

# Intermediate Data Transformation and Visualization with R

Catherine Barber

2022-09-26

## Goal and Learning Outcomes

**Goal:** The goal of this lesson is for you to learn to transform and visualize datasets using R packages.

**Learning outcomes:** During this lesson, you will demonstrate your ability to...

- Arrange data based on a variable
- Select and examine variables
- Filter a dataset based on a variable
- Count observations
- Group data based on one or more variables
- Import data
- Create a scatterplot
- Create facets

## R Refresher/Quiz

Before starting the lesson, refresh your knowledge on some key concepts from the introductory lesson, which you will need as you advance your knowledge:

1. What function do you call to install a package?
  - `install.packages()` with the name of the package in quotation marks
2. What function do you call to load a package?
  - `library()`, with no quotation marks around the name of the package
3. How do you get help on a function?
  - `help()` or `?help`
4. What are two main differences between a vector and a dataframe?
  - A vector is one-dimensional and contains a single data type, whereas a dataframe is two dimensional and can contain more than one data type.
5. What function do you call to look at the structure of an object?
  - `str()`



Figure 1: Data Investigation; Image by Dmitriy, Pixabay open license

## Scenario: Graduate Admissions Investigation

You have been contracted to assist your school with an evaluation of their recruitment and admissions practices. For the past year, the school has been making an effort to recruit and admit men and women more equitably, particularly in areas where one group or the other has been traditionally underrepresented.

After a year, the data are in, and your job is to determine whether the efforts have resulted in greater equity among candidates identifying as men and women.

For this scenario, you will use the `UCBAdmissions` dataset that “lives” in R. This dataset contains aggregated data on applicants to the UC Berkeley graduate school for the six largest departments in 1973.

Caveat: These data are purely for illustration purposes, and you will note that the variable `Gender` was coded as binary, which is an outdated practice. Although you will use the dataset as-is, be aware of social issues around classification of sex and gender and the limitations of omitting other gender options, such as non-binary.

## Load the Dataset and Examine Its Structure

To get started, load the `tidyverse` package from your library, look at the dataset by inputting its name, then call the function `glimpse()` on `UCBAdmissions`:

```
library(tidyverse)
```

```
UCBAdmissions
```

```
## , , Dept = A
##
##           Gender
## Admit      Male Female
## Admitted   512     89
## Rejected   313     19
##
## , , Dept = B
##
##           Gender
```

```
## Admit      Male Female
##   Admitted 353     17
##   Rejected 207      8
##
## , , Dept = C
##
##           Gender
## Admit      Male Female
##   Admitted 120     202
##   Rejected 205     391
##
## , , Dept = D
##
##           Gender
## Admit      Male Female
##   Admitted 138     131
##   Rejected 279     244
##
## , , Dept = E
##
##           Gender
## Admit      Male Female
##   Admitted  53      94
##   Rejected 138     299
##
## , , Dept = F
##
##           Gender
## Admit      Male Female
##   Admitted  22      24
##   Rejected 351     317
```

```
glimpse(UCBAdmissions)
```

```
## 'table' num [1:2, 1:2, 1:6] 512 313 89 19 353 207 17 8 120 205 ...
## - attr(*, "dimnames")=List of 3
##   ..$ Admit : chr [1:2] "Admitted" "Rejected"
##   ..$ Gender: chr [1:2] "Male" "Female"
##   ..$ Dept  : chr [1:6] "A" "B" "C" "D" ...
```

First, notice that the dataset looks like a table. There are six cross-tabs (one per Department) with frequency data for combinations of gender and admission status. In other words, there are three variables.

Second, the output of `glimpse()` shows us the structure of the data, similar to the function `str()`.

Sometimes, you just need to know the class of a dataset to determine if it is already in the desired format or not.

```
class(UCBAdmissions)
```

```
## [1] "table"
```

As we saw with the `glimpse()` function, the output shows that `UCBAdmissions` is a table. You need to convert the dataset to a dataframe using the function `as.data.frame()`, which coerces another object to a dataframe object.

Assign this new dataframe to an object called `admissions` and examine the structure of the object with `str()`:

```
(admissions <- as.data.frame(UCBAdmissions))
```

```
##      Admit Gender Dept Freq
## 1 Admitted   Male    A  512
## 2 Rejected   Male    A  313
## 3 Admitted Female    A   89
## 4 Rejected Female    A   19
## 5 Admitted   Male    B  353
## 6 Rejected   Male    B  207
## 7 Admitted Female    B   17
## 8 Rejected Female    B    8
## 9 Admitted   Male    C  120
##10 Rejected   Male    C  205
##11 Admitted Female    C  202
##12 Rejected Female    C  391
##13 Admitted   Male    D  138
##14 Rejected   Male    D  279
##15 Admitted Female    D  131
##16 Rejected Female    D  244
##17 Admitted   Male    E   53
##18 Rejected   Male    E  138
##19 Admitted Female    E   94
##20 Rejected Female    E  299
##21 Admitted   Male    F   22
##22 Rejected   Male    F  351
##23 Admitted Female    F   24
##24 Rejected Female    F  317
```

```
str(admissions)
```

```
## 'data.frame':   24 obs. of  4 variables:
## $ Admit : Factor w/ 2 levels "Admitted","Rejected": 1 2 1 2 1 2 1 2 1 2 ...
## $ Gender: Factor w/ 2 levels "Male","Female": 1 1 2 2 1 1 2 2 1 1 ...
## $ Dept  : Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 2 2 2 2 3 3 ...
## $ Freq  : num  512 313 89 19 353 207 17 8 120 205 ...
```

Now the data are in the right format! Each row represents a combination of Dept, Gender, and Admit subgroup, and Frequency is the number of people in that subgroup. For example, the first four rows are Department A, with row 1 being the number of male applicants who were admitted, row 2 the number of male applicants who were rejected, row 3 the number of female applicants who were admitted, etc. The three variables are factors.

## Tibbles

Small datasets such as this one are easy to see in their entirety when we assign them to a dataframe object. However, larger datasets are more unwieldy.

An alternative is to create a tidyverse object called a tibble. A tibble is a special dataframe that works well with various tidyverse functions. The function `as_tibble` converts a dataset or dataframe to a tibble.

Convert the `admissions` dataframe to a tibble. One of the default arguments for this function is `colnames = TRUE`, which indicates that the column names of the original object will become the column names of the tibble. This can be changed to `FALSE` if, for example, you are converting a table with no column names to a tibble.

```
(admissions_tib <- as_tibble(admissions, colnames = TRUE))
```

```
## # A tibble: 24 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Rejected Male   A     313
## 3 Admitted Female A      89
## 4 Rejected Female A      19
## 5 Admitted Male   B    353
## 6 Rejected Male   B    207
## 7 Admitted Female B      17
## 8 Rejected Female B       8
## 9 Admitted Male   C    120
## 10 Rejected Male  C    205
## # ... with 14 more rows
```

Notice that the tibble only displays the first 10 rows and as many columns as can fit on the screen, labeled with the type of data the columns contain.

Sometimes, however, you want to see additional rows. R gives you a hint about how to do this: `print(n = ...)` to see more rows. Try that out!

```
print(admissions_tib, n = 20)
```

```
## # A tibble: 24 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Rejected Male   A     313
## 3 Admitted Female A      89
## 4 Rejected Female A      19
## 5 Admitted Male   B    353
## 6 Rejected Male   B    207
## 7 Admitted Female B      17
## 8 Rejected Female B       8
## 9 Admitted Male   C    120
## 10 Rejected Male  C    205
## 11 Admitted Female C    202
## 12 Rejected Female C    391
## 13 Admitted Male   D    138
## 14 Rejected Male   D    279
## 15 Admitted Female D    131
## 16 Rejected Female D    244
## 17 Admitted Male   E     53
## 18 Rejected Male   E    138
## 19 Admitted Female E     94
## 20 Rejected Female E    299
## # ... with 4 more rows
```

This displays 20 rows of data!

## Find and Edit Commands in the Console

As a rule of thumb, it is best to work in the script editor rather than in the command console. However, occasionally you might want to try something out quickly without a plan to save it in the script. For example, you may want to practice changing the values of various arguments or call help on several functions.

For these purposes, you can use the command console rather than inputting commands in the script editor and then having to delete them. A shortcut to find previous commands that have been run is to use the up or down arrow to scroll through those commands in the console.

Practice by using the up arrow on your keyboard to scroll to the previous command `print(admissions_tib, n = 20)`. Edit this command so that `n = Inf`:

```
print(admissions_tib, n = Inf)
```

```
## # A tibble: 24 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Rejected Male   A     313
## 3 Admitted Female A      89
## 4 Rejected Female A      19
## 5 Admitted Male   B     353
## 6 Rejected Male   B     207
## 7 Admitted Female B      17
## 8 Rejected Female B       8
## 9 Admitted Male   C     120
## 10 Rejected Male  C     205
## 11 Admitted Female C     202
## 12 Rejected Female C     391
## 13 Admitted Male  D     138
## 14 Rejected Male  D     279
## 15 Admitted Female D     131
## 16 Rejected Female D     244
## 17 Admitted Male  E      53
## 18 Rejected Male  E     138
## 19 Admitted Female E      94
## 20 Rejected Female E     299
## 21 Admitted Male  F      22
## 22 Rejected Male  F     351
## 23 Admitted Female F      24
## 24 Rejected Female F     317
```

The output is the entire tibble.

## More about Tibbles

In the current tibble, the rows represent groups, and the frequency indicates the number of people in each group. In other tibbles you will encounter, each row will represent a person. In either case, tibbles prefer “tidy” data, which means that each row represents a single observation, each cell represents a single value, and each column represents a single variable.

**Caveat:** There are pros and cons to this approach. A major pro is that the format makes the data compatible with various tidyverse functions. A downside comes when variables represent repeated measures, such as pre and post test scores for the same person. Some types of statistical analyses, such as paired  $t$  tests, are more easily calculated when the two measures appear on the same row as separate variables, rather than two separate rows representing two observations for the same variable. You'll just need to think through the analyses you will perform before determining how to organize your data.

Nonetheless, all of the transformations in this lesson will work with the tidy format of the tibble.

## Overview of dplyr

Now you are ready to transform your dataset! You will use **dplyr**, a package in the tidyverse for data transformation. The **dplyr** package uses a pipe process to pass one or more functions (called “verbs”) to a dataframe or tibble.

The pipe function is `%>%`; make sure you do not put spaces between these characters.

**Tip:** If you like to chunk code by using the Enter/Return key to start a line for each new function within a pipe, put the pipe at the end of the previous line rather than at the beginning of the new line. Putting it at the beginning will return an error.

You can build a pipe with numerous verbs, each telling R how to transform the data in a step-by-step process. Note that this does not affect the original tibble, so if you want to save the output of a pipe, you will need to assign it to a new object.

## Arrange with arrange()

The verb **arrange()** is for arranging or sorting data based on one or more variables. The default is for R to arrange rows in ascending order based on the alphanumeric value of the variable specified, but you can add an argument **desc()** to arrange data in descending order.

### Exercise 1a: Arrange Data in Ascending Order

Return to the graduate admissions investigation scenario. Looking at the entire dataset, it appears at first glance that there are gender-related differences in the patterns of admission and rejection. Your task is to be more precise with your analysis of these differences.

As a starting point, you decide to look at the data for applicants who were admitted first by arranging the data so that Admitted rows appear before Rejected rows. From looking at the dataset previously, you already know that there are 12 rows for admitted applicants, so you specify that R should return 12 rows.

