Week 16: Final Exam Review

EMSE 6574 | John Paul Helveston | December 09, 2019

Download the `finalReview.zip` file for class today

Link in `slack/classroom`

Things to review

- Lesson pages (esp. tips sections)
- All lecture slides (esp. exercises covered in class)
- Practice your weaknesses, e.g. code tracing, writing functions, "tidyverse" functions, etc.
- Memorize syntax: functions, test functions, "tidyverse" functions, plotting with `ggplot2`, etc.
- Review hw, quiz, and exam solutions (on blackboard)

Data types

Numeric: Double vs. Integer

```
class(3.14)
```

```
## [1] "numeric"
```

```
class(3)
```

```
## [1] "numeric"
```

```
class(3L)
```

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

Logical:

```
x <- 2 == 2
x
```

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

Character / string:

```
s <- '3.14'
class(s)
```

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

```
s + 7
```

```
## Error in s + 7: non-numeric argument to binary operator
```

Checking / coercing data types

Check if number is integer *type*:

```
is.integer(3)
```

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

Check if number is integer *value*:

```
3 == as.integer(3)
```

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

Convert number to string:

```
as.character(3.14)
```

```
## [1] "3.14"
```

Convert number to logical:

```
as.logical(3.14)
```

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

Math with logical:

```
TRUE + TRUE
```

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

Logical & Relational operators

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

Logical: `&`, `|`, `!`

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

Relational: `==`, `!=`, `<`, `>`, `<=`, `>=`

Fill in the correct relational operators to make this statement return `TRUE`:

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

Number "chopping"

"Chopping" with orders of 10 - **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
```

The integer divide operator (`%/%`) "chops" a number and returns everything to the *left*

```
123456 %/% 1
```

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

```
123456 %/% 10
```

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

```
123456 %/% 100
```

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

Functions

```
FNAME <- function(ARG1, ARG2, ETC) {
    STATEMENT1
    STATEMENT2
    return(VALUE)
}
```

How I remember this:

"function name"	is a	function	of ()	that does...
`FNAME`	`<-`	`function`	`(ARG1, ARG2, ETC)`	`{}`

Example:

"function name"	is a	function	of ()	that does...
`mySqrt`	`<-`	`function`	`(n)`	`{ return(n^0.5) }`

Test Functions

```
testFNAME <- function() {
    cat('Testing FNAME()...')
    # Insert tests here
    stopifnot(FNAME(ARG1, ARG2, ETC) == TRUE)
    cat('Passed!')
}
```

`stopifnot()` stops the function if its argument is not `TRUE`.

Use `if` statements to filter function inputs

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` instead of causing an error.

Test cases:

- `isEvenNumber(42) == TRUE`
- `isEvenNumber(43) == FALSE`
- `isEvenNumber('not a number') == FALSE`

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

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

The `almostEqual()` function

```
almostEqual <- function(d1, d2) {
    epsilon = 0.00001
    return(abs(d1-d2) <= epsilon)
}
```

```
0.1 + 0.2 == 0.3
```

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

```
almostEqual(0.1 + 0.2, 0.3)
```

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

Iteration review: loop over a sequence

```
seq(5)
```

```
## [1] 1 2 3 4 5
```

`for` loop

```
for (i in seq(5)) {
    cat(i, '\n')
}
```

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

`while` loop

```
i <- 1
while (i <= 5) {
    cat(i, '\n')
    i <- i + 1
}
```

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

Searching for something in a sequence

```
seq(5)
```

```
## [1] 1 2 3 4 5
```

Count of **even** numbers in sequence:

`for` loop

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

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

`while` loop

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

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

`for` vs. `while`

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

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

Vectors with `c()`

```
# Numeric vectors
x <- c(1, 2, 3)
x
```

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

```
# Character vectors
y <- c('one', 'two', 'three')
y
```

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

```
# Logical vectors
y <- c(TRUE, FALSE, TRUE)
y
```

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

Vector coercion

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

```
## [1] "1" "2" "3"
```

Logicals become 1 and 0 as numbers:

```
c(1, 2, TRUE, FALSE)
```

```
## [1] 1 2 1 0
```

One float makes all integers floats:

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

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

Summation with loop vs. vector

```
x <- 1:10
x
```

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

Loop

```
total <- 0
for (item in x) {
    total <- total + item
}
total
```

```
## [1] 55
```

The `sum()` function:

```
sum(x)
```

```
## [1] 55
```

Vector slicing with brackets []

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

Return an element by index:

```
x[1]
```

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

Returns the last element:

```
x[length(x)]
```

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

Return sub-vector:

```
x[1:3]
```

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

```
x[c(2, 7)]
```

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

Negative integers remove elements:

```
x[-1] # Removes first element
```

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

```
x[-length(x)] # Removes last element
```

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

Comparing vectors

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

Check if *all* elements are same:

```
all(x == y)
```

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

```
all(x == z)
```

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

Check if *any* elements are same:

```
any(x == y)
```

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

```
any(x == z)
```

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

Creating a data frame with `tibble()`

```
library(tidyverse)
beatles <- tibble(
    firstName   = c("John", "Paul", "Ringo", "George"),
    lastName    = c("Lennon", "McCartney", "Starr", "Harrison"),
    instrument  = c("guitar", "bass", "drums", "guitar"),
    yearOfBirth = c(1940, 1942, 1940, 1943),
    deceased    = c(TRUE, FALSE, FALSE, TRUE)
)
beatles
```

```
## # A tibble: 4 x 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
```

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

Data frame *columns* are vectors:

```
beatles$firstName
```

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

Data frame *rows* are observations:

```
beatles[1,]
```

```
## # A tibble: 1 x 5
##   firstName lastName instrument yearOfBirth deceased
##   <chr>     <chr>    <chr>            <dbl> <lgl>
## 1 John      Lennon   guitar            1940 TRUE
```

Slicing data frames

Select specific elements using `[row, col]` indices:

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

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

```
beatles[1:2, c(2, 4)]
```

```
## # A tibble: 2 x 2
##   lastName  yearOfBirth
##   <chr>           <dbl>
## 1 Lennon           1940
## 2 McCartney        1942
```

Data frame dimensions

```
nrow(beatles) # Number of rows
```

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

```
ncol(beatles) # Number of columns
```

```
## [1] 5
```

```
dim(beatles)  # Number of rows and columns
```

```
## [1] 4 5
```

Data frame column names

```
names(beatles)
```

```
## [1] "firstName"   "lastName"    "instrument"  "yearOfBirth" "deceased"
```

```
colnames(beatles)
```

```
## [1] "firstName"   "lastName"    "instrument"  "yearOfBirth" "deceased"
```

Change column names:

```
colnames(beatles) <- c('one', 'two', 'three', 'four', 'five')
colnames(beatles)
```

```
## [1] "one"   "two"   "three" "four"  "five"
```

Combining data frames: `bind_cols()`:

```
names <- tibble(
    firstName = c("John", "Paul", "Ringo", "George"),
    lastName  = c("Lennon", "McCartney", "Starr", "Harrison"))
instruments <- tibble(
    instrument = c("guitar", "bass", "drums", "guitar"))
```

```
bind_cols(names, instruments)
```

```
## # A tibble: 4 x 3
##   firstName lastName  instrument
##   <chr>     <chr>     <chr>
## 1 John      Lennon    guitar
## 2 Paul      McCartney bass
## 3 Ringo     Starr     drums
## 4 George    Harrison  guitar
```

Combining data frames: `bind_rows()`:

```
members1 <- tibble(
    firstName = c("John", "Paul"),
    lastName  = c("Lennon", "McCartney"))
members2 <- tibble(
    firstName = c("Ringo", "George"),
    lastName  = c("Starr", "Harrison"))
```

```
bind_rows(members1, members2)
```

```
## # A tibble: 4 x 2
##   firstName lastName
##   <chr>     <chr>
## 1 John      Lennon
## 2 Paul      McCartney
## 3 Ringo     Starr
## 4 George    Harrison
```

Importing external data

Define path to the data with `file.path()`:

```
pathToData <- file.path('data', 'msleep.csv')
```

Read in the data with `read_csv()`:

```
library(readr)
msleep <- read_csv(pathToData)
```

Previewing data frames

```
glimpse(msleep)
```

```
## Observations: 83
## Variables: 11
## $ name         <chr> "Cheetah", "Owl monkey", "Mountain beaver", "Greate…
## $ 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, 0.3833…
## $ awake        <dbl> 11.9, 7.0, 9.6, 9.1, 20.0, 9.6, 15.3, 17.0, 13.9, 2…
## $ brainwt      <dbl> NA, 0.01550, NA, 0.00029, 0.42300, NA, NA, NA, 0.07…
## $ bodywt       <dbl> 50.000, 0.480, 1.350, 0.019, 600.000, 3.850, 20.490…
```

Previewing data frames

