Data Science for Public Policy

Aaron R. Williams and Alena Stern - Georgetown University

Introduction to the tidyverse

Reading

- R for Data Science – Chapters 5, 6, 8, & 12
- The Statistical Crisis in Science
  - This article, by Andrew Gelman and Eric Loken, discusses the importance of data manipulation choices in statistical inference.
- OPTIONAL: Tidy Data
  - Please ignore the R code as it is out-of-date.

Review

Assignment operator

`<-` is the assignment operator. An object created on the right side of an assignment operator is assigned to a name on the left side of an assignment operator. Assignment operators are important for saving the consequences of operations and functions. Without assignment, the result of a calculation is not saved for use in a future calculation. Operations without assignment operators will typically be printed to the console but not saved for future use.

Functions

Functions are collections of code that take inputs, perform operations, and return outputs. R functions are similar to mathematical functions.

R functions typically contain arguments. For example, `mean()` has `x`, `trim`, and `na.rm`. Many arguments have default values and don't need to be included in function calls. Default values can be seen in the documentation. `trim = 0` and `na.rm = FALSE` are the defaults for `mean()`.

== vs. =

== is a binary comparison operator.

```
1 == 1
```

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

```
1 == 2
```

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

= is an equals sign, it is most frequently used for passing arguments to functions.

```
mean(x = c(1, 2, 3))
```

Tidy data

tidyverse

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. ~ tidyverse.org

`library(tidyverse)` contains:

- ggplot2, for data visualization.
- dplyr, for data manipulation. 今天讲这个
- tidyr, for data tidying.
- readr, for data import.
- purrr, for functional programming.
- tibble, for tibbles, a modern re-imagining of data frames.
- stringr, for strings.
- forcats, for factors.

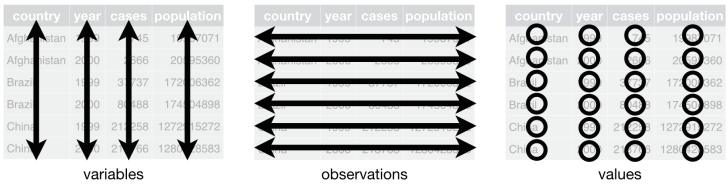
Opinionated software

> *Opinionated software* is a software product that believes a certain way of approaching a business process is inherently better and provides software crafted around that approach. ~ Stuart Eccles

Tidy data

The defining opinion of the tidyverse is its wholehearted adoption of tidy data. Tidy data has three features:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a dataframe. (This is from the paper, not the book)



Source: R4DS

Tidy data was formalized by Hadley Wickham in "Tidy Data" in the Journal of Statistical Software in 2014. It is equivalent to Codd's 3rd normal form (Codd, 1990) for relational databases.

> Tidy datasets are all alike, but every messy dataset is messy in its own way. ~ Hadley Wickham

The tidy approach to data science is powerful because it breaks data work into two distinct parts. First, get the data into a tidy format. Second, use tools optimized for tidy data. By standardizing the data structure for most community-created tools, the framework oriented diffuse development and reduced the friction of data work.

dplyr

`library(dplyr)` contains workhorse functions for manipulating and summarizing data once it is in a tidy format. `library(tidyr)` contains functions for getting data into a tidy format.

dplyr can be explicitly loaded with `library(dplyr)` or loaded with `library(tidyverse)`:

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.0     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.0     v forcats 0.5.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

We'll focus on the key dplyr syntax using the `storms` dataset, which is built in to dplyr. We can use `glimpse(storms)` to quickly view the data. We can use `?storms` to read about `storms` and `View(storms)` to open up `storms` in RStudio.

```
glimpse(x = storms)
```

```
## Rows: 10,010
## Columns: 13
## $ name        <chr> "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "Amy", "A~
## $ year        <dbl> 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975, 1975~
## $ month       <dbl> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7~
## $ day         <int> 27, 27, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30~
## $ hour        <dbl> 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0, 6, 12, 18, 0,~
## $ lat         <dbl> 27.5, 28.5, 29.5, 30.5, 31.5, 32.4, 33.3, 34.0, 34.4, 34.0~
## $ long        <dbl> -79.0, -79.0, -79.0, -79.0, -78.8, -78.7, -78.0, -77.0, -7~
## $ status      <chr> "tropical depression", "tropical depression", "tropical de~
## $ category    <ord> -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0,~
## $ wind        <int> 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 50, 55, 60~
## $ pressure    <int> 1013, 1013, 1013, 1013, 1012, 1012, 1011, 1006, 1004, 1002~
## $ ts_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
## $ hu_diameter <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA~
```

We're going to learn seven functions and one new piece of syntax from `library(dplyr)` that will be our main tools for manipulating tidy frames. These functions and a few extensions outlined in the Data Transformation Cheat Sheet are the core of data analysis in the

Tidyverse.

1. select()

`select()` reduces the number of columns in a dataframe or reorders the columns in a dataframe. The arguments after the name of the dataframe should be names of columns, without quotes.

```
select(.data = storms, name, category, wind, pressure)
```

```
## # A tibble: 10,010 x 4
##    name  category  wind pressure
##    <chr> <ord>    <int>    <int>
##  1 Amy   -1          25     1013
##  2 Amy   -1          25     1013
##  3 Amy   -1          25     1013
##  4 Amy   -1          25     1013
##  5 Amy   -1          25     1012
##  6 Amy   -1          25     1012
##  7 Amy   -1          25     1011
##  8 Amy   -1          30     1006
##  9 Amy   0           35     1004
## 10 Amy   0           40     1002
## # ... with 10,000 more rows
```

This works great until the goal is to select 99 of 100 variables. Fortunately, `-` can be used to remove variables. You can also select all but multiple variables by listing them with the `-` symbol separated by commas.

```
select(.data = storms, -hour)
```

```
## # A tibble: 10,010 x 12
##    name   year month   day   lat  long status            category  wind pressure
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <chr>             <ord>    <int>    <int>
##  1 Amy    1975     6    27  27.5 -79   tropical depress~ -1          25     1013
##  2 Amy    1975     6    27  28.5 -79   tropical depress~ -1          25     1013
##  3 Amy    1975     6    27  29.5 -79   tropical depress~ -1          25     1013
##  4 Amy    1975     6    27  30.5 -79   tropical depress~ -1          25     1013
##  5 Amy    1975     6    28  31.5 -78.8 tropical depress~ -1          25     1012
##  6 Amy    1975     6    28  32.4 -78.7 tropical depress~ -1          25     1012
##  7 Amy    1975     6    28  33.3 -78   tropical depress~ -1          25     1011
##  8 Amy    1975     6    28  34   -77   tropical depress~ -1          30     1006
##  9 Amy    1975     6    29  34.4 -75.8 tropical storm    0           35     1004
## 10 Amy    1975     6    29  34   -74.8 tropical storm    0           40     1002
## # ... with 10,000 more rows, and 2 more variables: ts_diameter <dbl>,
```

```
## #   hu_diameter <dbl>
```

Tidy data generally results in longer, wider data sets than other programming languages so iteratively selecting by column names is less important than in Stata or SAS. Still, dplyr contains powerful helper functions that can select variables based on patterns in column names:

- `contains()`: Contains a given string
- `starts_with()`: Starts with a prefix
- `ends_with()`: Ends with a suffix
- `matches()`: Matches a regular expression
- `num_range()`: Matches a numerical range

These are a subset of the `tidyselect` selection language and helpers which enable users to apply `library(dplyr)` functions to select variables.

Exercise

1. Select name and status from `storms`.
2. `pull()` is related to `select()` but can only select one variable. What is the other difference with `pull()`?

...

2. `rename()`

`rename()` renames columns in a data frame. The pattern is `new_name = old_name`.

```
rename(.data = storms, wind_mph = wind)
```

```
## # A tibble: 10,010 x 13
##    name   year month   day  hour   lat  long status   category wind_mph pressure
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>    <ord>       <int>    <int>
##  1 Amy    1975     6    27     0  27.5 -79   tropica~ -1             25     1013
##  2 Amy    1975     6    27     6  28.5 -79   tropica~ -1             25     1013
##  3 Amy    1975     6    27    12  29.5 -79   tropica~ -1             25     1013
##  4 Amy    1975     6    27    18  30.5 -79   tropica~ -1             25     1013
##  5 Amy    1975     6    28     0  31.5 -78.8 tropica~ -1             25     1012
##  6 Amy    1975     6    28     6  32.4 -78.7 tropica~ -1             25     1012
##  7 Amy    1975     6    28    12  33.3 -78   tropica~ -1             25     1011
##  8 Amy    1975     6    28    18  34   -77   tropica~ -1             30     1006
##  9 Amy    1975     6    29     0  34.4 -75.8 tropica~ 0              35     1004
## 10 Amy    1975     6    29     6  34   -74.8 tropica~ 0              40     1002
## # ... with 10,000 more rows, and 2 more variables: ts_diameter <dbl>,
## #   hu_diameter <dbl>
```

You can also rename a selection of variables using `rename_with()`. The `.cols` argument is used to select the columns to rename and takes a `tidyselect` statement like those we introduced above. Here, we're using the `where()` selection helper which selects all columns where a given condition is TRUE. The default value for the `.cols` argument is `everything()` which selects all columns in the dataset.

```
rename_with(.data = storms, .fn = toupper, .cols = where(is.numeric))
```

```
## # A tibble: 10,010 x 13
##    name   YEAR MONTH   DAY  HOUR   LAT  LONG status      category  WIND PRESSURE
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>       <ord>    <int>    <int>
##  1 Amy    1975     6    27     0  27.5 -79   tropical d~ -1          25     1013
##  2 Amy    1975     6    27     6  28.5 -79   tropical d~ -1          25     1013
##  3 Amy    1975     6    27    12  29.5 -79   tropical d~ -1          25     1013
##  4 Amy    1975     6    27    18  30.5 -79   tropical d~ -1          25     1013
##  5 Amy    1975     6    28     0  31.5 -78.8 tropical d~ -1          25     1012
##  6 Amy    1975     6    28     6  32.4 -78.7 tropical d~ -1          25     1012
##  7 Amy    1975     6    28    12  33.3 -78   tropical d~ -1          25     1011
##  8 Amy    1975     6    28    18  34   -77   tropical d~ -1          30     1006
##  9 Amy    1975     6    29     0  34.4 -75.8 tropical s~ 0           35     1004
## 10 Amy    1975     6    29     6  34   -74.8 tropical s~ 0           40     1002
## # ... with 10,000 more rows, and 2 more variables: TS_DIAMETER <dbl>,
## #   HU_DIAMETER <dbl>
```

在dplyr语句中，都可以使用new_name = 语法来重新命名一列。

Most `dplyr` functions can rename columns simply by prefacing the operation with `new_name =`. For example, this can be done with `select()`:

这些变化都要通过assign来保存在内存里，否则只会体现在console之中。

```
select(.data = storms, name, wind_mph = wind)
```

```
## # A tibble: 10,010 x 2
##    name  wind_mph
##    <chr>    <int>
##  1 Amy         25
##  2 Amy         25
##  3 Amy         25
##  4 Amy         25
##  5 Amy         25
##  6 Amy         25
##  7 Amy         25
##  8 Amy         30
##  9 Amy         35
## 10 Amy         40
## # ... with 10,000 more rows
```

3. `filter()` 筛选功能

`filter()` reduces the number of observations in a dataframe. Every column in a dataframe has a name. Rows do not necessarily have names in a dataframe, so rows need to be filtered based on logical conditions. 这说明这一项等于NA，是missing data

`==`, `<`, `>`, `<=`, `>=`, `!=`, `%in%`, and `is.na()` are all operators that can be used for logical conditions. `!` can be used to negate a condition and `&` and `|` can be used to combine conditions. `|` means or.

```
# return rows with wind speed greater than 100 mph and non-missing values
# for tropical storm conditions diameter
filter(.data = storms, wind > 100 & !is.na(ts_diameter))
```

加上！，是否认，说明不是missing data

```
## # A tibble: 241 x 13
##    name     year month   day  hour   lat  long status    category  wind pressure
##    <chr>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>     <ord>    <int>    <int>
##  1 Alex     2004     8     5     0  38.5 -66   hurricane 3          105      957
##  2 Alex     2004     8     5     6  39.5 -63.1 hurricane 3          105      957
##  3 Charley  2004     8    13     6  23   -82.6 hurricane 3          105      966
##  4 Charley  2004     8    13    18  26.1 -82.4 hurricane 4          125      947
##  5 Ivan     2004     9     5    18  10.2 -46.8 hurricane 3          110      955
##  6 Ivan     2004     9     6     0  10.6 -48.5 hurricane 4          115      948
##  7 Ivan     2004     9     6     6  10.8 -50.5 hurricane 3          110      950
##  8 Ivan     2004     9     6    12  11   -52.5 hurricane 3          110      955
##  9 Ivan     2004     9     7    18  11.8 -61.1 hurricane 3          105      956
## 10 Ivan     2004     9     8     0  12   -62.6 hurricane 4          115      950
## # ... with 231 more rows, and 2 more variables: ts_diameter <dbl>,
## #   hu_diameter <dbl>
```

Exercise

1. Filter `storms` to rows with `status` equal to "hurricane".
2. Filter `storms` to rows from the month August.
3. Filter `storms` to rows with `status` equal to "hurricane" or rows from the month August.

...

4. `arrange()`

`arrange()` sorts the rows of a data frame in alpha-numeric order based on the values of a variable or variables. The dataframe is sorted by the first variable first and each subsequent variable is used to break ties. `desc()` is used to reverse the sort order for a given variable.

```r
# sort name is descending order because the end of the alphabet is more
# interesting than the beginning of the alphabet
arrange(.data = storms, desc(name))
```

```
## # A tibble: 10,010 x 13
##    name   year month   day  hour   lat  long status      category  wind pressure
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>       <ord>    <int>    <int>
##  1 Zeta   2005    12    30     0  23.9 -35.6 tropical d~ -1          30     1009
##  2 Zeta   2005    12    30     6  24.2 -36.1 tropical s~ 0           40     1005
##  3 Zeta   2005    12    30    12  24.7 -36.6 tropical s~ 0           45     1002
##  4 Zeta   2005    12    30    18  25.2 -37   tropical s~ 0           45     1000
##  5 Zeta   2005    12    31     0  25.6 -37.3 tropical s~ 0           45     1000
##  6 Zeta   2005    12    31     6  25.7 -37.6 tropical s~ 0           50      997
##  7 Zeta   2005    12    31    12  25.7 -37.9 tropical s~ 0           50      997
##  8 Zeta   2005    12    31    18  25.7 -38.1 tropical s~ 0           45     1000
##  9 Zeta   2006     1     1     0  25.6 -38.3 tropical s~ 0           50      997
## 10 Zeta   2006     1     1     6  25.4 -38.4 tropical s~ 0           50      997
## # ... with 10,000 more rows, and 2 more variables: ts_diameter <dbl>,
## #   hu_diameter <dbl>
```

正确答案：arrange(storms, status, desc(name)) 因为顺序“ascending”不用写出来。先按status排序，再按name排序

Exercise

1. Sort `storms` in ascending order by status and descending order by name.

...

5. mutate()

`mutate()` creates new variables or edits existing variables. We can use arithmetic arguments, such as `+`, `-`, `*`, `/`, and `^`. We can also custom functions and functions from packages. For example, we can use `library(stringr)` for string manipulation and `library(lubridate)` for date manipulation.

Variables are created by adding a new column name, like `wind_mph`, to the left of `=` in `mutate()`.

```r
# convert wind from knots to miles per hour
mutate(.data = storms, wind_mph = wind * 1.15078)
```

```
## # A tibble: 10,010 x 14
##    name   year month   day  hour   lat  long status      category  wind pressure
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>       <ord>    <int>    <int>
##  1 Amy    1975     6    27     0  27.5 -79   tropical d~ -1          25     1013
##  2 Amy    1975     6    27     6  28.5 -79   tropical d~ -1          25     1013
```

```
##  3 Amy    1975     6    27    12  29.5 -79   tropical d~ -1          25     1013
##  4 Amy    1975     6    27    18  30.5 -79   tropical d~ -1          25     1013
##  5 Amy    1975     6    28     0  31.5 -78.8 tropical d~ -1          25     1012
##  6 Amy    1975     6    28     6  32.4 -78.7 tropical d~ -1          25     1012
##  7 Amy    1975     6    28    12  33.3 -78   tropical d~ -1          25     1011
##  8 Amy    1975     6    28    18  34   -77   tropical d~ -1          30     1006
##  9 Amy    1975     6    29     0  34.4 -75.8 tropical s~ 0           35     1004
## 10 Amy    1975     6    29     6  34   -74.8 tropical s~ 0           40     1002
## # ... with 10,000 more rows, and 3 more variables: ts_diameter <dbl>,
## #   hu_diameter <dbl>, wind_mph <dbl>
```

Variables are edited by including an existing column name, like `wind`, to the left of `=` in `mutate()`.

```
# adjust wind because the anemometer was improperly calibrated
mutate(.data = storms, wind = wind - 1)
```

```
## # A tibble: 10,010 x 13
##    name   year month   day  hour   lat  long status      category  wind pressure
##    <chr> <dbl> <dbl> <int> <dbl> <dbl> <dbl> <chr>       <ord>    <dbl>    <int>
##  1 Amy    1975     6    27     0  27.5 -79   tropical d~ -1          24     1013
##  2 Amy    1975     6    27     6  28.5 -79   tropical d~ -1          24     1013
##  3 Amy    1975     6    27    12  29.5 -79   tropical d~ -1          24     1013
##  4 Amy    1975     6    27    18  30.5 -79   tropical d~ -1          24     1013
##  5 Amy    1975     6    28     0  31.5 -78.8 tropical d~ -1          24     1012
##  6 Amy    1975     6    28     6  32.4 -78.7 tropical d~ -1          24     1012
##  7 Amy    1975     6    28    12  33.3 -78   tropical d~ -1          24     1011
##  8 Amy    1975     6    28    18  34   -77   tropical d~ -1          29     1006
##  9 Amy    1975     6    29     0  34.4 -75.8 tropical s~ 0           34     1004
## 10 Amy    1975     6    29     6  34   -74.8 tropical s~ 0           39     1002
## # ... with 10,000 more rows, and 2 more variables: ts_diameter <dbl>,
## #   hu_diameter <dbl>
```

Conditional logic inside of `mutate()` with functions like `if_else()` and `case_when()` is key to mastering data munging in R.

Exercise

1. Create a new variable called `storm_size` if the category of the storm is 3 or greater. Call storms >= 3 "bigger storms" and everything else "big storms". *Hint:* `if_else()` is useful and works like the IF command in Microsoft Excel.

...

mutate(storms, storm_size = if_else (condition = category >= 3, true = 'bigger storm", false = ''big storm"))

%>%

Data munging is tiring when each operation needs to be assigned to a name with <-. The pipe, %>%, allows lines of code to be chained together so the assignment operator only needs to be used once.

%>% passes the output from function as the first argument in a subsequent function. For example, this line can be rewritten:

```
# old way
mutate(.data = storms, wind_mph = wind * 1.15078)

# new way
storms %>%
  mutate(wind_mph = wind * 1.15078)
```

See the power:

这节省了很多指代功夫

```
new_storms <- storms %>%
  filter(year == 2005) %>%
  select(name, wind, pressure) %>%
  mutate(wind_mph = wind * 1.15078) %>%
  select(-wind)

new_storms
```

```
## # A tibble: 498 x 3
##    name  pressure wind_mph
##    <chr>    <int>    <dbl>
##  1 Emily     1010     28.8
##  2 Emily     1009     34.5
##  3 Emily     1009     34.5
##  4 Emily     1007     34.5
##  5 Emily     1006     40.3
##  6 Emily     1005     46.0
##  7 Emily     1004     51.8
##  8 Emily     1004     51.8
##  9 Emily     1003     51.8
## 10 Emily     1003     51.8
## # ... with 488 more rows
```

6. summarize()

summarize() collapses many rows in a dataframe into fewer rows with summary statistics of the many rows. n(), mean(), and sum() are common summary statistics. Renaming is useful with summarize()!

```
# summarize without renaming the statistics
storms %>%
  summarize(mean(wind), mean(pressure))
```

```
## # A tibble: 1 x 2
##   `mean(wind)` `mean(pressure)`
##          <dbl>            <dbl>
## 1         53.5             992.
```

```
# summarize and rename the statistics
storms %>%
  summarize(mean_wind = mean(wind), mean_pressure = mean(pressure))
```

```
## # A tibble: 1 x 2
##   mean_wind mean_pressure
##       <dbl>         <dbl>
## 1      53.5          992.
```

summarize() returns a data frame. This means all dplyr functions can be used on the output of summarize(). This is powerful! Manipulating summary statistics in Stata and SAS can be a chore. Here, it's just another dataframe that can be manipulated with a tool set optimized for dataframes: dplyr.

7. group_by() 这块没太跟上，下课了自学一下

group_by() groups a dataframe based on specified variables. summarize() with grouped dataframes creates subgroup summary statistics.

```
# one group
storms %>%
  group_by(year) %>%
  summarize(mean_wind = mean(wind),
            mean_pressure = mean(pressure))
```

```
## # A tibble: 41 x 3
##     year mean_wind mean_pressure
##    <dbl>     <dbl>         <dbl>
##  1  1975      50.9          995.
##  2  1976      59.9          989.
```

```
##  3  1977      54.0          995.
##  4  1978      40.5         1006.
##  5  1979      48.7          995.
##  6  1980      53.7          995.
##  7  1981      56.6          994.
##  8  1982      49.5          996.
##  9  1983      47.0         1001.
## 10  1984      51.4          995.
## # ... with 31 more rows
```

Dataframes can be grouped by multiple variables.

Grouped tibbles include metadata about groups. For example, `Groups:  status [3]`. One grouping is dropped each time `summarize()` is used. It is easy to forget if a dataframe is grouped, so it is safe to include `ungroup()` at the end of a section of functions.

```
# many groups
storms %>%
  group_by(status, category) %>%
  summarize(count = n(), mean_wind = mean(wind),
            mean_pressure = mean(pressure)) %>%
  arrange(category) %>%
  ungroup()
```

```
## `summarise()` has grouped output by 'status'. You can override using the `.groups` argument.
```

```
## # A tibble: 8 x 5
##   status              category count mean_wind mean_pressure
##   <chr>               <ord>    <int>     <dbl>         <dbl>
## 1 tropical depression -1        2545      27.3         1008.
## 2 tropical storm      0         4373      45.8          999.
## 3 hurricane           1         1684      70.9          982.
## 4 tropical storm      1            1      70            975
## 5 hurricane           2          628      89.4          967.
## 6 hurricane           3          363     105.           954.
## 7 hurricane           4          348     122.           940.
## 8 hurricane           5           68     145.           916.
```

是一个神奇的语法，不改变group的值，唯有在与其他dplyr语法互动的时候，会按照组来，例如summarize

Exercise

1. Create a new variable called `storm_size` if the category of the storm is 3 or greater. Call storms >= 3 "bigger storms" and everything else "big storms".
2. `group_by() storm_size`.
3. Find the mean and standard deviation of pressure using `summarize()`. It is always a good idea to include `n()` to evaluate cell size.

4. Add `ungroup()`.

...

BONUS: `count()`　非常非常有用，经常用到的语法。

`count()` is a shortcut to `df %>% group_by(var) %>% summarize(n())`. `count()` counts the number of observations with a level of a variable or levels of several variables. It is too useful to skip:

```
count(storms, status)
```

```
## # A tibble: 3 x 2
##   status                n
##   <chr>             <int>
## 1 hurricane          3091
## 2 tropical depression  2545
## 3 tropical storm       4374
```

```
count(x = storms, status, category)
```

```
## # A tibble: 8 x 3
##   status              category     n
##   <chr>               <ord>    <int>
## 1 hurricane           1         1684
## 2 hurricane           2          628
## 3 hurricane           3          363
## 4 hurricane           4          348
## 5 hurricane           5           68
## 6 tropical depression -1        2545
## 7 tropical storm      0         4373
## 8 tropical storm      1            1
```

Mutating joins multiple dataframes

Mutating joins join one dataframe to columns from another dataframe by matching values common in both dataframes. The syntax is derived from *Structured Query Language* (SQL).

Each function requires an `x` (or left) dataframe, a `y` (or right) data frame, and `by` variables that exist in both dataframes. Note that below we're creating dataframes using the `tribble()` function, which creates a tibble using a row-wise layout.

```
math_scores <- tribble(~name, ~math_score,
                       "Alec", 95,
                       "Bart", 97,
                       "Carrie", 100)

reading_scores <- tribble(~name, ~reading_score,
                          "Alec", 88,
                          "Bart", 67,
                          "Carrie", 100,
                          "Zeta", 100)
```

`left_join()`

`left_join()` matches observations from the `y` dataframe to the `x` dataframe. It only keeps observations from the `y` data frame that have a match in the `x` dataframe.

```
left_join(x = math_scores, y = reading_scores, by = "name")
```

```
## # A tibble: 3 x 3
##   name   math_score reading_score
##   <chr>       <dbl>         <dbl>
## 1 Alec           95            88
## 2 Bart           97            67
## 3 Carrie        100           100
```

Observations that exist in the `x` (left) dataframe but not in the `y` (right) dataframe result in `NA`s.

```
left_join(x = reading_scores, y = math_scores, by = "name")
```

```
## # A tibble: 4 x 3
##   name   reading_score math_score
##   <chr>          <dbl>      <dbl>
## 1 Alec              88         95
## 2 Bart              67         97
```

总是和x的dataframe的行数保持一致，所以有可能换一下x和y，能得到更多行（像这个例子中一样）

```
## 3 Carrie           100        100
## 4 Zeta             100         NA
```

inner_join()

`inner_join()` matches observations from the `y` dataframe to the `x` dataframe. It only keeps observations from either data frame that have a match.

a strict join

```
inner_join(x = reading_scores, y = math_scores, by = "name")
```

```
## # A tibble: 3 x 3
##   name   reading_score math_score
##   <chr>          <dbl>      <dbl>
## 1 Alec              88         95
## 2 Bart              67         97
## 3 Carrie           100        100
```

full_join()

`full_join()` matches observations from the `y` dataframe to the `x` dataframe. It keeps observations from both dataframes.

```
full_join(x = reading_scores, y = math_scores, by = "name")
```

```
## # A tibble: 4 x 3
##   name   reading_score math_score
##   <chr>          <dbl>      <dbl>
## 1 Alec              88         95
## 2 Bart              67         97
## 3 Carrie           100        100
## 4 Zeta             100         NA
```

anti_join()

`anti_join()` returns all rows from `x` where there are not matching values in `y`. `anti_join()` complements `inner_join()`. Together, they should exhaust the `x` dataframe.

```r
anti_join(x = reading_scores, y = math_scores, by = "name")
```

```
## # A tibble: 1 x 2
##   name  reading_score
##   <chr>         <dbl>
## 1 Zeta            100
```

The Combine Tables column in the Data Transformation Cheat Sheet is an invaluable resource for navigating joins. The "column matching for joins" section of that cheat sheet outlines how to join tables by matching on multiple columns or match on columns with different names in each table.

readr

`readr` is a core tidyverse package for reading and parsing rectangular data from text files (.csv, .tsv, etc.). `read_csv()` reads `.csv` files and has a bevy of advantages versus `read.csv()`. We recommend never using `read.csv()`.

Many `.csv`s can be read without issue with simple syntax `read_csv(file = "relative/path/to/data")`.

`readr` and `read_csv()` have powerful tools for resolving parsing issues. More can be learned in the data import section in R4DS.

readxl

readxl is a tidyverse package for reading data from Microsoft Excel files. It is not a core tidyverse package so it needs to be explicitly loaded in each R session.

The tidyverse website has a good tutorial on readxl.

Next Skills

- `across()` can be used with `library(dplyr)` functions such as `summarise()` and `mutate()` to apply the same transformations to multiple columns. For example, it can be used to calculate the mean of many columns with `summarize()`. `across()` uses the same `tidyselect` select language and helpers discussed earlier to select the columns to transform.
- `pivot_wider()` and `pivot_longer()` can be used to switch between wide and long formats of the data. This is important for tidying data and data visualization.