UTSA®

ALVAREZ
College of Business
The University of Texas at San Antonio

Introduction to Programming in R

Module 4:
Data Wrangling
Part 1

Learning Objectives

- Importing and exporting using *readr*
- Tibbles
- Filtering rows using *dplyr*
- Sorting using *dplyr*
- Selecting columns using *dplyr*
- Computing and concatenating new columns using *dplyr*
- Grouping using *dplyr*

Data Importing

- The *tidyverse* package has a *readr* package (much faster than Base R importing functions, such as *read.csv()*, and produces tibbles (next).
- Readr uses the following importing functions.

Function	
read_csv()	Comma delimited files
read_csv2()	Semicolon delimited files
read_tsv()	Tab delimited files
read_delim()	Any delimited files
read_fwf()	Fixed-width files
read_log	Apache style log files

- We will use read_csv(): most popular and the syntax is similar across all of them.
- 1st argument is the pathname (location) of the file including the filename.

Properties of *read_csv()*

- By default, the first line of the imported data will be the column names (i.e. the variable names).
- You can change this behavior by using argument *skip* =). For e.g. if the first 2 lines were unnecessary header data, use *skip* = 2.
- If the data does not have column names, then use argument *col_names = FALSE.* Then column names will be *X1, X2, …, Xn*.
- Otherwise, you can specify your own column names by providing *col_names = c().*
- For missing values, that maybe stored as a . or empty space, use argument *na* = *"."* or *na* = *" "*.
- This set of information will help you import most csv files, especially clean ones.
    - With more messy data with lots of missing values and mixed data types, we need to understand parsing a file.

Examples

- Before you can load, we need to check what is our working folder.
    - Use *getwd(), setwd(), and read_csv()*

- Read in *hmda_2017_tx_all_06.csv*.
- Check the data type of each variable.

1. Load the historical state populations file: introductory_state_example.csv
    - Make sure the columns are named appropriately.
2. Load the historical college information file: college_history.csv.
    - Caution the *original_name* column has missing values with different inputs.

How does *readr* work?

- Without going into too much detail – we need to understand <u>how readr automatically guesses each type of variable</u>, and <u>how to overwrite it, if needed</u>.
- Readr reads the first 1000 rows of a column, and the guesses the data type based on some rules.
- Uses *guess_parser()* and *parse_guess()*.

```
> guess_parser(c("TRUE", "FALSE", ""))
[1] "logical"
> parse_guess(c("TRUE", "FALSE", ""))
[1]  TRUE FALSE    NA
```

- You can have issues if your dealing with large files.
  - First 1000 rows might be special cases. For e.g. first 1000 rows are integer with numeric next.
  - First 1000 rows may be missing.
- Let's see an e.g. read in the file challenge.csv.
  - Again make sure either the data is in the working directory or provide the pathname.

How does *read csv()* work?

```
> read_csv(readr_example("Challenge.csv"))
Parsed with column specification:
cols(
  x = col_double(),
  y = col_logical()
)
Warning: 1000 parsing failures.
 row col           expected      actual
file
1001    y 1/0/T/F/TRUE/FALSE 2015-01-16 '/Library/Frameworks/R.framework/Versions/3.4/Resources/library/readr/extdata/C
hallenge.csv'
1002    y 1/0/T/F/TRUE/FALSE 2018-05-18 '/Library/Frameworks/R.framework/Versions/3.4/Resources/library/readr/extdata/C
hallenge.csv'
1003    y 1/0/T/F/TRUE/FALSE 2015-09-05 '/Library/Frameworks/R.framework/Versions/3.4/Resources/library/readr/extdata/C
hallenge.csv'
1004    y 1/0/T/F/TRUE/FALSE 2012-11-28 '/Library/Frameworks/R.framework/Versions/3.4/Resources/library/readr/extdata/C
hallenge.csv'
1005    y 1/0/T/F/TRUE/FALSE 2020-01-13 '/Library/Frameworks/R.framework/Versions/3.4/Resources/library/readr/extdata/C
hallenge.csv'
.... ... .................. .......... ............................................................................
.............
See problems(...) for more details.

# A tibble: 2,000 x 2
      x y
  <dbl> <lgl>
1   404 NA
2  4172 NA
3  3004 NA
4   787 NA
5    37 NA
6  2332 NA
7  2489 NA
```

- Based on the first 1000 rows (NAs), *readr* is expecting logicals. Remember NAs get stored as logicals. But after 1000rows it’s dates.

