Tidyverse examples

STAT215A Fall 2020 Week 1

October 28, 2021

This is an introduction to the basic functions in tidyverse, which contains multiple libaries that are useful for manipulating and plotting data in R.

dyplr

piping

Piping is a way to chain functions together to avoid redefining variables. Below, we look at the head of the iris data frame using piping.

```
# without using piping
head(iris)
```

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

```
# produces same result but with piping
iris %>% head()
```

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

We can also use multiple pipes in a single line of code. In the below example, we count the number of observations in each species using multiple pipes.

```
# without piping
iris_by_species <- group_by(iris, Species)
summarise(iris_by_species, n = n())
```

```
## # A tibble: 3 x 2
##   Species        n
##   <fct>      <int>
## 1 setosa        50
## 2 versicolor    50
## 3 virginica     50
```

```r
# produces same result but with piping
iris %>%
  group_by(Species) %>%
  summarise(n = n())
```

```
## # A tibble: 3 x 2
##   Species        n
##   <fct>      <int>
## 1 setosa        50
## 2 versicolor    50
## 3 virginica     50
```

Exercise 1. Rewrite the following code chunk using the pipe opertor.

```r
log_petal_length <- log(iris$Sepal.Length)
min(log_petal_length)
```

```
## [1] 1.458615
```

```r
# rewrite with pipe operator
iris %>%
  pull(Sepal.Length) %>%
  log() %>%
  min()
```

```
## [1] 1.458615
```

filter

Now, we look at `filter()`, which finds rows where the specified condition is true and returns those rows as a data frame. Here, we use `filter()` to get only the rows in the data frame that are from the versicolor species.

```r
# using filter() without piping
head(filter(iris, Species == "versicolor"))
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          7.0         3.2          4.7         1.4 versicolor
## 2          6.4         3.2          4.5         1.5 versicolor
## 3          6.9         3.1          4.9         1.5 versicolor
## 4          5.5         2.3          4.0         1.3 versicolor
## 5          6.5         2.8          4.6         1.5 versicolor
## 6          5.7         2.8          4.5         1.3 versicolor
```

```r
# same thing but using piping instead
iris %>%
  filter(Species == "versicolor") %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1          7.0         3.2          4.7         1.4 versicolor
## 2          6.4         3.2          4.5         1.5 versicolor
## 3          6.9         3.1          4.9         1.5 versicolor
## 4          5.5         2.3          4.0         1.3 versicolor
## 5          6.5         2.8          4.6         1.5 versicolor
## 6          5.7         2.8          4.5         1.3 versicolor
```

```r
# check dimension of data frame
iris %>%
  filter(Species == "versicolor") %>%
  dim()
```

```
## [1] 50  5
```

Instead of just filtering out one species, we could look at all observations that are versicolor as well as setosa.

```r
# save all rows that are veriscolor of setosa
iris %>%
  filter(Species %in% c("versicolor", "setosa")) %>%
  head()
```

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

```r
# check dimension of data frame
iris %>%
  filter(Species %in% c("versicolor", "setosa")) %>%
  dim()
```

```
## [1] 100   5
```

Exercise 2. How many observations have sepal length in the upper 50% quartile and petal width greater than 2?

```r
iris %>%
  filter(Sepal.Length > median(Sepal.Length),
         Petal.Width > 2) %>%
  nrow()
```

```
## [1] 22
```

Note that there are also variants of `filter()` named `filter_if()`, `filter_all()`, and `filter_at()`. If you run `? filter_if()`, you can learn more about these variants and see examples of how to use these functions.

select

We next look at `select()`, which keeps or removes only certain variables (selecting columns of the data frame whereas before we were filtering by row). Below, we use `select()` to keep only the columns that contain the species and sepal length information.

```r
iris %>%
  filter(Species == "versicolor") %>%
  select(Sepal.Length, Species) %>%
  head()
```

```
##   Sepal.Length    Species
## 1          7.0 versicolor
## 2          6.4 versicolor
## 3          6.9 versicolor
## 4          5.5 versicolor
## 5          6.5 versicolor
```

```
## 6          5.7 versicolor
```