Furthermore, because A (for Admitted) comes before R (for Rejected), you can use the default, which is to arrange rows in alphabetical order based on the values of the variable.

```
admissions_tib %>%  
  arrange(Admit) %>%  
  print(n = 12)
```

```
## # A tibble: 24 x 4  
##   Admit   Gender Dept   Freq  
##   <fct>   <fct> <fct> <dbl>  
## 1 Admitted Male   A     512  
## 2 Admitted Female A      89  
## 3 Admitted Male   B     353
```

```
## 4 Admitted Female B      17
## 5 Admitted Male   C     120
## 6 Admitted Female C     202
## 7 Admitted Male   D     138
## 8 Admitted Female D     131
## 9 Admitted Male   E      53
## 10 Admitted Female E      94
## 11 Admitted Male   F      22
## 12 Admitted Female F      24
## # ... with 12 more rows
```

The output is the first 12 rows of the tibble: all admitted applicants.

### Exercise 1b: Arrange Data Based on Two Variables

Now you want to do a quick comparison of male and female applicants who were admitted vs. rejected in each department. You can specify that you want to arrange first by department (Dept) and then by admission status (Admit).

Note that the order of variables in the argument matters: specify what to sort first and second, in that order. Also note that you want all rows, so you call `print(n = Inf)` again.

```
admissions_tib %>%
  arrange(Dept, Admit) %>%
  print(n = Inf)
```

```
## # A tibble: 24 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Admitted Female A      89
## 3 Rejected Male   A     313
## 4 Rejected Female A      19
## 5 Admitted Male   B     353
## 6 Admitted Female B      17
## 7 Rejected Male   B     207
## 8 Rejected Female B       8
## 9 Admitted Male   C     120
## 10 Admitted Female C     202
## 11 Rejected Male   C     205
## 12 Rejected Female C     391
## 13 Admitted Male   D     138
## 14 Admitted Female D     131
## 15 Rejected Male   D     279
## 16 Rejected Female D     244
## 17 Admitted Male   E      53
## 18 Admitted Female E      94
## 19 Rejected Male   E     138
## 20 Rejected Female E     299
## 21 Admitted Male   F      22
## 22 Admitted Female F      24
## 23 Rejected Male   F     351
## 24 Rejected Female F     317
```



The tibble groups by department first (A through F, alphabetically) and then by Admit status, in this case, Admitted first and then Rejected. Keep in mind that each of these tibbles is the same dataframe, just arranged in different ways.

### Exercise 2a: Arrange Data in Descending Order

Call the following and view the result.

**Tip:** Be sure to include both parentheses at the end of the script to avoid an error message!

```
admissions_tib %>%  
  arrange(desc(Gender))
```

```
## # A tibble: 24 x 4  
##   Admit   Gender Dept   Freq  
##   <fct>   <fct> <fct> <dbl>  
## 1 Admitted Female A      89  
## 2 Rejected Female A      19  
## 3 Admitted Female B      17  
## 4 Rejected Female B       8  
## 5 Admitted Female C     202  
## 6 Rejected Female C     391  
## 7 Admitted Female D     131  
## 8 Rejected Female D     244  
## 9 Admitted Female E      94  
## 10 Rejected Female E     299  
## # ... with 14 more rows
```

R has sorted the data by Gender—all female applicants appear first. But why do Females appear first when you included the argument `desc(Gender)`? Shouldn't males appear first, since you told R to sort in descending (i.e., reverse alphabetical) order?

There must be something in the variable Gender that is responsible for how the data are arranged. Look at the structure of the tibble again for a clue.

### Exercise 2b: Troubleshoot

```
str(admissions_tib)
```

```
## tibble [24 x 4] (S3: tbl_df/tbl/data.frame)  
## $ Admit : Factor w/ 2 levels "Admitted","Rejected": 1 2 1 2 1 2 1 2 1 2 ...  
## $ Gender: Factor w/ 2 levels "Male","Female": 1 1 2 2 1 1 2 2 1 1 ...  
## $ Dept  : Factor w/ 6 levels "A","B","C","D",...: 1 1 1 1 2 2 2 2 3 3 ...  
## $ Freq  : num [1:24] 512 313 89 19 353 207 17 8 120 205 ...
```

The output indicates that Gender is a factor that was coded as 1 for Male and 2 for Female. Thus, descending order returns Females first, as this level has a larger coded number than Males. For factors, the order will be based on the levels' coded values, not on their alphabetical value. Always be sure to review the coding system for your data, as this will play a part in how R interprets numbers assigned to categorical data.

## R Tip: Arrange

Keep in mind that `arrange()` sorts the data based on the selected variable(s). However, the order of the variables in the tibble does not change. If you have a big tibble with many variables, you might want to just look at certain variables, and you might want to specify the order in which these variables (i.e., columns) appear in the tibble. You can accomplish both of these tasks with the next verb: `select()`.

## Select Variables to Display with `select()`

You decide to narrow your focus for a moment to look at Gender, Department, and Frequency, omitting Admit. The order of variables specified in the argument determines their order in the resulting tibble.

```
admissions_tib %>%
  select(Gender, Dept, Freq)
```

```
## # A tibble: 24 x 3
##   Gender Dept   Freq
##   <fct> <fct> <dbl>
## 1 Male   A     512
## 2 Male   A     313
## 3 Female A      89
## 4 Female A      19
## 5 Male   B     353
## 6 Male   B     207
## 7 Female B      17
## 8 Female B       8
## 9 Male   C     120
## 10 Male  C     205
## # ... with 14 more rows
```

The output is a tighter tibble with only the variables selected. Note, however, that this function did not collapse any rows: Admitted and Rejected are still shown as separate rows (simply not labeled). With the current dataset, `select()` is not really necessary, but for large datasets with dozens of variables, `select()` is extremely handy for zooming in on just those variables of interest.

## Omit Variables to Display

Conversely, if you want to omit one or more variables, you can use `select()` and specify the variables to omit with a minus sign before the variable name. For example, you just want to see Gender and Freq:

```
admissions_tib %>%
  select(-Admit, -Dept)
```

```
## # A tibble: 24 x 2
##   Gender Freq
##   <fct> <dbl>
## 1 Male   512
## 2 Male   313
## 3 Female  89
## 4 Female  19
## 5 Male   353
```

```
## 6 Male      207
## 7 Female    17
## 8 Female     8
## 9 Male     120
## 10 Male    205
## # ... with 14 more rows
```

Now, only the two desired variables are displayed. Again, all original rows are preserved; only the columns are omitted.

### Select a Series of Variables to Display

If you want to select a series of consecutive variables, you can simply use a colon between the first and last variable inside `select()`. Note that if you specify the last variable in the sequence first, the variables will appear in reverse order.

Here is an example: You want all variables from Gender through Freq, but you want Freq to appear first.

```
admissions_tib %>%
  select(Freq:Gender)
```

```
## # A tibble: 24 x 3
##   Freq Dept  Gender
##   <dbl> <fct> <fct>
## 1   512 A    Male
## 2   313 A    Male
## 3    89 A    Female
## 4    19 A    Female
## 5   353 B    Male
## 6   207 B    Male
## 7    17 B    Female
## 8     8 B    Female
## 9   120 C    Male
## 10  205 C    Male
## # ... with 14 more rows
```

As before, all rows are preserved; however, Admit does not appear in the tibble because it was not in the range of variables specified in the argument.

### Exercise 3: Select Variables and Arrange Data

Now try creating a pipeline by calling two functions in the same script!

Back to the scenario: To protect departments' privacy, your boss asks you to generate a table with all of the Rejected rows shown at the top; the table should not specify which department each row is associated with. Thus, you need a table that omits Dept.

```
admissions_tib %>%
  select(-Dept) %>%
  arrange(desc(Admit)) %>%
  print(n = 12)
```

```
## # A tibble: 24 x 3
##   Admit   Gender Freq
##   <fct>   <fct> <dbl>
## 1 Rejected Male    313
## 2 Rejected Female   19
## 3 Rejected Male    207
## 4 Rejected Female    8
## 5 Rejected Male    205
## 6 Rejected Female   391
## 7 Rejected Male    279
## 8 Rejected Female   244
## 9 Rejected Male    138
## 10 Rejected Female  299
## 11 Rejected Male    351
## 12 Rejected Female  317
## # ... with 12 more rows
```

In this pipe, you select all variables except Dept by using `select(-Dept)`. That removes Dept from the tibble, protecting the departments' identities. Then you arrange the rows in descending order, as Rejected as coded as 2 and Admitted is coded as 1: `arrange(desc(Admit))`. Finally, you print the first 12 rows, which correspond to the rows for the rejected groups: `print(n = 12)`.

### R Tip: Helper Functions

`dplyr` has additional functions called “helpers” that operate inside the verb `select()`. Take a look at a few for future reference:

- `starts_with("string")` selects any variables that start with the string specified in parentheses. Be sure to include quotation marks around the string!
- `ends_with("string")` selects any variables that end with the string.
- `contains("string")` selects any variables that contain the string.
- `everything()` selects all variables and can be used with other variables to rearrange the order of variables in a tibble. This helper is handy if you want to move just a few columns to the far left of the tibble but don't care where other columns appear.

### Exercise 4: Helper Functions

Review the following command and guess what the output will be. Then check the output!

```
admissions_tib %>%
  select(Dept, everything()) %>%
  arrange(Dept)
```

```
## # A tibble: 24 x 4
##   Dept Admit   Gender Freq
##   <fct> <fct>   <fct> <dbl>
## 1 A      Admitted Male    512
## 2 A      Rejected Male    313
## 3 A      Admitted Female   89
## 4 A      Rejected Female   19
## 5 B      Admitted Male    353
## 6 B      Rejected Male    207
```

```
## 7 B      Admitted Female    17
## 8 B      Rejected Female     8
## 9 C      Admitted Male     120
## 10 C     Rejected Male     205
## # ... with 14 more rows
```

Did you guess that the tibble would include all variables and that Dept would appear first? Note, too, that rows are arranged by Dept, as specified by the second part of the pipeline.

## Count with count()

Now you are ready to transform your data further by collapsing some rows. The term “collapsing” means combining two or more rows and ignoring membership within a particular group. For instance, if you collapse across gender, you group male and female rows together rather than treating male rows and female rows separately. This point will become clearer in a moment.

The basis of this task is the verb `count()`, which counts the number of rows within each value of a specified variable. Take a look at how this operates:

```
admissions_tib %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1    24
```

If you call `count()` without an argument, R simply counts the number of rows in the dataset. Since you already know that there are 24 rows, this isn’t very helpful. However, when `count()` is used with various arguments and in conjunction with other verbs, it becomes extremely useful. But it’s important to know what the basic purpose of `count()` is: counting rows.

## Weighted Count

Now add an argument: `wt = variable_name`. This will weight the count by the variable specified, which translates to summing all of the values in that column.

```
admissions_tib %>%
  count(wt = Freq)
```

```
## # A tibble: 1 x 1
##       n
##   <dbl>
## 1  4526
```

R weighted each row by the value of `Freq`, or the frequency of applicants for that row. The output is the sum of all values, i.e., the total number of applicants: 4,526.

## Count Based on a Variable

There is one more piece to learn before putting everything together and using `count()` to its full extent. If you add a variable name as an argument in `count()`, this will call the output based on the variable. For example:

```
admissions_tib %>%  
  count(Dept)
```

```
## # A tibble: 6 x 2  
##   Dept      n  
##   <fct> <int>  
## 1 A         4  
## 2 B         4  
## 3 C         4  
## 4 D         4  
## 5 E         4  
## 6 F         4
```

This counts the number of rows associated with each value of `Dept`, that is, each department. The output shows that there are 6 departments and that each department has 4 rows of data.

