Cleaning the Data and Documenting the Process Using R Markdown

Introduction to R Programming

District Data Labs

The Data "Pipeline"

- Document Your Data Cleaning Process
- Tidy Data

Data cleaning methods

- Opening data files in R and viewing them
- Working with data frames
- Dropping variables
- Renaming variables
- Logical operators
- Dropping observations
- Reshaping
  - Reshaping from wide to long
  - Reshaping from long to wide
- Creating new variables

- Replacing existing variables
- Managing string/character variables
- Merging
  - Binding columns or rows
  - Matching on ID variables
  - Checking the merge
- Sorting rows

Additional data management techniques

- Managing the workspace
  - Deleting objects
  - Saving/loading the workspace
- Opening ASCII data files

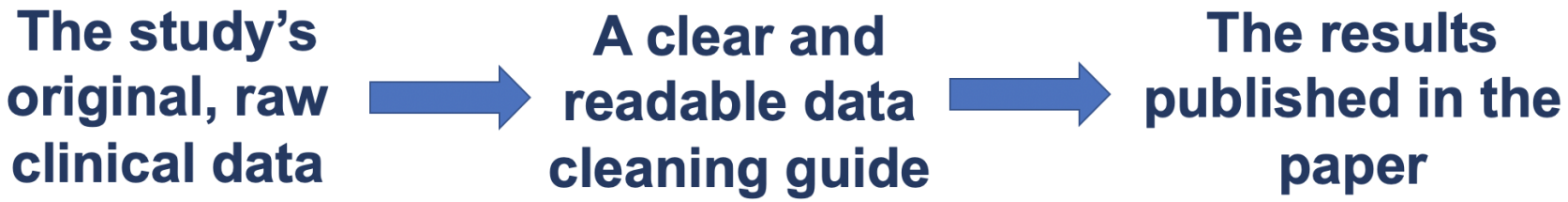
- Saving ASCII data files
- Arranging columns
- Missing data
- Managing categorical variables
  - Recoding/labeling values
  - Combining categories
  - Reordering categories
  - Combining categories across multiple variables
- More methods for managing string/character variables
- Managing date/time variables
- Piping
- Collapsing and within-group calculations
  - Using `group_by()`
  - Using `count()`

Document Your Data Cleaning Process

Remember the work by Baggerly and Coombes. They had to spend months trying to replicate a study in a "forensic" way because the authors did not document their data cleaning steps.

They had to deal with this:

Instead, we will learn how to do this:



Document Your Data Cleaning Process

Real-world data are always messy: that is, original data are never immediately ready for analysis. Data must be **cleaned** in order to be useful.

But small mistakes in the data cleaning stage can have profound effects for the analysis. If your data are corrupted, no amount of statistical-machine-learning-artificial-intelligence-magic will save the analysis.

This afternoon, we will review the **tidyverse** packages, and all the methods we have for cleaning data. We will also introduce a couple new methods.

We will work through how to do this work using R markdown, so that we document our steps **as we go along**. We can catch errors this way, and make them easier to fix.

What is Tidy Data?

"*Happy families are all alike; every unhappy family is unhappy in its own way.*" — Leo Tolstoy

"*Like families, tidy datasets are all alike but every messy dataset is messy in its own way.*" — Hadley Wickham

Data can come in many shapes and formats. The data can be assembled in one place or might be in many different places.

Tidy data is a philosophy about the best way to code a dataset to make data **easy to analyze** in a reproducible way.

What is Tidy Data?

A **tidy dataset** meets the following minimum standards:

1. The dataset exists as one table.

2. It's square: every row has an entry for every column (even if it is missing).

3. The rows represent observations.

4. The columns represent variables.

5. The observations are comparable units (for example: people, countries, but not a mix of the two)

What is Tidy Data?

Even better tidy data also:

6. Sorts the rows and columns into a logical order
7. Has descriptive variable names
8. Creates new variables to convey the important information
9. Deletes irrelevant variables (and sometime observations)
10. Reads date variables as dates
11. Uses consistent codes for missing values

These are all things to do with the edited, working data. **Never overwrite the original data file.** We may need to return to the original data many times.

Opening data files in R and viewing them

The most common type of data file is CSV (comma separated values). CSVs are space efficient, portable, and can be transferred to any data/statistics software.

To load a CSV file called `data.csv`, load the `tidyverse` package by typing

```
library(tidyverse)
```

Then type

```
data <- read_csv("data.csv")
```

Then to look at the data spreadsheet type `View(data)` in the **console**. (Please note the capital V.)

Looking at the data is the most useful way to understand the challenges ahead of you for preparing the data for analysis.

Working with data frames

All of the following commands start by assigning the data management step to a data frame.

If you assign the step to a new name, you create a **new object**. The step is applied to the NEW data frame, but the step is NOT applied to the original data frame object.

Example: If the data frame is called `data`,

- `data <- command` overwrites the `data` object,
- `data2 <- command` creates another object, `data2`, in which the command was applied. It leaves the `data` object alone.

In general: assign to the same object over and over, UNLESS you don't want the step to be permanent.

Dropping variables

Sometimes datasets have **hundreds or thousands** of variables. Only a small portion of them might be relevant to your work. You may have to curate the data by deleting variables.

Use the `select()` command to keep only the variables you specify. To keep only the `country_name` and `year` variables, type

```
vdem <- select(vdem, country_name, year)
```

You can keep `country_name`, `year`, and **every variable that starts with "v2x"** by typing

```
vdem <- select(vdem, country_name, year,
     starts_with("v2x"))
```

Dropping variables

You can keep `country_name`, `year`, and **every variable that ends with "ocracy"** by typing

```
vdem <- select(vdem, country_name, year,
                     ends_with("ocracy"))
```

You can also specify which variables to drop instead of keep by using a minus sign:

```
vdem <- select(vdem, -ends_with("error"))
```

Renaming variables

Tidy data should have descriptive variable names so you actually know what each variable is. It's crazy how often the data collectors use dumb names like XAB14G7B!

To rename the variable `v2x_polyarchy` to `democracy`, type:

```
vdem <- rename(vdem, democracy=v2x_polyarchy)
```

Note that the new name comes first, and the old name comes second.

You can rename many variables with one command. Just separate these statements with commas:

```
vdem <- rename(vdem, democracy=v2x_polyarchy,
                     corrupt=v2x_corr,
                     civil_lib=v2x_civlib)
```

Logical operators

A logical statement is one that can only take on two values: true or false. Logical operators are used to build logical statements. These operators are

- == is equal to?
- > is greater than?
- >= is greater than or equal to?
- < is less than?
- <= is less than or equal to?
- `%in%` is an element of the set?

A **!** sign in front of >, >=, <, or <= means "not greater than", "not greater than or equal to", etc.

Also, != means "not equal to", and `!(3 %in% c(4,5,6))` means "3 is not an element of the set 4, 5, 6"

Logical operators

Two symbols are used to combine logical operators:

- **&** "and"
- | "or" (shift + the button just above enter)

Parentheses work here too to designate an order of evaluation.

Examples:

1==1 true	2==3 false
(1==1)|(2==3) true	(1==1)&(2==3) false
(1!=1)|(2!=3) true	!((1==1)&(2==3)) true
(4*3-2) %in% c(10, 11, 12) true	!(4*3-2) %in% c(10, 11, 12) false

If variables are used in a logical statement, R puts the value `TRUE` on the rows for which the statement is true, and `FALSE` on the rows for which the statement is false.

If you change the class of a logical variable to numeric, the variable will be binary (0,1).

Dropping observations

The `filter()` command works just like `select()`, except it keeps observations instead of variables, and you use **logical statements** to identify rows instead of variable names.

To keep only the rows corresponding to the year 2014, type

```
vdem <- filter(vdem, year==2014)
```

To keep only the rows in 2014 when the `democracy` variable is greater than .5, type

```
vdem <- filter(vdem, year==2014 & democracy > .5)
```

To keep only the rows for 1997, 2003, and 2011, type

```
vdem <- filter(vdem, year %in% c(1997, 2003, 2011))
```

Reshaping

Reshaping is a way to transform a dataset by turning columns into rows, or rows into columns.

There are two kinds of reshapes:

1. `gather()`: turning columns into rows. (In Stata, this is called a **reshape long**.)
2. `spread()`: turning rows into columns. (In Stata, this is called a **reshape wide**.)

Remember: **tidy data** requires that rows are *observations* and columns are *variables*. Years are observations, not variables.

Reshaping is tricky. But any command can be broken down to its simple logical elements. We just need to examine the rules on which the `gather()` and `spread()` commands operate.

Reshaping

Here's an untidy dataset because observations are in the columns:

	country	1999	2000
1	Afghanistan	745	2666
2	Brazil	37737	80488
3	China	212258	213766

The `gather()` command brings the observations to the rows. This command has the following syntax:

```
gather(data, ... , key, value)
```

Let's clearly define what each argument does.

Reshaping

```
gather(data, ... , key, value)
```

`data` is the name of the data frame you want to reshape.

... represents the names of the variables that you will move to the rows.

- You can type all the **raw variable names** here, separated by commas
- If the variable names are numeric (as with years), place forward-slanted single quotes around the names
- You can type something like `var1:var10` to refer to all variables in between between `var1` and `var10` in the data frame.

Reshaping

```
gather(data, ... , key, value)
```

`key` is the name of the variable that will contain the names of the old variables you are moving. This variable doesn't yet exist. If the column names are numeric years, you should type `key="year"`.

`value` is the name of the variable that will contain the data inside the old variables you are moving. This variable also doesn't yet exist.

Reshaping

This data is an object named `table4a` in R:

	country	1999	2000
1	Afghanistan	745	2666
2	Brazil	37737	80488
3	China	212258	213766

The data inside the columns represent a variable named cases.

Reshaping

To reshape this data, we type:

```
gather(table4a, `1999`, `2000`, key="year", value="cases")
```

The data now looks like

country	year	cases
Afghanistan	1999	745
Brazil	1999	37737
China	1999	212258
Afghanistan	2000	2666
Brazil	2000	80488
China	2000	213766

Reshaping

Here's an untidy dataset because variables are in the rows:

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Reshaping

To clean this dataset, we use the `spread()` command, which works similarly (but not exactly like) the `gather()` command:

```
spread(data, key, value)
```

The notable difference here is the lack of an ... argument. We don't have to specify columns, because the variables are already in the rows, and **R knows what to place in the columns** automatically by looking at the levels of the factor (named `type` in this data).

`data` is the name of the data frame you want to reshape.

Reshaping

```
spread(data, key, value)
```

`key` is the name of the existing variable that contains the names of the variables you will move to the columns.

`value` is the name of the existing variable that contains the data inside the variables you will move to the columns.

To reshape the data (named `table2` in R), we type:

```
spread(table2, key="type", value="count")
```

Reshaping

The data now looks like this:

	country	year	cases	population
1	Afghanistan	1999	745	19987071
2	Afghanistan	2000	2666	20595360
3	Brazil	1999	37737	172006362
4	Brazil	2000	80488	174504898
5	China	1999	212258	1272915272
6	China	2000	213766	1280428583

Creating new variables

Use the `mutate()` command to create new variables.

Write the new variable name, one equal sign, then what the new variable equals. You can use math and logic operators and the names of existing variables. For example:

```
vdem <- mutate(vdem,
                    after2001 = (year>=2001),
                    adjusted_dem = democracy - corruption,
                    avg = (democracy + civil_rights)/2)
```

If you use **logical operators**, the new variable will be BINARY to express whether the logical statement is TRUE or FALSE.

Replacing existing variables

You can also use the `mutate()` command to replace old variables. The difference is that now you write old variable names instead of new variables.

In this example, the variables `democracy` and `corruption` **already exist**:

```
vdem <- mutate(vdem,
               democracy = 100*democracy,
               adjusted_dem = democracy - corruption)
```

The first transformation says "replace each value of `democracy` with 100 times its old value."

The second transformation says "create `adjusted_dem` to be the difference of `democracy` and `corruption`."

But there's a problem here. See it?

Replacing existing variables

```
vdem <- mutate(vdem,
               democracy = 100*democracy,
               adjusted_dem = democracy - corruption)
```

These transformations are done one at a time and in order.

In the second transformation, our intention was to use the **old version** of `democracy`. Instead it uses the **new version** ($\times 100$).

This command instead does what we want:

```
vdem <- mutate(vdem,
               adjusted_dem = democracy - corruption
               democracy = 100*democracy)
```

So pay attention to the order in which you create/replace variables.

Managing string/character variables

Some variables are coded as character (words), but in reality, they are categorical variables. We've covered how to deal with variables like that.

But other variables are character because they really are **open-ended responses**.

Open-ended questions contain a LOT of data, some of it novel and unexpected, since these responses are not constrained to follow any multiple choices.

But how to extract this data?

Managing string/character variables

The following commands are part of the `stringr` library, which is part of the `tidyverse` packages.

Here's an example of a string:

```
response <- "I have the faintest idea.
                I really don't watch the news"
```

String is just another word for character, text, words, etc. It has the character class in R:

```
> class(response)
[1] "character"
```

Managing string/character variables

The length of a string is the number of characters in the string, including spaces:

```
> str_length(response)
[1] 55
```

To pull out a subsection of this string, use the `str_sub()` command. Specify the string variable to work with, the NUMBER of the character that starts the substring, and the NUMBER of the last character of the substring:

```
> str_sub(response, 1, 10)
[1] "I have the"
> str_sub(response, 25, 37)
[1] ". I really do"
```

Managing string/character variables

Negative numbers with the `str_sub()` command tell R to count **backwards**. To get a substring starting 15 characters from the end, until the end, type:

```
> str_sub(response, -15, -1)
[1] " watch the news"
```

It might be useful to convert all letters to UPPERCASE, so we don't have to worry about case sensitivity when looking for specific patterns and words:

```
> str_to_upper(response)
[1] "I HAVE THE FAINTEST IDEA.  I REALLY DON'T WATCH
       THE NEWS"
```

Merging

Merging is the technique of combining two data frames. There are a few different ways to perform a merge.