We can also select all the columns except for sepal length by typing

```
iris %>%
  filter(Species == "versicolor") %>%
  select(-Sepal.Length) %>%
  head()
```

```
##   Sepal.Width Petal.Length Petal.Width    Species
## 1         3.2          4.7         1.4 versicolor
## 2         3.2          4.5         1.5 versicolor
## 3         3.1          4.9         1.5 versicolor
## 4         2.3          4.0         1.3 versicolor
## 5         2.8          4.6         1.5 versicolor
## 6         2.8          4.5         1.3 versicolor
```

It is often useful to combine `select()` with `rename()` to modify the selected column names.

```
iris %>%
  select(Sepal.Length, Sepal.Width) %>%
  rename(Length = Sepal.Length,
         Width = Sepal.Width) %>%
  head()
```

```
##   Length Width
## 1    5.1   3.5
## 2    4.9   3.0
## 3    4.7   3.2
## 4    4.6   3.1
## 5    5.0   3.6
## 6    5.4   3.9
```

Exercise 3. If you wanted to extract all columns in the data frame below that had the word "Length" in the column name, how would you go about doing this without explicitly typing the names of each individual column that you want? (Hint: The help page for `select_helpers` may be useful.)

```
# create fake dataset (please run lines)
idx <- sample(1:100, 25)  # generate random indices
cols <- c(paste0("Length", idx), paste0("Width", idx)) %>%  # generate column ids
  sample(., size = 50, replace = F)  # reorder column ids
data <- matrix(rnorm(500), nrow = 10, ncol = 50) %>%
  as.data.frame() %>%
  setNames(cols)

# view structure of data
str(data[, 1:10])
```

```
## 'data.frame':    10 obs. of  10 variables:
##  $ Length51: num  1.433 1.98 -0.367 -1.044 0.57 ...
##  $ Width79 : num  -0.743 0.189 -1.805 1.466 0.153 ...
##  $ Width54 : num  -1.25363 0.29145 -0.44329 0.00111 0.07434 ...
##  $ Width84 : num  0.594 0.333 1.063 -0.304 0.37 ...
##  $ Length97: num  1.587 0.558 -1.277 -0.573 -1.225 ...
##  $ Width1  : num  -0.655 1.767 0.717 0.91 0.384 ...
##  $ Width44 : num  -0.207 -0.393 -0.32 -0.279 0.494 ...
##  $ Length33: num  -0.1002 0.7127 -0.0736 -0.0376 -0.6817 ...
##  $ Length43: num  0.307 -1.536 -0.301 -0.528 -0.652 ...
```

```
##  $ Width7  : num  -1.1159 -0.7508 2.0872 0.0174 -1.2863 ...
```

```
# extract columns with "Length" in name
data %>%
  select(contains("Length")) %>%
  head()
```

```
##     Length51   Length97    Length33    Length43   Length70   Length35
## 1  1.4330237  1.5868335 -0.10019074  0.30655786 -1.4874603  0.5210227
## 2  1.9803999  0.5584864  0.71266631 -1.53644982 -1.0751923 -0.1587546
## 3 -0.3672215 -1.2765922 -0.07356440 -0.30097613  1.0000288  1.4645873
## 4 -1.0441346 -0.5732654 -0.03763417 -0.52827990 -0.6212667 -0.7660820
## 5  0.5697196 -1.2246126 -0.68166048 -0.65209478 -1.3844268 -0.4302118
## 6 -0.1350546 -0.4734006 -0.32427027 -0.05689678  1.8692906 -0.9261095
##      Length74    Length39   Length37     Length84   Length68   Length83
## 1 -2.28523554  0.98783827 -1.1565724 -0.059723276  1.7196273  0.1560117
## 2  2.49766159  1.51974503  1.8031419 -0.098178744  0.2700549  1.1302073
## 3  0.66706617 -0.30874057 -0.3311320  0.560820729 -0.4221840 -2.2891240
## 4  0.54132734 -1.25328976 -1.6055134 -1.186458639 -1.1891133  0.7410012
## 5 -0.01339952  0.64224131  0.1971934  1.096777044 -0.3310330 -1.3162452
## 6  0.51010842 -0.04470914  0.2631756 -0.005344028 -0.9398293  0.9198037
##     Length82   Length21   Length14   Length85     Length79    Length34
## 1  1.7784293  0.4820295  1.5778918 -0.1693183 -1.167662326  1.15482519
## 2  0.1344477  0.4561356  0.5962341  0.6122182 -0.008309014 -0.05652142
## 3  0.7655990 -0.3534003 -1.1735769  0.6783402  0.128855402 -2.12936065
## 4  0.9551367  0.1704895 -0.1556425  0.5679520 -0.145875628  0.34484576
## 5 -0.0505657 -0.8640360 -1.9189098 -0.5725426 -0.163910957 -1.90495545
## 6 -0.3058154  0.6792308 -0.1952588 -1.3632913  1.763552003 -0.81117015
##     Length54   Length44   Length59    Length87    Length7    Length73
## 1 -0.1101588 -0.4162220 -0.5080862  1.35099398 -0.4037368 -0.04773017
## 2 -0.9243128 -0.3756574  0.5236206 -1.31860948  0.2314961 -1.68452065
## 3  1.5929138 -0.3666309  1.0177542  0.36438459 -0.4223724 -0.14422656
## 4  0.0450106 -0.2956775 -0.2511646  0.23349984  0.3741184  1.18021367
## 5 -0.7151284  1.4418204 -1.4299934  1.19395526 -0.3660058  0.68139992
## 6  0.8652231 -0.6975383  1.7091210 -0.02790997  1.1901014  0.14324763
##      Length1
## 1  1.6420282
## 2 -0.7695923
## 3  0.3033610
## 4  1.2817374
## 5  0.6022228
## 6 -0.3070223
```

