Subsection 1

dplyr

dplyr::filter() [ROW OPERATIONS]

A much simpler and intuitive way to subset data is to use the `filter()` function

```
> filter(mtcars, mpg > 32)
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
2 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1

> filter(mtcars, mpg > 32, mpg < 33 )
   mpg cyl disp hp drat  wt  qsec vs am gear carb
1 32.4   4 78.7 66 4.08 2.2 19.47  1  1    4    1

> filter(mtcars, mpg > 32 & mpg < 33 )
   mpg cyl disp hp drat  wt  qsec vs am gear carb
1 32.4   4 78.7 66 4.08 2.2 19.47  1  1    4    1

> filter(mtcars, mpg > 32 & mpg < 33, wt > 2)
   mpg cyl disp hp drat  wt  qsec vs am gear carb
1 32.4   4 78.7 66 4.08 2.2 19.47  1  1    4    1
```

dplyr::slice() [ROW OPERATIONS]

Select specific rows from a data frame

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

> slice(iris, 3:5)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          4.7         3.2          1.3         0.2  setosa
2          4.6         3.1          1.5         0.2  setosa
3          5.0         3.6          1.4         0.2  setosa
```

dplyr::select() [COLUMN OPERATIONS]

```
> glimpse(mtcars)
Observations: 32
Variables: 11
$ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8, ....
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8, ...
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, ....
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, ...
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, ...
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, ...
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, ...
$ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, ...
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, ...
$ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, 4, 3, ...
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, ...

> glimpse(select(mtcars, mpg, cyl))
Observations: 32
Variables: 2
$ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8, ...
$ cyl <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, 4, 4, 8, ...
```

dplyr::select() [COLUMN OPERATIONS][CONT'D]

```
select(iris, contains("."))
```
Select columns whose name contains a character string.

```
select(iris, ends_with("Length"))
```
Select columns whose name ends with a character string.

```
select(iris, everything())
```
Select every column.

```
select(iris, matches(".t."))
```
Select columns whose name matches a regular expression.

```
select(iris, num_range("x", 1:5))
```
Select columns named x1, x2, x3, x4, x5.

```
select(iris, one_of(c("Species", "Genus")))
```
Select columns whose names are in a group of names.

```
select(iris, starts_with("Sepal"))
```
Select columns whose name starts with a character string.

```
select(iris, Sepal.Length:Petal.Width)
```
Select all columns between Sepal.Length and Petal.Width (inclusive).

```
select(iris, -Species)
```
Select all columns except Species.

Summarizing Data Using dplyr

The main dplyr functions used to summarize data are

1. summarise(), which summarizes data into a single row of values
2. summarise_all(), which applies multiple summary functions to each column

dplyr::summarise()

- Applies function(s) to column(s) of data
- Following examples use mammalSleep.csv

```
> summarise(mammals, avgSleep = mean(sleep_total))
# A tibble: 1  1
  meanSleep
      <dbl>
1  10.43373

> summarise(mammals, avgSleep = mean(sleep_total), medSleep = median(sleep_total))
# A tibble: 1  2
  meanSleep medSleep
      <dbl>    <dbl>
1  10.43373     10.1

> summarise(mammals, avgSleep = mean(sleep_total),
    avgREM = mean(sleep_rem, na.rm = T))
# A tibble: 1  2
  meanSleep meanREM
      <dbl>   <dbl>
1  10.43373 1.87541
```

dplyr::summarise_all()

Applies **same** function(s) to columns of data

```
> summarise_all(iris, funs(mean))
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1     5.843333    3.057333        3.758    1.199333      NA
Warning message:
In mean.default(c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L,  :
  argument is not numeric or logical: returning NA
```

- Note the errors generated by attempting to generate a descriptive statistic for non-numeric columns
- In the event you are generating a descriptive statistic of a *numeric* column that includes `NA`s, you are required to include the `na.rm = T` option in the function call

```
> summarise_all(mammals, funs(mean(., na.rm = T)))
```

dplyr::summarise_all() [CONT'D]

- Multiple functions can be called on columns of data

E.g. Computing the variance and median of all columns of the `iris` data with column names that begin with the word `Sepal`

```
> summarise_all(select(iris, starts_with("Sepal")), funs(var, median))
  Sepal.Length_var Sepal.Width_var Sepal.Length_median Sepal.Width_median
1        0.6856935       0.1899794                  5.8                 3
```

dplyr::group_by()

- Groups data into rows with the same values
- Once data is identified as being grouped, it can be summarized by group