```
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>
```

```
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
```

5 minute break!

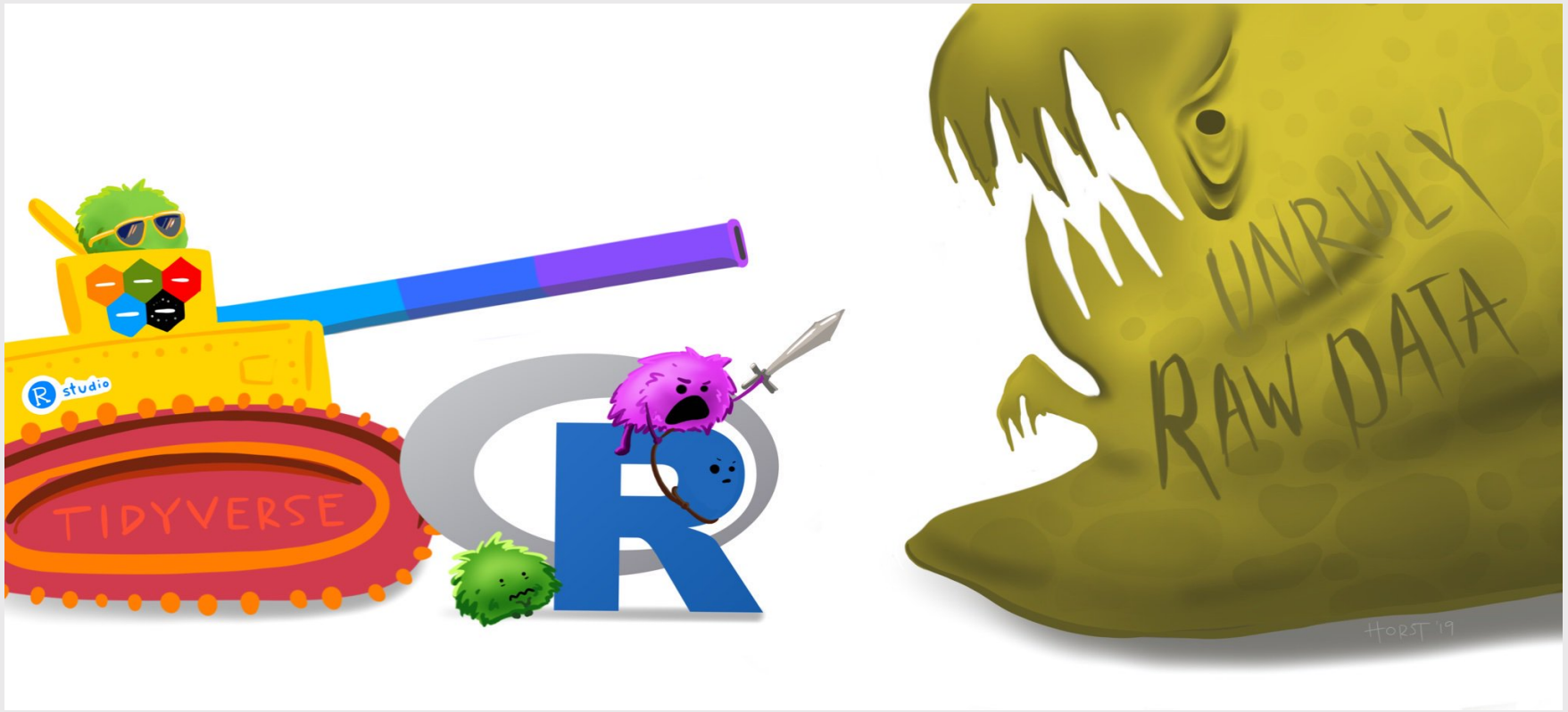
Stand up

Move around

Stretch!

The tidyverse

`stringr` + `dplyr` + `readr` + `ggplot2` + more = `tidyverse`



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 discrete rows

Select columns with `select()`

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

```
## # A tibble: 4 x 2
##   firstName lastName
##   <chr>     <chr>
## 1 John      Lennon
## 2 Paul      McCartney
## 3 Ringo     Starr
## 4 George    Harrison
```

Can also search column names:

- `contains()`
- `ends_with()`

```
beatles %>%
    select(contains('name'))
```

```
## # A tibble: 4 x 2
##   firstName lastName
##   <chr>     <chr>
## 1 John      Lennon
## 2 Paul      McCartney
## 3 Ringo     Starr
## 4 George    Harrison
```

Select columns with `select()`

Select all columns *except* certain ones with a `-` sign:

```
beatles %>%
    select(-firstName, -lastName)
```

```
## # A tibble: 4 x 3
##   instrument yearOfBirth deceased
##   <chr>            <dbl> <lgl>
## 1 guitar            1940 TRUE
## 2 bass              1942 FALSE
## 3 drums             1940 FALSE
## 4 guitar            1943 TRUE
```

Select rows with `filter()`

Filter the band members born after 1940

```
beatles %>%
    filter(yearOfBirth > 1940)
```

```
## # A tibble: 2 x 5
##   firstName lastName  instrument yearOfBirth deceased
##   <chr>     <chr>     <chr>            <dbl> <lgl>
## 1 Paul      McCartney bass              1942 FALSE
## 2 George    Harrison  guitar            1943 TRUE
```

Select rows with `filter()`

Filter the band members with 4-letter first names

```
beatles %>%
    filter(str_length(firstName) == 4)
```

```
## # A tibble: 2 x 5
##   firstName lastName  instrument yearOfBirth deceased
##   <chr>     <chr>     <chr>            <dbl> <lgl>
## 1 John      Lennon    guitar            1940 TRUE
## 2 Paul      McCartney bass              1942 FALSE
```

Sort rows with `arrange()`

Sort the data frame by year of birth:

```
beatles %>%
    arrange(yearOfBirth)
```

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

Sort rows with `arrange()`

Use the `desc()` function to sort in descending order:

```
beatles %>%
    arrange(desc(yearOfBirth))
```

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

Create new variables with `mutate()`

Compute the age of each band member from `yearOfBirth`

```
beatles %>%
    mutate(age = 2019 - yearOfBirth)
```

```
## # A tibble: 4 x 6
##   firstName lastName  instrument yearOfBirth deceased   age
##   <chr>     <chr>     <chr>            <dbl> <lgl>    <dbl>
## 1 John      Lennon    guitar            1940 TRUE        79
## 2 Paul      McCartney bass              1942 FALSE       77
## 3 Ringo     Starr     drums             1940 FALSE       79
## 4 George    Harrison  guitar            1943 TRUE        76
```

You can immediately use new variables

```
beatles %>%
    select(firstName, lastName, yearOfBirth) %>%
    mutate(
        age      = 2019 - yearOfBirth,
        meanAge  = mean(age),
        youngest = (age == min(age)),
        oldest   = (age == max(age)))
```

```
## # A tibble: 4 x 7
##   firstName lastName  yearOfBirth   age meanAge youngest oldest
##   <chr>     <chr>           <dbl> <dbl>   <dbl> <lgl>    <lgl>
## 1 John      Lennon           1940    79    77.8 FALSE    TRUE
## 2 Paul      McCartney        1942    77    77.8 FALSE    FALSE
## 3 Ringo     Starr            1940    79    77.8 FALSE    TRUE
## 4 George    Harrison         1943    76    77.8 TRUE     FALSE
```

if / else statements with `if_else()`

To create a new variable based on a condition, use `if_else()`

`if_else(<condition>, <if TRUE>, <else>)`

```
beatles %>%
    mutate(
        bornEvenOrOdd = if_else(yearOfBirth %% 2 == 0, 'even', 'odd'))
```

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

Create summary data frames

`mutate()` adds a new column:

```
beatles %>%
    mutate(age = 2019 - yearOfBirth) %>%
    mutate(mean_age = mean(age))
```

```
## # A tibble: 4 x 7
##   firstName lastName  instrument yearOfBirth a
##   <chr>     <chr>     <chr>            <dbl> <
## 1 John      Lennon    guitar            1940 T
## 2 Paul      McCartney bass              1942 F
## 3 Ringo     Starr     drums             1940 F
## 4 George    Harrison  guitar            1943 T
```

`summarise()` adds a new column and drops all others:

```
beatles %>%
    mutate(age = 2019 - yearOfBirth) %>%
    summarise(mean_age = mean(age))
```

```
## # A tibble: 1 x 1
##   mean_age
##      <dbl>
## 1     77.8
```

Operating on groups in the data

```
bands
```

```
## # A tibble: 9 x 5
##   firstName lastName  yearOfBirth deceased band
##   <chr>     <chr>           <dbl> <lgl>    <chr>
## 1 Melanie   Brown            1975 FALSE    spicegirls
## 2 Melanie   Chisholm         1974 FALSE    spicegirls
## 3 Emma      Bunton           1976 FALSE    spicegirls
## 4 Geri      Halliwell        1972 FALSE    spicegirls
## 5 Victoria  Beckham          1974 FALSE    spicegirls
## 6 John      Lennon           1940 TRUE     beatles
## 7 Paul      McCartney        1942 FALSE    beatles
## 8 Ringo     Starr            1940 FALSE    beatles
## 9 George    Harrison         1943 TRUE     beatles
```

Operating on groups in the data

```
bands %>%
    mutate(
        age = 2019 - yearOfBirth)
```

```
## # A tibble: 9 x 6
##   firstName lastName  yearOfBirth deceased band         age
##   <chr>     <chr>           <dbl> <lgl>    <chr>      <dbl>
## 1 Melanie   Brown            1975 FALSE    spicegirls    44
## 2 Melanie   Chisholm         1974 FALSE    spicegirls    45
## 3 Emma      Bunton           1976 FALSE    spicegirls    43
## 4 Geri      Halliwell        1972 FALSE    spicegirls    47
## 5 Victoria  Beckham          1974 FALSE    spicegirls    45
## 6 John      Lennon           1940 TRUE     beatles       79
## 7 Paul      McCartney        1942 FALSE    beatles       77
## 8 Ringo     Starr            1940 FALSE    beatles       79
## 9 George    Harrison         1943 TRUE     beatles       76
```

Operating on groups in the data

```
bands %>%
    mutate(
        age = 2019 - yearOfBirth,
        mean_age = mean(age))
```