When might this be useful? One application is for checking where missing data are. For example, if you have a large dataset with hundreds of rows and you expect each level of a variable to have an equal number of rows, you can call `count()` on that variable to check whether your expectations are met or not. Here, if you expected each department to provide four rows of data and one department has only three, then you will need to follow up with that department to locate the missing data.

Another time when this function can be useful is when the rows in the dataset represent person-level data, meaning measurements from individual people. You can call `count()` on a variable to determine how many people have each possible value for that measure.

## Weighted Count Based on a Variable

Now it's time to put everything together. Your boss asks you to calculate the total number of applicants for each department.

```
admissions_tib %>%  
  count(Dept, wt = Freq, sort = TRUE)
```

```
## # A tibble: 6 x 2  
##   Dept      n  
##   <fct> <dbl>  
## 1 A      933  
## 2 C      918  
## 3 D      792  
## 4 F      714  
## 5 B      585  
## 6 E      584
```

R collapses the data across the variables that were not specified (`Gender` and `Admit`), which means that all of the rows for each `Department` are treated together. R then weights each level of `Dept` by the value of

`Freq` for each row associated with that department. The count, shown in the tibble as `n`, is the sum of those weights. Finally, `sort = TRUE` sorts the results by the `n` column value, in descending order.

The output is a 6 x 2 tibble showing the total number of applicants for each department. The results indicate that Dept. A had the largest number of applicants, followed closely by Dept. C. The smallest numbers of applicants were in Depts. E and B.

### Exercise 5: Weighted Count

When you have frequency data such as those in this dataset, calling a weighted count on each categorical variable (i.e., factor) is a great way to determine how many observations are in the different subgroups in your data.

Try obtaining the frequencies for the other two variables. First, find the total number of male and female applicants. Then find the total number of admitted and rejected applicants.

```
admissions_tib %>%  
  count(Gender, wt = Freq, sort = TRUE)
```

```
## # A tibble: 2 x 2  
##   Gender      n  
##   <fct>   <dbl>  
## 1 Male    2691  
## 2 Female  1835
```

The first result is that there were 2,691 male applicants and 1,835 female, or approximately a 3:2 ratio of men to women.

```
admissions_tib %>%  
  count(Admit, wt = Freq, sort = TRUE)
```

```
## # A tibble: 2 x 2  
##   Admit      n  
##   <fct>   <dbl>  
## 1 Rejected 2771  
## 2 Admitted 1755
```

The second result is that there were 2,771 applicants rejected and 1,755 applicants admitted. This is also approximately a 3:2 ratio of rejected to admitted applicants.

### Filter with filter()

In the introductory R lesson, you used indexing (`[]` or `?`) to filter the data according to certain criteria. Within the tidyverse, you can use `filter()` to achieve the same end, with the added benefit of being able to filter within a longer pipeline.

As with any index, you can use conditionals such as `>`, `<`, and `==` to provide the criteria you want to filter by. Note that when you filter using a string variable, quotation marks are required; however, unlike indexing in base R, you can use the column names in the filter directly.

Here is an example:

As you are working on your investigation, the chair of Department A calls your office and wants to know that department's pattern of admissions and rejections. You can simply filter the data to obtain only rows for Department A.

```
admissions_tib %>%
  filter(Dept == "A")
```

```
## # A tibble: 4 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Rejected Male   A     313
## 3 Admitted Female A       89
## 4 Rejected Female A      19
```

By eyeballing the rows for this department, you can see that far more men than women applied, but the rate of admission was higher for women than for men.

### Filter on Multiple Variables

You can use `filter()` with multiple variables, and you may find that the code is simpler than the indexes you crafted in base R.

Look at the call below and see if you can determine what rows will be in the tibble before reviewing the output.

```
admissions_tib %>%
  filter(Admit == "Admitted" & Freq > 100)
```

```
## # A tibble: 6 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A     512
## 2 Admitted Male   B     353
## 3 Admitted Male   C     120
## 4 Admitted Female C     202
## 5 Admitted Male   D     138
## 6 Admitted Female D     131
```

Did you guess that R would first find all Admitted rows and then narrow those down to the rows that have a frequency of >100? The output is a tibble of departments that admitted more than 100 male applicants and/or more than 100 female applicants.

### Exercise 6: Filter on Multiple Variables

Use `filter()` to identify the departments that rejected 100 or more female applicants, then arrange the results in descending order of frequency. Hint: You'll need to use `filter()` with three criteria and include the `desc()` argument within `arrange()`.

```
admissions_tib %>%
  filter(Admit == "Rejected" & Gender == "Female" & Freq >= 100) %>%
  arrange(desc(Freq))
```



```
## # A tibble: 4 x 4
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Rejected Female C       391
## 2 Rejected Female F       317
## 3 Rejected Female E       299
## 4 Rejected Female D       244
```

The results show the departments that rejected 100 or more female applicants.

## Group Data by a Variable with `group_by()`

Perhaps the most powerful verb in the `dplyr` package is `group_by()`. On its own, it doesn't do much, but when combined with other verbs, it is extremely useful.

This verb groups all rows associated with a particular level of a specified variable and treats them as a group. When you call subsequent functions involving some sort of calculation, such as `count()`, `group_by()` will perform the calculations for groups of rows rather than for all rows.

First look at the basic `group_by()` call.

```
admissions_tib %>%
  group_by(Dept)
```

```
## # A tibble: 24 x 4
## # Groups:   Dept [6]
##   Admit   Gender Dept   Freq
##   <fct>   <fct> <fct> <dbl>
## 1 Admitted Male   A       512
## 2 Rejected Male   A       313
## 3 Admitted Female A        89
## 4 Rejected Female A        19
## 5 Admitted Male   B       353
## 6 Rejected Male   B       207
## 7 Admitted Female B        17
## 8 Rejected Female B         8
## 9 Admitted Male   C       120
## 10 Rejected Male  C       205
## # ... with 14 more rows
```

Note that the output doesn't look any different from the original tibble `admissions_tib`. That's because you haven't performed any calculations on the rows! However, you will notice under the description of the tibble (in this case, a 24 x 4 tibble) that `dplyr` indicates the groups that were formed by the call (in this case, 6 groups based on `Dept`).

## Combine `group_by()` with `count()`

Now look at an example of using `group_by()` with `count()`. Remember that `count()` counts the number of rows associated with a variable. A weighted count involves weighting each row by the value of some other variable, such as frequency.

As part of your investigation, you want to determine the number of admitted and rejected applicants for each department, collapsing across gender. You will group the data by `Dept` and perform a weighted count of `Admit`, which will calculate the total number of admissions and rejections for each department.

```
admissions_tib %>%
  group_by(Dept) %>%
  count(Admit, wt = Freq)
```

```
## # A tibble: 12 x 3
## # Groups:   Dept [6]
##   Dept Admit      n
##   <fct> <fct>   <dbl>
## 1 A     Admitted  601
## 2 A     Rejected  332
## 3 B     Admitted  370
## 4 B     Rejected  215
## 5 C     Admitted  322
## 6 C     Rejected  596
## 7 D     Admitted  269
## 8 D     Rejected  523
## 9 E     Admitted  147
## 10 E    Rejected  437
## 11 F    Admitted   46
## 12 F    Rejected  668
```

The output includes each department's admitted and rejected frequencies. Note that **Gender** is not included as a column because R has collapsed across gender. The results show that departments A and B admit far more applicants than they reject, whereas departments C, D, E, and F reject far more than they admit.

**Caveat:** Be sure to call `group_by()` before other verbs in the pipeline that you want to perform on the groups.

### Exercise 7a: Combine `group_by()` with `count()`

You are ready to start working on your investigation in earnest. You first want to find the school-wide admission and rejection numbers for men and women separately. This will involve counting total admissions and total rejections for men and for women, irrespective of the department that they applied to.

You will group by **Gender** and call `count()` on the weighted value of **Admit**.

```
admissions_tib %>%
  group_by(Gender) %>%
  count(Admit, wt = Freq)
```

```
## # A tibble: 4 x 3
## # Groups:   Gender [2]
##   Gender Admit      n
##   <fct> <fct>   <dbl>
## 1 Male   Admitted  1198
## 2 Male   Rejected  1493
## 3 Female Admitted   557
## 4 Female Rejected  1278
```

The output is a 4x3 tibble. You can see the overall numbers of admitted and rejected male applicants and admitted and rejected female applicants for the entire school.

One thing that stands out, just by glancing at the data, is that the school-wide admission rates are quite different for men and women. Men are accepted at a higher rate, overall.

## Parse `group_by()`

Take a minute to parse the `group_by()` function in more depth. When you pass a variable to `group_by()`, R identifies all of the rows associated with each level of that variable and groups them together.

For example, if you call `group_by(Gender)`, all of the rows with the level `Female` for `Gender` will be grouped together and all of rows with the level `Male` for `Gender` will be grouped together. This collapses the data across `Dept` and `Admit`.

The illustration below shows all of the rows associated with the `Female` level of `Gender`. These rows are all grouped together with the verb `group_by`.

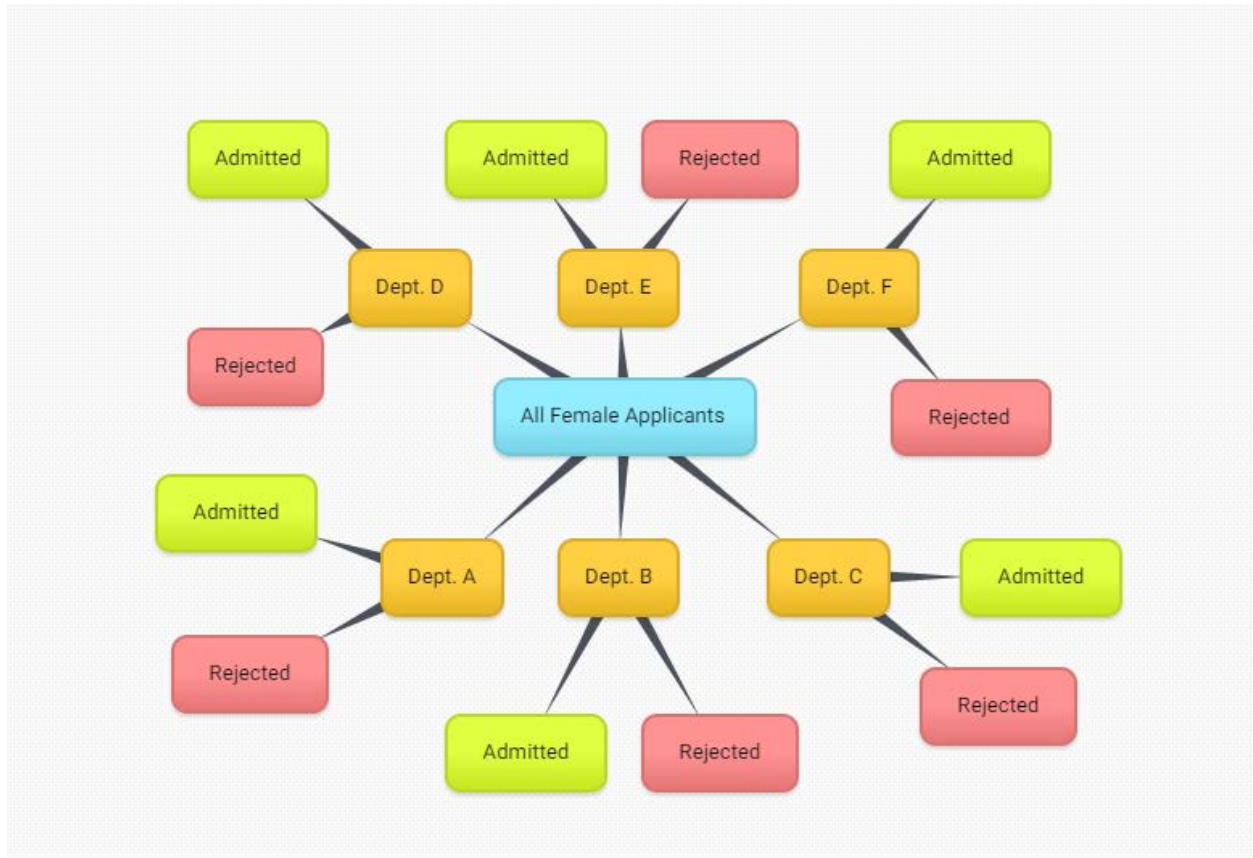


Figure 2: All Female Rows in the Dataset, Grouped Together by `group_by()`

Grouping by a variable is *not* the same as collapsing on that variable. To keep this clear, you can think of the `group_by()` function as *grouping rows with the same level of a variable, for the purpose of performing calculations separately on each level of the variable*.