```
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))
# A tibble: 5  2
     vore meanSleepTime
    <chr>         <dbl>
1   carni     10.378947
2   herbi      9.509375
3 insecti     14.940000
4    omni     10.925000
5    <NA>     10.185714
```

dplyr::group_by()

- Groups data into rows with the same values
- Once data is identified as being grouped, it can be summarized by group

```
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))
# A tibble: 5  2
     vore meanSleepTime
    <chr>         <dbl>
1   carni     10.378947
2   herbi      9.509375
3 insecti     14.940000
4    omni     10.925000
5    <NA>     10.185714
```

- Observe how cumbersome it is becoming to unpack the above code
- Thankfully, piping will help simplify our code

dplyr::count()

- Counts the number of rows with each unique value of a variable
- Using the `hflights` dataset from the `hflights` package, compute the total number of flights per carrier

```
> count(hflights, UniqueCarrier)
# A tibble: 15  2
   UniqueCarrier     n
           <chr> <int>
1             AA  3244
2             AS   365
3             B6   695
4             CO 70032
5             DL  2641
6             EV  2204
7             F9   838
8             FL  2139
9             MQ  4648
10            OO 16061
11            UA  2072
12            US  4082
13            WN 45343
14            XE 73053
15            YV    79
```

count() vs. group_by()

Be sure to understand the distinction between these functions

- `group_by()` will not transform data in any tangible way, it solely groups the data according to the grouping criteria, *preparing* it for a future arithmetic operation
- `count()` executes two functions: (1) it performs a `group_by()` according to the grouping criteria and (2) it then preforms a tally (count) of the number of items within each grouping

A Brief Digression: magrittr

- magrittr is package which allows for the use of the piping operator %>%
- There are various piping operators, but for the purposes of this brief introduction, the focus will be solely on %>%; you are encouraged to read the magrittr documentation to learn more about the different piping operators
- Piping allows for a significant reduction in typing, as well as making code intuitive to external code reviewers
- Piping is very useful when using data manipulation packages such tidyr and dplyr
- Piping operator keystoke: SHIFT + COMMAND + M

A Brief Digression: magrittr [HOW IT WORKS]

- Data is piped into a function, and the data argument in the function can be dropped
  - Without the piping operator: `mean(x, na.rm = T)`
  - With the piping operator: `x %>% mean(na.rm = T)`

```
# w/o magrittr
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))

#-----------------------
library(magrittr)

mammals %>%
  group_by(vore) %>%
    summarise(meanSleepTime = mean(sleep_total, na.rm = T))

    # A tibble: 5  2
     vore meanSleepTime
    <chr>         <dbl>
1   carni      10.378947
2   herbi       9.509375
3 insecti      14.940000
4    omni      10.925000
5    <NA>      10.185714
```

IMPORTANT: Syntactical Flow for Pipes

- Generally speaking, when using pipes in R, employ the following generic sequence of steps to achieve the desired results
  1. Pipe in the data
  2. Physically manipulate the data (e.g., group data, select columns, select rows, etc.)
  3. Lastly, perform arithmetic operations on the data
- The above is not a rule but strong a recommendation which will typically help avoid execution errors

IMPORTANT: Syntactical Flow for Pipes

- Generally speaking, when using pipes in R, employ the following generic sequence of steps to achieve the desired results
  1. Pipe in the data
  2. Physically manipulate the data (e.g., group data, select columns, select rows, etc.)
  3. Lastly, perform arithmetic operations on the data
- The above is not a rule but strong a recommendation which will typically help avoid execution errors

n.b. Although a single set of pipes is preferred, do not contort yourself to try and chain together an elaborate set of pipes; if the sequence of operations on the data is too complex, write blocks of code with pipes, breaking things down into bite-sized chunks

dplyr::mutate()

- Computes **and** appends one or more new columns
- Using the `hflights` dataset from the `hflights` package, compute the percent of time spent airborne during a flight for each flight

```
myFlights <- hflights  # create a local copy we can modify

myFlights <- myFlights %>%
  mutate(actualAirbornePct = AirTime / ActualElapsedTime)

# using a more advanced piping operator that pipes LHS data, but rather than
# returning the result of the entire chain, overwrites LHS with updated data

myFlights %<>%
  mutate(actualAirbornePct = AirTime / ActualElapsedTime)

glimpse(myFlights$actualAirbornePct)
 num [1:227496] 0.667 0.75 0.686 0.557 0.71 ...
```

summarise() vs. mutate()

Be sure to understand the distinction between these functions

- `summarise()` and `summarise_all()` operate on a column of data, and will generate a **single** summary statistic for each column or group of data in a column

E.g. If you have a data frame with two columns, `age` and `gender`, computing a mean age using a `summarise()` will generate a single value; alternatively, `group_by()` `gender` and then computing a mean age using a `summarise()` will generate a mean age for each gender

- `mutate()` will take the values in one or more columns and perform arithmetic operations on them row-wise, resulting in a new column with the same number of rows as the original columns, e.g., computing BMI for each person in a data frame, where $BMI = weight/height^2$

Using piping notation, `dplyr` functions and the `babynames` dataset from the `babynames` package,

YOU TRY IT

1. Compute the total number of births per year beginning in 1880, in 20-year increments; the output should be as follows