```
## # A tibble: 9 x 7
##   firstName lastName  yearOfBirth deceased band         age mean_age
##   <chr>     <chr>           <dbl> <lgl>    <chr>      <dbl>    <dbl>
## 1 Melanie   Brown            1975 FALSE    spicegirls    44     59.4
## 2 Melanie   Chisholm         1974 FALSE    spicegirls    45     59.4
## 3 Emma      Bunton           1976 FALSE    spicegirls    43     59.4
## 4 Geri      Halliwell        1972 FALSE    spicegirls    47     59.4
## 5 Victoria  Beckham          1974 FALSE    spicegirls    45     59.4
## 6 John      Lennon           1940 TRUE     beatles       79     59.4
## 7 Paul      McCartney        1942 FALSE    beatles       77     59.4
## 8 Ringo     Starr            1940 FALSE    beatles       79     59.4
## 9 George    Harrison         1943 TRUE     beatles       76     59.4
```

Operating on groups in the data

```
bands %>%
    mutate(age = 2019 - yearOfBirth) %>%
    group_by(band) %>%
    mutate(mean_age = mean(age))
```

```
## # A tibble: 9 x 7
## # Groups:   band [2]
##   firstName lastName  yearOfBirth deceased band         age mean_age
##   <chr>     <chr>           <dbl> <lgl>    <chr>      <dbl>    <dbl>
## 1 Melanie   Brown            1975 FALSE    spicegirls    44     44.8
## 2 Melanie   Chisholm         1974 FALSE    spicegirls    45     44.8
## 3 Emma      Bunton           1976 FALSE    spicegirls    43     44.8
## 4 Geri      Halliwell        1972 FALSE    spicegirls    47     44.8
## 5 Victoria  Beckham          1974 FALSE    spicegirls    45     44.8
## 6 John      Lennon           1940 TRUE     beatles       79     77.8
## 7 Paul      McCartney        1942 FALSE    beatles       77     77.8
## 8 Ringo     Starr            1940 FALSE    beatles       79     77.8
## 9 George    Harrison         1943 TRUE     beatles       76     77.8
```

Operating on groups in the data

```
bands %>%
    mutate(age = 2019 - yearOfBirth) %>%
    group_by(band) %>%
    summarise(mean_age = mean(age))
```

```
## # A tibble: 2 x 2
##   band       mean_age
##   <chr>         <dbl>
## 1 beatles        77.8
## 2 spicegirls     44.8
```

Count observations with `count()`

How many members are in each band?

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

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

```
## # A tibble: 2 x 2
##   band       count
##   <chr>      <int>
## 1 beatles        4
## 2 spicegirls     5
```

Method 2: Use `count()`

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

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

Exporting data

```
ageSummary <- bands %>%
    mutate(age = 2019 - yearOfBirth) %>%
    group_by(band) %>%
    summarise(mean_age = mean(age))
ageSummary
```

```
## # A tibble: 2 x 2
##   band       mean_age
##   <chr>         <dbl>
## 1 beatles        77.8
## 2 spicegirls     44.8
```

Save `ageSummary` data frame:

```
savePath <- file.path('data', 'ageSummary.csv')
write_csv(ageSummary, savePath)
```

Plotting with ggplot2

Layers:

1. The data: `ggplot(data = ___)`
2. The aesthetics: `aes(x = ____, y = ____)`
3. The geometries: `geom_point()`, `geom_bar()`

Example data:

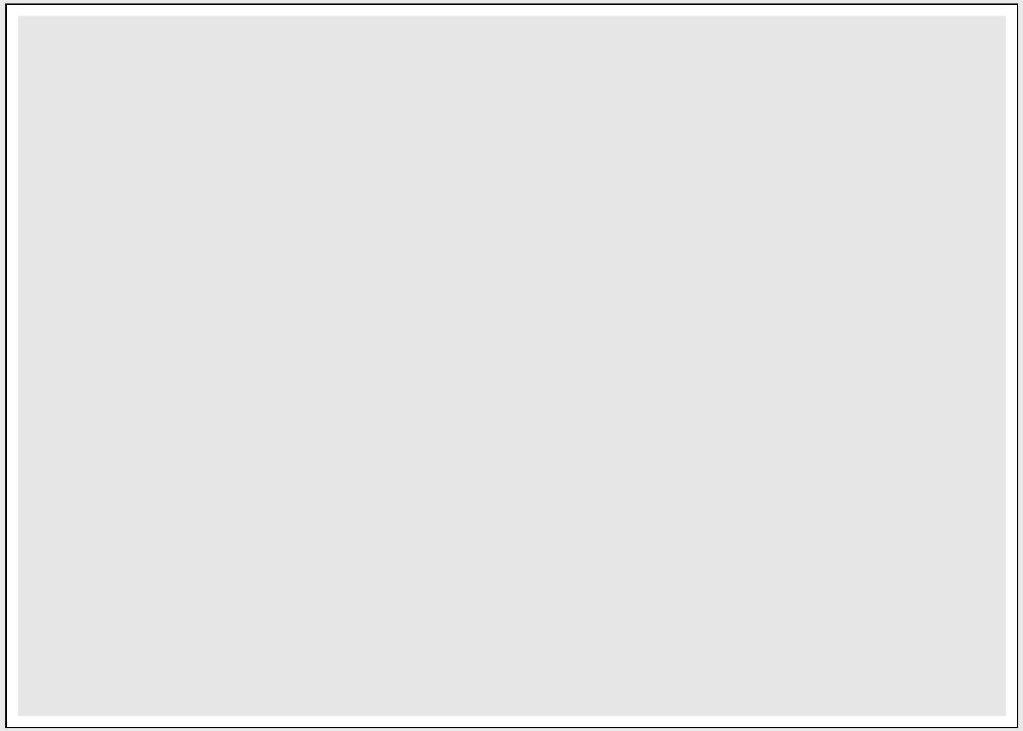
```
glimpse(bears)
```

```
## Observations: 166
## Variables: 14
## $ name          <chr> "Mary Porterfield", "Wilie Porterfield"
## $ age           <dbl> 3, 5, 7, 18, 1, 61, 60, 9, 52, NA, 60,
## $ gender        <chr> "female", "male", "male", "male", NA, "
## $ date          <chr> "19/05/1901", "19/05/1901", "19/05/1901
## $ month         <dbl> 5, 5, 5, 11, 10, 9, 6, 8, 9, 10, 6, 7,
## $ year          <dbl> 1901, 1901, 1901, 1906, 1908, 1916, 192
## $ wildOrCaptive <chr> "Wild", "Wild", "Wild", "Wild", "Captiv
## $ location      <chr> "Job, West Virginia", "Job, West Virgir
## $ description   <chr> "The children were gathering flowers ne
## $ bearType      <chr> "Black", "Black", "Black", "Black", "Bl
## $ hunter        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
## $ grizzly       <dbl> 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
## $ hiker         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
## $ onlyOneKilled <dbl> 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Layer 1: The data

The `ggplot()` function initializes the plot:

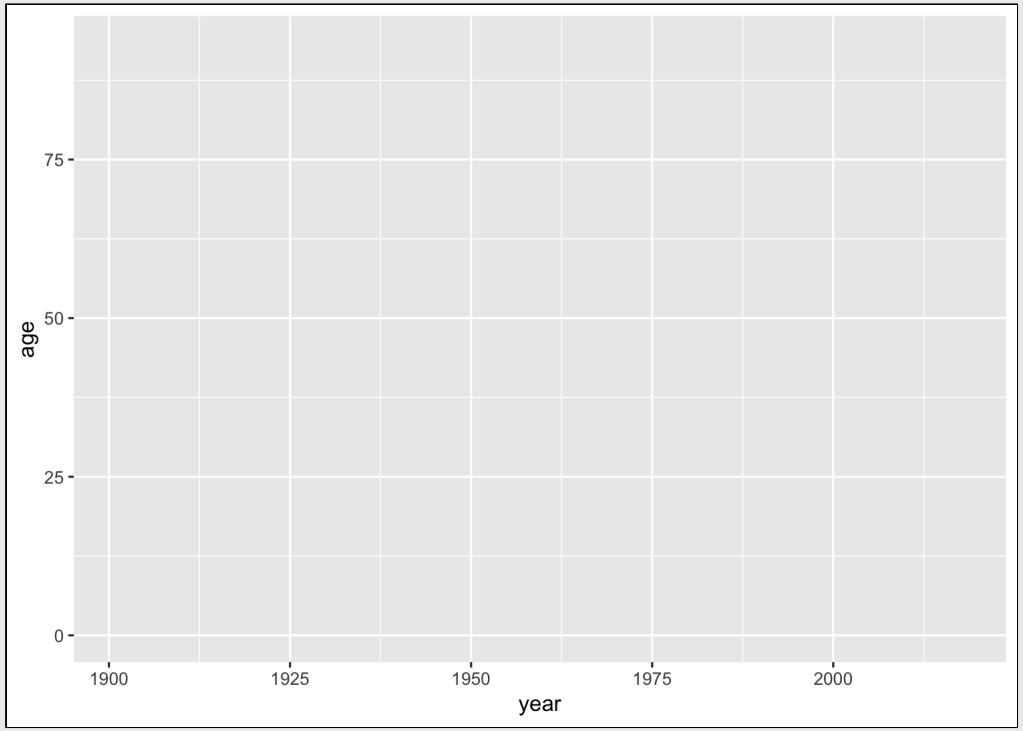
```
ggplot(data = bears)
```



Layer 2: The aesthetics

The `aes()` function determines which variables will be *mapped* to the axes:

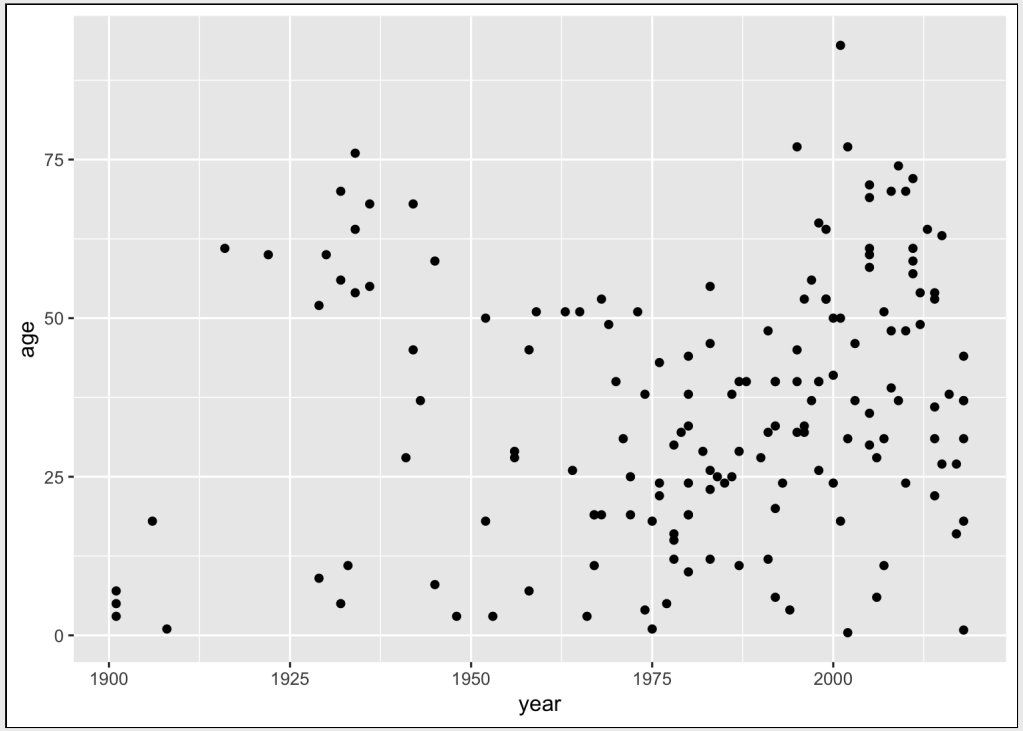
```
ggplot(data = bears, aes(x = year, y = age))
```



Layer 3: The geometries

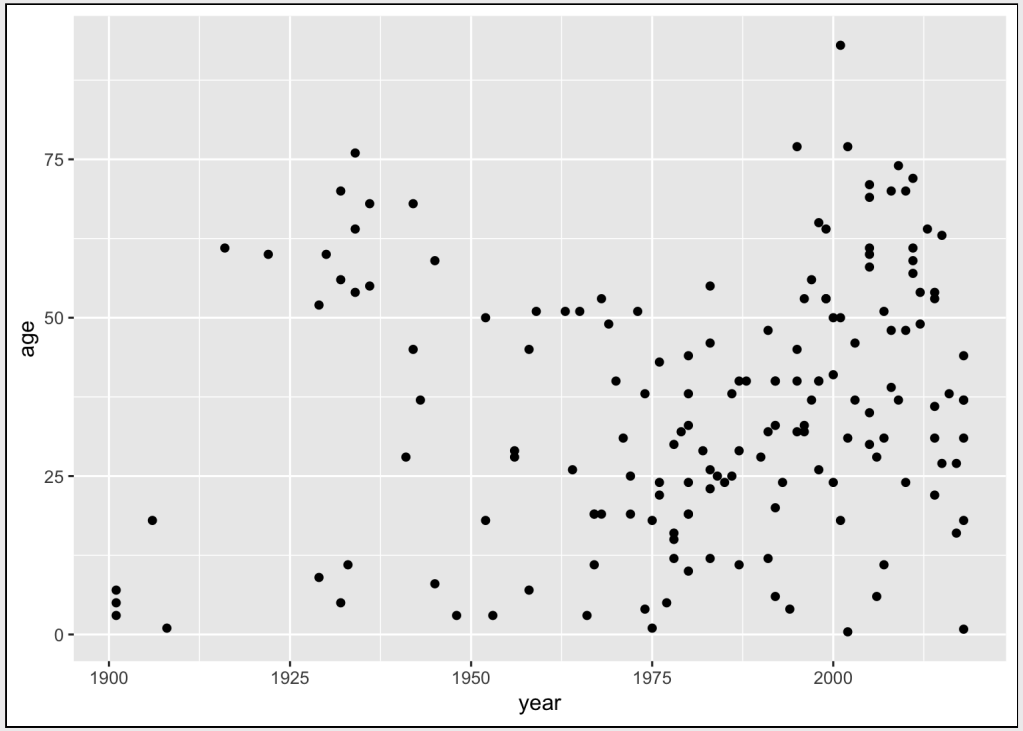
Use `+` to add geometries (e.g. points) to the plot:

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



Scatterplots with `geom_point()`

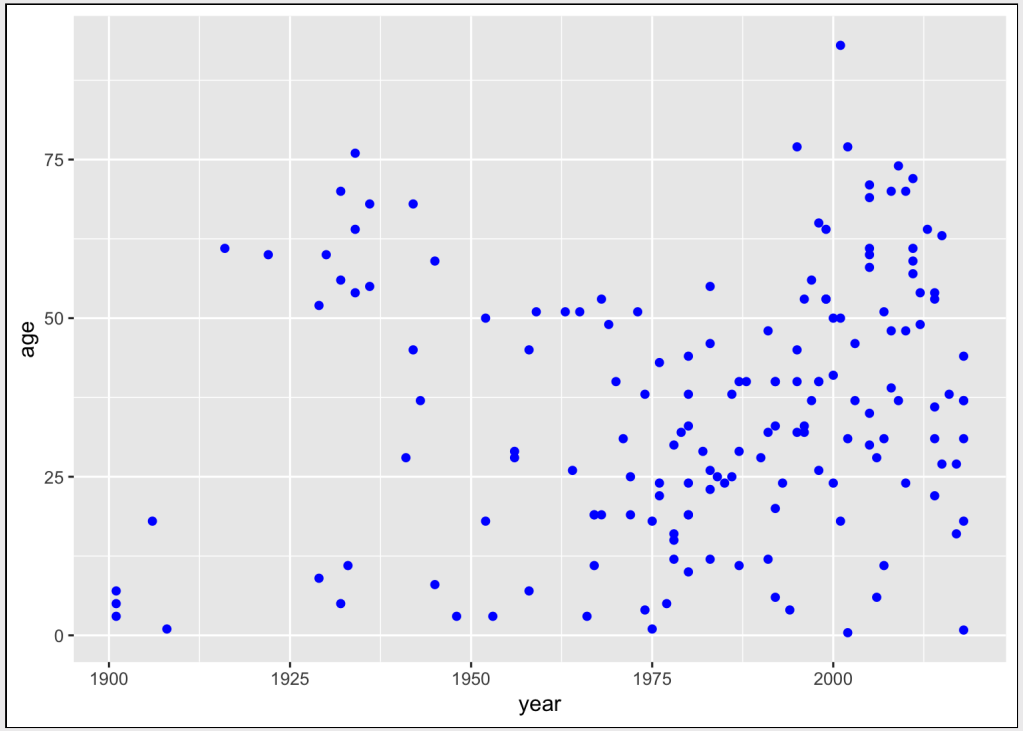
```
ggplot(data = bears, aes(x = year, y = age)) +
    geom_point()
```



Scatterplots with `geom_point()`

Change the point color:

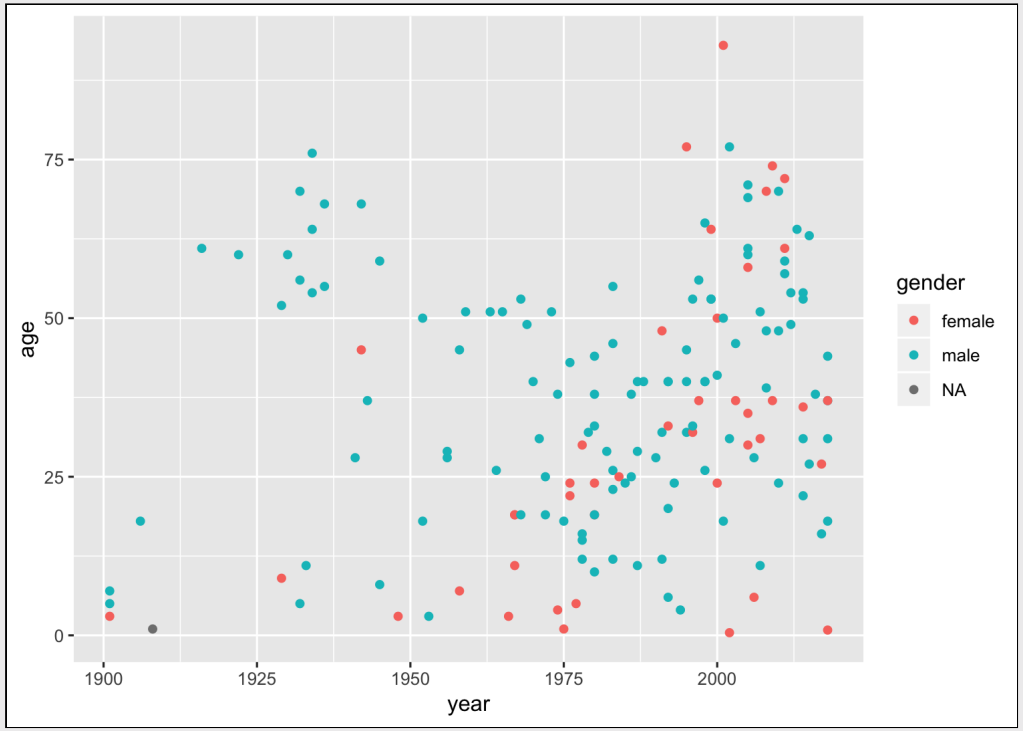
```
ggplot(data = bears, aes(x = year, y = age)) +
    geom_point(color = 'blue')
```