There are also variants of `select()` named `select_if()`, `select_all()`, and `select_at()`. If you run `?select_if()`, you can learn more about these variants and see examples of how to use these functions. Below, we show how to use `select_if()` to select all columns that are numeric.

```
iris %>%
  select_if(is.numeric) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3.0          1.4         0.2
## 3          4.7         3.2          1.3         0.2
## 4          4.6         3.1          1.5         0.2
```

```
## 5          5.0         3.6          1.4         0.2
## 6          5.4         3.9          1.7         0.4
```

Exercise 4. Extract the second column of the `iris` dataset using `select()`. Next, extract the second column of the `iris` dataset using `pull()`. What is the difference between `select()` and `pull()`?

```
# using select
iris %>%
  select(2) %>%
  head()
```

```
##   Sepal.Width
## 1         3.5
## 2         3.0
## 3         3.2
## 4         3.1
## 5         3.6
## 6         3.9
```

```
# using pull
iris %>%
  pull(2) %>%
  head()
```

```
## [1] 3.5 3.0 3.2 3.1 3.6 3.9
```

```
# The difference is that select() always returns a data frame or tibble while pull() returns a vector.
```

mutate

`mutate()` is a function which creates new variables consisting of functions of existing variables. We will first use `mutate()` to create a new variable that is the sum of sepal length and sepal width.

```
# mutate without piping
iris_vc <- iris[iris$Species == "versicolor", ]
iris_vc$Sepal.Sum <- iris_vc$Sepal.Width + iris_vc$Sepal.Length
head(iris_vc)
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Sepal.Sum
## 51          7.0         3.2          4.7         1.4 versicolor      10.2
## 52          6.4         3.2          4.5         1.5 versicolor       9.6
## 53          6.9         3.1          4.9         1.5 versicolor      10.0
## 54          5.5         2.3          4.0         1.3 versicolor       7.8
## 55          6.5         2.8          4.6         1.5 versicolor       9.3
## 56          5.7         2.8          4.5         1.3 versicolor       8.5
```

```
# mutate with piping
iris %>%
  filter(Species == "versicolor") %>%
  mutate(Sepal.Sum = Sepal.Length + Sepal.Width) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species Sepal.Sum
## 1          7.0         3.2          4.7         1.4 versicolor      10.2
## 2          6.4         3.2          4.5         1.5 versicolor       9.6
## 3          6.9         3.1          4.9         1.5 versicolor      10.0
## 4          5.5         2.3          4.0         1.3 versicolor       7.8
## 5          6.5         2.8          4.6         1.5 versicolor       9.3
```

```
## 6          5.7         2.8          4.5         1.3 versicolor       8.5
```

Like `select()`, `mutate()` also has the close variants `mutate_if()`, `mutate_at()`, and `mutate_all()`. Next, we use `mutate_at()` to multiply each Sepal.length and Sepal.Width by 2.

```
iris %>%
  filter(Species == "versicolor") %>%
  mutate_at(vars(contains("Sepal")), list(~ 2 * .)) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1         14.0         6.4          4.7         1.4 versicolor
## 2         12.8         6.4          4.5         1.5 versicolor
## 3         13.8         6.2          4.9         1.5 versicolor
## 4         11.0         4.6          4.0         1.3 versicolor
## 5         13.0         5.6          4.6         1.5 versicolor
## 6         11.4         5.6          4.5         1.3 versicolor
```

Exercise 5. Using the `iris_new` dataset, convert all character columns into factors using `mutate_if()`.

```
# create iris_new dataset
iris_new <- iris %>%
  rownames_to_column("id") %>%
  mutate(Species = as.character(Species))

# look at structure of iris_new
str(iris_new)
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ id          : chr  "1" "2" "3" "4" ...
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : chr  "setosa" "setosa" "setosa" "setosa" ...
```

```
# convert character columns to factors
iris_new %>%
  mutate_if(is.character, as.factor) %>%
  str()
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ id          : Factor w/ 150 levels "1","10","100",..: 1 63 74 85 96 107 118 129 140 2 ...
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

group_by and summarize

`group_by()` is a function that changes from operating on the entire dataset to operating on it group-by-group. `summarize()` allows us to summarize the group into a single value. Next, we group the dataset by species and compute the mean and median sepal length for each species.

```
iris %>%
  group_by(Species) %>%
  summarise(Sepal.Length.mean = mean(Sepal.Length),
```

```
            Sepal.Length.median = median(Sepal.Length))
```

```
## # A tibble: 3 x 3
##   Species    Sepal.Length.mean Sepal.Length.median
##   <fct>                  <dbl>               <dbl>
## 1 setosa                  5.01                 5
## 2 versicolor              5.94                 5.9
## 3 virginica               6.59                 6.5
```

Exercise 6. For each species, randomly select half of the observations and compute the 25^{th} quartile for each feature (i.e., `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`). Hint: see `? sample_frac`

```
iris %>%
  group_by(Species) %>%
  sample_frac(size = .5, replace = F) %>%
  summarise_all(list(Q1 = quantile), probs = 0.25)
```

```
## # A tibble: 3 x 5
##   Species    Sepal.Length_Q1 Sepal.Width_Q1 Petal.Length_Q1 Petal.Width_Q1
##   <fct>                <dbl>          <dbl>           <dbl>          <dbl>
## 1 setosa                 4.8            3.1             1.4            0.2
## 2 versicolor             5.6            2.5             3.9            1.1
## 3 virginica              6.3            2.8             5.1            1.8
```

```
# check to see if we did the sampling correct (and got the right number of obs)
iris %>%
  group_by(Species) %>%
  sample_frac(size = .5, replace = F) %>%
  select(Species) %>%
  table()
```

```
## .
##     setosa versicolor  virginica
##         25         25         25
```

arrange

`arrange()` is a function for ordering rows in a data.frame (or tibble) based upon some expression involving its variables/columns. By default, `arrange()` orders the rows based upon increasing order of the specified column (in the example, by petal length).

```
iris %>%
  arrange(Petal.Length) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          4.6         3.6          1.0         0.2  setosa
## 2          4.3         3.0          1.1         0.1  setosa
## 3          5.8         4.0          1.2         0.2  setosa
## 4          5.0         3.2          1.2         0.2  setosa
## 5          4.7         3.2          1.3         0.2  setosa
## 6          5.4         3.9          1.3         0.4  setosa
```

We can also arrange by multiple columns and by decreasing order of column(s). In the example below, we order the rows first by decreasing petal length and then by increasing sepal width (if there are ties among petal length).

```
iris %>%
  arrange(desc(Petal.Length), Sepal.Width) %>%
  head()
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 1          7.7         2.6          6.9         2.3 virginica
## 2          7.7         2.8          6.7         2.0 virginica
## 3          7.7         3.8          6.7         2.2 virginica
## 4          7.6         3.0          6.6         2.1 virginica
## 5          7.9         3.8          6.4         2.0 virginica
## 6          7.3         2.9          6.3         1.8 virginica
```

Exercise 7. For each species, only keep the observations with the largest 10 sepal lengths. Then, sort the rows in order of decreasing sepal length. Hint: see `? top_n`

```
iris %>%
  group_by(Species) %>%
  top_n(n = 10, wt = Sepal.Length) %>%
  ungroup() %>%
  arrange(desc(Sepal.Length))
```

```
## # A tibble: 32 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          7.9         3.8          6.4         2   virginica
##  2          7.7         3.8          6.7         2.2 virginica
##  3          7.7         2.6          6.9         2.3 virginica
##  4          7.7         2.8          6.7         2   virginica
##  5          7.7         3            6.1         2.3 virginica
##  6          7.6         3            6.6         2.1 virginica
##  7          7.4         2.8          6.1         1.9 virginica
##  8          7.3         2.9          6.3         1.8 virginica
##  9          7.2         3.6          6.1         2.5 virginica
## 10          7.2         3.2          6           1.8 virginica
## # ... with 22 more rows
```

More dplyr

Beyond the functions discussed previously, there are a few other `dplyr` functions that can be quite useful when cleaning and working with real data. For instance, suppose you need to merge two datasets based upon some designated id column. There are several ways to merge or join two datasets. See the help page for `dplyr::join` to learn about the various ways of joining two datasets. The most common ways are to use `inner_join()`, `left_join()`, `right_join()`, or `full_join()`. Below, we will use `left_join()` to merge the two play datasets based upon the `id` column.

```
lowercase_data <- data.frame(id = 1:6, lower = letters[1:6])
uppercase_data <- data.frame(id = 5:1, upper = LETTERS[5:1])
lowercase_data
```

```
##   id lower
## 1  1     a
## 2  2     b
## 3  3     c
## 4  4     d
## 5  5     e
## 6  6     f
```

```
uppercase_data
```

```
##   id upper
## 1  5     E
## 2  4     D
## 3  3     C
## 4  2     B
## 5  1     A
```

```
left_join(x = lowercase_data, y = uppercase_data, by = "id")
```

```
##   id lower upper
## 1  1     a     A
## 2  2     b     B
## 3  3     c     C
## 4  4     d     D
## 5  5     e     E
## 6  6     f  <NA>
```

Exercise 8. Using the `lowercase_data` and `uppercase_data` from the previous code chunk, merge the two datasets so that only rows with `id` matches in both datasets are kept. In other words, the resulting merged dataset should only include observations with `id` $= 1, \ldots, 5$.

```
inner_join(x = lowercase_data, y = uppercase_data, by = "id")
```

```
##   id lower upper
## 1  1     a     A
## 2  2     b     B
## 3  3     c     C
## 4  4     d     D
## 5  5     e     E
```

Exercise 9. The purpose of this exercise is to introduce a couple useful (but perhaps not well-known) functions for data cleaning. In the following `iris_messy` dataset, some rows are duplicates of another row, and some other rows have missing values. Delete both the rows with missing values and the duplicated rows. Hint: this can be easily done with two simple `dplyr` functions.

```
# add messiness to the iris data (please run)
iris_messy <- rbind(
  iris,
  iris %>% sample_n(size = 50, replace = T)  # add duplicates
) %>%
  slice(sample(1:n()))  # randomly shuffle rows
iris_messy[sample(1:nrow(iris_messy), size = nrow(iris_messy) / 4),
           "Species"] <- NA  # randomly add NAs to Species column
# sum(is.na(iris_messy))
# summary(iris_messy)

# delete duplicates and NAs
iris_clean <- iris_messy %>%
  drop_na() %>%  # can also use stats::na.omit() here
  distinct()

# look at cleaned data
head(iris_clean)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
```

```
## 1          5.4         3.7          1.5         0.2     setosa
## 2          7.7         2.8          6.7         2.0  virginica
## 3          4.8         3.0          1.4         0.1     setosa
## 4          6.0         2.7          5.1         1.6 versicolor
## 5          5.9         3.2          4.8         1.8 versicolor
## 6          5.2         3.4          1.4         0.2     setosa
```

```
str(iris_clean)
```

```
## 'data.frame':    118 obs. of  5 variables:
##  $ Sepal.Length: num  5.4 7.7 4.8 6 5.9 5.2 5.5 5 4.9 6.9 ...
##  $ Sepal.Width : num  3.7 2.8 3 2.7 3.2 3.4 2.4 3.4 3.6 3.2 ...
##  $ Petal.Length: num  1.5 6.7 1.4 5.1 4.8 1.4 3.8 1.5 1.4 5.7 ...
##  $ Petal.Width : num  0.2 2 0.1 1.6 1.8 0.2 1.1 0.2 0.1 2.3 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 3 1 2 2 1 2 1 1 3 ...
```

```
sum(is.na(iris_clean))  # count number of NAs; no NAs
```

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

```
nrow(unique(iris_clean)) == nrow(iris_clean)  # check if all rows in iris_clean are unique; yes
```

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

tidyr

tidyr contains functions for changing the shape of the data, allowing you to transition columns into rows and rows into columns. The two main functions are `spread()` and `gather()`, which can be viewed as inverses of each other.

gather

`gather()` converts data from wide format to long format.

```
# wide to long
iris_long <- iris %>%
  rownames_to_column("id") %>%
  gather(key = "Variable", value = "Value", -Species, -id)

# top
head(iris_long)
```

```
##   id Species     Variable Value
## 1  1  setosa Sepal.Length   5.1
## 2  2  setosa Sepal.Length   4.9
## 3  3  setosa Sepal.Length   4.7
## 4  4  setosa Sepal.Length   4.6
## 5  5  setosa Sepal.Length   5.0
## 6  6  setosa Sepal.Length   5.4
```

```
# bottoms
tail(iris_long)
```

```
##      id   Species    Variable Value
## 595 145 virginica Petal.Width   2.5
## 596 146 virginica Petal.Width   2.3
## 597 147 virginica Petal.Width   1.9
```

```
## 598 148 virginica Petal.Width   2.0
## 599 149 virginica Petal.Width   2.3
## 600 150 virginica Petal.Width   1.8
```

```
# overall structure
str(iris_long)
```

```
## 'data.frame':    600 obs. of  4 variables:
##  $ id      : chr  "1" "2" "3" "4" ...
##  $ Species : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ Variable: chr  "Sepal.Length" "Sepal.Length" "Sepal.Length" "Sepal.Length" ...
##  $ Value   : num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
```

spread

`spread()` converts data from long format to wide format

```
# long to wide
iris_wide <- iris_long %>%
  spread(key = "Variable", value = "Value")

# top
head(iris_wide)
```

```
##    id    Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 1   1     setosa          1.4         0.2          5.1         3.5
## 2  10     setosa          1.5         0.1          4.9         3.1
## 3 100 versicolor          4.1         1.3          5.7         2.8
## 4 101  virginica          6.0         2.5          6.3         3.3
## 5 102  virginica          5.1         1.9          5.8         2.7
## 6 103  virginica          5.9         2.1          7.1         3.0
```

```
# bottoms
tail(iris_wide)
```

```
##     id    Species Petal.Length Petal.Width Sepal.Length Sepal.Width
## 145 94 versicolor          3.3         1.0          5.0         2.3
## 146 95 versicolor          4.2         1.3          5.6         2.7
## 147 96 versicolor          4.2         1.2          5.7         3.0
## 148 97 versicolor          4.2         1.3          5.7         2.9
## 149 98 versicolor          4.3         1.3          6.2         2.9
## 150 99 versicolor          3.0         1.1          5.1         2.5
```

```
# overall structure
str(iris_wide)
```

```
## 'data.frame':    150 obs. of  6 variables:
##  $ id          : chr  "1" "10" "100" "101" ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 2 3 3 3 3 3 3 3 ...
##  $ Petal.Length: num  1.4 1.5 4.1 6 5.1 5.9 5.6 5.8 6.6 4.5 ...
##  $ Petal.Width : num  0.2 0.1 1.3 2.5 1.9 2.1 1.8 2.2 2.1 1.7 ...
##  $ Sepal.Length: num  5.1 4.9 5.7 6.3 5.8 7.1 6.3 6.5 7.6 4.9 ...
##  $ Sepal.Width : num  3.5 3.1 2.8 3.3 2.7 3 2.9 3 3 2.5 ...
```

Exercise 10. Currently, the `mtcars` dataset is in wide format. Remove the `cyl`, `vs`, `am`, `gear`, and `carb` columns. Then create an `id` column containing the rownmaes. Finally, with the `id` column as the `key` and all other features as values, put the wide `mtcars` dataset into long format. Name the resulting data frame `mtcars_long`.

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
mtcars_long <- mtcars %>%
  select(-cyl, -vs, -am, -gear, -carb) %>%
  rownames_to_column("id") %>%
  gather(key = "Variable", value = "Value", -id) %>%
  mutate_if(is.character, as.factor)
str(mtcars_long)
```

```
## 'data.frame':    192 obs. of  3 variables:
##  $ id      : Factor w/ 32 levels "AMC Javelin",..: 18 19 5 13 14 31 7 21 20 22 ...
##  $ Variable: Factor w/ 6 levels "disp","drat",..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ Value   : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
```

```
head(mtcars_long)
```

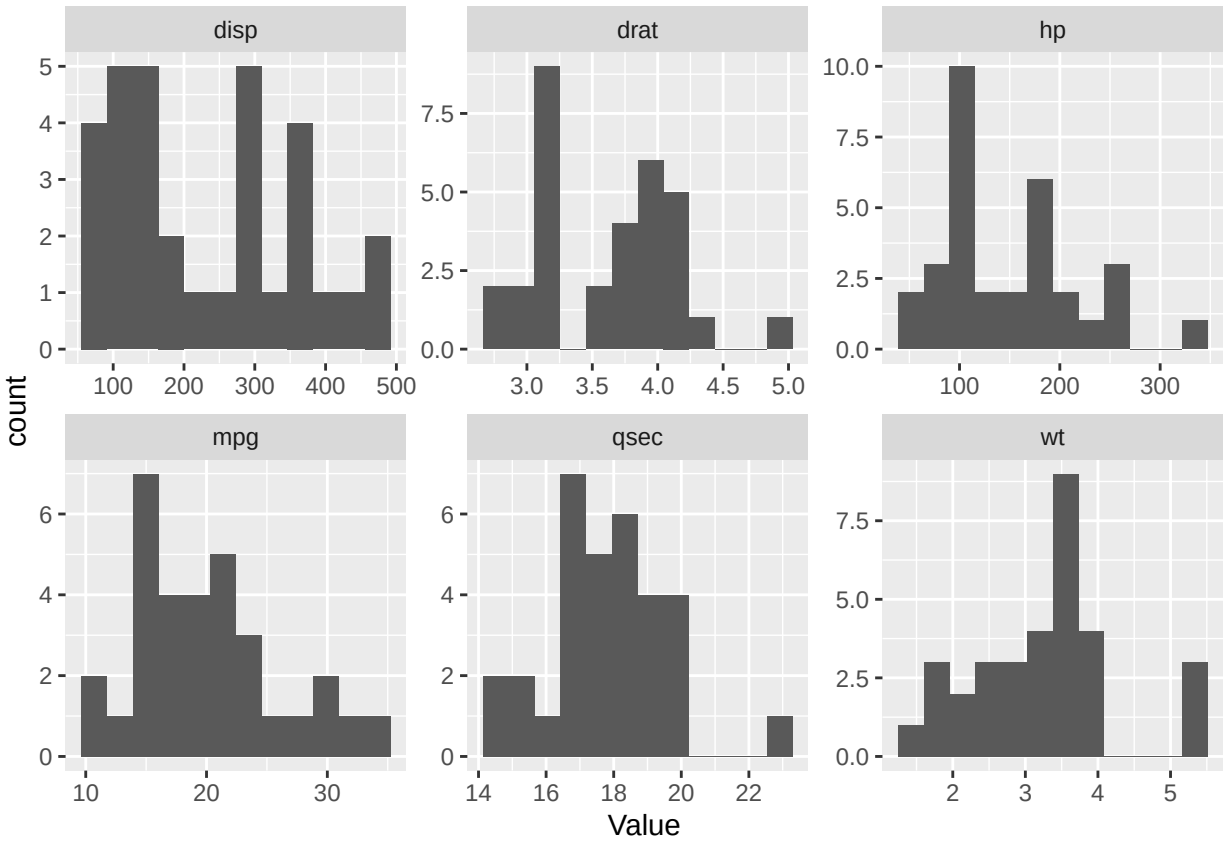
```
##                   id Variable Value
## 1          Mazda RX4      mpg  21.0
## 2      Mazda RX4 Wag      mpg  21.0
## 3         Datsun 710      mpg  22.8
## 4     Hornet 4 Drive      mpg  21.4
## 5  Hornet Sportabout      mpg  18.7
## 6            Valiant      mpg  18.1
```

```
tail(mtcars_long)
```

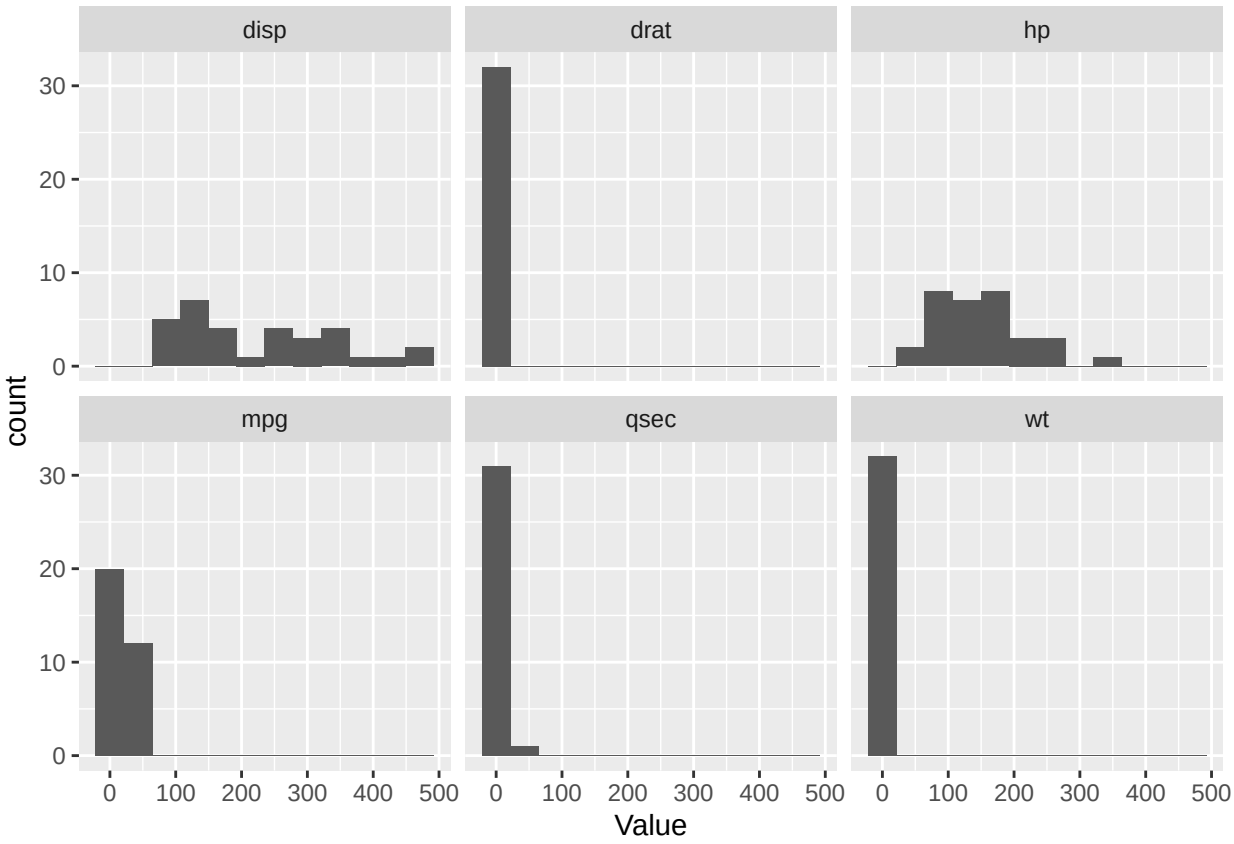
```
##                 id Variable Value
## 187  Porsche 914-2     qsec  16.7
## 188   Lotus Europa     qsec  16.9
## 189 Ford Pantera L     qsec  14.5
## 190   Ferrari Dino     qsec  15.5
## 191  Maserati Bora     qsec  14.6
## 192     Volvo 142E     qsec  18.6
```

Challenge Exercise. `gather()` can be particularly useful in conjunction with `ggplot()` and `facet_grid()` or `facet_wrap()`. Using the `mtcars_long` dataset you created in the previous exercise along with `ggplot()` and `facet_wrap()`, plot a histogram of the values for each variable. In other words, you should end up with one histogram for each of `disp`, `drat`, `hp`, `mpg`, `qsec`, and `wt`.

```
ggplot(mtcars_long) +
  aes(x = Value) +
  facet_wrap(~Variable, scales = "free") +
  geom_histogram(bins = 12)
```




```
# compare without scales = "free"
ggplot(mtcars_long) +
  aes(x = Value) +
  facet_wrap(~Variable) +
  geom_histogram(bins = 12)
```


disp
drat
hp
mpg
qsec
wt
count
Value
0
10
20
30
0
100
200
300
400
500