```
1  1880    201482
2  1900    450312
3  1920    2262732
...
```

2. Compute the total number of **unique** names in each year beginning in 1880, in 20-year increments; the output should be as follows

```
1  1880    2000
2  1900    3731
3  1920    10756
...
```

SOLUTION

1.
```
babyNames %>%
  group_by(year) %>%
    summarise(totalBirthsPerYear = sum(n)) %>%
      slice(seq(1, n(), by = 20))

      # A tibble: 7  2
   year totalBirthsPerYear
  <dbl>              <int>
1  1880             201482
2  1900             450312
...
```

2.
```
babynames %>%
  count(year) %>%
    slice(seq(1, n(), by = 20))

# A tibble: 7  2
   year    nn
  <dbl> <int>
1  1880  2000
2  1900  3731
...
```

Combine/Append Data Frames with `rbind()`

- **When at least one data frame already exists**, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `rbind()` to row-bind two data frames or a data frame with a vector

```
> myDataFrame_01 <- data.frame(x = 1:3, y = c("A", "B", "c"))

> myDataFrame_03 <- rbind(myDataFrame_01, data.frame(x = -99, y = "ZzZ"))
    x   y
1   1   A
2   2   B
3   3   c
4 -99 ZzZ

> (myDataFrame_04 <- rbind(myDataFrame_01, c(x = -99, y = "ZzZ")))
    x    y
1   1    A
2   2    B
3   3    c
4 -99 <NA>
Warning message:
In `[<-.factor`(`*tmp*`, ri, value = "ZzZ") :
  invalid factor level, NA generated
```

Combine/Append Data Frames `rbind()` [CONT'D]

- **When at least one data frame already exists**, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `rbind()` to row-bind a data frame with a vector

```
> (myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))
  x   y    z
1 1  98 1000
2 2  99 1001
3 3 100 1002

> (myDataFrame_06 <- rbind(myDataFrame_05, abc = -1:-3))
     x   y    z
1    1  98 1000
2    2  99 1001
3    3 100 1002
abc -1  -2   -3
```

n.b. behavior can quickly become quirky if the length of your vector is not equal to the number of columns in the data frame

PREAMBLE

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

TRY THIS

Based on the myDataFrame_06 code, what happens if we replace ??? with:

(a) `abc = -1`

(b) `abc = -1:-2`

(c) `abc = -1:-99`

(d) `abc = c(-1, -2)`

(e) `abc = c("-1", -2)`

(f) `abc = c("a", -2, -3)))`

PREAMBLE

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

SOLUTION

(a) Entire additional row of -1's

(b) Entire additional row of repeating -1's and -2's

(c) Additional row: -1, -2, -3

(d) Entire additional row of repeating -1's and -2's

(e) Entire additional row of repeating -1's and -2's as characters (non numeric), thereby changing **all** all data frame column types to characters

(f) Additional row: a, -2, -3 as characters (non numeric), thereby changing **all** all data frame columns types to characters

dplyr::bind_rows

bind_rows() from the dplyr package the faster and more predictable cousin of rbind(), although it is not without its own quirks

- bind_rows() **only permits the binding of data frames to each other**; rbind() allows the binding of a vector to a data fame
- Even though it is part of the tidyverse, the result of a call to bind_rows() creates a data frame, not a tibble

PREAMBLE

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- bind_rows(myDataFrame_07, ???)
```

TRY THIS

Based on the myDataFrame_08 code, what happens if we replace ??? with:

(a) `data.frame(-1)`

(b) `data.frame(-1:-2)`

(c) `data.frame(c(-1:-2))`

(d) `data.frame(-1:-99)`

(e) `data.frame("-1", -2)`

(f) `data.frame(c("-1", -2))`

SOLUTION

(a)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1)))
   x   y    z X.1
1  1  98 1000  NA
2  2  99 1001  NA
3  3 100 1002  NA
4 NA  NA   NA  -1
```

(b)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1:-2)))
   x   y    z X.1..2
1  1  98 1000     NA
2  2  99 1001     NA
3  3 100 1002     NA
4 NA  NA   NA     -1
5 NA  NA   NA     -2
```