- So if you store this as a variable, all the date information will be lost, and only NAs will be stored.
- You can see this by view(challenge) and scrolling down.

How does *read_csv()* work? (cont.)

```
> read_csv(
+     readr_example("challenge.csv"),
+     col_types = cols(
+         x = col_number(),
+         y = col_date()
+     )
+ )
# A tibble: 2,000 x 2
      x y
  <dbl> <date>
 1   404 NA
 2  4172 NA
 3  3004 NA
 4   787 NA
 5    37 NA
 6  2332 NA
 7  2489 NA
 8  1449 NA
 9  3665 NA
10  3863 NA
# … with 1,990 more rows
```

- You can add argument *col_types* to *read_csv()* if you already know.
- Here, *col_double()* (numeric) & *col_date()* works.

```
> read_csv(readr_example("Challenge.csv"), guess_max = 1001)
Parsed with column specification:
cols(
  x = col_double(),
  y = col_date(format = "")
)
# A tibble: 2,000 x 2
      x y
  <dbl> <date>
 1   404 NA
 2  4172 NA
 3  3004 NA
 4   787 NA
 5    37 NA
 6  2332 NA
 7  2489 NA
 8  1449 NA
 9  3665 NA
10  3863 NA
# … with 1,990 more rows
```

- You can also use guess_max = as an argument.
- Here, if you used guess_max = 1001, it'd work.

Data Exporting

- You can use *write_csv()* (comma) and *write_tsv()* (tab).
- Save the tidy *college_hist* data to *college_history.csv* file.
- However, these functions do not save data types. So when you read it in again, you'll have to go through parsing again.
    - This is bad for intermediate data manipulations and storing.

For e.g. do *write_csv(challenge,"test_challenge.csv")*, then *read_csv("test_challenge.csv").*

- For intermediate storing, use *write_rds()* and *read_rds()*. This keeps the data type information.

For e.g. do Try *write_rds(challenge,"test_challenge.rds")*, then *read_rds("test_challenge.rds")*.

Tibble

- Very much like a data frame, a bit more modernized.
- Load built-in data frame mtcars.
  - *data("mtcars")* and *head(mtcars)*
- Since we have a data frame, we can coerce it using *as_tibble(mtcars)*

```
> as_tibble(mtcars)
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# … with 22 more rows
```

Advantages:

✓ Does not convert strings to factors!

✓ Tibble column names are more flexible.

✓ A single value variable will be auto-replicated to match size of longest variable.

✓ Can create variables that depend on existing variables in the tibble.

Tibble (cont.)

- You can also manually create a tibble using *tibble()*.
- Here, *b* is a singleton. It's replicated to match size of other variables.
- Also, *c* depends on *a*, so once *a* is entered, *c* gets auto-populated.

```
> tibble(a = c(1:10), b = "x", c = 4*a)
# A tibble: 10 x 3
       a b         c
   <int> <chr> <dbl>
 1     1 x         4
 2     2 x         8
 3     3 x        12
 4     4 x        16
 5     5 x        20
 6     6 x        24
 7     7 x        28
 8     8 x        32
 9     9 x        36
10    10 x        40
```

```
> tibble(`$$` = c("USD","AUD","PES","Yen"), `.1` = 20)
# A tibble: 4 x 2
  `$$`   `.1`
  <chr> <dbl>
1 USD      20
2 AUD      20
3 PES      20
4 Yen      20
```

- You can also name variables in tibble that would be invalid outside of the tidyverse package.
- You must put such names inside single quotes ` `.

Tibble (cont..)

- Another way to manually enter a tibble is using *tribble()*. Stands for transposed tibble.
- The tilde ~ denotes the name of the variables.

```
> tribble(~a, ~b, ~c,
+         1, "x", 4,
+         2, "x", 8,
+         3, "x", 12)
# A tibble: 3 x 3
      a b         c
  <dbl> <chr> <dbl>
1     1 x         4
2     2 x         8
3     3 x        12
```

Differences with data.frame:

- Printing and subsetting.
  - Object stored as a data frame will print the whole thing. But as a tibble, only print the first 10 obs, by default.
  - Let's try with the built-in dataset *Iris*. Do *data("iris").*
  - It is stored as a data frame. Do *print(iris)*
  - Then do *tib_iris = as_tibble(iris)* and *print(tib_iris).*