After grouping by `Gender`, when you called a weighted count on `Admit`, R pulled out the two levels of `Admit` and counted the frequency for each of those levels.

The result of this call answered the question, “Among the men, how many applicants were admitted vs. rejected, and among the women, how many applicants were admitted vs. rejected?”

**Caveat:** Choose your grouping variable(s) carefully! Consider the illustration below.

In the second illustration, you are grouping by `Admit` and then counting the number of men and women within each of the `Admit` levels:

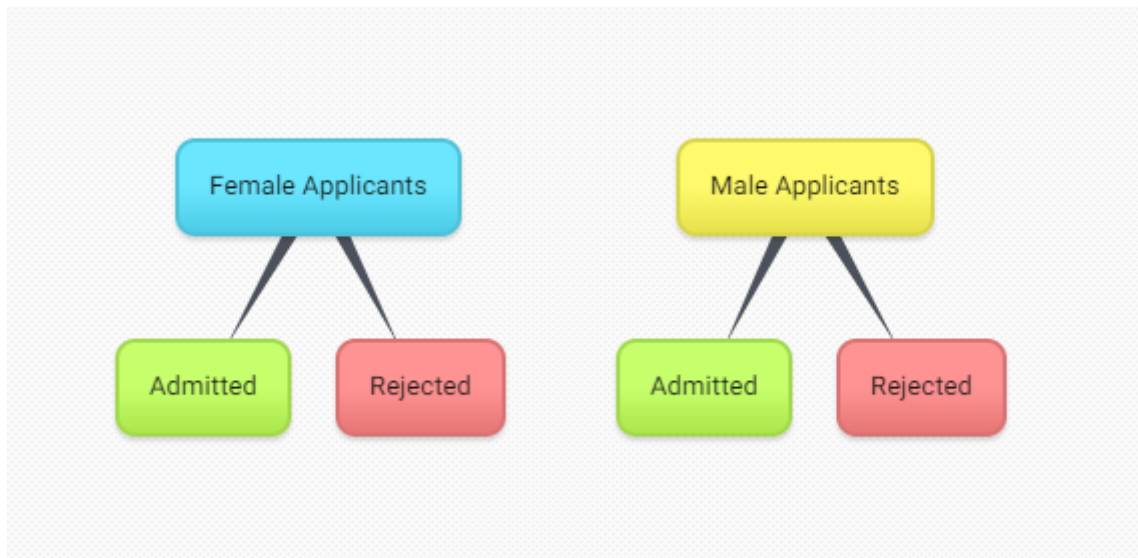


Figure 3: How R Parses `group_by(Gender)` and `count(Admit)`

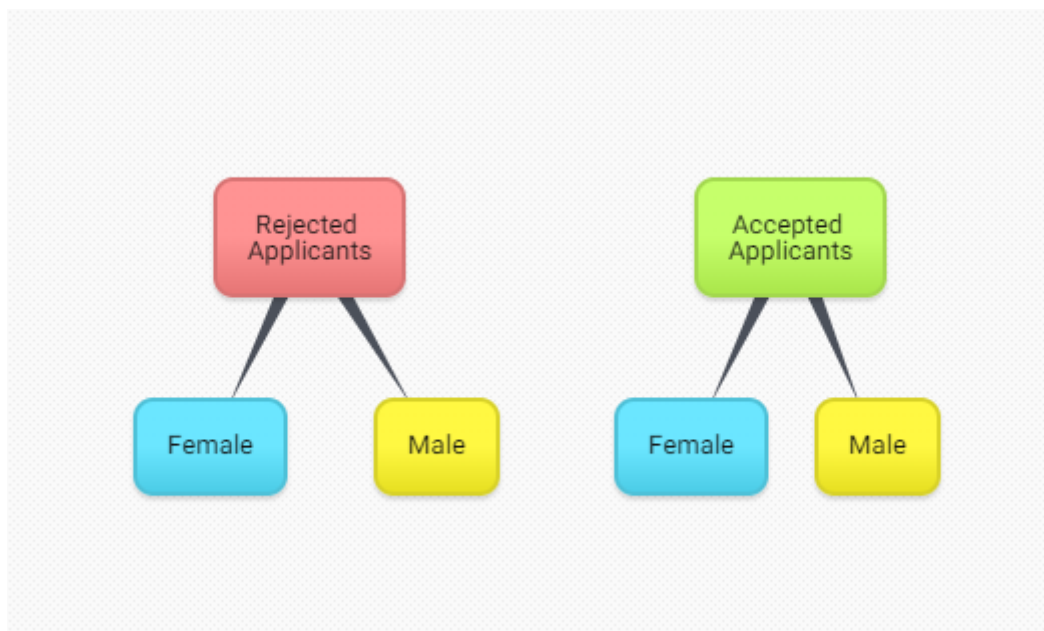


Figure 4: How R Parses `group_by(Admit)` and `count (Gender)`

```
admissions_tib %>%
  group_by(Admit) %>%
  count(Gender, wt = Freq)
```

```
## # A tibble: 4 x 3
## # Groups:   Admit [2]
##   Admit   Gender     n
##   <fct>   <fct> <dbl>
## 1 Admitted Male   1198
## 2 Admitted Female  557
## 3 Rejected Male   1493
## 4 Rejected Female 1278
```

The actual results of this call are identical to those of the previous call! But you will notice that the output is arranged differently. The second call answers the question, “Among those admitted, how many were men vs. women, and among those rejected, how many were men vs. women?”

This difference may not matter for simple counts, but it does matter when you perform calculations, such as percentages and means, on specific subgroups. The grouping variable determines the denominator of such calculations, as you will see in a moment.

### Exercise 7b: Combine `group_by()` and `count()`

At this point, your school’s admission team wants to know how many men and women applied to each department. You create a pipe to answer this question.

```
admissions_tib %>%
  group_by(Dept) %>%
  count(Gender, wt = Freq)
```

```
## # A tibble: 12 x 3
## # Groups:   Dept [6]
##   Dept Gender     n
##   <fct> <fct> <dbl>
## 1 A     Male   825
## 2 A     Female 108
## 3 B     Male   560
## 4 B     Female  25
## 5 C     Male   325
## 6 C     Female 593
## 7 D     Male   417
## 8 D     Female 375
## 9 E     Male   191
## 10 E    Female 393
## 11 F    Male   373
## 12 F    Female 341
```

The output shows that two departments, D and F, had somewhat similar numbers of male and female applicants. In contrast, there was a big difference between the number of male and female applicants in departments A and B, for example.

## Recap

So far, you have conducted several analyses on this dataset with the goal of answering the initial question: how equitable are the school's application and admission outcomes in terms of gender? You looked at the following:

- Unweighted counts with `count()`:
  - `count()`: total number of rows in the dataset
  - `count(Dept)`: number of rows for each department
- Weighted counts with `count(..., wt = Freq)`
  - `count(wt = Freq)`: total number of applicants
  - `count(Dept, wt = Freq)`: number of applicants in each department
  - `count(Gender, wt = Freq)`: number of male and female applicants
  - `count(Admit, wt = Freq)`: number of admitted and rejected applicants
- Weighted group counts with `group_by() %>% count(..., wt = Freq)`
  - `group_by(Dept) %>% count(Admit, wt = Freq)`: number of admissions vs. rejections in each department
  - `group_by(Gender) %>% count(Admit, wt = Freq)`: number of admitted vs. rejected applicants among males vs. females
  - `group_by(Admit) %>% count(Gender, wt = Freq)`: number of males vs. females among admitted vs. rejected applicants
  - `group_by(Dept) %>% count(Gender, wt = Freq)`: number of males vs. females applying to each department

Here is a summary of the findings from these analyses:

- The total number of applicants to this school is 4,526.
- Departments A and B have the largest number of applicants.
- The ratio of male to female applicants is approximately 3:2.
- The ratio of rejected to admitted applicants is approximately 3:2.
- School-wide, men have a higher rate of admission than women.
- Departments A and B are least selective (i.e., higher rates of Admit).
- Departments E and F are most selective.
- Far more men than women apply to Departments A and B.
- Far more women than men apply to Departments C and E.

These are interesting observations. There does appear to be a gender discrepancy at the school level in terms of applications and admissions, but there are some interesting patterns at play at the departmental level that need to be investigated.

Next you will drill down into department-level data and calculate some percentages.

## Create a New Variable with `mutate()`

So far, you have been working with counts/frequencies. Often, however, you will want to look at other calculations, such as percentages or means. For example, to determine whether men and women have comparable outcomes, you need to convert frequencies to percentages.

A handy function for accomplishing this task is `mutate()`, which creates a new variable based on existing variables, using a specified formula. The formula can include other functions, such as `sum()` or `mean()`. The

formula for obtaining a percentage for a row involves dividing the row frequency (i.e., the `Freq` column in the current dataset) by the sum of all frequencies and then multiplying that number by 100:

```
Freq / sum(Freq) * 100
```

Note that if the frequency column were named something else, such as `n` or `count`, then you would simply replace `Freq` with that column name.

The format for creating a new variable is `mutate(new_variable = formula(existing_variable))`. Here is an example:

```
admissions_tib %>%  
  mutate(percent = Freq / sum(Freq) * 100) %>%  
  print(n = Inf)
```

```
## # A tibble: 24 x 5  
##   Admit   Gender Dept   Freq percent  
##   <fct>   <fct> <fct> <dbl>   <dbl>  
## 1 Admitted Male   A     512    11.3  
## 2 Rejected Male   A     313     6.92  
## 3 Admitted Female A      89     1.97  
## 4 Rejected Female A      19     0.420  
## 5 Admitted Male   B     353     7.80  
## 6 Rejected Male   B     207     4.57  
## 7 Admitted Female B      17     0.376  
## 8 Rejected Female B       8     0.177  
## 9 Admitted Male   C     120     2.65  
## 10 Rejected Male   C     205     4.53  
## 11 Admitted Female C     202     4.46  
## 12 Rejected Female C     391     8.64  
## 13 Admitted Male   D     138     3.05  
## 14 Rejected Male   D     279     6.16  
## 15 Admitted Female D     131     2.89  
## 16 Rejected Female D     244     5.39  
## 17 Admitted Male   E      53     1.17  
## 18 Rejected Male   E     138     3.05  
## 19 Admitted Female E      94     2.08  
## 20 Rejected Female E     299     6.61  
## 21 Admitted Male   F      22     0.486  
## 22 Rejected Male   F     351     7.76  
## 23 Admitted Female F      24     0.530  
## 24 Rejected Female F     317     7.00
```

The `mutate()` function adds a column named `percent` to the tibble. The percentages represent the row percentage of the overall total. For example, among all applicants to the school, males admitted to Department A make up 11.3%.

Sometimes it can be helpful to know each row's percentage of the grand total, depending on the nature of the dataset. However, you will usually want to calculate percentages based on subgroups, which you will do next.

### Combine `count()` and `mutate()`

Recall that a weighted `count()` calculates the sum of frequencies across rows, based on whatever variable you specify. You can combine `count()` with `mutate()` to obtain the percentages as well. Here is an example:

Recall that there were 2,691 men and 1,835 women who applied to the school. What percentages of the total applicants were men vs. women?

```
admissions_tib %>%
  count(Gender, wt = Freq) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 2 x 3
##   Gender      n percent
##   <fct>   <dbl>   <dbl>
## 1 Male    2691    59.5
## 2 Female  1835    40.5
```

The output shows that 59.5% of the applicants were male, while 40.5% of the applicants were female. Calculating percentages provides a more precise look at the data, but note that the result corresponds to the 3:2 ratio that you obtained from “eyeballing” the numbers in a previous exercise.

One thing to note: When combining these two functions, you will need to use `n` in your formula for percent. This is because the `count()` function returns frequencies (counts) as a column labeled `n`.

Now try another example!

### Exercise 8: Combine `count()` and `mutate()`

Calculate the percentage of applicants who were admitted and the percentage of applicants who were rejected (overall).

```
admissions_tib %>%
  count(Admit, wt = Freq) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 2 x 3
##   Admit      n percent
##   <fct>   <dbl>   <dbl>
## 1 Admitted 1755    38.8
## 2 Rejected 2771    61.2
```

The output is the overall admission rate: 38.8% of all applicants were admitted.

### Combine `group_by()` and `mutate()`

Combining `count()` and `mutate()` is useful for sample-wide calculations. But what if you want to determine the percentages within subgroups? You will need to use `group_by()`.

Recall that the grouping variable determines the denominator of calculations. For example, when you group by `Dept`, R will calculate each row as a percentage of its Department total (rather than as a percentage of the grand total). Go ahead and try it!

```
admissions_tib %>%
  group_by(Dept) %>%
  mutate(percent = Freq / sum(Freq) * 100) %>%
  print(n = Inf)
```



```
## # A tibble: 24 x 5
## # Groups:   Dept [6]
##   Admit   Gender Dept   Freq percent
##   <fct>   <fct> <fct> <dbl>   <dbl>
## 1 Admitted Male   A     512    54.9
## 2 Rejected Male   A     313    33.5
## 3 Admitted Female A      89     9.54
## 4 Rejected Female A      19     2.04
## 5 Admitted Male   B     353    60.3
## 6 Rejected Male   B     207    35.4
## 7 Admitted Female B      17     2.91
## 8 Rejected Female B       8     1.37
## 9 Admitted Male   C     120    13.1
## 10 Rejected Male   C     205    22.3
## 11 Admitted Female C     202    22.0
## 12 Rejected Female C     391    42.6
## 13 Admitted Male   D     138    17.4
## 14 Rejected Male   D     279    35.2
## 15 Admitted Female D     131    16.5
## 16 Rejected Female D     244    30.8
## 17 Admitted Male   E      53     9.08
## 18 Rejected Male   E     138    23.6
## 19 Admitted Female E      94    16.1
## 20 Rejected Female E     299    51.2
## 21 Admitted Male   F      22     3.08
## 22 Rejected Male   F     351    49.2
## 23 Admitted Female F      24     3.36
## 24 Rejected Female F     317    44.4
```

The output shows four percentages per department (male admitted, male rejected, female admitted, and female rejected) that should sum to approximately 100%. This can be handy when you want to know the percentage breakdown within each level of another variable. For example, if the four rows of each department represented four levels of a single variable (e.g., applicants' country of origin), that would be useful information.

However, in the current analysis, you are interested in both gender and admission status. You want to compare the admission rate for men to the admission rate for women within each department. Thus, you will need to group by two variables. Can you determine which ones?

### Exercise 9: Combine `group_by()` and `mutate()`

Did you guess **Dept** and **Gender**? Think through the logic of this analysis:

- You want an outcome for each department, so that is the first variable to group by.
- You also want to compare men and women, so that is the second variable to group by.
- Grouping by both variables creates 12 Dept-Gender subgroups (Dept. A-Male, Dept. A-Female, Dept. B- Male, Dept. B-Female, etc.).
- Finally, you will call `mutate()` to calculate the percentage of each row (admitted vs. rejected) within its Dept-Gender subgroup.

Does this seem a little complicated? Take a look at the code and the output!

```
admissions_tib %>%
  group_by(Dept, Gender) %>%
  mutate(percent = Freq / sum(Freq) * 100) %>%
  print(n = Inf)
```

```
## # A tibble: 24 x 5
## # Groups:   Dept, Gender [12]
##   Admit   Gender Dept   Freq percent
##   <fct>   <fct> <fct> <dbl>   <dbl>
## 1 Admitted Male   A     512    62.1
## 2 Rejected Male   A     313    37.9
## 3 Admitted Female A      89    82.4
## 4 Rejected Female A      19    17.6
## 5 Admitted Male   B     353    63.0
## 6 Rejected Male   B     207    37.0
## 7 Admitted Female B      17     68
## 8 Rejected Female B       8     32
## 9 Admitted Male   C     120    36.9
## 10 Rejected Male   C     205    63.1
## 11 Admitted Female C     202    34.1
## 12 Rejected Female C     391    65.9
## 13 Admitted Male   D     138    33.1
## 14 Rejected Male   D     279    66.9
## 15 Admitted Female D     131    34.9
## 16 Rejected Female D     244    65.1
## 17 Admitted Male   E      53    27.7
## 18 Rejected Male   E     138    72.3
## 19 Admitted Female E      94    23.9
## 20 Rejected Female E     299    76.1
## 21 Admitted Male   F      22     5.90
## 22 Rejected Male   F     351    94.1
## 23 Admitted Female F      24     7.04
## 24 Rejected Female F     317    93.0
```

Now you can compare apples to apples! In four of the six departments (A, B, D, and F), women had a higher admission rate than men, whereas in the other two departments (C and E), men had a higher admission rate.

### R Tips: `group_by()` and `mutate()`

- Think carefully about what variable(s) you want R to group by; in other words, what subgroups do you want to compare? This decision will determine how R calculates the new variable.
- Always check the output to make sure the results make sense. If not, the issue may be with the denominator of the calculation, so check the `group_by()` variable(s).
- It may be necessary to group by two (or more) variables!

### Exercise 10: Combine Multiple Verbs

In the previous exercise, you saw that admission rates differed between women and men for each department. How does this outcome compare to the school-wide admission rates for men and women?

To answer this question, you will combine `group_by()`, `count()`, and `mutate()`.

```
admissions_tib %>%
  group_by(Gender) %>%
  count(Admit, wt = Freq) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 4 x 4
## # Groups:   Gender [2]
##   Gender Admit      n percent
##   <fct> <fct>   <dbl>   <dbl>
## 1 Male   Admitted  1198    44.5
## 2 Male   Rejected  1493    55.5
## 3 Female Admitted   557    30.4
## 4 Female Rejected  1278    69.6
```

The output shows the result for the entire school: 44.5% of men were admitted, while 30.4% of women were admitted. This seems strange, as the previous analysis showed that within four of the six departments, women's admissions rates were higher than men's.

Did something go wrong with the analysis? Maybe you used the wrong grouping variable? In a situation like this, it would be reasonable to go back and check your code. However, when you go back and check, you confirm that you grouped appropriately and called `count()` on the correct variable (`Admit`). So what happened?

## A Paradox

The current finding is an illustration of Simpson's paradox, which is a statistical phenomenon in which the overall finding (such as the lower admission rate for women at the school level) evaporates or even reverses when subgroups are examined (such as the higher admission rate for women in four out of six departments). How does this happen? It is usually due to some third variable—a confound—that is only apparent at the subgroup level.

In the current scenario, the explanation is that department popularity (i.e., the number of applications to a department) differs between men and women. Compare the following two outputs:

```
admissions_tib %>%
  group_by(Dept) %>%
  count(Gender, wt = Freq) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 12 x 4
## # Groups:   Dept [6]
##   Dept Gender      n percent
##   <fct> <fct>   <dbl>   <dbl>
## 1 A     Male    825    88.4
## 2 A     Female  108    11.6
## 3 B     Male    560    95.7
## 4 B     Female   25     4.27
## 5 C     Male    325    35.4
## 6 C     Female  593    64.6
## 7 D     Male    417    52.7
## 8 D     Female  375    47.3
## 9 E     Male    191    32.7
## 10 E    Female  393    67.3
```

```
## 11 F      Male      373    52.2
## 12 F      Female    341    47.8
```

This output shows department-level application patterns of men and women. Two departments, C and E, had more female than male applicants. Two additional departments, D and F, also had large numbers of female applicants (even if not the majority).

```
admissions_tib %>%
  group_by(Dept) %>%
  count(Admit, wt = Freq) %>%
  mutate(percent = n / sum(n) * 100)
```

```
## # A tibble: 12 x 4
## # Groups:   Dept [6]
##   Dept Admit      n percent
##   <fct> <fct>   <dbl>   <dbl>
## 1 A     Admitted  601    64.4
## 2 A     Rejected  332    35.6
## 3 B     Admitted  370    63.2
## 4 B     Rejected  215    36.8
## 5 C     Admitted  322    35.1
## 6 C     Rejected  596    64.9
## 7 D     Admitted  269    34.0
## 8 D     Rejected  523    66.0
## 9 E     Admitted  147    25.2
## 10 E    Rejected  437    74.8
## 11 F    Admitted   46     6.44
## 12 F    Rejected  668    93.6
```

This output shows department-level admission patterns. The four departments with the highest rejection rates (C, D, E, and F) are the same departments that were popular among women (i.e., high percentages of female applicants).

What is the implication? At this school, women tend to apply to departments that have higher rejection rates, which leads to an overall (school-wide) lower admission rate for women. If you only looked at school-wide admission rates, then the picture would be incomplete.

Fortunately, dplyr verbs made it easy for you to look at various subgroups and to drill down in your analysis! As you complete your investigation, you recommend to your boss that the school examine its recruitment strategies to determine how to increase the appeal of departments that have lower numbers of male or female applicants.

## Recap: dplyr Verbs

Here is a recap of the functions you have learned in this lesson so far:

- **arrange()**: sorts the data in ascending order, based on a selected variable. You can add **desc()** as an optional argument to obtain descending order instead.
- **select()**: selects the variables to display in the tibble and allows you to specify their placement.
- **count()**: counts the rows associated with a variable. You can add the argument **wt =** to weight each row by some other variable, such as frequency.
- **filter()**: selects specific observations based on the criteria for variables of interest.

- `group_by()`: groups rows based on a variable so that the subsequent calculations are performed by group rather than for the entire sample.
- `mutate()`: creates a new variable, based on a formula and/or existing variable(s). Examples include percentage and mean.

## Recommended Pipelines for Frequency Data

There are two key pipelines that you have seen in these exercises. The first pipeline focuses on the entire sample:

- Create the initial tibble with the desired variables and observations in the desired order.
  - `tibble_name %>% select() %>% filter() %>% arrange() %>%`
- Call a weighted count to obtain the sum of frequencies for each level of each variable of interest.
  - `count(variable, wt = ) %>%`
- Mutate the tibble to calculate percentages associated with these sums (once per variable of interest).
  - `mutate(percent = n / sum(n) * 100)`

The results will be each subgroup's percentage of the overall total for each variable.

The second pipeline focuses on subgroups.

- Use the same initial cleaning process.
  - `tibble_name %>% select() %>% filter() %>% arrange() %>%`
- Group by the variable(s) of interest.
  - `group_by(variable)`
- Call a weighted count to obtain the sum of frequencies.
  - `count(variable, wt = ) %>%`
  - Note that this might not be needed if you have grouped by multiple variables.
- Mutate to calculate subgroup percentages.
  - `mutate(percent = n / sum(n) * 100)`

## Scenario: Your New Job

Congratulations! Due to your outstanding work for your school, you caught the eye of Netflix. The company has hired you to analyze their database of movies and shows to reveal key business insights.

Your new boss has compiled a CSV file that contains a database of their content as well as viewer rating information from IMDB and TMDB. You will need to access the file (see the webpage where you accessed this lesson) and save it to your working directory.

Note: the file `Netflix_titles_OR.csv` is a cleaned version of an open dataset found on Kaggle (Soero, 2022). It was cleaned with OpenRefine.



Figure 5: A Fun Job; Image by Devon Breen, Pixabay open license

## Import and Clean a Dataset

You will begin by importing the dataset. There are numerous options for importing data into R; the best option depends on the file type and structure. For most CSV files, `read.csv()`, followed by the path to the file in quotation marks, will be useful. You will use the function with the default arguments, but it is helpful to know a few of these key arguments in case the dataset you are importing has a slightly different structure.

- `header = TRUE` tells R that the first row of data contains column names. You can change this to `FALSE` if that is not true for your dataset.
- `stringsAsFactors = FALSE` tells R that you do not want to treat columns with strings as factors. Note that you can always convert a variable to a factor later.
- `sep = ","` tells R that values in the CSV file are separated by commas. If your values are separated by a different (e.g., `;`), you can add that argument and change the comma to the correct separator.

### Tip: Getting and Setting the Working Directory

It's best to put the file in your working directory ahead of time so that you can easily specify the path. If you have forgotten how to check your working directory, here is a reminder: call `getwd()`.

If you want to change your working directory, you can call `setwd()` with the directory path in quotation marks. However, be cautious about using this script, and never include `setwd()` and a specified path in a script you share with others, as this can mess up their directory!

### Exercise 11: Import Data from a .CSV File

To practice, first ensure that the `Netflix_titles_OR.csv` file is in your working directory. Next, import the file into R and assign it to an object named `titles`. You will also add an argument called `na.strings`, which will convert blank cells to NA.

```
titles <- read.csv("Netflix_titles_OR.csv", stringsAsFactors = FALSE, na.strings = "")
```

You will see that an object named `titles` appears in your Global Environment.

Next, check the structure of `titles` with the function `glimpse()`.

```
glimpse(titles)
```

```
## Rows: 5,850
## Columns: 16
## $ id          <chr> "ts300399", "tm84618", "tm154986", "tm127384", ~
## $ title       <chr> "Five Came Back: The Reference Films", "Taxi Dr~
## $ type        <chr> "SHOW", "MOVIE", "MOVIE", "MOVIE", "MOVIE", "SH~
## $ description <chr> "This collection includes 12 World War II-era p~
## $ release_year <int> 1945, 1976, 1972, 1975, 1967, 1969, 1979, 1971,~
## $ age_certification <chr> "TV-MA", "R", "R", "PG", NA, "TV-14", "R", "R",~
## $ runtime      <int> 51, 114, 109, 91, 150, 30, 94, 102, 110, 104, 1~
## $ genres.1     <chr> "documentation", "drama", "drama", "fantasy", "~
## $ genres.2     <chr> NA, "crime", "action", "action", "action", "eur~
## $ genres.3     <chr> NA, NA, "thriller", "comedy", NA, NA, NA, "crim~
## $ production_countries.1 <chr> "US", "US", "US", "GB", "GB", "GB", "GB", "US",~
## $ production_countries.2 <chr> NA, NA, NA, NA, "US", NA, NA, NA, NA, NA, "US",~
## $ imdb_score   <dbl> NA, 8.2, 7.7, 8.2, 7.7, 8.8, 8.0, 7.7, 7.7, 5.8~
## $ imdb_votes   <dbl> NA, 808582, 107673, 534486, 72662, 73424, 39502~
## $ tmdb_popularity <dbl> 0.600, 40.965, 10.010, 15.461, 20.398, 17.617, ~
## $ tmdb_score   <dbl> NA, 8.179, 7.300, 7.811, 7.600, 8.306, 7.800, 7~
```

The output shows that the dataset has 5,860 rows and 16 columns.

## Exercise 12: Cleaning - Rename Variables

Next, you will change the names of two variables with the function `rename()` and assign the new dataset to `titles_tidy`. Note that the arguments for `rename()` are pairs of names: `newname = oldname`.

```
titles_tidy <- rename(titles, genre1 = genres.1, rating = age_certification)
glimpse(titles_tidy)
```

```
## Rows: 5,850
## Columns: 16
## $ id          <chr> "ts300399", "tm84618", "tm154986", "tm127384", ~
## $ title       <chr> "Five Came Back: The Reference Films", "Taxi Dr~
## $ type        <chr> "SHOW", "MOVIE", "MOVIE", "MOVIE", "MOVIE", "SH~
## $ description <chr> "This collection includes 12 World War II-era p~
## $ release_year <int> 1945, 1976, 1972, 1975, 1967, 1969, 1979, 1971,~
## $ rating      <chr> "TV-MA", "R", "R", "PG", NA, "TV-14", "R", "R",~
## $ runtime      <int> 51, 114, 109, 91, 150, 30, 94, 102, 110, 104, 1~
## $ genre1      <chr> "documentation", "drama", "drama", "fantasy", "~
## $ genres.2     <chr> NA, "crime", "action", "action", "action", "eur~
## $ genres.3     <chr> NA, NA, "thriller", "comedy", NA, NA, NA, "crim~
## $ production_countries.1 <chr> "US", "US", "US", "GB", "GB", "GB", "GB", "US",~
## $ production_countries.2 <chr> NA, NA, NA, NA, "US", NA, NA, NA, NA, NA, "US",~
## $ imdb_score   <dbl> NA, 8.2, 7.7, 8.2, 7.7, 8.8, 8.0, 7.7, 7.7, 5.8~
## $ imdb_votes   <dbl> NA, 808582, 107673, 534486, 72662, 73424, 39502~
## $ tmdb_popularity <dbl> 0.600, 40.965, 10.010, 15.461, 20.398, 17.617, ~
## $ tmdb_score   <dbl> NA, 8.179, 7.300, 7.811, 7.600, 8.306, 7.800, 7~
```

The main thing to notice in the output is that `genres.1` is now called `genre1`, and `age_certification` is now called `rating`. Renaming to simpler variable names can save you some energy!

## Exercise 12b: Cleaning - Create a Factor

Next, you will convert one of the variables to a factor. Remember that you did not change the `stringsAsFactors` argument to `TRUE`, so all string values were read in as strings. However, you do want one of your variables to be a factor: `rating`. Use the function `as.factor` to make this conversion.

```
titles_tidy$rating <- as.factor(titles_tidy$rating)
glimpse(titles_tidy)
```

```
## Rows: 5,850
## Columns: 16
## $ id          <chr> "ts300399", "tm84618", "tm154986", "tm127384", ~
## $ title       <chr> "Five Came Back: The Reference Films", "Taxi Dr~
## $ type        <chr> "SHOW", "MOVIE", "MOVIE", "MOVIE", "MOVIE", "SH~
## $ description <chr> "This collection includes 12 World War II-era p~
## $ release_year <int> 1945, 1976, 1972, 1975, 1967, 1969, 1979, 1971,~
## $ rating      <fct> TV-MA, R, R, PG, NA, TV-14, R, R, R, R, NA, PG--
## $ runtime     <int> 51, 114, 109, 91, 150, 30, 94, 102, 110, 104, 1~
## $ genre1      <chr> "documentation", "drama", "drama", "fantasy", "~
## $ genres.2    <chr> NA, "crime", "action", "action", "action", "eur~
## $ genres.3    <chr> NA, NA, "thriller", "comedy", NA, NA, NA, "crim~
## $ production_countries.1 <chr> "US", "US", "US", "GB", "GB", "GB", "GB", "US",~
## $ production_countries.2 <chr> NA, NA, NA, NA, "US", NA, NA, NA, NA, NA, "US",~
## $ imdb_score  <dbl> NA, 8.2, 7.7, 8.2, 7.7, 8.8, 8.0, 7.7, 7.7, 5.8~
## $ imdb_votes  <dbl> NA, 808582, 107673, 534486, 72662, 73424, 39502~
## $ tmdb_popularity <dbl> 0.600, 40.965, 10.010, 15.461, 20.398, 17.617, ~
## $ tmdb_score  <dbl> NA, 8.179, 7.300, 7.811, 7.600, 8.306, 7.800, 7~
```

Notice that the line for the variable `rating` now says that the datatype is `fct`, which means factor.

Now try calling the function `levels()` on that factor.

```
levels(titles_tidy$rating)
```

```
## [1] "G"      "NC-17" "PG"     "PG-13" "R"      "TV-14" "TV-G"  "TV-MA" "TV-PG"
## [10] "TV-Y"   "TV-Y7"
```

This returns all levels for this factor! There appear to be five levels for movies and six levels for TV shows.

## Exercise 13: Narrow the Dataframe

Now you are ready to use `dplyr` verbs to narrow the dataframe to a smaller subset of data. Your manager at Netflix is interested in all movies released in 2020 or later, so you need to filter to obtain only those movies. In addition, you don't need all of the variables, just a few of interest. The remaining analyses will focus on this narrower dataframe, so you can save this script as an object called `movies_recent`.

```
movies_recent <- titles_tidy %>%
  filter(type == "MOVIE", release_year > 2019) %>%
  select(title, rating, release_year, genre1, runtime, imdb_score, tmdb_score)
glimpse(movies_recent)
```



```
## Rows: 1,173
## Columns: 7
## $ title      <chr> "The Rental", "I'm Thinking of Ending Things", "Extractio~
## $ rating     <fct> R, R, R, NA, R, PG-13, R, NA, R, PG, R, R, R, PG-13, R, N~
## $ release_year <int> 2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020, 2020, 202~
## $ genre1     <chr> "drama", "drama", "crime", "documentation", "comedy", "cr~
## $ runtime    <int> 88, 134, 116, 84, 78, 123, 125, 93, 138, 102, 127, 131, 1~
## $ imdb_score <dbl> 5.7, 6.6, 6.7, 8.1, 6.1, 6.6, 6.6, 6.5, 7.1, 7.0, 7.0, 6.~
## $ tmdb_score <dbl> 5.600, 6.600, 7.400, 8.000, 6.400, 7.300, 7.200, 6.500, 7~
```

The output shows that this object has 1,173 rows of data (i.e., movies) and seven columns. Now you can start visualizing the data!