(c)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(c(-1:-2))))
   x   y    z c..1..2.
1  1  98 1000       NA
2  2  99 1001       NA
3  3 100 1002       NA
4 NA  NA   NA       -1
5 NA  NA   NA       -2
```

SOLUTION [CONT'D]

(d)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1:-99)))
     x   y    z X.1..99
1    1  98 1000      NA
2    2  99 1001      NA
3    3 100 1002      NA
4   NA  NA   NA      -1
5   NA  NA   NA      -2
...
102 NA  NA   NA     -99
```

(e)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame("-1", -2)))
   x   y    z X..1. X.2
1  1  98 1000  <NA>  NA
2  2  99 1001  <NA>  NA
3  3 100 1002  <NA>  NA
4 NA  NA   NA    -1  -2
```

(f)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(c("-1", -2))))
   x   y    z c...1....2.
1  1  98 1000        <NA>
2  2  99 1001        <NA>
3  3 100 1002        <NA>
4 NA  NA   NA          -1
5 NA  NA   NA          -2
```

Combine/Append Data Frames with `cbind()`

- **When at least one data frame already exists**, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `cbind()` to column-bind two data frames or a data frame with a vector

```
> (myDataFrame_01 <- data.frame(x = 1:3, y = c("A", "B", "c")))
  x y
1 1 A
2 2 B
3 3 c

> (myDataFrame_02 <- cbind(myDataFrame_01, z = -1:-3))
  x y  z
1 1 A -1
2 2 B -2
3 3 c -3
```

Combine/Append Data Frames with `cbind()` [CONT'D]

- **When a data frame already exists**, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `cbind()` to column-bind a data frame with a vector
    - **n.b.** behavior can become quirky if the length of your vector is not equal to the number of rows in the data frame

```
> (myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))
  x   y    z
1 1  98 1000
2 2  99 1001
3 3 100 1002

> (myDataFrame_08 <- cbind(myDataFrame_05, abc = -1:-3))
  x   y    z abc
1 1  98 1000  -1
2 2  99 1001  -2
3 3 100 1002  -3
```

PREAMBLE

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- cbind(myDataFrame_07, ???)
```

TRY THIS

Based on the myDataFrame_08 code, what happens if we replace ??? with:

(a) `abc = -1`

(b) `abc = -1:-2`

(c) `abc = -1:-99`

(d) `abc = c("-1", -2)`

(e) `abc = c("a", -2, -3)))`

PREAMBLE

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- cbind(myDataFrame_05, ???)
```

SOLUTION

(a) Entire additional column of −1's

(b) `<arguments imply differing number of rows: 3, 2>`

(c) Extends the length of all other columns and repeats those values until −99

(d) `<arguments imply differing number of rows: 3, 2>`

(e) `<arguments imply differing number of rows: 3, 2>`

(f) Additional column: a, -2, -3 as factors (non numeric)

dplyr::bind_cols

`bind_cols()` from the `dplyr` package the faster and more predictable cousin of `cbind()`, although it is not without its own quirks

- `bind_cols()` only permits the binding of data frames to each other; `cbind()` allows the binding of a vector to a data fame
- Even though it is part of the `tidyverse`, the result of a call to `bind_cols()` creates a data frame, not a tibble

PREAMBLE

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)

> myDataFrame_08 <- bind_cols(myDataFrame_07, ???)
```

TRY THIS

Based on the myDataFrame_08 code, what happens if we replace ??? with:

(a) `abc = data.frame(-1)`

(b) `abc = data.frame(-1:-2)`

(c) `abc = data.frame(-1:-99)`

(d) `abc = data.frame(c("-1", -2))`

(e) `abc = data.frame(c("a", -2, -3))`

SOLUTION

(a)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1))
Error: incompatible number of rows (1, expecting 3)
```

(b)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1:-2))
Error: incompatible number of rows (2, expecting 3)
```

(c)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1:-99))
Error: incompatible number of rows (99, expecting 3)
```

(d)

```
> bind_cols(myDataFrame_07, data.frame(abc = c("-1", -2)))
Error: incompatible number of rows (2, expecting 3)
```

(e)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1))
Error: incompatible number of rows (1, expecting 3)
```

(f)

```
> bind_cols(myDataFrame_07, data.frame(abc = c("a", -2, -3)))
  x   y    z qqq
1 1  98 1000   a
2 2  99 1001  -2
3 3 100 1002  -3
```

LAB

Case Study in `dplyr`: `ggplot2` Movies

Paul Intrevado

July 19, 2017

The `ggplot2movies` package contains a dataset named `movies`, which provides information on 58,788 movies, datging as far back as 1893! Here is a quick graphical summary of how many movies were realeased each year: