Code ▾

GCDS: Data Frame Manipulation and Wrangling

1 Libraries for Data Frame Manipulation

- `magrittr` : for piping data frame objects
- `dplyr` : for selecting, filtering, and mutating
- `stringr` : for working with strings

Hide

```
library(magrittr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

Hide

```
library(stringr)
```

2 A Grammar for Data Wrangling

The `dplyr` package presents a type of grammar for wrangling data (H. Wickham and Francois 2020). `dplyr` is part of the `tidyverse` ecosystem and loads using `library(tidyverse)` . `dplyr` also imports functions from `tidyselect` , which is also part of the `tidyverse` ecosystem.

The functions from the packages in the `tidyverse` can can be used without referencing the packages explicitly but to avoid functions of the same name from other packages being confused, we will reference the package/library and the function (e.g., `package::function()`).

Working with data involves creating new column variables, removing or renaming column variables, removing or deleting row observations, sorting, and summarizing data, often by groups. Consequently, there are five main function verbs for working with data in a data frame: `select` , `filter` , `mutate` , `arange` , and `summarize` .

- `select(dataframe, variables_to_select)` : subset by columns
- `mutate(dataframe, variables_to_create)` and `dplyr::rename()` : add or modify existing columns
- `filter(dataframe, rows_to_select)` : subset by rows
- `arrange(dataframe, variable_to_sort_by)` : sort rows
- `summarize(dataframe, procedures_for_summarizing)` in conjunction with `dplyr::group_by()` : aggregate the data in some way

3 Some Common Ways for Selecting Variables Using `dplyr`

Using `select()` , you can select columns/variables from a data frame. The variables you select are retained and those that you don't select are not included in the returned data frame.

If not using `%>%` , the first argument passed into `select()` will be the data frame from which to select the variables and the second and subsequent arguments can be variables.

```
select(mydataframe, myvars)
```

If piping a data frame with `magrittr` , the “.” or “.data” will serve to reference the inherited data frame.

```
dataframe %>%  select(., myvars)
```

Variables can be passed separately without quotes or collectively as a character vector.

Hide

```
# passing variables by name (does not work with base R manipulation)

# passing variables separately
USArrests %>%
  select(., Murder, Assault) %>% head()
```

```
##            Murder Assault
## Alabama      13.2     236
## Alaska       10.0     263
## Arizona       8.1     294
## Arkansas      8.8     190
## California    9.0     276
## Colorado      7.9     204
```

Hide

```
# passing variables separately as characters
# though combining then with c() is probably more clear
USArrests %>%
  select(., "Murder", "Assault") %>% head()
```

```
##            Murder Assault
## Alabama      13.2     236
## Alaska       10.0     263
## Arizona       8.1     294
## Arkansas      8.8     190
## California    9.0     276
## Colorado      7.9     204
```

Hide

```
# passing a character vector
USArrests %>%
  select(., c("Murder", "Assault")) %>% head()
```

```
##            Murder Assault
## Alabama      13.2     236
## Alaska       10.0     263
## Arizona       8.1     294
## Arkansas      8.8     190
## California    9.0     276
## Colorado      7.9     204
```

Hide

```
# passing an object holding a character vector
keep_vars <- c("Murder", "Assault") %>% head()



USArrests %>%
  select(., keep_vars)
```

```
## Note: Using an external vector in selections is ambiguous.
## ℹ Use `all_of(keep_vars)` instead of `keep_vars` to silence this message.
## ℹ See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
##                Murder Assault
## Alabama          13.2      236
## Alaska           10.0      263
## Arizona           8.1      294
## Arkansas          8.8      190
## California        9.0      276
## Colorado          7.9      204
## Connecticut       3.3      110
## Delaware          5.9      238
## Florida          15.4      335
## Georgia          17.4      211
## Hawaii            5.3       46
## Idaho             2.6      120
## Illinois         10.4      249
## Indiana           7.2      113
## Iowa              2.2       56
## Kansas            6.0      115
## Kentucky          9.7      109
## Louisiana        15.4      249
## Maine             2.1       83
## Maryland         11.3      300
## Massachusetts     4.4      149
## Michigan         12.1      255
## Minnesota         2.7       72
## Mississippi      16.1      259
## Missouri          9.0      178
## Montana           6.0      109
## Nebraska          4.3      102
## Nevada           12.2      252
## New Hampshire     2.1       57
## New Jersey        7.4      159
## New Mexico       11.4      285
## New York         11.1      254
## North Carolina   13.0      337
## North Dakota      0.8       45
## Ohio              7.3      120
## Oklahoma          6.6      151
## Oregon            4.9      159
## Pennsylvania      6.3      106
## Rhode Island      3.4      174
## South Carolina   14.4      279
## South Dakota      3.8       86
## Tennessee        13.2      188
## Texas            12.7      201
## Utah              3.2      120
## Vermont           2.2       48
## Virginia          8.5      156
## Washington        4.0      145
## West Virginia     5.7       81
## Wisconsin         2.6       53
## Wyoming           6.8      161
```

3.1 *Select Variables Starting with or Ending with Certain Characters*

One thing about `dplyr` , when you load the library, there are functions from other libraries that are imported along with `dplyr` s own functions. These important functions are designed to work with each other, so the people who maintain the libraries have packeged them up nicely so you don't have to load separate libraries.

Many of the functions are imported from the `tidyselect` library and these functions are what give you additional manipulation ability. Some imported functions are: `all_of()` , `any_of()` , `contains()` , `ends_with()` , `everything()` , `last_col()` , `matches()` , `starts_with()` .

With functions like `starts_with()` , `contains()` , and `ends_with()` , you can select variables with patterns in their names.

Rather that passing the names of the variables as the second argument (e.g., `c("Murder", "Assault")`), you would pass the helper function, say `starts_with()` . Whatever `starts_with()` returns is what gets passed to `select()` as the variables. This is what is referred to as functional programming. Rather than coding specifically what to do, you with utilize the task of another function to passed its returned object as an arguemn to another function.

But first, we need to see what's going on with these functions, like `starts_with()` .

```
starts_with(match, ignore.case = TRUE, vars = NULL)
```

Notice the arguments we need to pass:

- `match` : A character vector
- `ignore.case` : If `TRUE` , the default, ignores case when matching names. This is most flexible.
- `vars` : A character vector of variable names. If not supplied, the variables are taken from the current selection context (as established by functions like `select()` or `pivot_longer()`).

Let's just try out `starts_with()` on its own. Let's set a required pattern `match = some character` and because `vars = NULL` by default, let's just set `vars = some character vector` . Note that `vars` is not the second argument, so you will want to name it in the function call.

Hide

```
starts_with(match = "a", vars = c("Hello", "Hi", "Bye"))
```

```
## integer(0)
```

Returns `integer(0)` which is speak for "there is no match". Hmm, OK. Let's try another character.

Hide

```
starts_with(match = "b", vars = c("Hello", "Hi", "Bye"))
```

```
## [1] 3
```

OK, so now an integer is returned (yes, try `is.integer()` if you don't believe me).

Hide

```
is.integer(starts_with("b", vars = c("Hello", "Hi", "Bye")))
```

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

Importantly, the value refers to the element index/position in the `vars` vector. Because the third string `"Bye"` starts with `"b"` , that’s what is returned.

Try something else…

Hide

```
starts_with("h", vars = c("Hello", "Hi", "Bye"))
```

```
## [1] 1 2
```

Now a vector with length = 2 is returned, representing both the first and the second elements start with “h”.

Hide

```
length(starts_with("h", vars = c("Hello", "Hi", "Bye")))
```

```
## [1] 2
```

See, it’s really a vector containing the element(s) of the `vars` vector matching the pattern.

And yes, this the letter casing is ignored because the default `ignore.case = TRUE` . Set to `FALSE` if you want your match to be case sensitive.

Hide

```
starts_with("h",
            vars = c("Hello", "Hi", "Bye"),
            ignore.case = F)
```

```
## integer(0)
```

OK, no matches.

You will typically use `starts_with()` along with other functions. When using `starts_with()` in the context of `select()` , the `vars` argument is essentially passing `vars = the names of the columns of the data frame passed to select()` .

Example:

`select(mydataframe, starts_with(match = "my pattern", vars = "var names of mydataframe"))`

Without piping…

Hide

```
select(USArrests, starts_with("m")) %>% head()
```

```
##            Murder
## Alabama      13.2
## Alaska       10.0
## Arizona       8.1
## Arkansas      8.8
## California    9.0
## Colorado      7.9
```

With piping…

Hide

```
USArrests %>%
  select(., starts_with("m")) %>% head()
```

```
##            Murder
## Alabama      13.2
## Alaska       10.0
## Arizona       8.1
## Arkansas      8.8
## California    9.0
## Colorado      7.9
```

Hide

```
USArrests %>%
  select(., ends_with("t")) %>% head()
```

```
##            Assault
## Alabama        236
## Alaska         263
## Arizona        294
## Arkansas       190
## California     276
## Colorado       204
```

3.2 *Selecting and Selecting Out Variables By/Between Index*

- `select(., 1,2)` : select first and second
- `select(., c(1,2))` : select first and second
- `select(., -c(1,2))` : select out first and second
- `select(., 1:2)` : select first through second

- `select(., c(1:2))` : select first through second

- `select(., -c(1:2))` : select out first through second

Recommendation: use options utilizing `c()` as this habit will be more versatile with base R functionality.

Let’s make a data frame to work with first.

Hide

```
data <- data.frame(
  Id  = c(100, 101, 102, 103, 104, 100, 105),
  Sex = c('male', 'female', 'Male', NA, 'man', "male", "neither"),
  Age = c(25, 33, 27, 40, 44, 25, 40),
  Renting = c("yes", NA, "yes", NA, "no", "yes", "yes")
)
```

Hide

```
data %>%
  select(., 1,2) # select this and that
```

```
##    Id     Sex
## 1 100    male
## 2 101  female
## 3 102    Male
## 4 103    <NA>
## 5 104     man
## 6 100    male
## 7 105 neither
```

Hide

```
data %>%
  select(., c(1,2)) # select this and that
```

```
##    Id     Sex
## 1 100    male
## 2 101  female
## 3 102    Male
## 4 103    <NA>
## 5 104     man
## 6 100    male
## 7 105 neither
```

Hide

```
data %>%
  select(., -c(1,2)) # select out this and that
```

```
##   Age Renting
## 1  25     yes
## 2  33    <NA>
## 3  27     yes
## 4  40    <NA>
## 5  44      no
## 6  25     yes
## 7  40     yes
```

Hide

```
data %>%
  select(., 1:2) # select from here to there
```

```
##    Id     Sex
## 1 100    male
## 2 101  female
## 3 102    Male
## 4 103    <NA>
## 5 104     man
## 6 100    male
## 7 105 neither
```

Hide

```
data %>%
  select(., c(1:3)) # select from here to there
```

```
##    Id     Sex Age
## 1 100    male  25
## 2 101  female  33
## 3 102    Male  27
## 4 103    <NA>  40
## 5 104     man  44
## 6 100    male  25
## 7 105 neither  40
```

Hide

```
data %>%
  select(., -c(1:3))   # select out from here to there
```

```
##   Renting
## 1     yes
## 2    <NA>
## 3     yes
## 4    <NA>
## 5      no
## 6     yes
## 7     yes
```

3.3 *Selecting and Selecting Out Variables By or Between Character Name*

- `select(., "var1", "var2")`

- `select(., c("var1", "var2"))`

- `select(., -c("var1", "var2"))`

- `select(., var1:var2))`

- `select(., c("var1":"var2))`

- `select(., -c("var1":"var2))`

Recommendation: use options utilizing `c()` as this will be more versatile with base R functionality.

These also work but may lead to some confusion regarding usage of quotes:

- `select(., var1, var2)`
- `select(., c(var1, var2))`
- `select(., -c(var1, var2))`

Hide

```
data %>%
  select(., Id:Age) # select from here to there
```

```
##    Id     Sex Age
## 1 100    male  25
## 2 101  female  33
## 3 102    Male  27
## 4 103    <NA>  40
## 5 104     man  44
## 6 100    male  25
## 7 105 neither  40
```

Hide

```
data %>%
  select(., "Id":"Age") # select from here to there
```

```
##    Id     Sex Age
## 1 100    male  25
## 2 101  female  33
## 3 102    Male  27
## 4 103    <NA>  40
## 5 104     man  44
## 6 100    male  25
## 7 105 neither  40
```

Hide

```
data %>%
  select(., c("Id":"Age")) # select from here to there
```

```
##    Id     Sex Age
## 1 100    male  25
## 2 101  female  33
## 3 102    Male  27
## 4 103    <NA>  40
## 5 104     man  44
## 6 100    male  25
## 7 105 neither  40
```

Hide

```
data %>%
  select(., -c("Id":"Age"))   # select out from here to there
```

```
##   Renting
## 1     yes
## 2    <NA>
## 3     yes
## 4    <NA>
## 5      no
## 6     yes
## 7     yes
```

Hide

```
# you can also use the ! operator to select NOT these variables (therefore, all others)
data %>%
  select(., !c("Id":"Age"))   # select out from here to there
```

```
##   Renting
## 1     yes
## 2    <NA>
## 3     yes
## 4    <NA>
## 5      no
## 6     yes
## 7     yes
```

3.4 *Selecting and Selecting Out Variables Characters in Their Names*

- `select(., starts_with("character/s"))`
- `select(., ends_with("character/s"))`
- `select(., contains('e'))`

Hide

```
data %>% select(., starts_with('i'))
```

```
##    Id
## 1 100
## 2 101
## 3 102
## 4 103
## 5 104
## 6 100
## 7 105
```

Hide

```
data %>% select(., -starts_with('s'))
```

```
##    Id Age Renting
## 1 100  25     yes
## 2 101  33    <NA>
## 3 102  27     yes
## 4 103  40    <NA>
## 5 104  44      no
## 6 100  25     yes
## 7 105  40     yes
```

Hide

```
data %>% select(., ends_with('e'))
```

```
##   Age
## 1  25
## 2  33
## 3  27
## 4  40
## 5  44
## 6  25
## 7  40
```

Hide

```
data %>% select(., -ends_with('e'))
```

```
##    Id     Sex Renting
## 1 100    male     yes
## 2 101  female    <NA>
## 3 102    Male     yes
## 4 103    <NA>    <NA>
## 5 104     man      no
## 6 100    male     yes
## 7 105 neither     yes
```

Hide

```
data %>% select(., contains('g'))
```

```
##   Age Renting
## 1  25     yes
## 2  33    <NA>
## 3  27     yes
## 4  40    <NA>
## 5  44      no
## 6  25     yes
## 7  40     yes
```

Hide

```
data %>% select(., -contains('g'))
```

```
##    Id     Sex
## 1 100    male
## 2 101  female
## 3 102    Male
## 4 103    <NA>
## 5 104     man
## 6 100    male
## 7 105 neither
```

3.5 *Selecting and Selecting Out Variables by Type*

Hide

```
data %>% select(., where(is.numeric))
```

```
##    Id Age
## 1 100  25
## 2 101  33
## 3 102  27
## 4 103  40
## 5 104  44
## 6 100  25
## 7 105  40
```

Hide

```
data %>% select(., -where(is.numeric))
```

```
##       Sex Renting
## 1    male     yes
## 2  female    <NA>
## 3    Male     yes
## 4    <NA>    <NA>
## 5     man      no
## 6    male     yes
## 7 neither     yes
```

Hide

```
data %>% select(., where(is.character))
```

```
##       Sex Renting
## 1    male     yes
## 2  female    <NA>
## 3    Male     yes
## 4    <NA>    <NA>
## 5     man      no
## 6    male     yes
## 7 neither     yes
```

Hide

```
data %>% select(., -where(is.character))
```

```
##    Id Age
## 1 100  25
## 2 101  33
## 3 102  27
## 4 103  40
## 5 104  44
## 6 100  25
## 7 105  40
```

Hide

```
data %>% select(., where(is.logical))
```

```
## data frame with 0 columns and 7 rows
```

Hide

```
data %>% select(., -where(is.logical))
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 101  female  33    <NA>
## 3 102    Male  27     yes
## 4 103    <NA>  40    <NA>
## 5 104     man  44      no
## 6 100    male  25     yes
## 7 105 neither  40     yes
```

4 **Cleaning Data**

Data files are messy and as a result require cleaning. You will have missing rows, incorrect variable names, files with columns named the same, `NA` s, strings for numbers, duplicate rows of data, people who completed a survey twice, and all sorts of unimaginable and unbelievable data problems. So cleaning is important.

Whereas `select()` is used for columns, `filter()` operates on rows. Data frame manipulation may involve keeping only certain rows for data, for example, male or female respondents, male respondents, those who do not contain missing values (e.g., `NA` s) for a specific column variable, who are of a certain age (or born in in certain year), who are above (or below) some acceptable criterion, etc.

When a column variable has more than one value (e.g., check using `unique()` to determine the unique elements contained), you may wish to filter on some but not others.

You may even need to filter rows in a data frame that are distinct (e.g., not duplicate responses). This is often a good first step in order to determine the size of the usable data set. `dplyr::distinct()` makes de-duplicating easy as this function will return only distinct rows.

4.1 *Removing duplicate rows using `distinct()`*

- `dplyr::distinct()` : remove duplicate rows
- `dplyr::distinct(., column)` : remove duplicate rows by column
- `na.omit()` : remove any row with NA’s

Let’s use the simple `data` data frame.

Hide

```
data %>% view(.)     # the full data frame
```

Show 100 **entries** **Search:**

	Id	**Sex**	**Age**	**Renting**
	All	All	All	All
1	100	male	25	yes
2	101	female	33	
3	102	Male	27	yes
4	103		40	
5	104	man	44	no
6	100	male	25	yes
7	105	neither	40	yes

Showing 1 to 7 of 7 entries Previous 1 Next

Notice that rows 1 and 6 are the same person (e.g., Id) and have exactly the same data for all variables.

Hide

```
data[1,] == data[6,]
```

```
##     Id  Sex  Age Renting
## 1 TRUE TRUE TRUE    TRUE
```

Great that they are consistent but you don’t want their data twice. So let’s just remove any rows that are identical.

Hide

```
data %>%
  distinct(.) %>%    # Remove exact duplicates
  view(.)
```

Show 100 entries Search:

	Id	Sex	Age	Renting
	All	All	All	All
1	100	male	25	yes
2	101	female	33	
3	102	Male	27	yes
4	103		40	
5	104	man	44	no
6	105	neither	40	yes

Showing 1 to 6 of 6 entries Previous 1 Next

If you know each row is unique based on a variable in the data frame, you can use `distinct()` for that variable.

Hide

```
data %>%
  distinct(., Id) %>% view(.) # Remove duplicates by variable; passes unique values for data frame
```

Show 100 entries Search:

	Id
	All
1	100
2	101
3	102
4	103
5	104
6	105

Showing 1 to 6 of 6 entries

Previous 1 Next

But this just returns the unique values in `Id`. To retain the variables, set `.keep_all = T`.

Hide

```
data %>%
  distinct(., Id, .keep_all = T) %>% view(.)
```

Show 100 **entries** **Search:**

	Id	Sex	Age	Renting
	All	All	All	All
1	100	male	25	yes
2	101	female	33	
3	102	Male	27	yes
4	103		40	
5	104	man	44	no
6	105	neither	40	yes

Showing 1 to 6 of 6 entries

Previous 1 Next

Notice, however, this only removed the last instance or `Id == 100`. Which row to include is a judgment call. The first, the last, neither, the average? Is there a correct answer?

5 Filtering using `dplyr` and Understanding Filtering Operators

Filtering cases using the `dplyr::filter()` verbs works by removing rows that do not match a specific criterion and then by returning the data frame that omits the mismatched condition.

Some useful filtering operators and functions include: `==`, `>`, `>=`, `&`, `|`, `!`, `xor()`, `c()`, `is.na()`, `between()`, `near()`.

Row/Obersvations/Cases can be filtered to “include” only certain matched conditions or can be filtered to “exclude” by negating those matched conditions. If the column variable `Sex` is in the data frame and cases are `'male'`, `'men'`, `'female'`, `'women'`, `'neither'`, `NA`, etc., you can specify the column `Sex` variable and then the row matching condition(s).

The first argument in `dplyr::filter()` is a data frame, and the function all `dplyr::filter(data, Sex == 'female')` will filter the data frame named `data` to include rows for which the `sex` column equals `'female'` . In other words, `TRUE` rows.

Hide

```
dplyr::filter(data, Sex == 'female')
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
```

Similarly, the function call `dplyr::filter(., Sex == 'male')` can be read “filter the data frame to include rows for which the value of `Sex == 'male'` is `TRUE` ”.

More flexibly, however, you could specify a vector containing acceptable strings using `c()` . `dplyr::filter(., Sex %in% c('male'))` filters the rows to include only those for which the value for `sex` is in the string vector which includes a single string, `'male'` whereas `dplyr::filter(., Sex %in% c('male', 'Man'))` filters the rows to include only those for which the value for `Sex` is in the string vector which includes `'male'` and `'Man'` . Cases containing `'Male'` , `'Men'` (R is a case-sensitive language), or `'female'` , for example, will not be included in the returned data frame because they do not match values in the string vector.

6 **Piping Multiple Filter Function Calls**

In many cases, data filtering will involve different conditions for different column variables, so specifying them separately as separate lines of code is most appropriate.

When passing a data frame using `%>%` from `magrittr` , the first argument for the data frame can be specified using a `.` because the function inherits the data frame manipulated. However, `dplyr` also understand this so the `.` can also be omitted for convenience; this is the general practice you will see in forums like (stackoverflow.com)[stackoverflow.com]. Example to follow.

7 **Filtering Cases by Character Names/String Values**

7.1 *Filter Cases using* `==`

Hide

```
data %>%
  dplyr::filter(., Sex == 'female')
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
```

Hide

```
# not equal
data %>%
  dplyr::filter(., Sex != 'female')
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 102    Male  27     yes
## 3 104     man  44      no
## 4 100    male  25     yes
## 5 105 neither  40     yes
```

Hide

```
# multiple filters
data %>%
  dplyr::filter(., Sex == 'female' & Age > 27) # this "AND" that
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
```

Hide

```
data %>%
  dplyr::filter(., Sex == 'female' | Age > 27) # this "OR" that
```

```
##    Id     Sex Age Renting
## 1 101  female  33    <NA>
## 2 103    <NA>  40    <NA>
## 3 104     man  44      no
## 4 105 neither  40     yes
```

A cleaner method involves separate lines of code. Although cleaner, this will not allow the “OR” option because the data frame that is returned from the first `filter()` is passed to the second `filter()` and all cases other than `"female"` have already been removed from the data frame.

Hide

```
data %>%
  dplyr::filter(., Sex == 'female') %>%   # keep female (and add another pipe)
  dplyr::filter(., Age >= 27)             # keep only those equal to or older than 27
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
```

8 Filtering Cases by Value

8.1 *Filter by `<` and `>` or `<=` or `>=` …*

Hide

```
data %>% dplyr::filter(., Age < 40)  # keep those less than
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 102   Male  27     yes
## 4 100   male  25     yes
```

Hide

```
data %>% dplyr::filter(., Age > 40)  # keep older than
```

```
##    Id Sex Age Renting
## 1 104 man  44      no
```

Hide

```
data %>% dplyr::filter(., Age >= 40)  # keep equal to or older than
```

```
##    Id     Sex Age Renting
## 1 103    <NA>  40    <NA>
## 2 104     man  44      no
## 3 105 neither  40     yes
```

9 Filter Cases by Conditional X or Y Using `|` Operator…

Using the “OR” operator, `|`, cases can be included if “this” OR “that” condition.

Hide

```
data %>%
  dplyr::filter(., Age == 25 | Age == 40)    # filter out numeric values IN a range
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 103    <NA>  40    <NA>
## 3 100    male  25     yes
## 4 105 neither  40     yes
```

Hide

```
data %>%
  dplyr::filter(., Sex == 'male' | Sex == 'female')
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 100   male  25     yes
```

Hide

```
# although dplyr::filter(sex %in% c('male', 'female')) would be easier


data %>%
  dplyr::filter(., Sex == 'male' | Age == 27)
```

```
##    Id  Sex Age Renting
## 1 100 male  25     yes
## 2 102 Male  27     yes
## 3 100 male  25     yes
```

Hide

```
data %>%
  dplyr::filter(., between(Age, 27, 33))
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
## 2 102   Male  27     yes
```

Hide

```
data %>%
  dplyr::filter(., between(Age, 27, 33))
```

```
##    Id    Sex Age Renting
## 1 101 female  33    <NA>
## 2 102   Male  27     yes
```

9.1 *Filter by range using `%in%` …*

Though less flexible than using `between()` …

Hide

```
data %>%
  dplyr::filter(., Age %in% 20:43)    # filter out numeric values IN a range
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 101  female  33    <NA>
## 3 102    Male  27     yes
## 4 103    <NA>  40    <NA>
## 5 100    male  25     yes
## 6 105 neither  40     yes
```

If a vector object is already defined (e.g., `my_levels = c('male', 'female')`), it can be used for filtering also. Such approaches are useful when data manipulation involves reusing a reference as it simplifies coding and reduces errors because the specification is defined only once.

Hide

```
my_levels = c('male', 'female')

data %>%
  dplyr::filter(., Sex %in% my_levels)
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 100   male  25     yes
```

When inclusion is inappropriate, exclusion may be useful. The `!` operator means “NOT” in `R` so it is great to accomplish the opposite. For example, `dplyr::filter(. !sex %in% c('male', NA))` will “filter the data frame to include rows in the `sex` column for which the value is NOT in the vector”.

Hide

```
# exclusion
data %>%
  dplyr::filter(., !Sex %in% c('male', NA))  # keep only if NOT in vector
```

```
##    Id     Sex Age Renting
## 1 101  female  33    <NA>
## 2 102    Male  27     yes
## 3 104     man  44      no
## 4 105 neither  40     yes
```

Hide

```
data %>%
  dplyr::filter(., !Sex %in% c('male', 'Men'))  # keep only if NOT in vector
```

```
##    Id     Sex Age Renting
## 1 101  female  33    <NA>
## 2 102    Male  27     yes
## 3 103    <NA>  40    <NA>
## 4 104     man  44      no
## 5 105 neither  40     yes
```

9.2 *Filter by conditional X and Y using `&` operator…*

Hide

```
data %>%
  dplyr::filter(., Id >= 102 & Age <= 43)    # filter by range
```

```
##    Id     Sex Age Renting
## 1 102    Male  27     yes
## 2 103    <NA>  40    <NA>
## 3 105 neither  40     yes
```

Hide

```
data %>%
  dplyr::filter(., Age >= 20 & Age <= 43)    # filter by range
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 101  female  33    <NA>
## 3 102    Male  27     yes
## 4 103    <NA>  40    <NA>
## 5 100    male  25     yes
## 6 105 neither  40     yes
```

Hide

```
#Note: Age 20:43 won't work. Can you figure out why?
```

10 **Filter Cases containing characters Using `str_detect()` and `%in%`…**

- `stringr::str_detect()` : returns matching conditions

Hide

```
data %>% dplyr::filter(., stringr::str_detect(Sex, "ma"))
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 104    man  44      no
## 4 100   male  25     yes
```

But the case for which `Sex = Male` is now missing. This is because `stringr::str_detect()` is a case-sensitive pattern match.

You can fix this is a couple ways:

1. make the cases in `Sex` all lower case to `mutate()` the fix or
2. to wrap `Sex` in `tolower()` to make cases lowercase.

The first option might be better if you want to fix the problem in the data frame.

Other casing functions are:

- `tolower()` : returns lower case of string
- `toupper()` : returns upper case of string
- `tools::toTitleCase()` : returns Title Case (capitalize first letter of string)

Hide

```
data %>%
  mutate(., Sex = tolower(Sex)) %>%
  filter(., stringr::str_detect(tolower(Sex), "ma"))
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 102   male  27     yes
## 4 104    man  44      no
## 5 100   male  25     yes
```

Hide

```
data %>%
  filter(., stringr::str_detect(tolower(Sex), "ma"))
```

```
##    Id    Sex Age Renting
## 1 100   male  25     yes
## 2 101 female  33    <NA>
## 3 102   Male  27     yes
## 4 104    man  44      no
## 5 100   male  25     yes
```

Hide

```
# along with toTitleCase() makes it better
data %>%
  mutate(., Sex = tools::toTitleCase(tolower(Sex))) %>%
  filter(., stringr::str_detect(tolower(Sex), "ma"))
```

```
##    Id    Sex Age Renting
## 1 100   Male  25     yes
## 2 101 Female  33    <NA>
## 3 102   Male  27     yes
## 4 104    Man  44      no
## 5 100   Male  25     yes
```

But notice the `male` and `man` issue is still a problem. Hard code a fix using `%in%` and `case_when()` …

Hide

```
data %>%
  mutate(.,
         Sex = case_when(
           tolower(Sex) %in% c("male", "man") ~ "Male",
           tolower(Sex) %in% c("female", "woman") ~ "Female"
         ))
```

```
##    Id    Sex Age Renting
## 1 100   Male  25     yes
## 2 101 Female  33    <NA>
## 3 102   Male  27     yes
## 4 103   <NA>  40    <NA>
## 5 104   Male  44      no
## 6 100   Male  25     yes
## 7 105   <NA>  40     yes
```

Or for a more flexible fix using `str_detect()` …

BUT beware that the order of operations matters here. Because the string “female” contains “ma”, you could accidentally recode all “female” cases to “male” if you perform the `case_when()` conversion on “male” first.

Hide

```
data %>%
  mutate(.,
         Sex = case_when(
           stringr::str_detect(tolower(Sex), "fe") ~ "Female",
           stringr::str_detect(tolower(Sex), "ma") ~ "Male",
         ))
```

```
##    Id    Sex Age Renting
## 1 100   Male  25     yes
## 2 101 Female  33    <NA>
## 3 102   Male  27     yes
## 4 103   <NA>  40    <NA>
## 5 104   Male  44      no
## 6 100   Male  25     yes
## 7 105   <NA>  40     yes
```

11 Filtering Missing Data (`NA`’s)

`is.na()` will return a logical vector for which `TRUE` represents there are missing values.

Try on the entire data frame…

Hide

```
is.na(data)
```

```
##         Id   Sex   Age Renting
## [1,] FALSE FALSE FALSE   FALSE
## [2,] FALSE FALSE FALSE    TRUE
## [3,] FALSE FALSE FALSE   FALSE
## [4,] FALSE  TRUE FALSE    TRUE
## [5,] FALSE FALSE FALSE   FALSE
## [6,] FALSE FALSE FALSE   FALSE
## [7,] FALSE FALSE FALSE   FALSE
```

You can see that some columns contain cases/rows with `TRUE` indicating the cell contains `NA` .

The negation operator, `!` , will be used to illustrate some filtering approaches. Because `filter()` will filter out `FALSE` cases and retain `TRUE` ones, so you may sometimes need to negate a function so that you keep the rows you want to keep.

- `na.omit()` : removes rows with NAs
- `dplyr::filter(., is.na(column_name))` : keep rows with NA in specific variable
- `dplyr::filter(., !is.na(column_name))` : remove rows with NA in specific variable
- `dplyr::filter(., complete.cases(.))` : remove rows

11.1 *Filter using `na.omit()` …*

Hide

```
data %>%
  na.omit(.) %>%     # omit any rows with NAs
  view(.)
```

Show 100 **entries**

Search:

	Id	Sex	Age	Renting
	All	All	All	All
1	100	male	25	yes
3	102	Male	27	yes
5	104	man	44	no
6	100	male	25	yes
7	105	neither	40	yes

Showing 1 to 5 of 5 entries Previous 1 Next

11.2 *Filter using* `is.na()` *and* `!is.na()` *...*

Hide

```
data %>%
  filter(., is.na(Sex)) %>%      # keep NAs by variable
  view(.)
```

Show 100 **entries** **Search:**

	Id	Sex	Age	Renting
	All	All	All	All
1	103		40	

Showing 1 to 1 of 1 entries Previous 1 Next

Hide

```
data %>%
  filter(., !is.na(Sex)) %>%      # remove NAs by variable
  view(.)
```

Show 100 **entries** **Search:**

	Id	Sex	Age	Renting

	All	All	All	All
1	100	male	25	yes
2	101	female	33	
3	102	Male	27	yes
4	104	man	44	no
5	100	male	25	yes
6	105	neither	40	yes

Showing 1 to 6 of 6 entries

Previous 1 Next

Hide

```
data %>%
  filter(., !is.na(Sex)) %>%
  filter(., !is.na(Renting)) %>%
  view(.)
```

Show **100** **entries**

Search:

	Id	Sex	Age	Renting
	All	All	All	All
1	100	male	25	yes
2	102	Male	27	yes
3	104	man	44	no
4	100	male	25	yes
5	105	neither	40	yes

Showing 1 to 5 of 5 entries

Previous 1 Next

Hide

```
# separate calls on separate lines can also make code inclusion/exclusion easy
data %>%
  #filter(., !is.na(Sex)) %>%
  filter(., !is.na(Renting)) %>%
  view(.)
```

Show 100 **entries** **Search:**

	Id	Sex	Age	Renting
	All	All	All	All
1	100	male	25	yes
2	102	Male	27	yes
3	104	man	44	no
4	100	male	25	yes
5	105	neither	40	yes

Showing 1 to 5 of 5 entries Previous 1 Next

11.3 *Filter using `complete.cases()` …*

The `complete.cases()` function returns a logical vector for which `TRUE` reflects the row has complete information and no missing cases. Using `complete.cases()` along with `filter()`, you would retain all rows `TRUE` rows.

Hide

```
data %>%
  dplyr::filter(., complete.cases(.))
```

```
##    Id     Sex Age Renting
## 1 100    male  25     yes
## 2 102    Male  27     yes
## 3 104     man  44      no
## 4 100    male  25     yes
## 5 105 neither  40     yes
```

12 **Summarizing Data Using `dplyr`**

If you want a quick summary of data, `summary()` will provide some basic information for you.

Hide

```
summary(data)
```

```
##        Id            Sex                 Age          Renting
##  Min.   :100.0   Length:7           Min.   :25.00   Length:7
##  1st Qu.:100.5   Class :character   1st Qu.:26.00   Class :character
##  Median :102.0   Mode  :character   Median :33.00   Mode  :character
##  Mean   :102.1                      Mean   :33.43
##  3rd Qu.:103.5                      3rd Qu.:40.00
##  Max.   :105.0                      Max.   :44.00
```

However, there are many other ways to summarize data. To introduce data summary techniques using `dplyr` , we will open the `diamonds` data set from the `ggplot2` library. Then, we will use `dplyr::summarise()` or `dplyr::summarize()` to summarize the data. The `summarise()` function works similar to `mutate()` insofar as variables are created but differs insofar as the data frame returned from `summarise()` contains only the variable(s) referenced in `summarize()` .

In the example below, we summarize by creating a new variable which is set to represent some data summary technique. In essence, summarizing is for descriptive statistics. Using `mean()` , we can summarize the data by taking the mean of the `price` variable.

Hide

```
diamonds <- ggplot2::diamonds   # assign data to object

diamonds %>%
  summarise(.,
            mean = mean(price, na.rm = T),
            )
```

```
## # A tibble: 1 × 1
##    mean
##   <dbl>
## 1 3933.
```

Notice what is returned is a single value reflecting the mean of all the data in the data frame. We could have obtained the same without using `dplyr` .

Hide

```
mean(diamonds$price, na.rm = T)        # $ notation
```

```
## [1] 3932.8
```

But we lose flexibility of easily adding new summary procedures.

Hide

```
diamonds %>%
  summarise(.,
            mean = mean(price, na.rm = T),
            sd   = sd(price, na.rm = T)
            )
```

```
## # A tibble: 1 × 2
##    mean    sd
##   <dbl> <dbl>
## 1 3933. 3989.
```

Now there is a mean and standard deviation for price. You can also add the sample size using `dplyr::n()` .

Hide

```
diamonds %>%
  summarise(.,
            mean = mean(price, na.rm = T),
            sd   = sd(price, na.rm = T),
            n    = n()
            )
```

```
## # A tibble: 1 × 3
##    mean    sd     n
##   <dbl> <dbl> <int>
## 1 3933. 3989. 53940
```

13 Summarizing `across()` Multiple Variables

Summarizing a single variable is useful but if you want to summarize by many, you likely don't want to code a new line for each variable. In such cases, you can use `across()` as a helper function as was used for creating new variables with `mutate()` (see previous lesson).

Remember `across()` will want you to pass the columns to summarize by, `.cols` , the function for how to summarize, `.fns` , and the names for how to name the new variables, `.names` (which will be `NULL` by default). The `.x` here stands for passing the vector to the mean function and not the data frame. More on `~` and `.x` later.

13.1 *Summarize across by numeric variables…*

Hide

```
diamonds %>%
  summarise(., across(.cols = where(is.numeric),
                      .fns  = ~mean(.x, na.rm = TRUE))
            )
```

```
## # A tibble: 1 × 7
##   carat depth table price     x     y     z
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 0.798  61.7  57.5 3933.  5.73  5.73  3.54
```

Passing `.names` …

Hide

```
diamonds %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = ~mean(.x, na.rm = TRUE),
                      .names = "{.col}_mean")
            )
```

```
## # A tibble: 1 × 3
##   carat_mean depth_mean table_mean
##        <dbl>      <dbl>      <dbl>
## 1      0.798       61.7       57.5
```

13.2 *Summarize across by variable name vector…*

Hide

```
diamonds %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = ~mean(.x, na.rm = TRUE),
                      .names = "{.col}_mean")
            )
```

```
## # A tibble: 1 × 3
##   carat_mean depth_mean table_mean
##        <dbl>      <dbl>      <dbl>
## 1      0.798       61.7       57.5
```

14 **Summarizing Data using `group_by()`**

A summary of the entire data set is fine for understanding grand mean and other metrics but this lacks information at a group level, for example, by sex, ethnicity, personality, city, job title, or in the `diamonds` data by diamont `cut` , `clarity` , or other variation.

When summarizing data, you will often want to summarize by levels of other variables, either categorical or numeric. In such cases, you can `group_by()` another variable and then summarize.

Let’s summarize in several ways after grouping by diamond `cut` .

Hide

```
diamonds %>%
  group_by(., cut) %>%
  summarise(.,
            n = n(),  # or also length()
            mean = mean(price, na.rm = T),
            sd   = sd(price, na.rm = T)
            )
```

```
## # A tibble: 5 × 4
##   cut           n  mean    sd
##   <ord>     <int> <dbl> <dbl>
## 1 Fair       1610 4359. 3560.
## 2 Good       4906 3929. 3682.
## 3 Very Good 12082 3982. 3936.
## 4 Premium   13791 4584. 4349.
## 5 Ideal     21551 3458. 3808.
```

You see that price increases with `cut` quality. Using `str()` or `glimpse()` you can see `cut` is an ordered factor.

Hide

```
diamonds %>% str()
```

```
## tibble [53,940 × 10] (S3: tbl_df/tbl/data.frame)
##  $ carat  : num [1:53940] 0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num [1:53940] 61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num [1:53940] 55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int [1:53940] 326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num [1:53940] 3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num [1:53940] 3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num [1:53940] 2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

But wow, that was a lot of coding lines. You might want to write your code to be more flexible so every variable is not hard coded. Could you imagine having 50 variables?

14.1 *Passing functions in a `list()`*

Doing this requires a little fancy coding by passing two functions as a `list` object. A `list` is a special object (e.g., container) for which its elements can be different types of objects. Whereas elements of `vectors` can be only character or only numeric, elements of lists can hold different object. One element can be a numeric vector, another element a data frame, another element a character vector, etc. Many functions used in R will actually return lists for which elements contain different types of objects.

Back to two or more functions. If you pass a `list()` with arguments for the `mean` and the `sd` , you can summarize by both. if you want to prevent errors (yes you do) and want to keep the summaries separate, you can modify `.names` to pass the column and the function, `"{.col}_{.fn}"` . The underscore is not needed; it only helps with readability of the variables. Let's pass the summary procedures as a `list` . Unfortunately, `n()` will throw an error but `length()` will also return the length of the vector, after of course, the grouping variable.

Here are a couple ways to do this. If you pass only the functions into the list, `.names` will take on the order of the functions such that mean would be `"_1"` . This is confusing.

Hide

```
diamonds %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean, sd, length),
                      .names = "{.col}_{.fn}")
            )
```

```
## # A tibble: 5 × 10
##   cut    carat_1 carat_2 carat_3 depth_1 depth_2 depth_3 table_1 table_2 table_3
##   <ord>    <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>
## 1 Fair     1.05    0.516    1610    64.0   3.64     1610    59.1    3.95    1610
## 2 Good     0.849   0.454    4906    62.4   2.17     4906    58.7    2.85    4906
## 3 Very …   0.806   0.459   12082    61.8   1.38    12082    58.0    2.12   12082
## 4 Premi…   0.892   0.515   13791    61.3   1.16    13791    58.7    1.48   13791
## 5 Ideal    0.703   0.433   21551    61.7   0.719   21551    56.0    1.25   21551
```

A better approach would be to assign the function a name in the `list()` function call so that the name is appended and the variable is named meaningfully.

Hide

```
diamonds %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean = mean,
                                   sd = sd,
                                   n = length
                                   ),
                      .names = "{.col}_{.fn}")
            )
```

```
## # A tibble: 5 × 10
##   cut    carat…¹ carat…² carat_n depth…³ depth…⁴ depth_n table…⁵ table…⁶ table_n
##   <ord>    <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>
## 1 Fair     1.05    0.516    1610    64.0   3.64     1610    59.1    3.95    1610
## 2 Good     0.849   0.454    4906    62.4   2.17     4906    58.7    2.85    4906
## 3 Very …   0.806   0.459   12082    61.8   1.38    12082    58.0    2.12   12082
## 4 Premi…   0.892   0.515   13791    61.3   1.16    13791    58.7    1.48   13791
## 5 Ideal    0.703   0.433   21551    61.7   0.719   21551    56.0    1.25   21551
## # … with abbreviated variable names ¹carat_mean, ²carat_sd, ³depth_mean,
## #   ⁴depth_sd, ⁵table_mean, ⁶table_sd
```

Importantly, however, if a vector contains even one `NA` , remember the returned statistic for the entire vector will also be `NA` . The previous code will *NOT* return correct statistics if you have an `NA` .

You will want to remove them either by filtering rows for `complete.cases()` or omitting `NA` s from the summary statistic function. To illustrate, let's make some data missing in the data frame. Because there are 53,940 rows in the data frame, we can simply make some data in the last row missing. Let's change the data in `depth` and `table` variables to `NA` just to illustrate the point.

Hide

```
diamonds[53940, c("depth", "table")]   # the current data
```

```
## # A tibble: 1 × 2
##   depth table
##   <dbl> <dbl>
## 1  62.2    55
```

Hide

```
diamonds[53940, c("depth", "table")] <- NA # set these cells to NA

tail(diamonds)               # see missing values
```

```
## # A tibble: 6 × 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.72 Premium   D     SI1      62.7    59  2757  5.69  5.73  3.58
## 2  0.72 Ideal     D     SI1      60.8    57  2757  5.75  5.76  3.5
## 3  0.72 Good      D     SI1      63.1    55  2757  5.69  5.75  3.61
## 4  0.7  Very Good D     SI1      62.8    60  2757  5.66  5.68  3.56
## 5  0.86 Premium   H     SI2      61      58  2757  6.15  6.12  3.74
## 6  0.75 Ideal     D     SI2      NA      NA  2757  5.83  5.87  3.64
```

Now filter out missing cases…

Hide

```
diamonds %>%
  filter(., complete.cases(.)) %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean = mean,
                                   sd = sd,
                                   n = length),
                      .names = "{.col}_{.fn}"
            ))
```

```
## # A tibble: 5 × 10
##   cut    carat…¹ carat…² carat_n depth…³ depth…⁴ depth_n table…⁵ table…⁶ table_n
##   <ord>    <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>
## 1 Fair     1.05    0.516    1610    64.0   3.64     1610    59.1    3.95    1610
## 2 Good     0.849   0.454    4906    62.4   2.17     4906    58.7    2.85    4906
## 3 Very …   0.806   0.459   12082    61.8   1.38    12082    58.0    2.12   12082
## 4 Premi…   0.892   0.515   13791    61.3   1.16    13791    58.7    1.48   13791
## 5 Ideal    0.703   0.433   21550    61.7   0.719   21550    56.0    1.25   21550
## # … with abbreviated variable names ¹carat_mean, ²carat_sd, ³depth_mean,
## #   ⁴depth_sd, ⁵table_mean, ⁶table_sd
```

That's likely the easiest approach but if there is an `NA` in some variables and not others (e.g., only `depth` and `table` here), the variables with complete cases will also lose data. Note how `n` is the same for all variables.

What you may really want to do would be to compute the statistics by omitting `NA` s at the variable level. Now, `across()` has a way to handle this which involves passing an additional argument `na.rm = TRUE` which will adjust the functions, in this case mean and sd functions to include `na.rm = TRUE` . We can add this after `.names` . Unfortunately, however, `length()` annoyingly has no argument for removing `NA` s. If we drop it out of the list, we get the means and standard deviations.

Hide

```
diamonds %>%
  filter(., complete.cases(.)) %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean = mean,
                                   sd = sd
                                   ),
                      .names = "{.col}_{.fn}",
                      na.rm = TRUE
            ))
```

```
## # A tibble: 5 × 7
##   cut       carat_mean carat_sd depth_mean depth_sd table_mean table_sd
##   <ord>          <dbl>    <dbl>      <dbl>    <dbl>      <dbl>    <dbl>
## 1 Fair           1.05     0.516       64.0    3.64        59.1     3.95
## 2 Good           0.849    0.454       62.4    2.17        58.7     2.85
## 3 Very Good      0.806    0.459       61.8    1.38        58.0     2.12
## 4 Premium        0.892    0.515       61.3    1.16        58.7     1.48
## 5 Ideal          0.703    0.433       61.7    0.719       56.0     1.25
```

So this doesn't solve the problem for all metrics you want to compute. What now?

Approach #1:

We can write our own function that contains `NA` removal in the same way. We just need to ensure we use `na.rm` and not something like `remove.na` . Let's name the function `length_na` .

Hide

```
length_na <- function(x, na.rm = FALSE) {
# as length() substitute that calculates length with and without NAs
  if (na.rm) {
    x = length(na.omit(x))
  } else {
    x = length(x)
  }
  return(x)
}
```

And add `length_na` to the list…

Hide

```
diamonds %>%
  filter(., complete.cases(.)) %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean = mean,
                                   sd = sd,
                                   n = length_na
                                   ),
                      .names = "{.col}_{.fn}",
                      na.rm = TRUE
            ))
```

```
## # A tibble: 5 × 10
##   cut    carat…¹ carat…² carat_n depth…³ depth…⁴ depth_n table…⁵ table…⁶ table_n
##   <ord>    <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>
## 1 Fair     1.05    0.516    1610    64.0   3.64     1610    59.1    3.95    1610
## 2 Good     0.849   0.454    4906    62.4   2.17     4906    58.7    2.85    4906
## 3 Very …   0.806   0.459   12082    61.8   1.38    12082    58.0    2.12   12082
## 4 Premi…   0.892   0.515   13791    61.3   1.16    13791    58.7    1.48   13791
## 5 Ideal    0.703   0.433   21550    61.7   0.719   21550    56.0    1.25   21550
## # … with abbreviated variable names ¹carat_mean, ²carat_sd, ³depth_mean,
## #   ⁴depth_sd, ⁵table_mean, ⁶table_sd
```

But this removed `NA` row-wise and not for each column variable independently. The `carat` variable should not have the name number of cases as `depth` and `table`, both of which contain missing cases. This approach just did the same thing as filtering by complete cases as done earlier with `filter(., complete.cases(.))`.

Approach #2:

When functions do not contain argument for dealing with `NA` s, there is `na.omit()`, a function that takes an object and removes `NA` s. So you can just pass the variable to `na.omit()` and then wrap it in the metric function of interest. Also, because `na.rm = T` cannot be used for `length()`, `na.omit()` offers consistency across all functions and as a result, I believe, less confusion.

Unfortunately, accomplishing this task can be rather tricky and requires some new syntax. This requires usage of what's called a "lamba" technique. Using this type of syntax, we can pass functions to the `.fns` argument. The `?across()` documentation calls it "a purrr-style lambda" in the arguments section (for clarity, `purrr` is a library). This approach can be a little bit confusing, so I'm going to show you an example, and then walk through it step by step.

We need to precede the function with `~` and reference the vector using `.x`. Let's do this and change the `.fns` argument slightly.

General Example:

```
name = ~function(na.omit(.x))
```

Hide

```
diamonds %>%
  group_by(., cut) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                      .fns  = list(mean = ~mean(na.omit(.x)),
                                   sd = ~sd(na.omit(.x)),
                                   n = ~length(na.omit(.x))),
                      .names = "{.col}_{.fn}"
                      )
            )
```

```
## # A tibble: 5 × 10
##   cut    carat…¹ carat…² carat_n depth…³ depth…⁴ depth_n table…⁵ table…⁶ table_n
##   <ord>    <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>   <dbl>   <dbl>   <int>
## 1 Fair     1.05    0.516    1610    64.0   3.64     1610    59.1    3.95    1610
## 2 Good     0.849   0.454    4906    62.4   2.17     4906    58.7    2.85    4906
## 3 Very …   0.806   0.459   12082    61.8   1.38    12082    58.0    2.12   12082
## 4 Premi…   0.892   0.515   13791    61.3   1.16    13791    58.7    1.48   13791
## 5 Ideal    0.703   0.433   21551    61.7   0.719   21550    56.0    1.25   21550
## # … with abbreviated variable names ¹carat_mean, ²carat_sd, ³depth_mean,
## #   ⁴depth_sd, ⁵table_mean, ⁶table_sd
```

Great! Now `carat` contains one more case than the other variables. So what's the point of all of this? Well, you need to be careful not to apply functions and assume they are doing what you believe you are doing. You always need to be smarter than the code you use. Also, there is no single answer for dealing with data. Sometimes one approach will be appropriate and in other instances another approch will be. You as the data scientist need to know that there are different methods so that you an decide where to apply those different methods.

14.2 *Grouping by multiple variables`*

OK, now this gets exciting. When you want to group by additional variables, pass a new one to `group_by()`. Just to keep the output more simple, let's remove one summary function.

Hide

```
diamonds %>%
  group_by(., cut, clarity) %>%
  summarise(., across(.cols = c("carat", "depth", "table"),
                    .fns  = list(mean = ~mean(na.omit(.x)),
#                                 sd = ~sd(na.omit(.x)),
                                 n = ~length(na.omit(.x))),
                    .names = "{.col}_{.fn}"
                    )
            )
```

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

```
## # A tibble: 40 × 8
## # Groups:   cut [5]
##    cut   clarity carat_mean carat_n depth_mean depth_n table_mean table_n
##    <ord> <ord>        <dbl>   <int>      <dbl>   <int>      <dbl>   <int>
##  1 Fair  I1           1.36      210       65.7     210       58.1     210
##  2 Fair  SI2          1.20      466       64.4     466       58.8     466
##  3 Fair  SI1          0.965     408       63.9     408       59.1     408
##  4 Fair  VS2          0.885     261       63.6     261       59.1     261
##  5 Fair  VS1          0.880     170       62.9     170       60.4     170
##  6 Fair  VVS2         0.692      69       62.8      69       59.2      69
##  7 Fair  VVS1         0.665      17       60.4      17       61.2      17
##  8 Fair  IF           0.474       9       60.1       9       59.1       9
##  9 Good  I1           1.20       96       62.1      96       59.5      96
## 10 Good  SI2          1.04     1081       62.2    1081       58.9    1081
## # … with 30 more rows
```

15 **The Data Manipulation Workflow: Putting It All Together**

Of course, all of this can be paired with `select()` , `mutate()` , `filter()` , etc. Here is the data manipulation workflow

```
dataframe %>%  select(., ...) %>%     # select variables of interest
mutate(., ...) %>%     # then create new variables  filter(., ...) %>%     # then filter by rows
group_by(., ...) %>%   # then group for subsetting  summarize(., ...)      # then summarize
```