Tibble (cont…)

- When printing, it does not overwhelm your console.
- Only shows the number of variables that fit on screen.
- Tibbles also show the type of data stored in each variable, like *str()*.
- It also shows the remaining number of obs, after the first 10.
    - You can modify the number to show, using *n* = argument inside the print function.
- When subsetting using *$* or *[[ ]]*, data frame & tibble provides a *NULL* if you have an error, but tibble also provides a *"unknown"* warning.

```
> print(tib_iris)
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            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           3.6          1.4         0.2 setosa
 6          5.4         3.9          1.7         0.4 setosa
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
# … with 140 more rows
```

```
> iris$special
NULL
> tib_iris$special
NULL
Warning message:
Unknown or uninitialised column: 'special'.
```

Transformations

- Using dplyr package.
  - Can pick observations based on values: *filter()*
  - Reorder observations/rows: *arrange()*
  - Extract variables using names: *select()*
  - Create a new variable using functions on existing ones: *mutate()*
  - Collapse multiple observations to a single summary: *summarize()*
- You can also use *group_by()* to use the above functions on the entire data or on results of each other.
- With the above functions, the first argument is the data, and the following argument state what to do with variables.
- Result is a tibble (data frame).

filter()

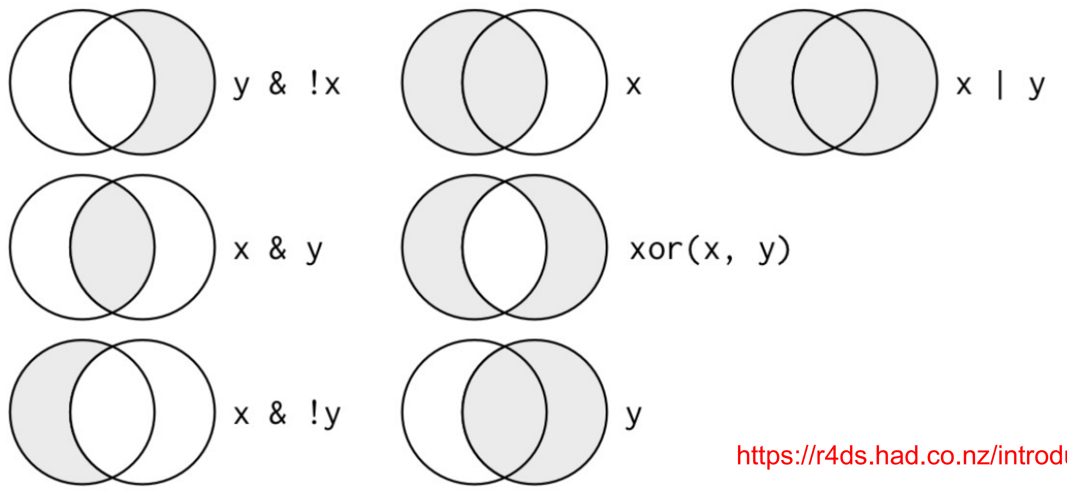
- You can subset a tibble based on values of observations.
- Let's use built-in data mtcars. Do data("mtcars").
- You want to see the cars that have 1 carburetor and 4 gears:

```
> filter(mtcars,carb==1,gear==4)
   mpg cyl  disp hp drat    wt  qsec vs am gear carb
1 22.8   4 108.0 93 3.85 2.320 18.61  1  1    4    1
2 32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
3 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
4 27.3   4  79.0 66 4.08 1.935 18.90  1  1    4    1
```

  - Note that if you want to save this output, you'll have to use <-
  - R will either assign or print the result of the expression.
  - To do both, wrap the expression in parenthesis *( )*.
- For evaluating a criteria do not use a single =, as that is for assigning.
  - Here I used == for testing the criteria. You can also use >, >=, <, <=, *!=, or between().*

filter() (cont.)

- Boolean operations:



https://r4ds.had.co.nz/introduction.html

Figure 5.1: Complete set of boolean operations. `x` is the left-hand circle, `y` is the right-hand circle, and the shaded region show which parts each operator selects.

- Find all cars with either 6 cylinders or 5 gears: *filter(mtcars, cyl==6 | gear==5)*
- Find all cars with with 6 cylinders and 5 gears: *filter(mtcars, cyl==6 & gear==5)*
- Find all cars with mileage between 20 and 25mpg: *mtcars[between(mtcars$mpg, 20, 25),]*

arrange()

- Change the order of rows – sorting.
- Provide column name(s). If you provide more than one, after the first, each columns is used to break ties when sorting.
    - E.g. Sort cars by cylinder, then gears, then carburetor.
- Use *desc()* to reorder in descending order.
    - E.g. Same sort as above, but with gears in descending order.
- If you have NAs, they will be sorted and put at the bottom.

```
> arrange(mtcars, cyl, gear, carb)
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
2  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
3  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
4  33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
5  27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
6  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
7  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
8  30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
9  21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
10 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
```

```
> arrange(mtcars, cyl, desc(gear), carb)
    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
2  30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
3  22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4  32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
5  33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
6  27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
7  24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
8  22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
9  30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
10 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Examples (cont.)

1. Find the colleges in our college history data that have a “Secular” sponsorship.
2. Order them from most recent to oldest.
   - Is there any college in this list that had a different original name.
   - Retrieve that name, without looking at the data.
3. Find all the states in 1840 and order them in decreasing population size.
4. Print the top 5.

Extra: Using the Texas data, find the largest population where the median family income is >75K.

select()

- Quickly select a subset of variable from the full data frame.
- After the data, subsequent arguments are the variable to select.
  - E.g. For the cars data, only extract mileage, cylinders, horsepower, and transmission data for the cars.
- You can also select a range of columns using colon (:) operator.
  - E.g. If you only want to extract the first 4 columns of mtcars.
- Or if you want to exclude columns use –
  - E.g. Get all columns except the first 4.

```
> select(mtcars, mpg, cyl, hp, am)
                   mpg cyl  hp am
Mazda RX4         21.0   6 110  1
Mazda RX4 Wag     21.0   6 110  1
Datsun 710        22.8   4  93  1
Hornet 4 Drive    21.4   6 110  0
Hornet Sportabout 18.7   8 175  0
Valiant           18.1   6 105  0
```

```
> select(mtcars, mpg:hp)
                   mpg cyl  disp  hp
Mazda RX4         21.0   6 160.0 110
Mazda RX4 Wag     21.0   6 160.0 110
Datsun 710        22.8   4 108.0  93
Hornet 4 Drive    21.4   6 258.0 110
Hornet Sportabout 18.7   8 360.0 175
Valiant           18.1   6 225.0 105
```

```
> select(mtcars, -c(mpg:hp))
                  drat    wt  qsec vs am gear carb
Mazda RX4         3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     3.90 2.875 17.02  0  1    4    4
Datsun 710        3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 3.15 3.440 17.02  0  0    3    2
```

select() (cont.)

Built-in functions within *select():*

- *starts_with("abc")* will find variables that start with abc.
- ends_with("xyz") will find variables that end with xyz.
- contains("ijk") will find variables that contain ijk in exact order.
- And others…

You can also rename columns (variables) using *rename()*

Suppose you want to reorder variables and move some to the front, use *everything().*

- E.g. Move gears and cylinders to the front: *select(mtcars, gear, cyl, everything())*

```
> select(mtcars, starts_with("mp"))
                   mpg
Mazda RX4         21.0
Mazda RX4 Wag     21.0
Datsun 710        22.8
Hornet 4 Drive    21.4
Hornet Sportabout 18.7
```

```
> select(mtcars, contains("se"))
                   qsec
Mazda RX4         16.46
Mazda RX4 Wag     17.02
Datsun 710        18.61
Hornet 4 Drive    19.44
Hornet Sportabout 17.02
```

```
> rename(mtcars, cylind = cyl)
                mpg cylind  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0      6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0      6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8      4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive 21.4      6 258.0 110 3.08 3.215 19.44  1  0    3    1
```

mutate()

- Add new columns to the end using functions on existing columns.
- For example, when we look at mileage of a car, we also look at number of cylinders.
- So, we can create a new variable, MileagePerCylinder = mpg / cyl

```
> mutate(mtcars, MileagePerCylinder = mpg/cyl)
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb MileagePerCylinder
1 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4           3.500000
2 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4           3.500000
3 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1           5.700000
4 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1           3.566667
5 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2           2.337500
6 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1           3.016667
7 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4           1.787500
8 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2           6.100000
```

- Something in the middle: High mpg and many cylinders.
- You can also right away create new columns based on the one you just created.
- If you only want the new variables, use *transmute()*.

```
> transmute(mtcars, MileagePerCylinder = mpg/cyl)
  MileagePerCylinder
1           3.500000
2           3.500000
3           5.700000
```

Other functions for *mutate()*

- Arithmetic operations +, -, /, *, ^ all work. Vector inputs are expected. If you enter a singleton, it will be replicated.
- You can also use *%/%* for integer division and *%%* for the remainder.
- *log(), log2(),* and *log10()* all help when data ranges across multiple orders of magnitude. Logs can convert multiplicative to additive: *log a + log b = log (a*b)*
- *lead()* (leading) and *lag()* (lagging) is very handy when studying temporal trends.

```
> a = c(1:10)
> lag(a)
 [1] NA  1  2  3  4  5  6  7  8  9
> lead(a)
 [1]  2  3  4  5  6  7  8  9 10 NA
```

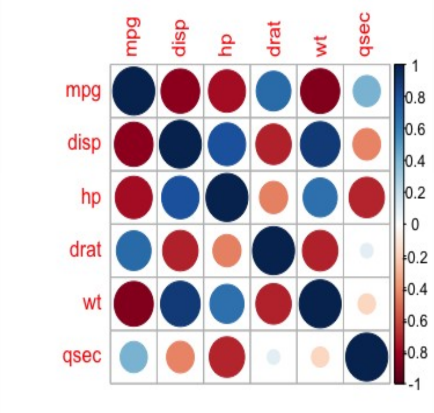
- Cumulative and rolling functions: *cumsum()* (sum), *cumprod()* (product), *cumin()* (minimum), *cummax()* (maximum), and *cummean()* (mean)

```
> cumprod(a)
 [1]       1       2       6      24     120     720    5040   40320  362880 3628800
> cummin(a)
 [1] 1 1 1 1 1 1 1 1 1 1
> cummean(a)
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
> cummax(a)
 [1]  1  2  3  4  5  6  7  8  9 10
> cumsum(a)
 [1]  1  3  6 10 15 21 28 36 45 55
```

Examples (cont..)

Install package and load *corrplot.*

1. Compute a correlation matrix from the mtcars dataset
    - Variables: mileage, displacement, horsepower, rear axle ratio, weight, and 1/4 mi time.
2. Make a correlation matrix plot using *corrplot()*.
3. Add a new indicator variable that is 1 if mpg > 23.
    - Find the mean mpg for these efficient vehicles.



<u>Extra</u>: Using the Texas data, create a new variable
minority population % = minority population * 100 / population.

summarise() & group_by()

- *summarise()* collapses the entire data frame into one row.

```
> summarize(mtcars, avgmpg = mean(mpg), minmpg = min(mpg), maxmpg = max(mpg))
    avgmpg minmpg maxmpg
1 20.09062   10.4   33.9
```

- Not too useful, unless used in conjunction with *group_by().*
- Let's see mileage stats of mtcars grouped by cylinders, gears, and carburetors:

grp_data = group_by(mtcars, cyl, gear, carb)

- Then, we can summarize it

```
> summarise(grp_data, avgmpg = mean(mpg), minmpg = min(mpg), maxmpg = max(mpg))
# A tibble: 12 x 6
# Groups:   cyl, gear [8]
    cyl  gear  carb avgmpg minmpg maxmpg
  <dbl> <dbl> <dbl>  <dbl>  <dbl>  <dbl>
1     4     3     1   21.5   21.5   21.5
2     4     4     1   29.1   22.8   33.9
3     4     4     2   24.8   21.4   30.4
4     4     5     2   28.2   26     30.4
5     6     3     1   19.8   18.1   21.4
6     6     4     4   19.8   17.8   21
```

- Note: ungroup() removes grouping

group_by(), filter() & mutate()

- Find out the mileage per cylinder of grp_data with mileage > 27mpg.

```
> filter(grp_data, mpg>27)
# A tibble: 5 x 11
# Groups:   cyl, gear, carb [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1
2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2
3  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1
4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1
5  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2
```

```
> mutate(filter(grp_data, mpg>27), MileagePerCylinder = mpg/cyl)
# A tibble: 5 x 12
# Groups:   cyl, gear, carb [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb MileagePerCylinder
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>              <dbl>
1  32.4     4  78.7    66  4.08  2.2   19.5     1     1     4     1               8.1
2  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2               7.6
3  33.9     4  71.1    65  4.22  1.84  19.9     1     1     4     1               8.48
4  27.3     4  79      66  4.08  1.94  18.9     1     1     4     1               6.82
5  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2               7.6
```

Other Useful Functions

- *dplyr* has other useful functions also
  - *sample_n()*: randomly samples *n* rows from a data frame.
    - For e.g. *sample_n(*data*, 20)* would randomly sample 20 rows from a data frame with more than 20 rows.
  - *sample_frac()*: randomly samples a percentage of rows from the data frame.
    - For e.g. *sample_frac(*data*, 0.2)* would randomly sample 20% of the rows.
- If we want 5 random observation from the mtcars data.

```
> sample_n(mtcars,5)
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb efficient
1 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4         0
2 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2         0
3 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2         1
4 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2         0
5 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4         0
```

- *replace* argument forces sampling to occur with or without replacement.

Other Useful Functions (cont.)

- Sampling functions can be useful when the data is not balanced well.
  - When there are far more observations of a value of a variable than the other values.
  - For e.g. in mtcars there are 15 cars with 3 gears, 12 cars with 4 gears, 5 cars with 5 gears.
  - If this doesn't represent the population well, we can force the sampling to occur within each of these groups.
  - Say we want 9 samples, 3 from each of the gear groups:

```
> by_GEAR <- group_by(mtcars, gear)
> sample_n(by_GEAR, 3)
# A tibble: 9 x 12
# Groups:   gear [3]
    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb efficient
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>     <dbl>
1  10.4     8 460     215  3     5.42  17.8     0     0     3     4         0
2  16.4     8 276.    180  3.07  4.07  17.4     0     0     3     3         0
3  14.7     8 440     230  3.23  5.34  17.4     0     0     3     4         0
4  21       6 160     110  3.9   2.62  16.5     0     1     4     4         0
5  22.8     4 108      93  3.85  2.32  18.6     1     1     4     1         0
6  30.4     4  75.7    52  4.93  1.62  18.5     1     1     4     2         1
7  26       4 120.     91  4.43  2.14  16.7     0     1     5     2         1
8  15       8 301     335  3.54  3.57  14.6     0     1     5     8         0
9  19.7     6 145     175  3.62  2.77  15.5     0     1     5     6         0
```

Other Useful Functions (cont..)

- Sometimes the data may need to be recoded.
  - For e.g. male/female stored as logical (1/0).
  - Or student status: freshman, sophomore, junior, senior stored as 1, 2, 3, 4.
- For analysis, we want the values to be descriptive and the class to be correct, so we need to recode it.
- We can use *case_when()* to achieve this.
- Load data("ChickWeight") and see head(ChickWeight).
  - There are 4 types of diets – coded as 1 (veges), 2 (fruits), 3 (candy), and 4 (meat).
  - We want to show the actual words instead of the numbers.

```
> mutate(ChickWeight, diet_name = case_when(
+     Diet == 1 ~ "vegetables", Diet == 2 ~ "fruit",
+     Diet == 3 ~ "candy", Diet == 4 ~ "meat"))
# A tibble: 578 x 5
   weight  Time Chick Diet  diet_name
    <dbl> <dbl> <ord> <fct> <chr>
 1     42     0 1     1     vegetables
 2     51     2 1     1     vegetables
 3     59     4 1     1     vegetables
 4     64     6 1     1     vegetables
 5     76     8 1     1     vegetables
 6     93    10 1     1     vegetables
 7    106    12 1     1     vegetables
 8    125    14 1     1     vegetables
 9    149    16 1     1     vegetables
10    171    18 1     1     vegetables
# … with 568 more rows
```

Example

- Install the package *nycflights13* and load it.
- This dataset contains all 336776 flights that departed from New York City in 2013.
- The data comes from the US Bureau of Transportation Statistics, and is documented in ?nycflights13

1. Show all flights on January 1st.
2. Find the flights on January 1st flights which had the 5 longest arrival delays.
   - Print the carrier and flight numbers.
3. Compute 2 new variables and add it onto the flights table
   - gain = arrival delay – departure delay
   - speed = (distance / airtime) * 60.
   - Print the top 5 flights (carrier and number) for the best gain in the full flights table.