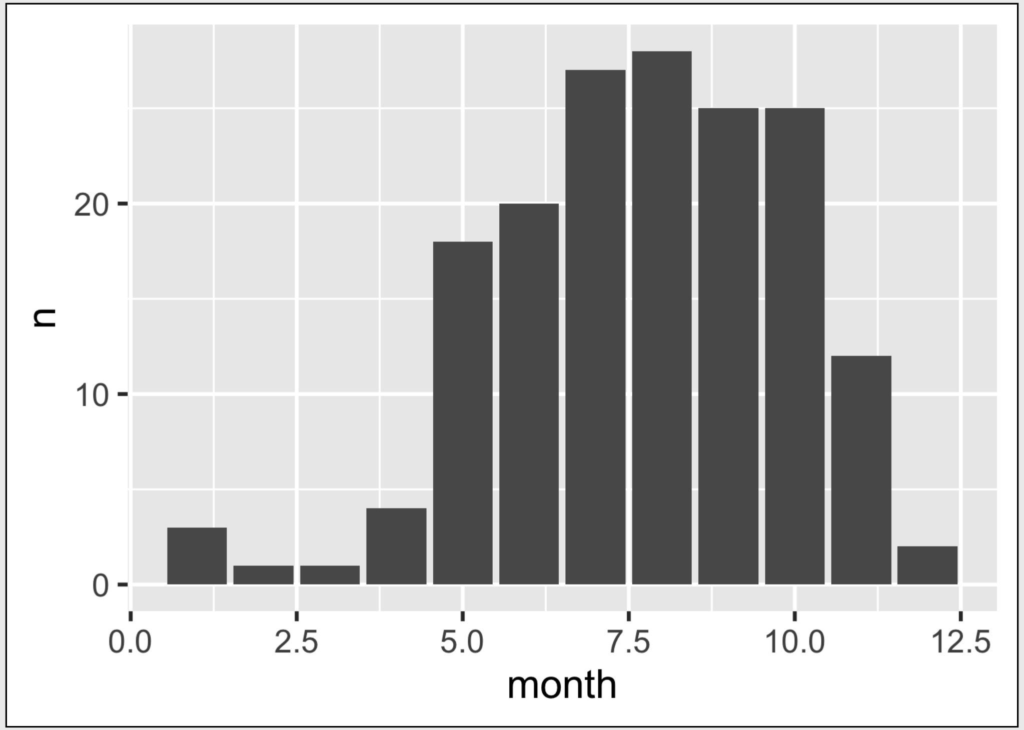
```
ggplot(data = bears,
       aes(x = year, y = age)) +
    geom_point(aes(color = gender))
```