Scatterplots with `geom_point()`

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

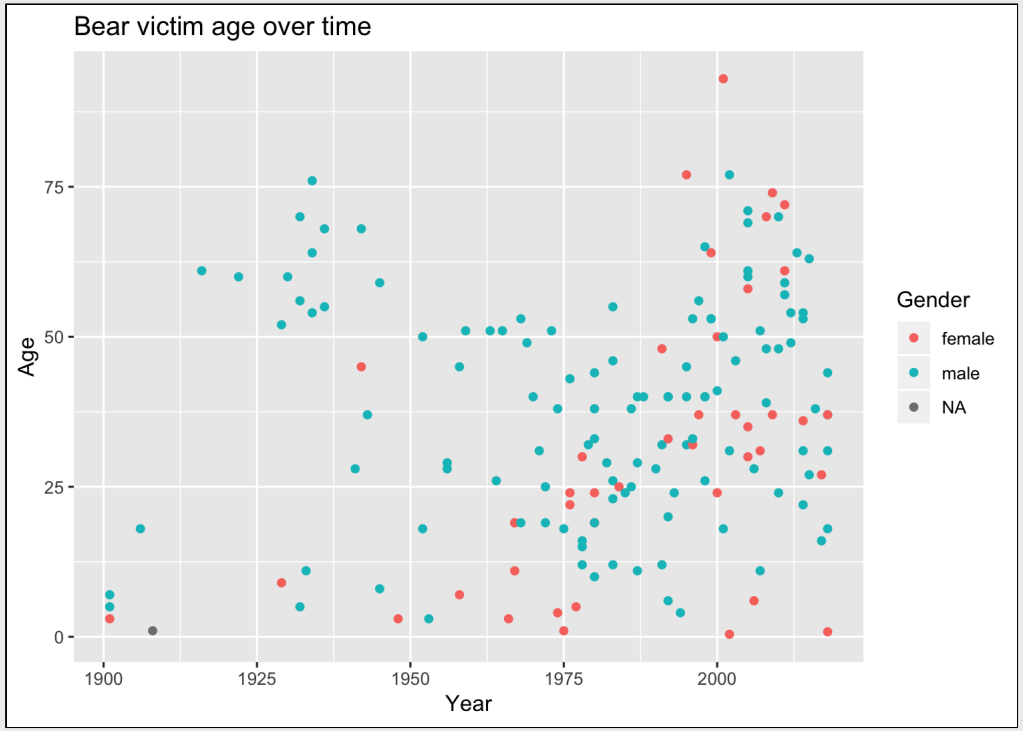
```
ggplot(data = bears, aes(x = year, y = age)) +
    geom_point(aes(color = gender))
```



Scatterplots with `geom_point()`

Adjust labels with `labs()` layer:

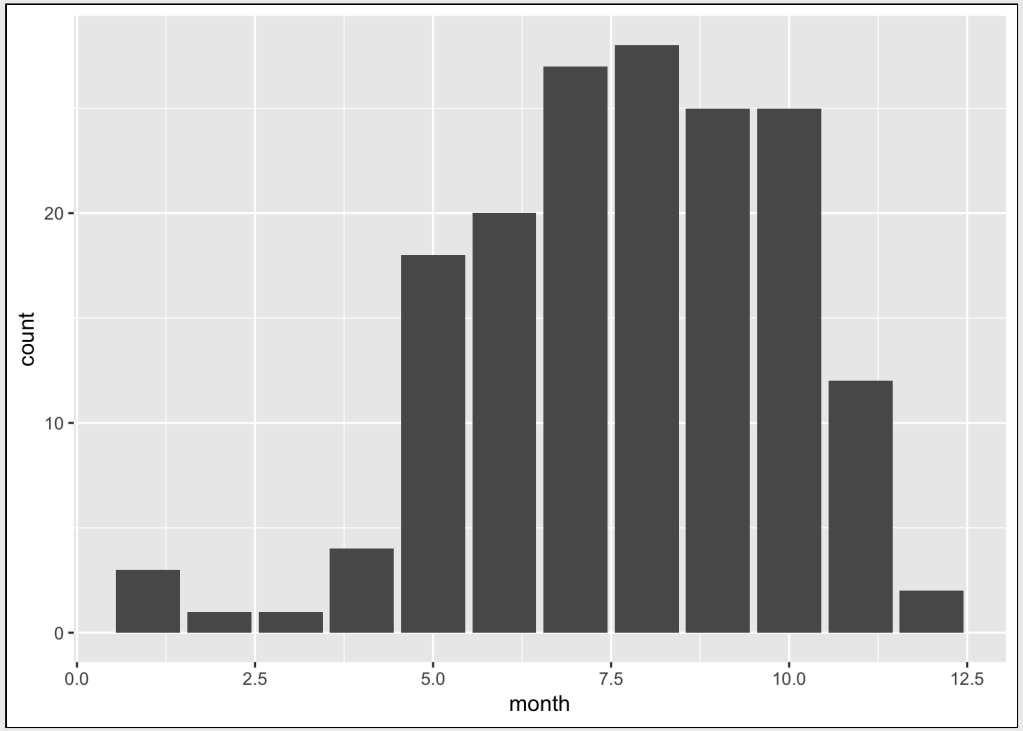
```
ggplot(data = bears, aes(x = year, y = age)) +
    geom_point(aes(color = gender)) +
    labs(x = "Year",
         y = "Age",
         title = "Bear victim age over time",
         color = "Gender")
```



Histograms with `geom_bar()`

The default `geom_bar()` is a count of `x`

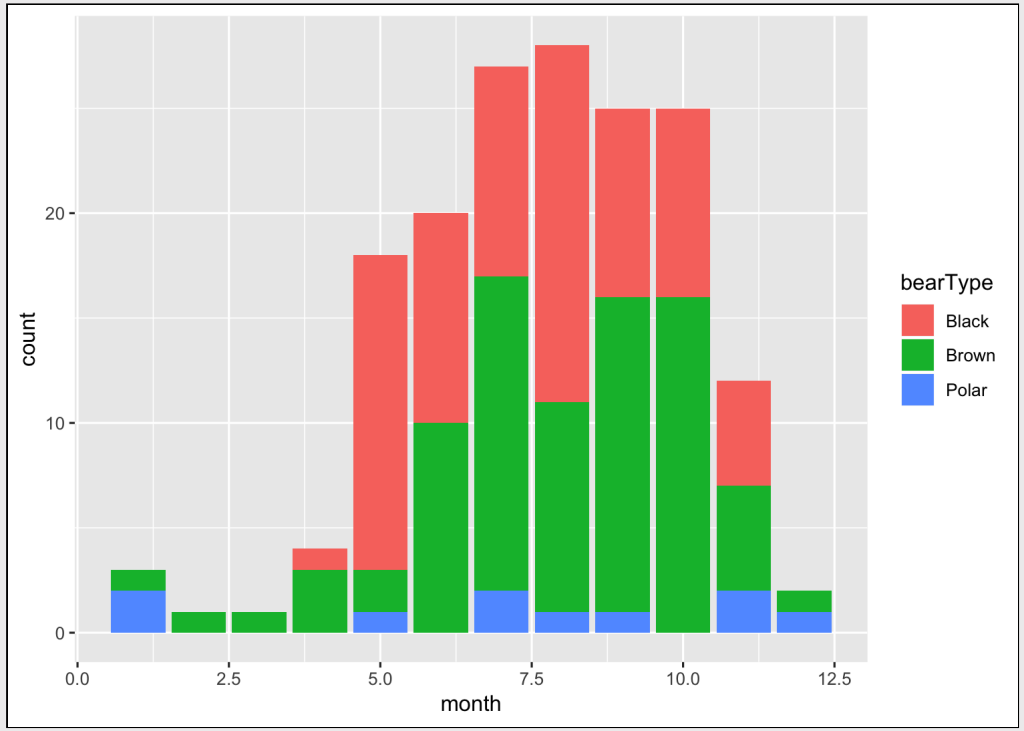
```
ggplot(data = bears, aes(x = month)) +
    geom_bar()
```



Histograms with `geom_bar()`

Use `fill` (not `color`) to change the color of the bars:

```
ggplot(data = bears, aes(x = month)) +
    geom_bar(aes(fill = bearType))
```

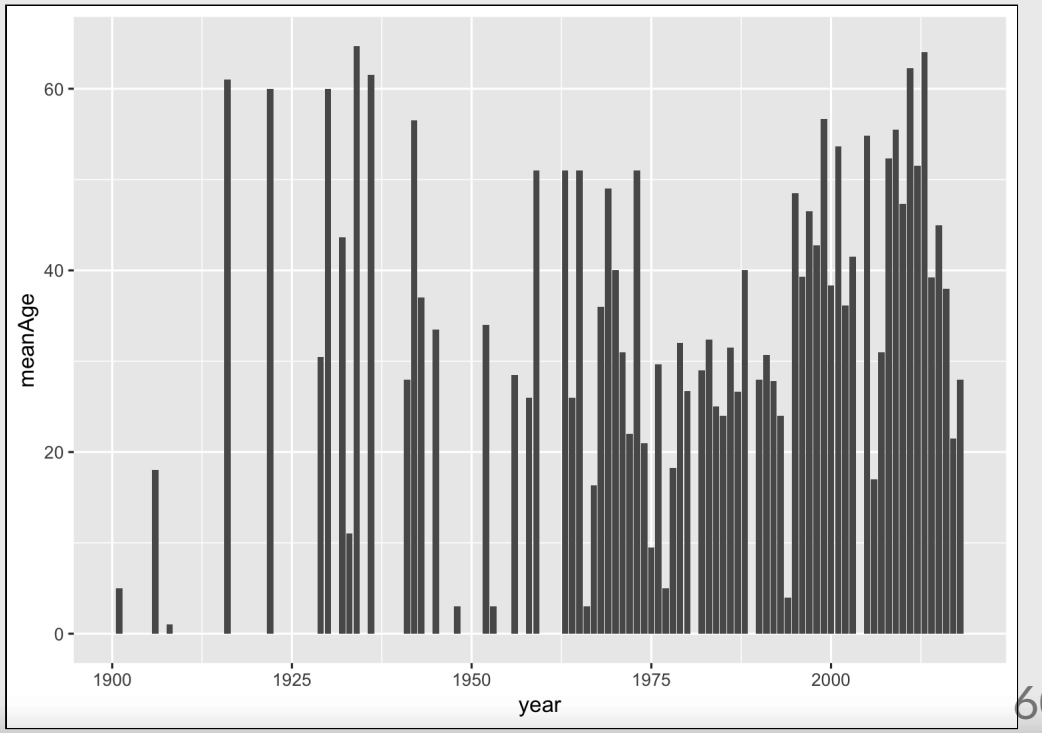


Bar charts with `geom_bar()`

To plot a `y` variable other than "count", add `stat = 'identity'`:

(Default "stat" is `count`)

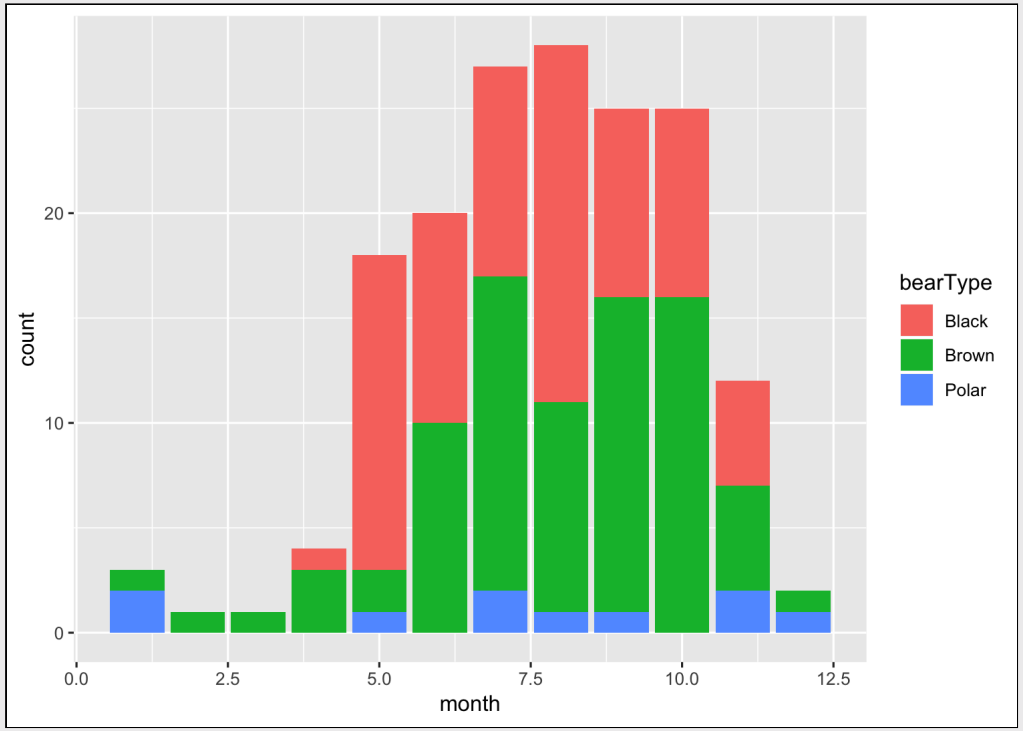
```
meanAgeAnnual <- bears %>%
  filter(!is.na(age)) %>%
  group_by(year) %>%
  summarise(meanAge = mean(age))

ggplot(data=meanAgeAnnual, aes(x=year, y=meanAge)
    geom_bar(stat = 'identity')
```



"Factors" = Categorical variables

By default, R makes numeric variables *continuous*

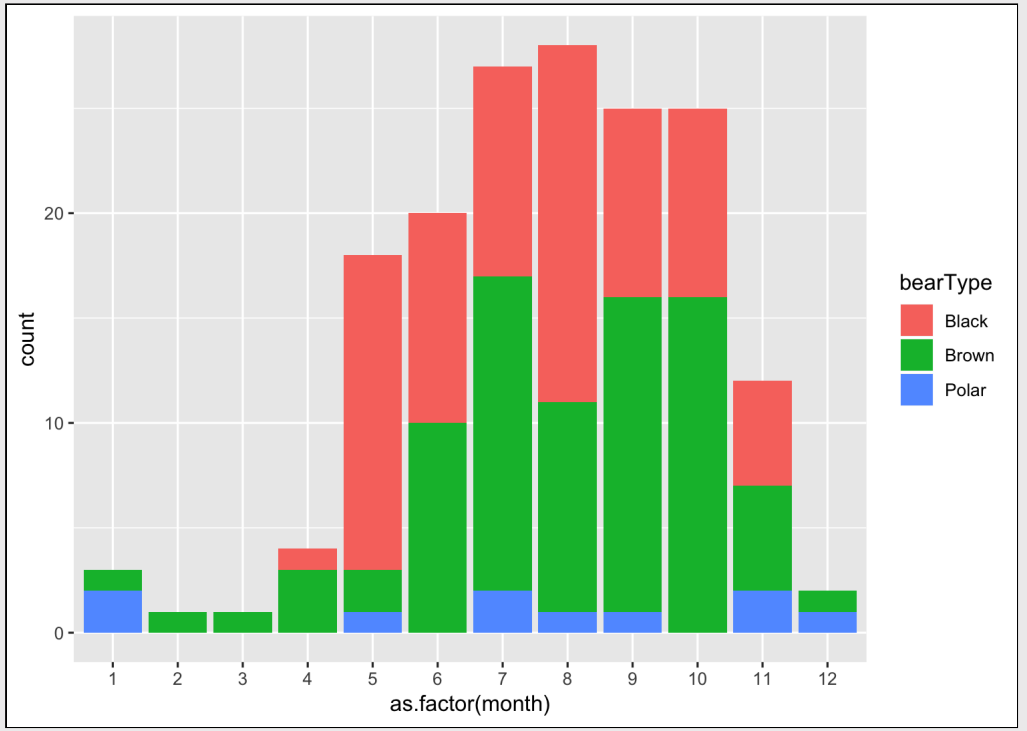
```
ggplot(data = bears, aes(x = month)) +
    geom_bar(aes(fill = bearType))
```



"Factors" = Categorical variables

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

```
ggplot(data = bears, aes(x = as.factor(month))) +
    geom_bar(aes(fill = bearType))
```



Save figures 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 = file.path('data', 'scatterPlot.pdf'),
       plot   = scatterPlot,
       width  = 6,
       height = 4)
```

RMarkdown

Know the basic elements:

- YAML
- Markdown formatting
- Code chunks
- Tables: markdown & `kable()`

Elements of an RMarkdown file (`.Rmd`)

1) YAML metadata

```
---
title: "This is a demo"
author: "John Helveston"
date: "11/18/2019"
output: html_document
---
```

2) Markdown text

```
# Section 1

This is a sentence...
```

3) R Code

````
```{r}
plot(x = 1:5, y = 1:5)
```
````

4) Code output

Markdown tables

Basic tables

```
Table Header  | Second Header
------------- | -------------
Cell 1, 1     | Cell 2, 1
Cell 1, 2     | Cell 2, 2
```

Table Header	Second Header
Cell 1, 1	Cell 2, 1
Cell 1, 2	Cell 2, 2

Tables with `kable()`