## Overview of ggplot2

You will use the **ggplot2** package within the tidyverse to create your plots. This is a powerful package that is informed by a theory called the grammar of graphics (Wilkinson, 2005). The theory guides the layered approach that ggplot's creators use as a template for building plots (Wickham & Grolemund, 2017).

The approach has seven parameters:

- data
- geom (a set of visual marks that represent datapoints)
- mappings
- statistics
- position
- coordinate system
- faceting scheme

In practice, you just have to know three of these parameters to get started: the dataset, the geom function, and the mappings. The other parameters are covered in the defaults for whatever geom function you call.

The **ggplot2** package has a huge number of features, but for now, here are a few need-to-know concepts. The basic format for a plot is

```
ggplot(dataset, aes(variable1, variable2, aesthetic3 = variable3)) + geom_type(attributes)
```

The first layer is **aes()**, which means the aesthetic mappings of the plot. This takes one or more arguments, depending on the number of variables in the plot.

The first two variables are mapped to the x and y axes, respectively; additional variables can be represented by additional aesthetics, which are visual aspects mapped onto variables. Examples of aesthetics include color, size, and shape representing levels of a variable.

The plot type will depend on the number of variables you are plotting. For example, a histogram plots one variable, a scatterplot plots two variables, and a third variable can be added to a scatterplot as the size or color of the dot representing different levels of the third variable.

The second layer of the script is the **geom** function. Some common geom functions include **geom\_freqpoly()** for frequency polygrams, **geom\_histogram()** for histograms, **geom\_point()** for scatterplots, and **geom\_line()** for line plots. Note that this layer is added with a plus sign.

Finally, additional visual aspects that are applied to the entire plot are called attributes, and these are generally arguments within the geom function. Such attributes include things like the color or placement of bars or the transparency of dots for the entire plot.

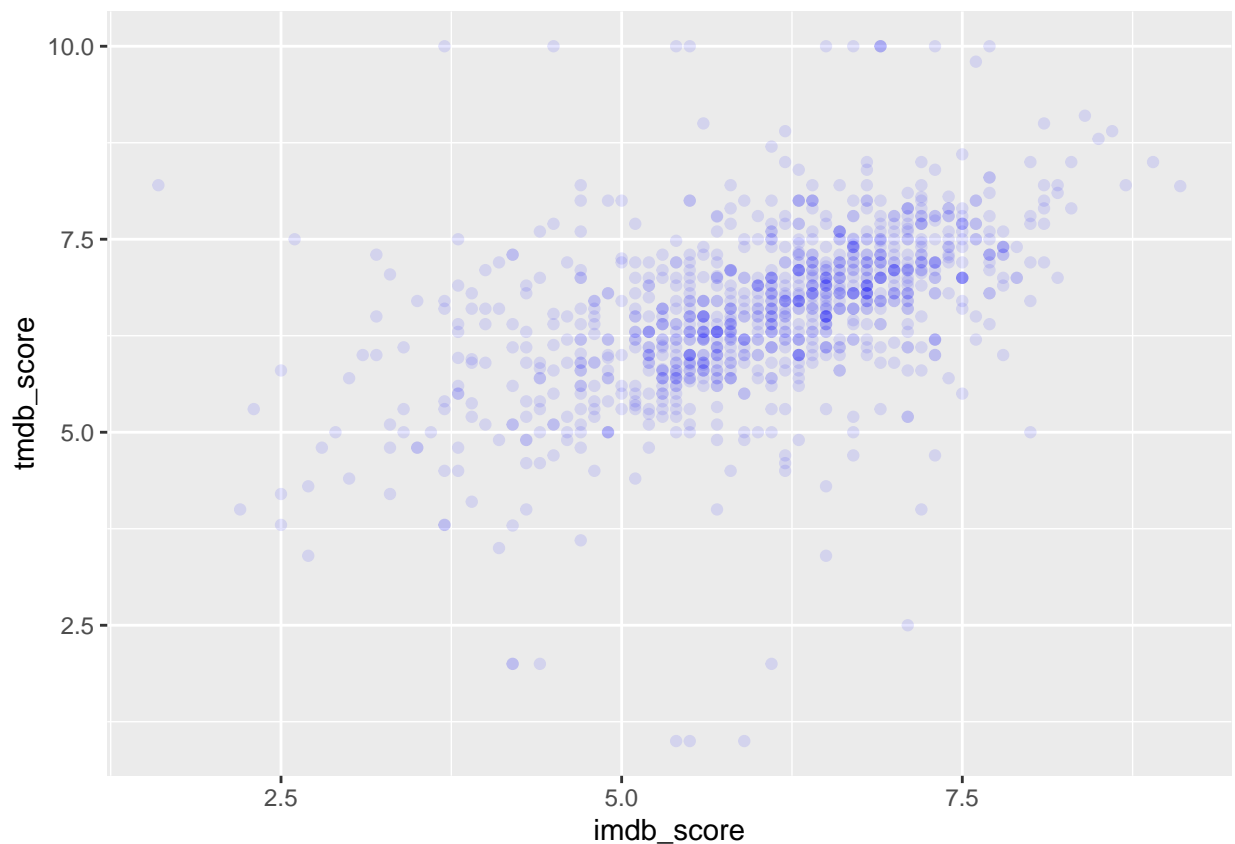
Now it's time to practice!

### Exercise 14a: Create a Scatter Plot with Two Variables

Your manager wants to know if there is a relationship between the two measures of audience rating: IMDB and TMDB. You will create a scatterplot to examine these two variables in tandem. Note that R will give you a warning that observations with missing values on x and/or y have been removed. That warning is shown below but will be suppressed in subsequent output in this lesson. However, you will still see the warnings.

```
ggplot(movies_recent, aes(x = imdb_score, y = tmdb_score)) +  
  geom_point(alpha = .10, color = "blue")
```

```
## Warning: Removed 197 rows containing missing values (geom_point).
```

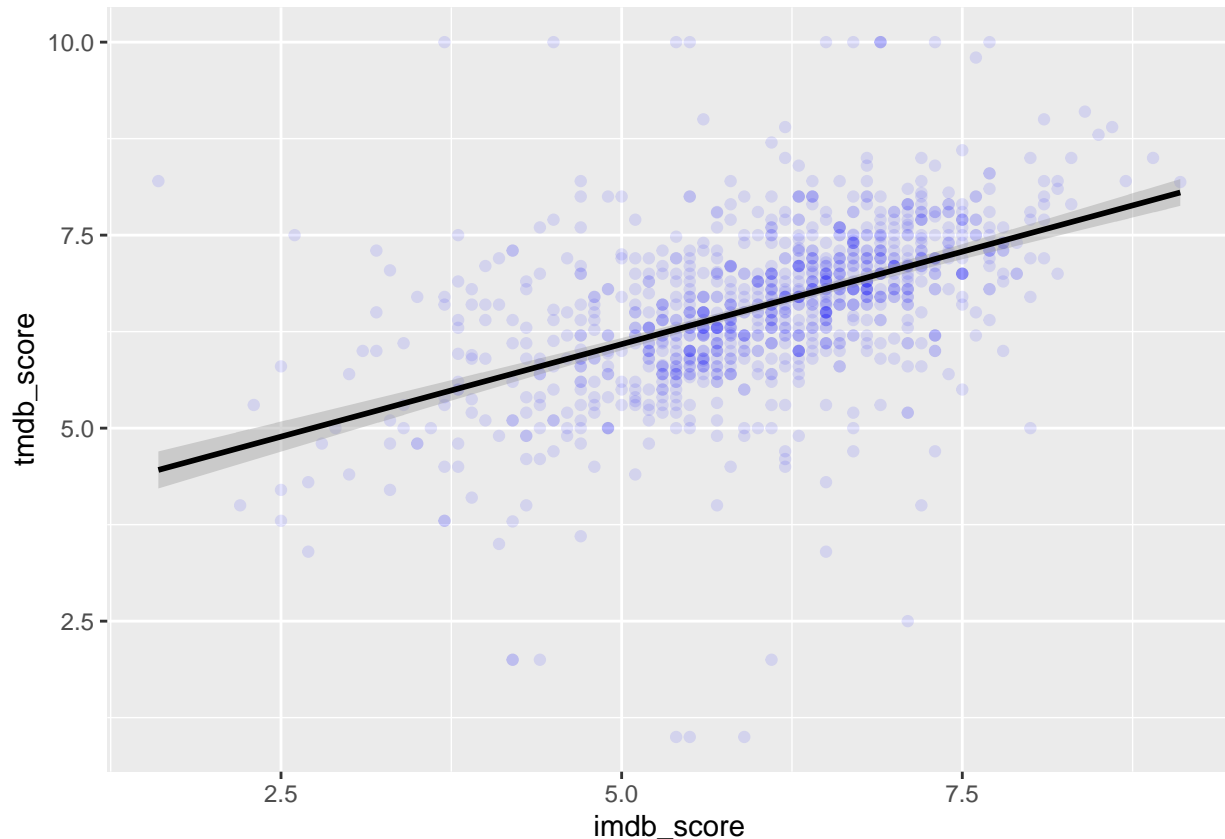


Note that the attribute color specifies the color of all dots, while alpha specifies the transparency of dots. Transparency can be useful when there are many datapoints and a lot of overlap. The darker points indicate more observations at the same location on the plot.

There appears to be a fairly positive relationship between the two variables, but you want to see the degree of relationship with a regression line. You can add another geom, `geom_smooth()` and specify the method as `lm`, which stands for linear model.

### Exercise 14b: Add a Regression Line

```
ggplot(movies_recent, aes(x = imdb_score, y = tmdb_score)) +
  geom_point(alpha = .10, color = "blue") +
  geom_smooth(method = lm, color = "black")
```



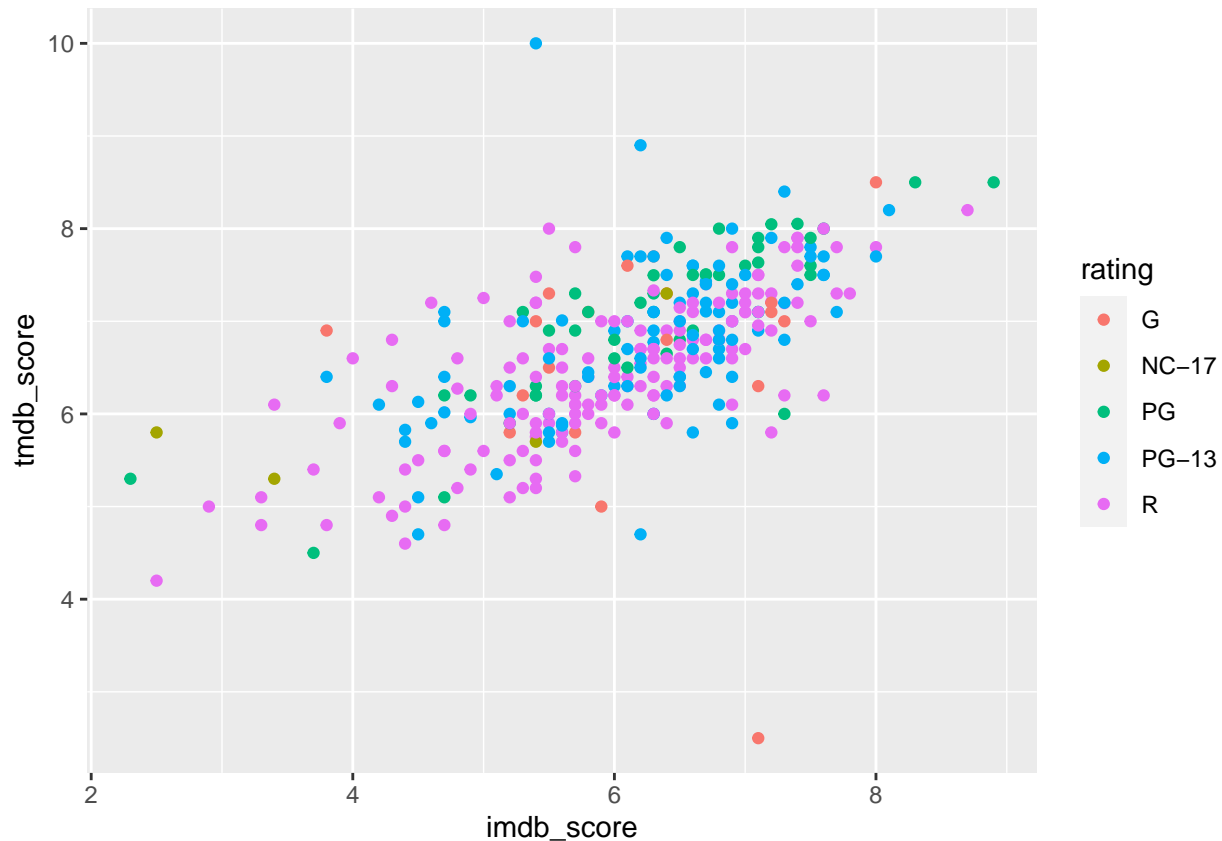
The black regression line shows the relationship with a bit more clarity, and the error is shown with shading around the line.

### Exercise 15: Add a Factor and Drop NAs

Your manager wants to see how movie rating relates to the relationship you have just found, so you will need to add movie rating as a variable. Because `rating` is a factor, you can use an aesthetic such as `color` to indicate the different levels of that factor.

There were a few movies that did not include a rating, but you don't want to include those in your plot, so you will also use the function `drop_na(rating)` to drop any movies that had NA for their rating. Note that `drop_na()` is a function from another tidyverse package called `tidyr`. It uses the `%>%` pipe. You do not need to specify `movies_recent` as an argument in `ggplot()` because the name of the dataset was included at the beginning of the pipe.

```
movies_recent %>%
  drop_na(rating) %>%
  ggplot(aes(x = imdb_score, y = tmdb_score, color = rating)) +
  geom_point()
```



The output is the same plot, but with each movie rating appearing as a different color. We see a few outliers: For example, there was a G movie that was rated well in IMDB but poorly in TMDb, and there is a PG-13 movie that has a very high TMDb score but a more moderate IMDB score. But adding color doesn't necessarily add clarity to this plot. There are too many rating levels to see a clear picture emerge. This is where facets can come in handy.

## Facets

Facets are plots that are repeated for each level of a factor. Also called tiny multiples (Tufte, 2001), facets can be used to show how the relationship between two variables differs at various levels of a third variable.

### Exercise 16: Create a Pipeline that Includes Facets

Your manager has been getting pushback from the board of directors because new Netflix directors want to make long movies. These directors claim that longer movies are more highly rated. Your manager asks you to investigate whether that is the case, focusing on the four top genres. To answer this question, you will need to draw upon your new `dplyr` and `ggplot2` skills while experimenting with an additional aspect of `ggplot2`: the facet wrap.

Here is the approach you will take:

- Change `genre1` to a factor.
- Count the frequencies for `genre1`, sorted in descending order.
- Filter to include only the top four genres, using `%in%`.

- Plot `tmdb_score` on `run_time`, with `genre1` as a third aesthetic (color).
- Include a regression line.
- Add a facet wrap on `genre1` so that each genre's plot appears in its own facet.

Note that the function `%in%` will look at the specified column and find values that are identified in the vector of values after `%in%` (in the case below, the four top genres).

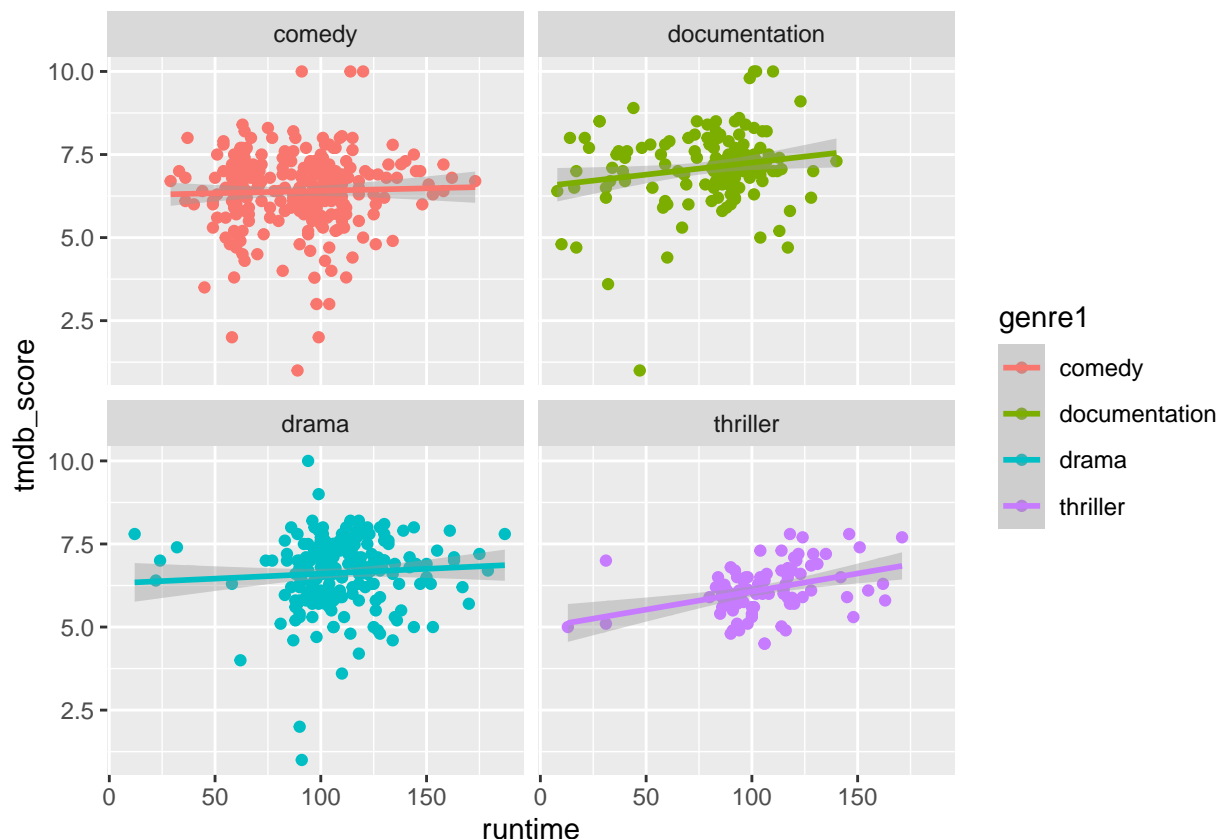
```
movies_recent$genre1 <- as.factor(movies_recent$genre1)
```

```
movies_recent %>%
  count(genre1, sort = TRUE)
```

```
##      genre1  n
## 1      comedy 273
## 2       drama 242
## 3 documentation 162
## 4      thriller 90
## 5      romance 76
## 6       action 67
## 7      animation 61
## 8        crime 45
## 9       fantasy 32
## 10      horror 31
## 11       scifi 23
## 12       music 21
## 13       family 17
## 14        <NA> 16
## 15        war 8
## 16      history 5
## 17       sport 2
## 18      western 2
```

```
top_genres <- movies_recent %>%
  filter(genre1 %in% c("drama", "comedy", "documentation", "thriller"))
```

```
ggplot(top_genres, aes(x = runtime, y = tmdb_score, color = genre1)) +
  geom_point() +
  geom_smooth(method = lm) +
  facet_wrap(~genre1)
```



From these facets, you can see that there was virtually no relationship between run time and TMDB score for comedy or drama.

However, the slightly upward slope and the tighter clustering of points for the documentation and thriller genres suggests a small positive relationship. People rated longer documentaries and thrillers as slightly better!

You suggest to your manager that more research into this possible relationship is needed. Your manager is also reassured that the longer movies aren't turning off viewers—that is, the relationship between length and rating isn't negative.

## Summary and Quick Guide

There is much more to discover about `dplyr` and `ggplot2`, but hopefully this lesson has given you a foundation in some of the key aspects of these packages. Here is a list of the functions you have learned:

- `glimpse()`: shows the structure of an object; especially useful for tibbles.
- `class()`: shows the class of an object (e.g., vector, table, dataframe, tibble).
- `as.data.frame()`: coerces an object to a dataframe.
- `as_tibble()`: coerces an object to a tibble.
- `print(n = )`: displays specified number of rows in a tibble.
- `%>%`: pipe function that passes subsequent functions to a dataframe/tibble.
- `arrange()`: arranges data in ascending order based on specified variable(s).
- `arrange(desc())`: arranges data in descending order based on specified variable(s).
- `select()`: selects variables to display and their order in the tibble.

- `select(-)`: omits variables from the tibble
- `select(:)`: provides a range of variables to display in the tibble.
- `select(starts_with(""))`: selects variables that start with the specified string.
- `select(ends_with(""))`: selects variables that end with the specified string.
- `select(contains(""))`: selects variables that contain the specified string.
- `select(everything())`: selects all variables and can be used with specified variables to rearrange the order of columns in a tibble.
- `count()`: counts rows associated with a variable
  - `count(wt = )`: weights rows and sums their values
- `filter()`: indexes the data to include rows that meet specified criteria.
  - `filter(... %in% ...)`: finds observations that match a set of values for a variable.
- `group_by()`: groups the data based on levels of a specified variable (factor).
- `mutate()`: creates a new variable based on one or more existing variables, using a formula.
  - `sum()`: sums values and can be used in a `mutate()` formula to obtain percentages.
  - `mutate(percent = n/sum(n) * 100)`: formula for calculating percent when `n` is a frequency.
- `read.csv`: one option for reading in data from a CSV file.
- `setwd()`: sets the working directory; *avoid calling this function on a specific path in shared scripts!*
- `rename()`: renames variables.
- `as.factor()`: converts a variable to a factor.
- `levels()`: displays the levels of a factor.
- `ggplot()`: basic function for creating a plot.
  - `aes()`: specifies the aesthetics of the plot.
  - `geom_point()`: creates a scatterplot.
  - `geom_smooth(method = lm)`: adds a regression line.
  - `facet_wrap()`: creates separate plots for each level of a factor.

## References and Recommended Reading

- Chang, W. (2022). *R graphics cookbook*. O'Reilly. <https://r-graphics.org/>
- OpenRefine. (n.d.). <https://openrefine.org/>
- R Project for Statistical Computing. (n.d.). <https://www.r-project.org/>
- R Studio. (n.d.). R Studio Cheatsheets. <https://www.rstudio.com/resources/cheatsheets/>
- Soero, V. (2022). Netflix TV Shows and Movies. <https://www.kaggle.com/datasets/victorsoeiro/netflix-tv-shows-and-movies>
- Springer, J., & Weinberger, N. (2021). Simpson's paradox. *Stanford encyclopedia of philosophy*. Springer. <https://plato.stanford.edu/entries/paradox-simpson/>
- Tufte, E. (1997). *The visual display of quantitative information*. Graphics Press LLC.
- Wickham, H., & Grolemund, G. (2017). *R for data science*. O'Reilly.