Make bar charts with `geom_col()`

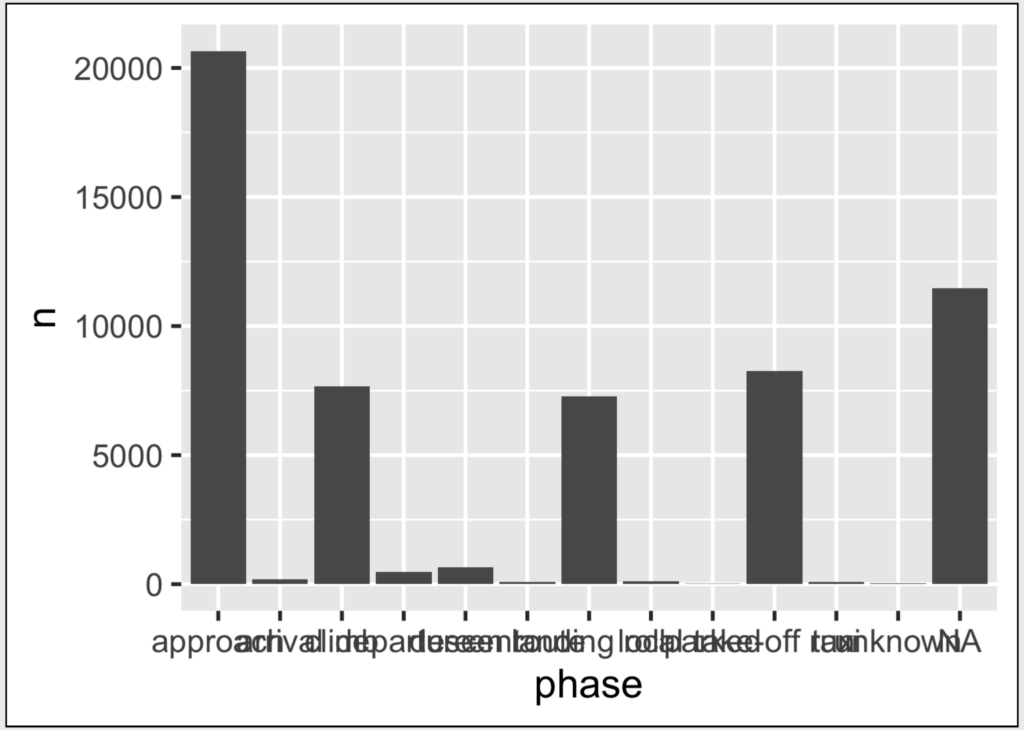
Piping directly into ggplot

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



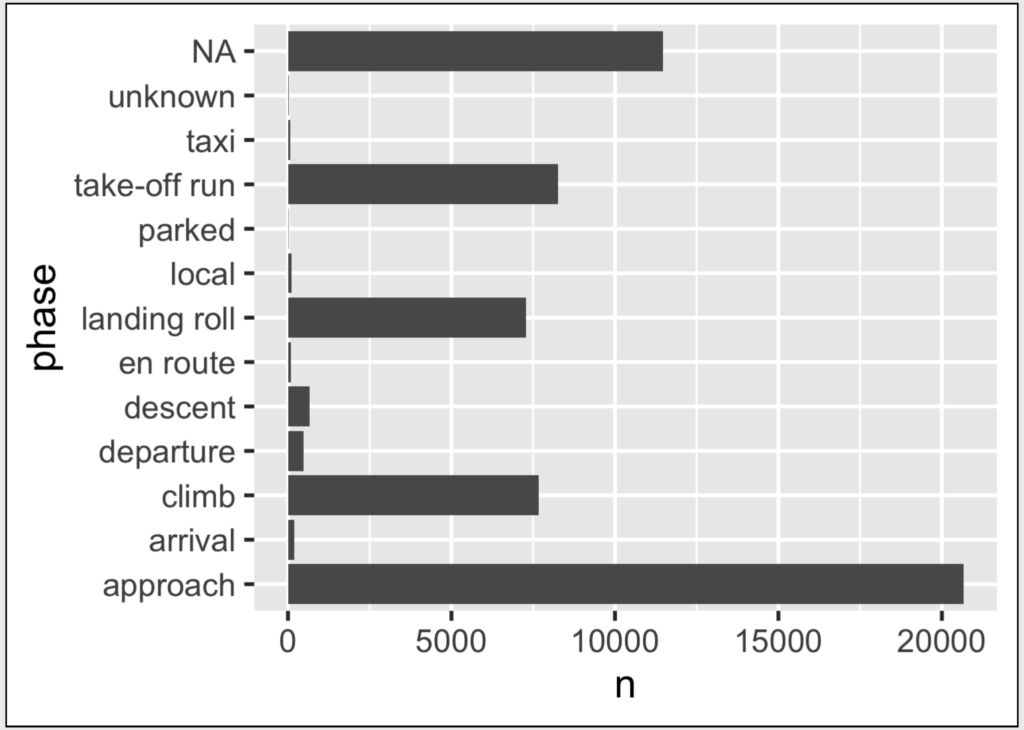
If you can't read the axes...flip them

```
birds %>%
  mutate(phase =
    str_to_lower(phase_of_flt)) %>%
  count(phase) %>%
  ggplot() +
  geom_col(aes(x = phase, y = n))
```



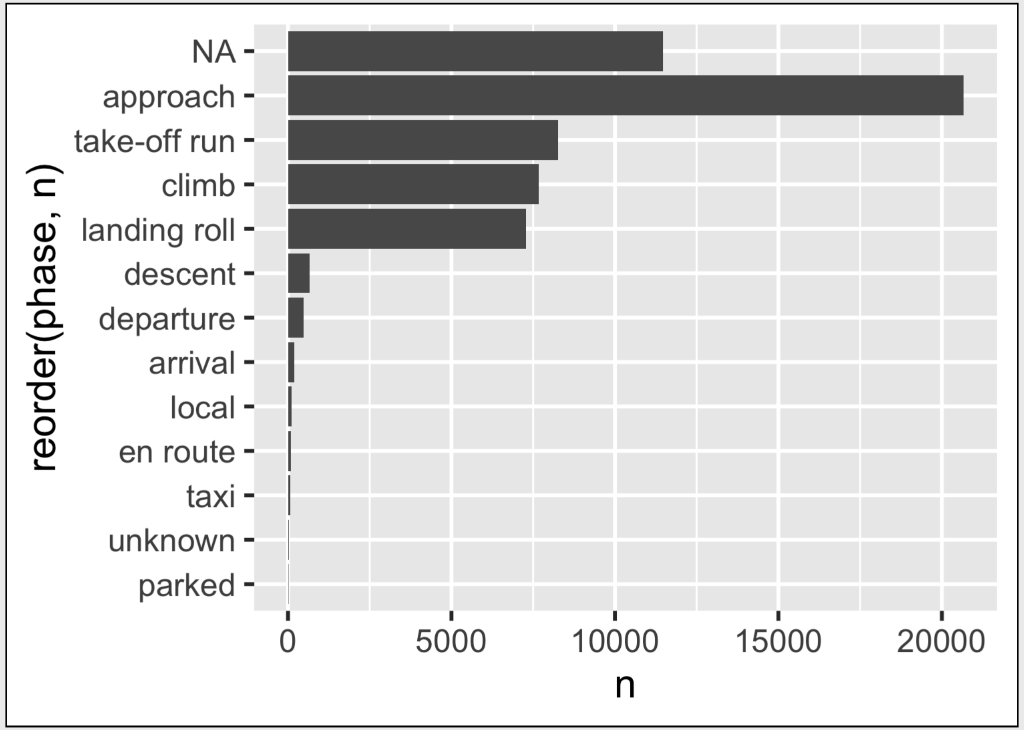
If you can't read the axes...flip them

```
birds %>%
  mutate(phase =
    str_to_lower(phase_of_flt)) %>%
  count(phase) %>%
  ggplot() +
  geom_col(aes(y = phase, x = n))
```



Use `reorder()` to sort an axis

```
birds %>%
  mutate(phase =
    str_to_lower(phase_of_flt)) %>%
  count(phase) %>%
  ggplot() +
  geom_col(aes(y = reorder(phase, n),
x = n))
```

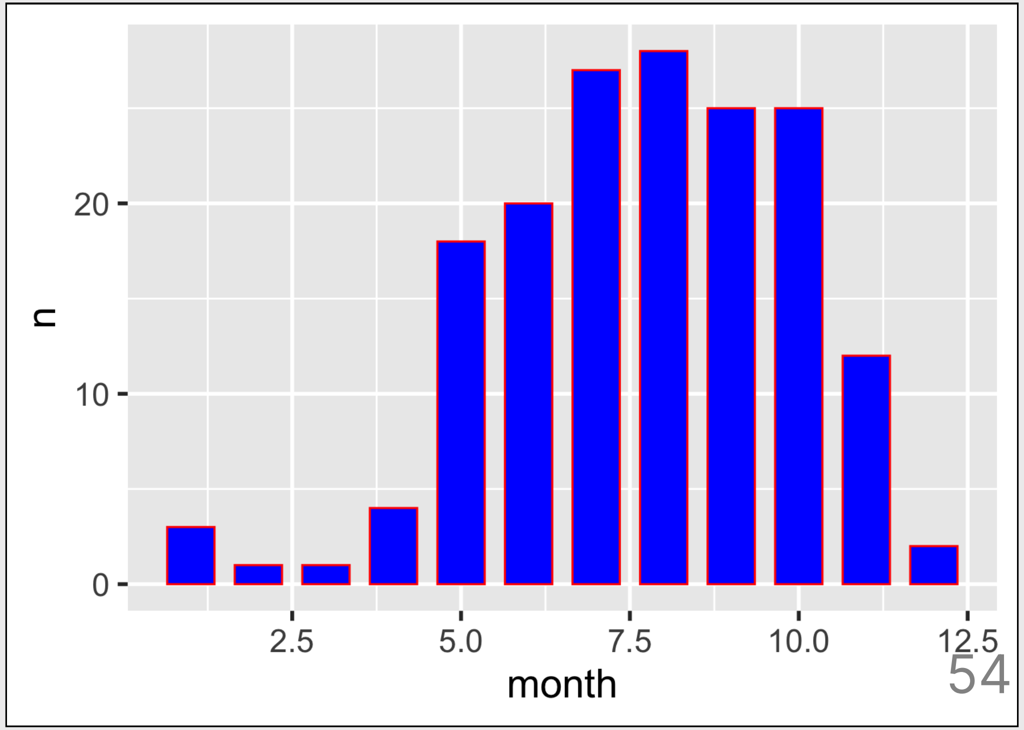


Change bar width: `width`

Change bar color: `fill`

Change bar outline: `color`

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



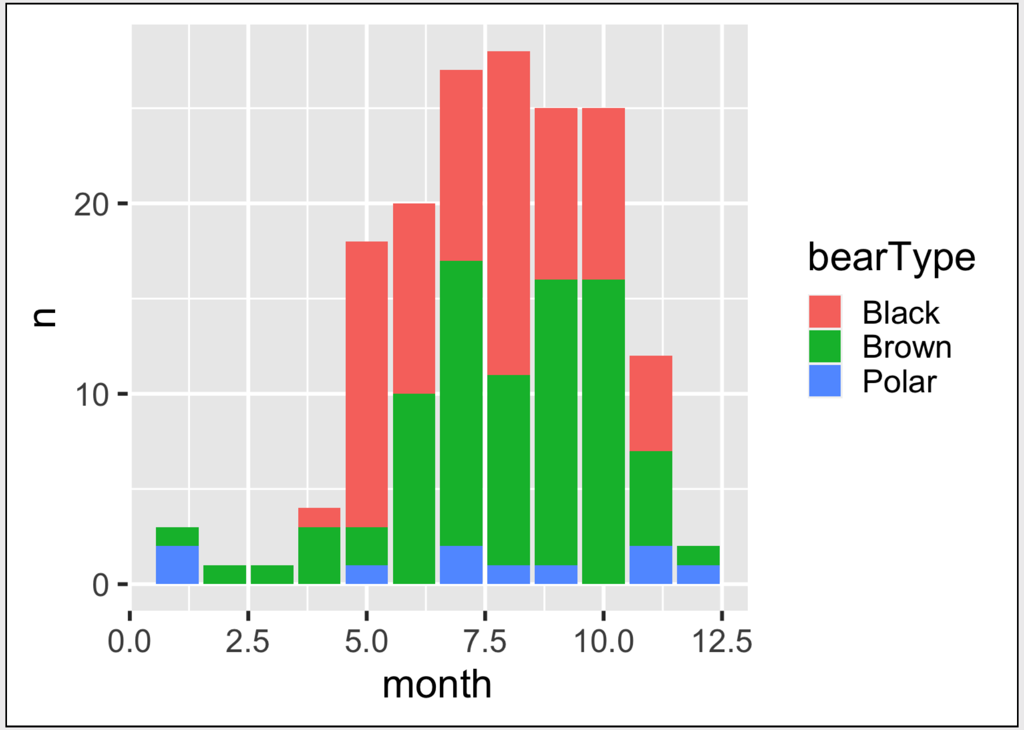
Map the `fill` to `bearType`

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

Note that I had to summarize the count by both `month` and `bearType`

```
bears %>%
  count(month, bearType)
```

```
#> # A tibble: 27 x 3
#>    month bearType     n
#>    <dbl> <chr>    <int>
#>  1     1 Brown        1
#>  2     1 Polar        2
#>  3     2 Brown        1
#>  4     3 Brown        1
#>  5     4 Black        1
#>  6     4 Brown        3
#>  7     5 Black       15
```