```
bears %>%
  count(bearType, wildOrCaptive) %>%
  kable()
```

bearType	wildOrCaptive	n
Black	Captive	16
Black	Wild	60
Brown	Captive	8
Brown	Wild	72
Polar	Captive	4
Polar	Wild	6

List of all class practice problems

Number chopping

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

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

`onesDigit(x)` tests:

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

`tensDigit(x)` tests:

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

General function writing

1) `eggCartons(eggs)`: Write a program that reads in a non-negative number of eggs and prints the number of egg cartons required to hold that many eggs (given that each egg carton holds one dozen eggs, and you cannot buy fractional egg cartons). Be sure your program works for multiples of 12, including 0.

- eggCartons(0) == 0
- eggCartons(1) == 1
- eggCartons(12) == 1
- eggCartons(13) == 2
- eggCartons(24) == 2
- eggCartons(25) == 3

2) `militaryTimeToStandardTime(n)`: Write a program that takes an integer between 0 and 23 (representing the hour in military time), and returns the same hour in standard time. For example, 17 in military time is 5 o'clock in standard time.

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

Conditionals (if / else)

Write the function `getInRange(x, bound1, bound2)` which takes 3 numeric values: `x`, `bound1`, and `bound2`, where `bound1` is not necessarily less than `bound2`. If `x` is between the two bounds, just return it unmodified. Otherwise, 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 the range [3,5])
- `getInRange(4, 3, 5)` returns `4` (the original value, since 4 is below the range [3,5])
- `getInRange(6, 3, 5)` returns `5` (the upper bound, since 6 is above the range [3,5])
- `getInRange(6, 5, 3)` returns `5` (the upper bound, since 6 is above the range [3,5])

Start by writing the test function `testGetInRange()` that tests for a variety of values of `x`, `bound1`, `bound2`.

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

Loops / Vectors

1) `sumFromMToN(m, n)`: Write a function that sums the total of the integers between `m` and `n`. **Challenge**: Try solving this without a loop (it's possible - Google it!).

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

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

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

3) `sumOfOddsFromMToN(m, n)`: Write a function that sums every *odd* integer between `m` and `n`. **Challenge**: Try solving this without a loop (Hint: use a vector operation...we'll cover this next week!).

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

Loops / Vectors

1) `isMultipleOf4Or7(n)`

Write a function that returns `TRUE` if `n` is a multiple of 4 or 7 and `FALSE` otherwise. Here's some test cases:

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

2) `nthMultipleOf4Or7(n)`

Write a function that returns the nth positive integer that is a multiple of either 4 or 7. Hint: use `isMultipleOf4Or7(n)` as a helper function! Here's some test cases:

- `nthMultipleOf4Or7(1) == 4`
- `nthMultipleOf4Or7(2) == 7`
- `nthMultipleOf4Or7(3) == 8`
- `nthMultipleOf4Or7(4) == 12`
- `nthMultipleOf4Or7(5) == 14`
- `nthMultipleOf4Or7(6) == 16`
- `nthMultipleOf4Or7(10) == 28`

Loops / Vectors

1) `isPrime(n)`

Write a function that takes a non-negative integer, `n`, and returns `TRUE` if it is a prime number and `FALSE` otherwise. Here's some test cases:

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

2) `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 `isPrime(n)` as a helper function! Here's some test cases:

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

Vectors

1) `reverse(x)`

Write a function that returns the vector in reverse order. You cannot use the `rev()` function. You cannot use loops! Test cases:

- `reverse(c(5, 1, 3)) == c(3, 1, 5)`
- `reverse(c('a', 'b', 'c')) == c('c', 'b', 'a')`
- `reverse(c(FALSE, TRUE, TRUE)) == c(TRUE, TRUE, FALSE)`
- `reverse(seq(10)) == seq(10, 1, -1)`

2) `middleValue(a)`

Write a function that takes a vector of numbers `a` and returns the value of the middle element (or the average of the two middle elements). Test cases:

- `middleValue(c(0,0,0)) == 0`
- `middleValue(c(1,2,3)) == 2`
- `middleValue(c(4,5,6,7,8)) == 6`
- `middleValue(c(5,3,8,4)) == mean(c(3,8))`
- `middleValue(c(4,5,6,7)) == mean(c(5,6))`

Vectors

1) `dotProduct(a, b)`

The "dot product" of two vectors is the sum of the products of the corresponding terms. So the dot product of the vectors `c(1,2,3)` and `c(4,5,6)` is `(1*4) + (2*5) + (3*6)`, or `4 + 10 + 18 = 32`. Write a function that takes two vectors and returns the dot product of those vectors. If the vectors are not equal length, ignore the extra elements in the longer vector. Try not to use loops! Test cases:

- `dotProduct(c(1,2,3), c(4,5,6)) == 32`
- `dotProduct(c(1,2), c(4,5,6)) == 14`
- `dotProduct(c(2,3,4), c(-7,1,9)) == 25`
- `dotProduct(c(0,0,0), c(-7,1,9)) == 0`

2) `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. Test cases:

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

Data frames

```
animals_farm = tibble(
    name           = c("cow", "horse"),
    sound          = c("moo", "neigh"),
    aveWeightLbs   = c(2400, 1500),
    aveLifeSpanYrs = c(20, 25)
)
animals_pet = tibble(
    name           = c("dog", "cat"),
    sound          = c("woof", "meow"),
    aveWeightLbs   = c(40, 8),
    aveLifeSpanYrs = c(10, 12)
)
```

Use R code to find answers to these questions:

1. How many rows are in the `animals_farm` data frame?
2. How many columns are in the `animals_pet` data frame?
3. Create a new data frame, `animals`, by combining `animals_farm` and `animals_pet`.
4. Create a new column in `animals` called `type` and set the values to `"farm"` or `"pet"`.
5. Change the column names of `animals` to title case.

Data frames