(1) Adding columns, without trying to match observations.

The `bind_cols()` (or the older `cbind()`) function pastes two data frames together, side by side, exactly *as they are*. So if you sort one or both data frames, it changes the result of the `bind_cols()` command.

For example, consider `data1` and `data2`:

country	x	y
USA	-0.10	-0.90
Canada	0.18	-0.88
UK	0.50	0.92

country	x	y
China	-0.61	-1.23
Japan	-0.24	-1.05
S. Korea	-0.11	-1.10

Merging

Using `bind_cols()` to combine these data frames has the following result:

```
cbind(data1,data2)
```

country	x	y	country1	x1	y1
USA	-0.10	-0.90	China	-0.61	-1.23
Canada	0.18	-0.88	Japan	-0.24	-1.05
UK	0.50	0.92	S. Korea	-0.11	-1.10

To avoid duplicate column names, R placed an arbitrary "1" after the variable names for `data2`.

R pasted the two data frames together without any regard for the data's structure or meaning. In general, this is NOT what we want to do.

Merging

(2) Adding rows (in Stata this is called "appending")

The `bind_rows()` (or the older `rbind()`) function is similar to `bind_cols()`, but with two important differences:

1. `bind_rows()` pastes data frames together by pasting one on top of the other, resulting in more rows but not more columns
2. `bind_rows()` rearranges the columns of each data frame to try to match corresponding variables.

This is a good option to group observations that belong together, but are **stored in separate places** for whatever reason.

Merging

Using `bind_rows()` to combine these data frames has the following result:

```
rbind(data1,data2)
```

	country	x	y
1	USA	-0.10	-0.90
2	Canada	0.18	-0.88
3	UK	0.50	0.92
4	China	-0.61	-1.23
5	Japan	-0.24	-1.05
6	S. Korea	-0.11	-1.10

Merging

(3) Adding columns, matching observations that share the same ID variable(s).

This is the most important kind of merge. It's also the trickiest to do correctly.

Suppose you have two data frames that look like this:

country	x	y
USA	0.97	0.76
China	-0.96	-2.12
Russia	1.10	0.24

country	w	z
China	0.99	0.33
USA	0.37	-0.64
Russia	0.10	-0.49

How do you combine the two data frames?

Merging

You can sort both data frames alphabetically by country, then use `cbind()`. But while that works here, that approach won't work in general.

We need a way for R to recognize that **country is the ID variable**, and that the two data frames share IDs. Then R needs to match rows that share the same ID.

The most important merge command is `full_join()`. Applying this command gives the following result:

```
full_join(data1, data2)
```

country	x	y	w	z
USA	0.97	0.76	0.37	-0.64
China	-0.96	-2.12	0.99	0.33
Russia	1.10	0.24	0.10	-0.49

Merging

Here's what `full_join()` does:

1. It looks at the **variable names** for each dataset, and determines whether there is any overlap in these names.

2. It assumes that the shared variable names are the ID variables you want to use to match the observations in each data frame.

3. It combines observations with the same IDs.

`full_join()` is very useful and easy to use compared to merge commands in other packages and software.

But it tends to **mess up without displaying any warnings or errors**! You have to understand how `full_join()` works, and you have to visualize what you want the data to look like after using `full_join()`.

Merging

If an observation has no match in the other data frame, it still appears in the merged data **with `NA` values** for the variables from the other data frame.

There are related commands that treat **unmatched observations** differently:

- `inner_join()`: drops all unmatched observations
- `left_join(data1, data2)`: keeps all observations from `data1`, but drops all unmatched observations in `data2`
- `right_join(data1, data2)`: keeps all observations from `data2`, but drops all unmatched observations in `data1`

In general, it is safer to stick with `full_join()`. These other commands might delete observations you don't want to delete.

Merging

Suppose for example we want to merge these two data frames:

country	x	y
USA	0.43	-0.57
China	-0.70	-1.38
Russia	1.64	0.60
France	-0.38	-0.30

country	w	z
USA	0.99	-1.27
China	1.06	0.13
Russia	1.65	0.72
Japan	0.28	-0.09

Note that some, but not all, of the ID values have a match in the other data frame.

Merging

`full_join()` keeps all observations that appear in either data frame, but creates missing values for unmatched observations:

```
data3 <- full_join(data1, data2)
```

country	x	y	w	z
USA	0.43	-0.57	0.99	-1.27
China	-0.70	-1.38	1.06	0.13
Russia	1.64	0.60	1.65	0.72
France	-0.38	-0.30	NA	NA
Japan	NA	NA	0.28	-0.09

Merging

`inner_join()` keeps the observations that have a match in both observations, and deletes all other observations:

```
data3 <- inner_join(data1, data2)
```

country	x	y	w	z
USA	0.43	-0.57	0.99	-1.27
China	-0.70	-1.38	1.06	0.13
Russia	1.64	0.60	1.65	0.72

Merging

`left_join()` keeps all observations from `data1` (the data frame typed FIRST), keeps all observations from `data2` (the data frame typed SECOND) that have a match in `data1`, and deletes observations from `data2` without a match in `data1`:

```
data3 <- left_join(data1, data2)
```

country	x	y	w	z
USA	0.43	-0.57	0.99	-1.27
China	-0.70	-1.38	1.06	0.13
Russia	1.64	0.60	1.65	0.72
France	-0.38	-0.30	NA	NA

Merging

`right_join()` keeps all observations from `data2` (the data frame typed SECOND), keeps all observations from `data1` (the data frame typed FIRST) that have a match in `data2`, and deletes observations from `data1` without a match in `data2`:

```
data3 <- left_join(data1, data2)
```

country	x	y	w	z
USA	0.43	-0.57	0.99	-1.27
China	-0.70	-1.38	1.06	0.13
Russia	1.64	0.60	1.65	0.72
Japan	NA	NA	0.28	-0.09

Merging

Sometimes there is more than one ID variable that identifies an observation. For example, we might have data on

- U.S States *within* years
- Political parties *within* countries *within* years
- Students *within* classes *within* schools *within* school districts *within* states

A set of ID variables are called **unique identifiers** of the rows if no two rows share the same values of every ID variable. Two rows might represent China at different years. So country alone is not a unique identifier, but country and year TOGETHER are.

The `full_join()` command can match on multiple ID variables, as long as corresponding ID variables have the **same name in each data frame**.

Merging

Suppose we want to merge the following two data frames:

country	year	x	y
USA	2014	0.53	0.26
USA	2015	-1.12	-1.68
USA	2016	0.75	-1.24
China	2014	-0.01	0.17
China	2015	-0.09	0.14
China	2016	-0.32	0.07
Russia	2014	1.49	0.83
Russia	2015	-0.96	1.04
Russia	2016	2.50	-0.43

country	year	w	z
USA	2014	0.27	-1.95
USA	2015	0.88	0.14
USA	2016	-0.06	0.49
China	2014	-0.25	-1.02
China	2015	0.72	0.22
China	2016	0.89	1.20
Russia	2014	0.59	0.45
Russia	2015	0.06	-1.57
Russia	2016	-0.93	-0.01

There are two ID variables, country and year. Because the have the same name in each data frame, we can use `full_join()` like we did before.

Merging

We use the following command:

```
data3 <- full_join(data1, data2)
```

country	year	x	y	w	z
USA	2014	0.53	0.26	0.27	-1.95
USA	2015	-1.12	-1.68	0.88	0.14
USA	2016	0.75	-1.24	-0.06	0.49
China	2014	-0.01	0.17	-0.25	-1.02
China	2015	-0.09	0.14	0.72	0.22
China	2016	-0.32	0.07	0.89	1.20
Russia	2014	1.49	0.83	0.59	0.45
Russia	2015	-0.96	1.04	0.06	-1.57
Russia	2016	2.50	-0.43	-0.93	-0.01

Merging

Sometimes one data frame has more unique identifying variables than the other. For example:

country	year	x	y
USA	2014	0.53	0.26
USA	2015	-1.12	-1.68
USA	2016	0.75	-1.24
China	2014	-0.01	0.17
China	2015	-0.09	0.14
China	2016	-0.32	0.07
Russia	2014	1.49	0.83
Russia	2015	-0.96	1.04
Russia	2016	2.50	-0.43

country	w	z
USA	0.99	-1.27
China	1.06	0.13
Russia	1.65	0.72

Merging

`full_join()` handles this situation too. It recognizes the shared ID of country, and places the data for a country in `data2` on every row for that country:

```
data3 <- full_join(data1, data2)
```

country	year	x	y	w	z
USA	2014	0.53	0.26	0.99	-1.27
USA	2015	-1.12	-1.68	0.99	-1.27
USA	2016	0.75	-1.24	0.99	-1.27
China	2014	-0.01	0.17	1.06	0.13
China	2015	-0.09	0.14	1.06	0.13
China	2016	-0.32	0.07	1.06	0.13
Russia	2014	1.49	0.83	1.65	0.72
Russia	2015	-0.96	1.04	1.65	0.72
Russia	2016	2.50	-0.43	1.65	0.72

What can go wrong while merging

`full_join()` doesn't always know if something went wrong. So there's no automated procedure to check errors. But there are common problems you can check for before merging the data.

Prior to merging, we will perform three checks:

1. **ID name check**: do the shared ID variables have the same name? Are these variables the ONLY ones that share the same name in both data frames?
2. **Unique ID check**: do we expect the ID variables to be unique identifiers in one or both data frames? If so, are they?
3. **ID value check**: are there discrepancies in the values of the ID variable?

If all three checks pass, then the merge will work without problems.

What can go wrong while merging

ID name check: First, look at each data frame with the `View()`, `summary()`, and `head()` commands. Decide on what the unique ID variables are in each data frame, and which you will use to match in the merge.

Use the `names()` command to display the names for each data frame, and use the `intersect()` command to see which variable names are shared by both data frames:

```
> names(data1)
[1] "country" "year"    "x"         "y"
> names(data2)
[1] "country" "year"    "w"         "z"
> intersect(names(data1), names(data2))
[1] "country" "year"
```

There are **two problems** this check can reveal.

What can go wrong while merging

Problem 1: the ID variables don't have the same name in each data frame. In this case R won't know which variable to match on.

That can happen if one data frame has "state" while the other has "State" (case sensitive!). Or "year" vs. "yr", "countrycode" vs. "ccode", etc.

Solution: **prior to merging**, rename the ID variable in one of the two data frames, so that they have the same name.

Problem 2: non-ID variables in each data frame unexpectedly have the same name. In this case, R will mistakenly think this variable is an ID.

Solution: **prior to merging**, rename the non-ID variable in one of the two data frames, so that they have they DON'T same name anymore.

What can go wrong while merging

Unique ID check: If the ID variables are not unique identifiers in either data frame, then R places *ALL combinations of matching observations* in the merged data.

Here's a simple example:

name	x
A	2
A	7
A	1

name	y
A	6
A	5
A	8

These two data frames share an ID variable named "name". But name is NOT a unique ID in either data frame since multiple rows have the same value of "A".

There's no second ID variable like year that we can use to identify the rows.

What can go wrong while merging

If we just go ahead and merge, the result is

```
data3 <- full_join(data1, data2)
```

name	x	y
A	2	6
A	2	5
A	2	8
A	7	6
A	7	5
A	7	8
A	1	6
A	1	5
A	1	8

What can go wrong while merging

R didn't have enough information to match each row to a single row in the other data frame, so it matched each row to every possible row. **That's not what we want.**

To check whether this is a problem:

1. Create a second data frame with just the ID variables
2. Use the `unique()` command to keep only the non-repeated rows
3. Use `nrow()` to compare the number of rows of the unique ID data frame to the number of rows in the original data. If these numbers are the same, then the ID variables are unique.

What can go wrong while merging

For example, to check whether `country` and `year` are unique IDs in this data frame:

country	year	x	y
USA	2014	0.53	0.26
USA	2015	-1.12	-1.68
USA	2016	0.75	-1.24
China	2014	-0.01	0.17
China	2015	-0.09	0.14
China	2016	-0.32	0.07
Russia	2014	1.49	0.83
Russia	2015	-0.96	1.04
Russia	2016	2.50	-0.43

What can go wrong while merging

(1) Create a second data frame with just the ID variables

```
data.temp <- select(data1, country, year)
```

(2) Use the `unique()` command to keep only the non-repeated rows

```
data.temp <- unique(data.temp)
```

(3) Use `nrow()` to compare the number of rows of the unique ID data frame to the number of rows in the original data. If these numbers are the same, then the ID variables are unique.

```
> nrow(data.temp)
[1] 9
> nrow(data1)
[1] 9
```

Because the numbers of rows are equal, country and year are unique IDs.

What can go wrong while merging

IDs don't always have to be unique. In this example:

country	year	x	y
USA	2014	0.53	0.26
USA	2015	-1.12	-1.68
USA	2016	0.75	-1.24
China	2014	-0.01	0.17
China	2015	-0.09	0.14
China	2016	-0.32	0.07
Russia	2014	1.49	0.83
Russia	2015	-0.96	1.04
Russia	2016	2.50	-0.43

country	w	z
USA	0.99	-1.27
China	1.06	0.13
Russia	1.65	0.72

We merge on `country`. We didn't expect `country` to be unique in the first data frame, just the second. That's okay! Make sure the results of this test match your expectations.

What can go wrong while merging

ID value check: It's common for two different datasets to code the same observations with slightly different IDs. Some examples:

- "District of Columbia" vs. "DC"
- "South Korea" vs. "S. Korea" vs. "ROK"
- 1998 vs 98

If you don't catch these discrepancies, **the data won't get matched for this observation**.

Also, two datasets might not cover the exact same cases. Or they might use different time frames. You have to decide whether or not that's okay.

Sometimes data contain thousands of unique IDs. That's too many to read through manually. We need a reliable way to identify the unmatched IDs.

What can go wrong while merging

If we merge two data frames, `data1` and `data2`, there are **two ways** for an ID value to be unmatched:

- An ID appears in `data1` but not `data2`, or
- An ID appears in `data2` but not `data1`

We can look at each type of unmatched ID separately.

The `anti_join()` command is a kind of "reverse merging". It drops the matched observations and leaves the unmatched ones. Specifically:

- `anti_join(data1, data2)` keeps only **observations in** `data1` **that have no match in** `data2`
- `anti_join(data2, data1)` keeps only **observations in** `data2` **that have no match in** `data1`

What can go wrong while merging

To identify the unmatched observations, type:

```
check1 <- anti_join(data1, data2, by=c("id1", "id2"))
check2 <- anti_join(data2, data1, by=c("id1", "id2"))
```

Unlike `full_join()`, these commands require the `by` argument, which takes a character vector with the names of the ID variables to match on. (If there's only one ID, type its name in quotes without the `c()` command.)

Now **check1** has all the observations in `data1` that have no match in `data2`, and **check2** has all the observations in `data2` that have no match in `data1`.

You can use `View()` to see these observations, and `nrow()` to count the number of observations in `check1` and `check2`. If both have 0 observations, then **every observation was matched**.

Sorting rows

Sorting means rearranging the rows/columns in a way that does not break any row or column apart.

You can sort rows based on the values of one variable **numerically** (from smallest to largest, or from largest to smallest) or **alphabetically**.

You can move the columns to appear in any order you like.

These steps are mostly cosmetic (they won't change statistical results) but they make the data much easier to look at.

Sorting rows

To sort observations, use the `arrange()` function.

Example: The state legislature data lists estimates of left/right ideologies for U.S. state legislatures from 1993-2014. To sort the rows from the most liberal (smallest value) to the most conservative (largest value) state house, type

```
stateleg <- arrange(stateleg, hou_chamber)
```

To sort from most conservative (largest value) to most liberal (smallest value), use a minus sign in front of the sorting variable:

```
stateleg <- arrange(stateleg, -hou_chamber)
```

Sometimes I get an error when I use the minus sign. If that happens, this should work instead:

```
stateleg <- arrange(stateleg, desc(hou_chamber))
```

Sorting rows

You can specify more than one sorting variable:

1. The data are sorted by the **first** variable.
2. If there are ties, they are broken by sorting the **second** variable within values of the first.
3. The **third** variable breaks ties with the first two variables, and so on.

To sort alphabetically by state name, then sort the observations from the same state by year, type

```
stateleg <- arrange(stateleg, st, year)
```

Deleting Objects

The memory that R sets aside for all the objects you load and create is called the workspace.

To see all of the objects that currently exist in the workspace, type `ls()`.

To delete an object, use `rm()`. To delete an object named `data`, type

```
rm(data)
```

To delete three objects named `data`, `polity`, and `cow`, type

```
rm(list=c("data", "polity", "cow"))
```

Note: you need to put object names in quotes here.

Deleting Objects

When you start a new R session (by closing and restarting R), there are no objects in the workspace.

Sometimes you switch from working on one project to another. When you do, it might be a good idea to **delete ALL objects in the workspace**. That way, there's less chance of confusing new objects with old ones. To delete all objects, type

```
rm(list=ls())
```

Sometimes this command is compared to `clear` in Stata. It's similar, but two big differences:

1. `clear` closes datasets only, `rm(list=ls())` removes any object.
2. `clear` closes a file without saving it. `rm(list=ls())` DELETES objects.

Saving and Loading the Workspace

The workspace (the set of all of the objects you've made) is temporary.

But, when you close R and R Studio, they always ask you if you want to save the "workspace image", **even when there's nothing in it**.