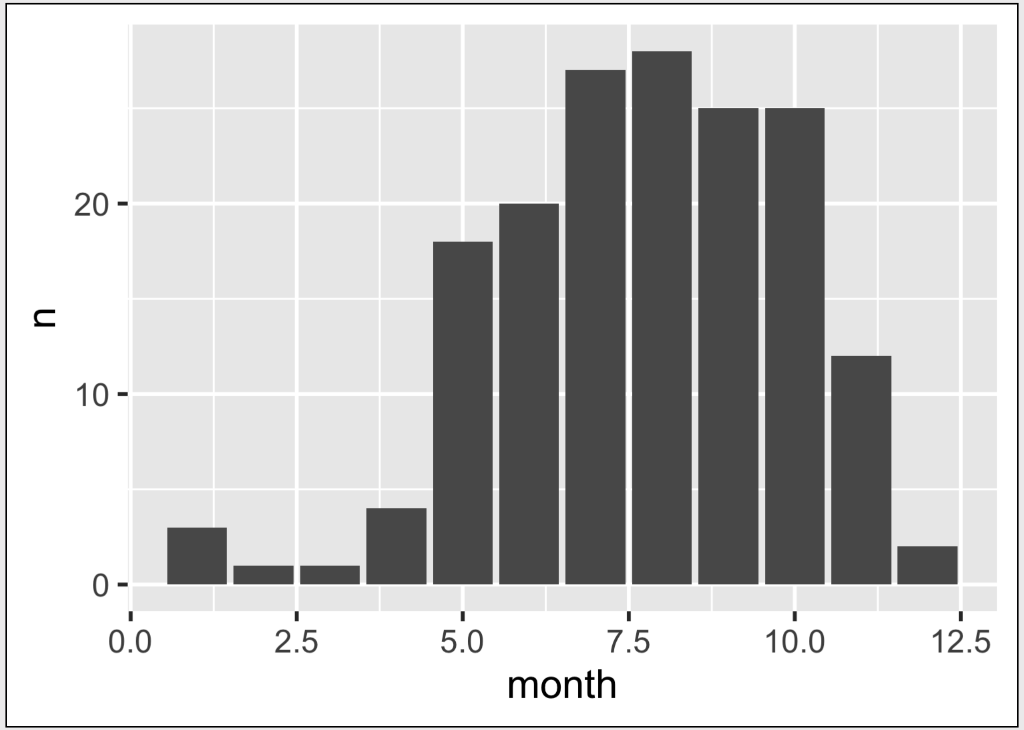


"Factors" = Categorical variables

By default, R makes numeric variables *continuous*

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

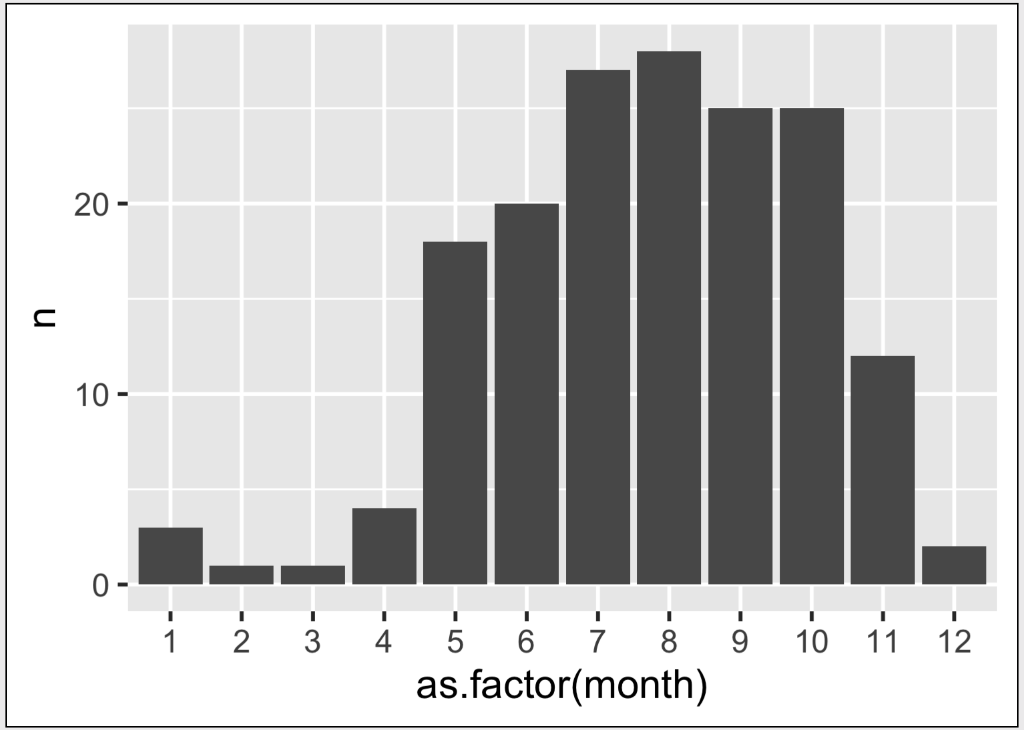
The variable `month` is a *number*



"Factors" = Categorical variables

You can make a continuous variable *categorical* using `as.factor()`

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

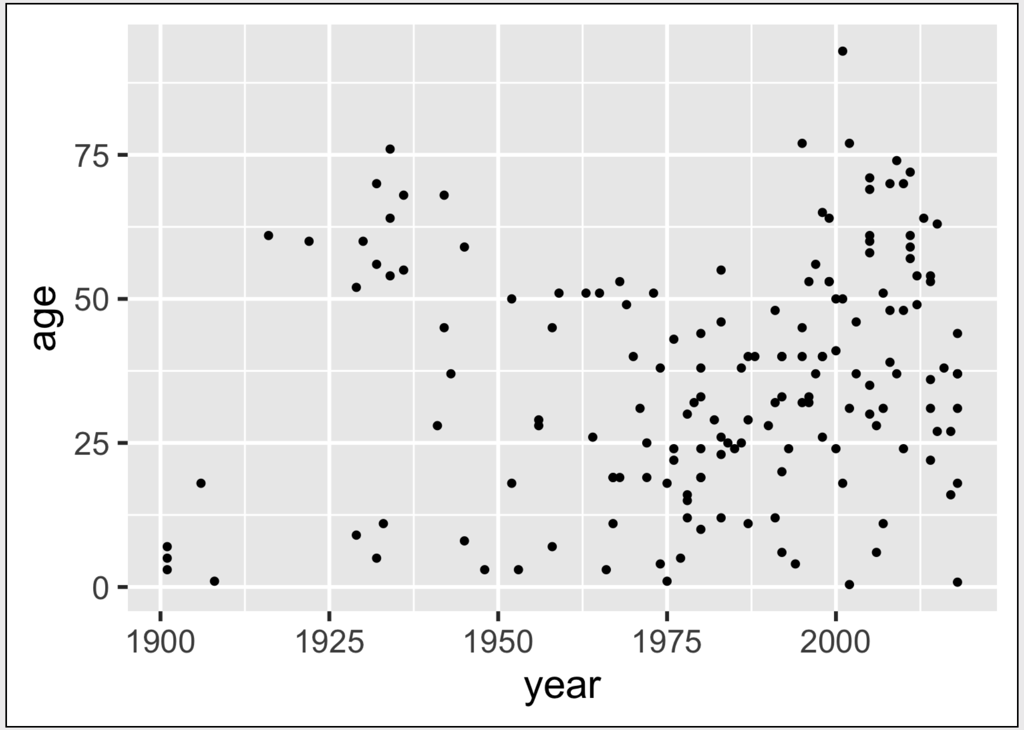


The variable `month` is a *factor*

Working with themes

Themes change *global* features of your plot, like the background color, grid lines, etc.

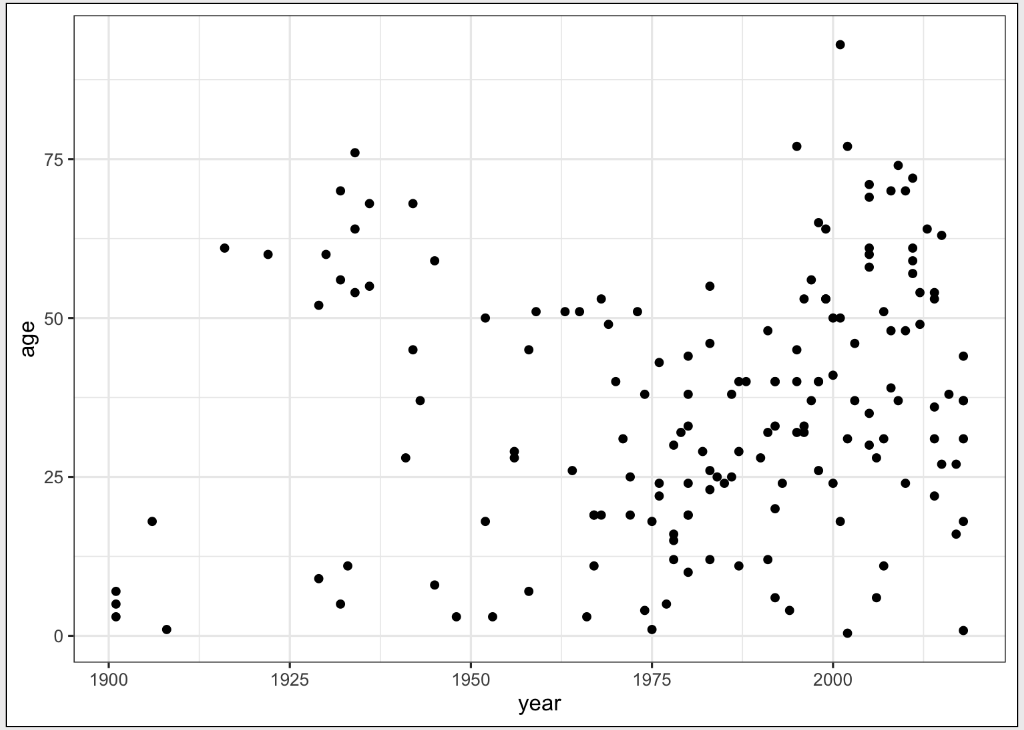
```
ggplot(data = bears,
       aes(x = year, y = age)) +
    geom_point()
```



Working with themes

Themes change *global* features of your plot, like the background color, grid lines, etc.

```
ggplot(data = bears,
       aes(x = year, y = age)) +
    geom_point() +
    theme_bw()
```



Save ggplots with `ggsave()`

First, assign the plot to an object name:

```
scatterPlot <- ggplot(data = bears) +
    geom_point(aes(x = year, y = age))
```

Then use `ggsave()` to save the plot:

```
ggsave(filename = here('plots', 'scatterPlot.png'),
       plot   = scatterPlot,
       width  = 6, # inches
       height = 4)
```

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.

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

```r
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)`

Data frames

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"`)

Think-Pair-Share

1) Use the `here()` and `read_csv()` functions to load the `birds.csv` file that is in the `data` folder. Name the data frame object `df`.

2) Use the `df` object to answer the following questions:

- How many rows and columns are in the data frame?
- What type of data is each column?
- Preview the different columns - what do you think this data is about? What might one row represent?
- How many unique airports are in the data frame?
- What is the earliest and latest observation in the data frame?
- What is the lowest and highest cost of any one repair in the data frame?

Think pair share: wildlife impacts data

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `birds.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?

Think pair share: wildlife impacts data

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `birds.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?

Think pair share: wildlife impacts data

1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `birds.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

Think pair share