```
beatles <- tibble(
    firstName   = c("John", "Paul", "Ringo", "George"),
    lastName    = c("Lennon", "McCartney", "Starr", "Harrison"),
    instrument  = c("guitar", "bass", "drums", "guitar"),
    yearOfBirth = c(1940, 1942, 1940, 1943),
    deceased    = c(TRUE, FALSE, FALSE, TRUE)
)
```

Use R code to find answers to these questions:

1. Create a new column, `playsGuitar`, which is `TRUE` if the band member plays the guitar and `FALSE` otherwise.
2. Select 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` columns & `filter` rows

1) Create the data frame object `df` by using `file.path()` and `read_csv()` to load the `birds.csv` file that is 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`, and `%>%`

1) Create the data frame object `df` by using `file.path()` and `read_csv()` to load the `birds.csv` file that is 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, `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`

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

2) Use the `df` object and the `mutate()` functions to add the following new variables:

- `height_miles`: The `height` variable converted to miles (Hint: there are 5,280 feet in a mile).
- `cost_mil`: `TRUE` if the repair costs was greater or equal to $1 million, `FALSE` otherwise.

BONUS: Use the `incident_month` variable to create a new variable `season`, which takes one of four values based on the incident month:

- `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 `file.path()` and `read_csv()` to load the `birds.csv` file that is in the `data` folder.

2) Use the `df` object and the `group_by()` and `summarise` functions 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.

Data wrangling: `count`

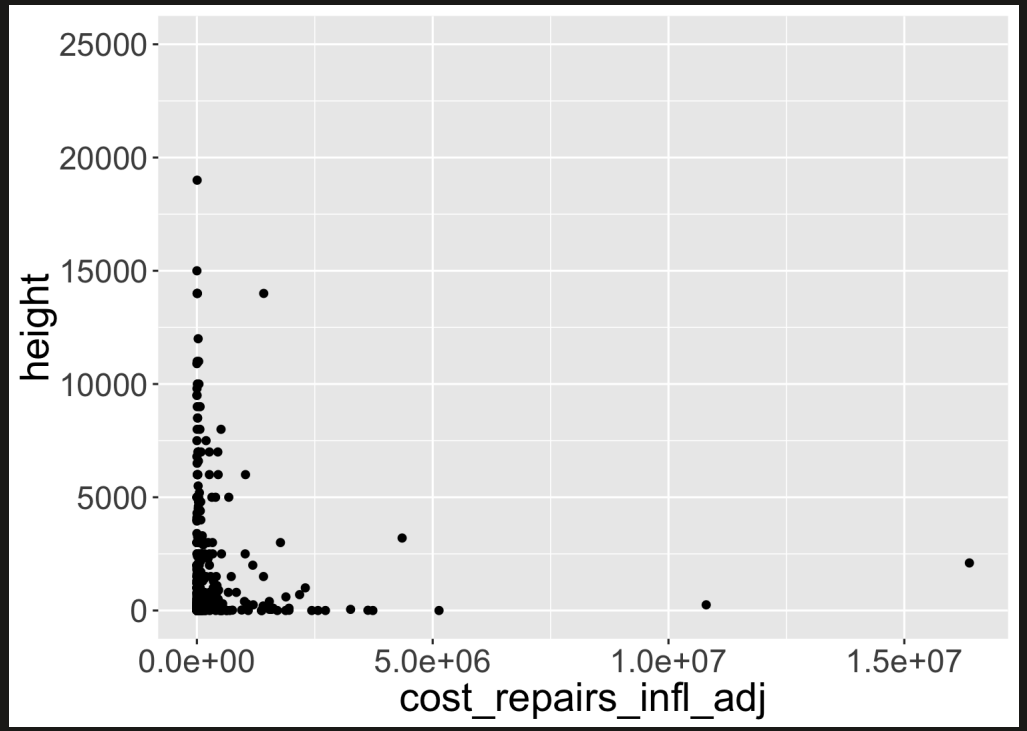
1) Create the data frame object `df` by using `file.path()` and `read_csv()` to load the `birds.csv` file that is in the `data` folder.

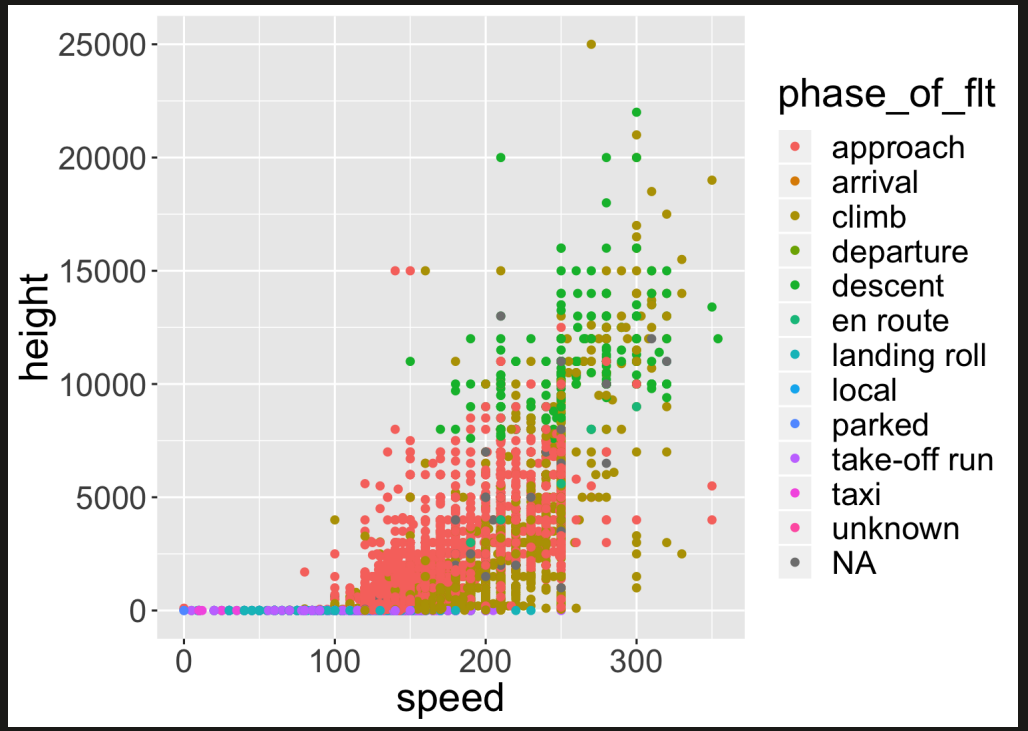
2) Use the `df` object and the `count()` function to answer the following questions:

- Which month has had the greatest number of reported incidents?
- Which year has had the greatest number of reported incidents?

Practice: `geom_point()`

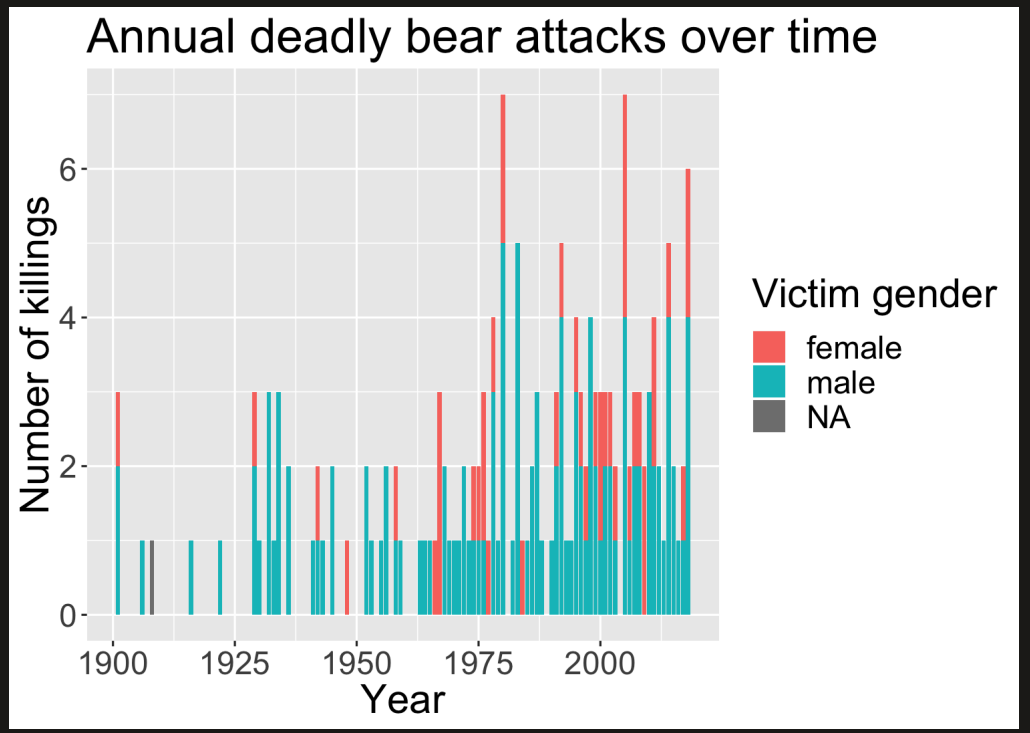
Use the `birds` data frame and `geom_point()` to create the following plots

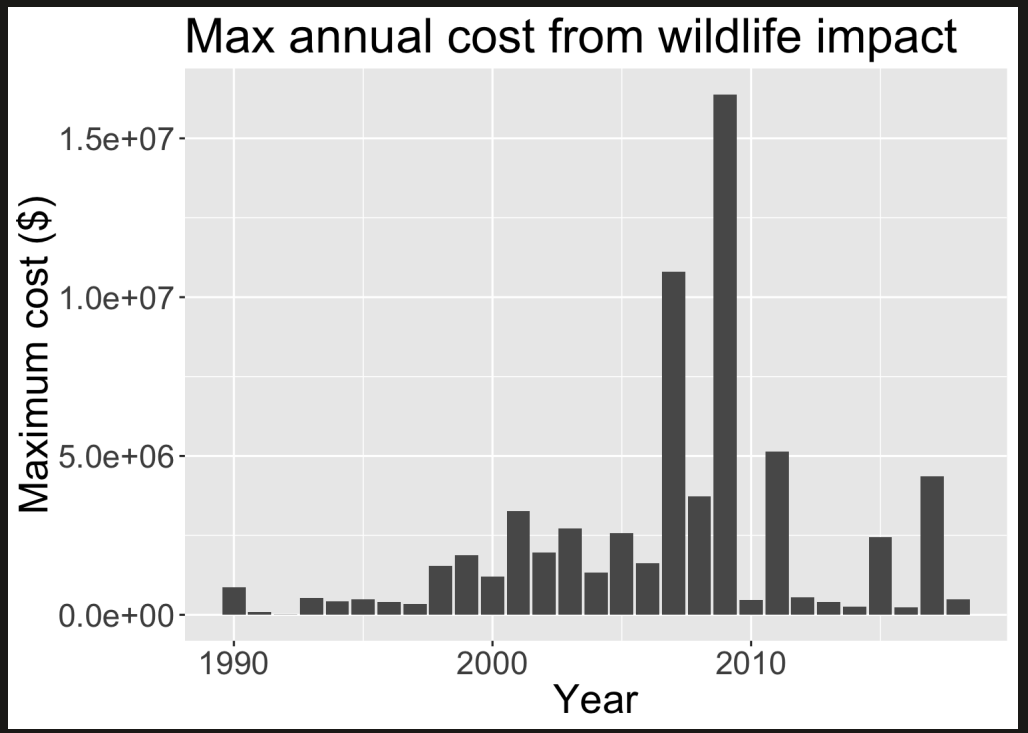




Practice: `geom_bar()`

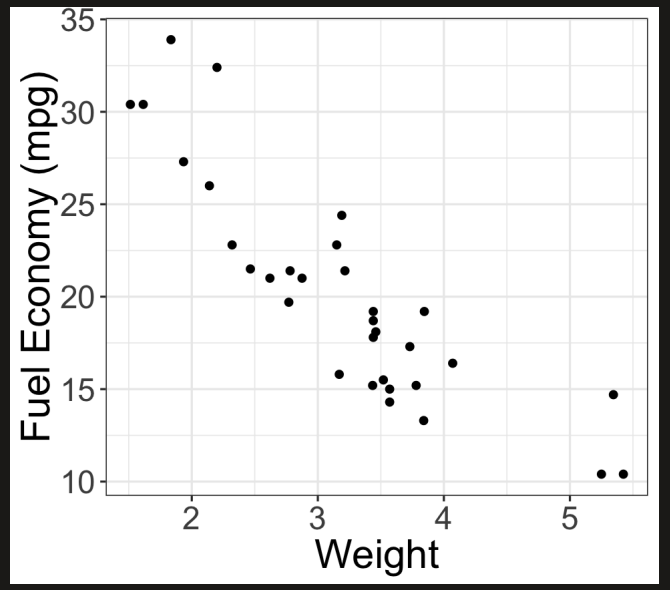
Use the `bears` and `birds` data frames with `geom_bar()` to create the following plots

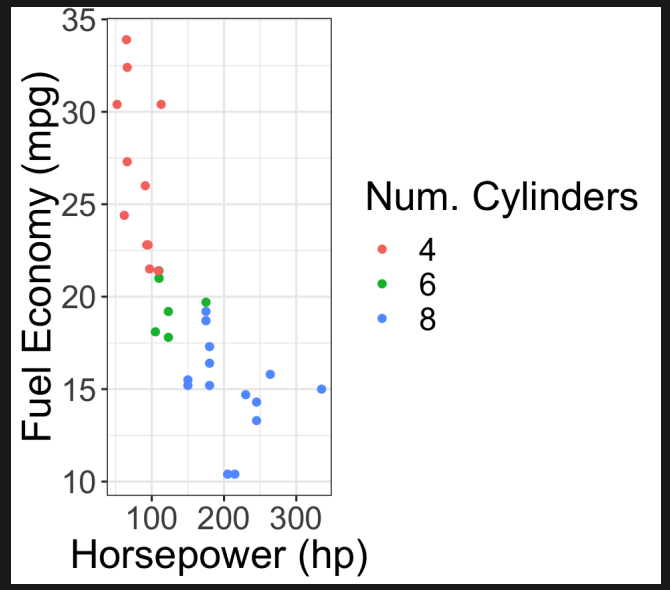


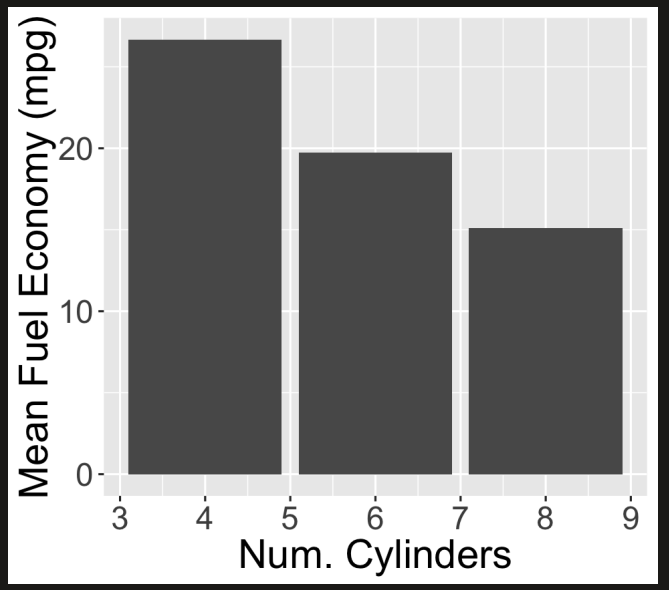


Extra plots practice 1

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







Extra plots practice 2

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