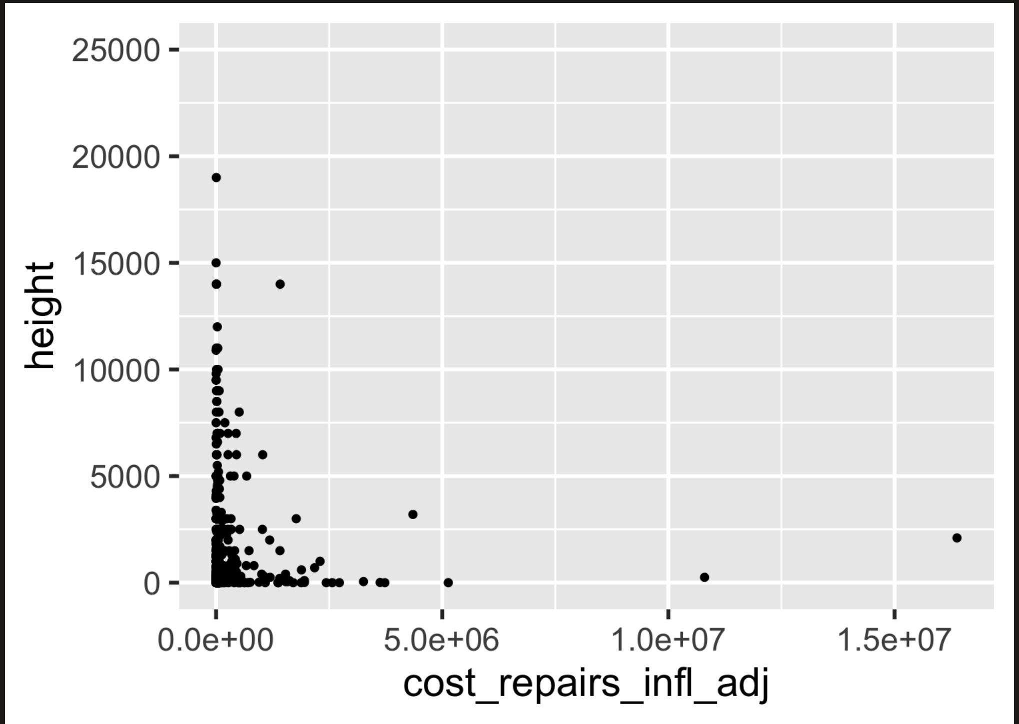
1) Create the data frame object `df` by using `here()` and `read_csv()` to load the `birds.csv` file in the `data` folder.

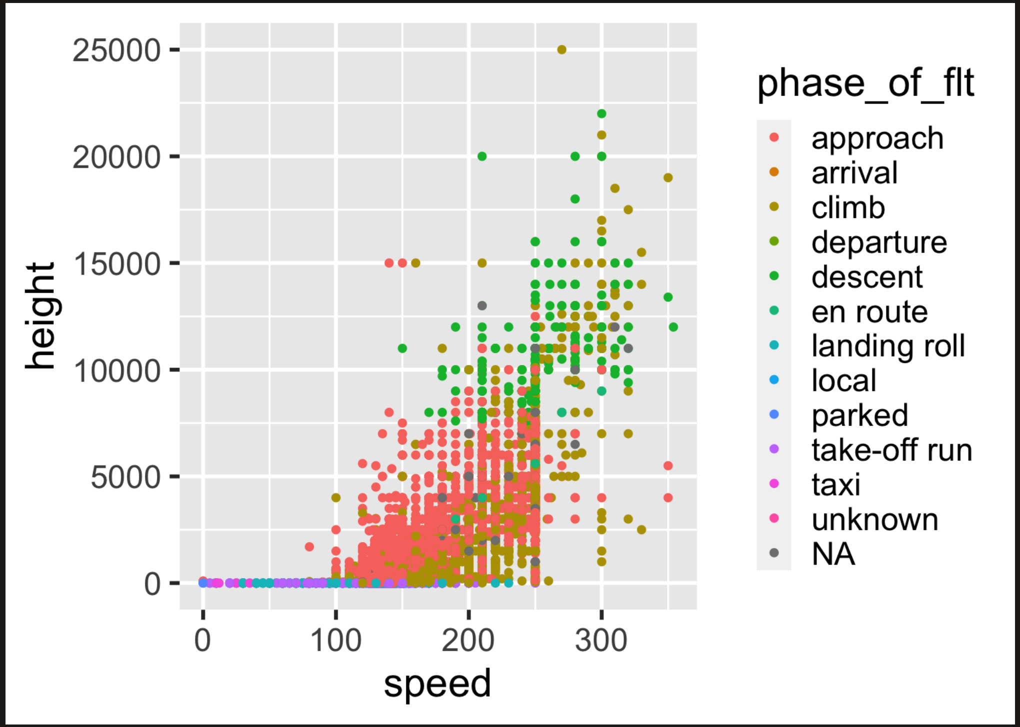
2) Use the `df` object and `group_by()`, `summarise()`, `count()`, and `%>%` to answer the following questions:

- 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?

Think pair share: `geom_point()`

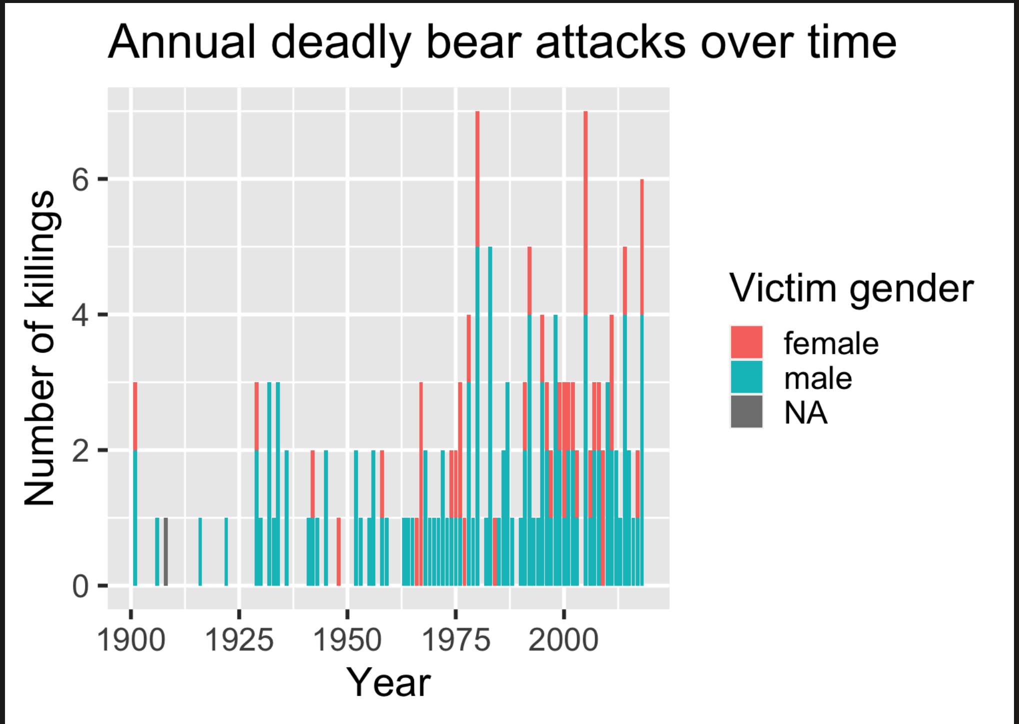
Use the `birds` data frame to create the following plots

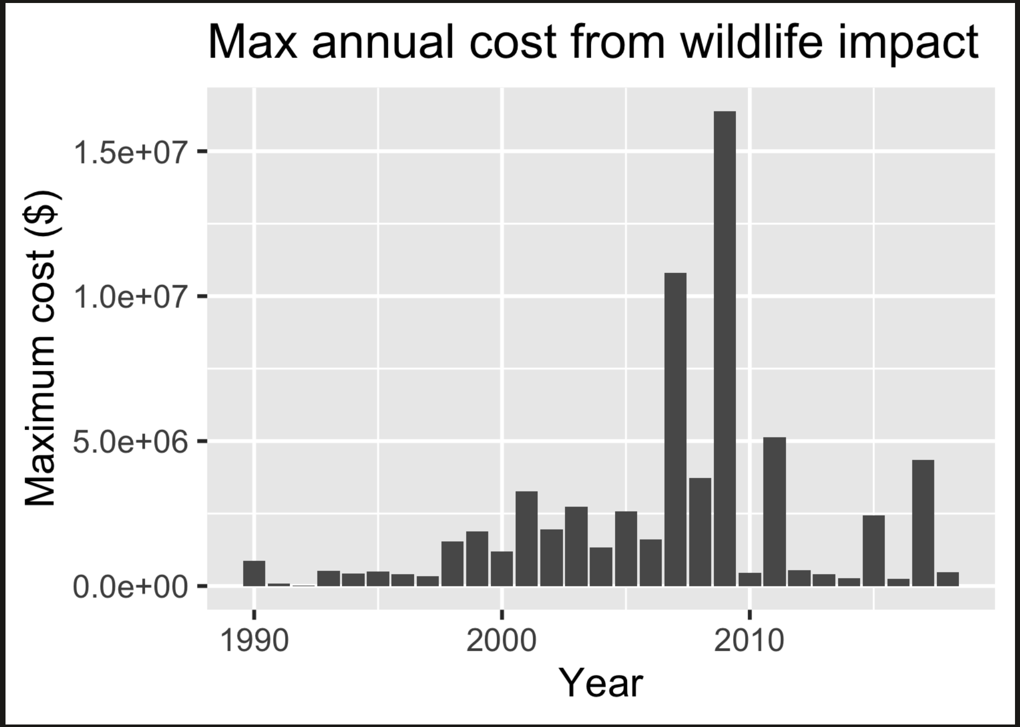




Think pair share: `geom_col()`

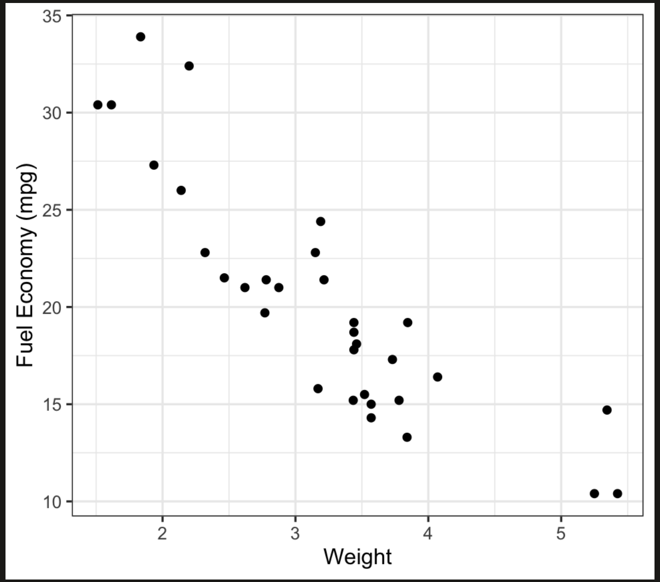
Use the `bears` and `birds` data frame to create the following plots

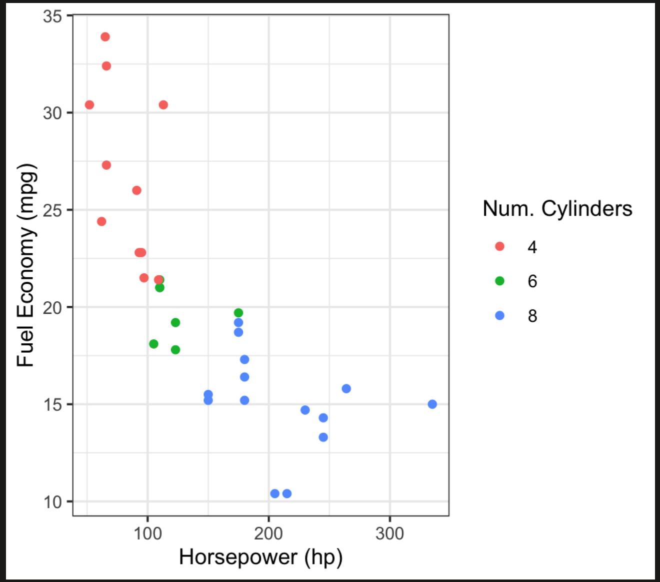


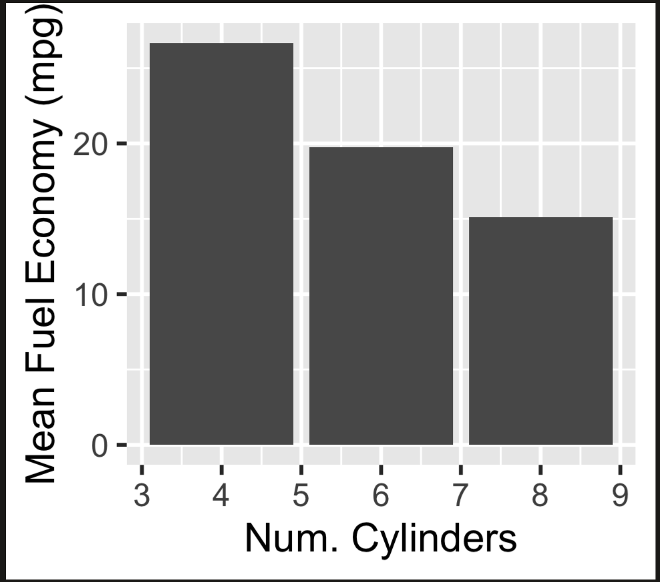


Extra practice 1

Use the `mtcars` data frame to create the following plots







Extra practice 2

Use the `mpg` data frame to create the following plot