You can also save the whole workspace without being prompted by typing

```
save(list=ls(), file="filename.Rdata")
```

If you save the workspace, it gets saved as an `.Rdata` file in your working directory.

This is NOT the same as a data file. It can only be read by R and R Studio. It contains many objects, potentially, not just data frames.

Managing the workspace

You can save just a few of the objects by typing something like

```
save(list=c("data", "polity", "cow"),
        file="filename.Rdata")
```

Caution: if "filename.Rdata" already exists in the working directory, this command OVERWRITES it. Be very careful if the raw data is stored in an `.Rdata` file.

You can load a saved workspace when you start a new R session by typing

```
load("filename.Rdata")
```

Now all the objects you had saved are in the workspace again.

Managing the workspace

I don't recommend relying on the `save()` and `load()` commands or `.Rdata` files. Here's why:

- `.Rdata` files are specific to R and do not transfer to other programs.
- `.Rdata` files are too easy to **overwrite**.
- `.Rdata` files encourage saving data in disparate objects instead of in a clean dataset.

Avoiding `.Rdata` files is **counter-intuitive** since it's natural to equate `.Rdata` files to `.dta` files for Stata or `.xls` files for Excel.

I use `.Rdata` files only when there is a very **specific reason** for saving objects instead of a data frame.

Much more often, I use commands to load and save data in **ASCII format files**.

Opening and Saving ASCII Data Files

By default, `read_csv()` assumes row 1 contains the variable names. But if the data does not have names in row 1, type

```
data <- read_csv("data.csv", col_names=FALSE)
```

To replace the given variable names with **names you choose**, type something like

```
data <- read_csv("data.csv",
        col_names=c("id", "vote", "age"))
```

Sometimes data files have a few lines of text at the top with information like authors, title, grant number, etc. To skip 3 lines at the top, type

```
data <- read_csv("data.csv", skip=3)
```

Opening and Saving ASCII Data Files

Sometimes certain rows in the ASCII file are **commented out** with some symbol the data's authors chose. To avoid reading rows that are marked with % at the beginning as data, type

```
data <- read_csv("data.csv", comment="%")
```

You might need to convert missing codes. For example, Stata marks missing values as `.` and R marks them as `NA`. To read the `.` markers as missing, type

```
data <- read_csv("data.csv", na=".")
```

Opening and Saving ASCII Data Files

`read_csv2()` is for semi-colon separated files.

`read_tsv()` is for tab separated files.

`read_table()` is for files where datapoints are separated by white space, not necessary tab.

`read_fwf()` is for fixed width files. You will have to also specify which characters (counting left to right) correspond to which variables.

To read data that is delimited in any other way (by &, for example), type

```
data <- read_delim("data.txt", delim="&")
```

Saving CSV data files

Remember the workflow: First, load the raw data. Then do stuff. End by **saving the cleaned/edited data under a different name**, and NEVER overwrite the original data.

To save a data frame object `data` as a new CSV file, type

```
write_csv(data, "different_name.csv")
```

To save a data frame object `data` as a new tab separated file, type

```
write_tsv(data, "different_name.txt")
```

These two commands will be enough the vast majority of the time. See *R for Data Science* for some commands for very special circumstances.

Arranging columns

You can arrange columns from left to right. Mostly, this task just makes the data nicer to look at when you use `View()`.

Note, however, that it will **change the column numbers**. So `stateleg[,15]` no longer refers to the same variable as before.

To arrange the `stateleg` data so that state name comes first, then year, then everything else, type:

```
stateleg <- select(stateleg, st, year, everything())
```

Careful: The `select()` command is also used to delete variables. If you forget to include `everything()` in the above command:

```
stateleg <- select(stateleg, st, year)
```

then every variable EXCEPT for `st` and `year` gets deleted.

Missing data

A data point is missing if for some reason we can't know what it is. R denotes missing values as `NA`.

"**Missing values are contagious.**" Since we don't know what an `NA` is, we can't know what operations with an `NA` are either:

$$1 + \texttt{NA} = \texttt{NA}$$

If you want to see the mean of a variable, this might happen:

```
> mean(stateleg$hou_dem)
[1] NA
```

That happened because the variable `hou_dem` has at least one missing value. To ignore missing values when using functions like the mean, use the `na.rm=TRUE` argument:

```
> mean(stateleg$hou_dem, na.rm=TRUE)
[1] -0.7418481
```

Missing data

The `summary()` command for a data frame will show you the number of missing values for each variable.

To see a giant matrix of TRUE/FALSE values, indicating whether each data point is missing, type

```
is.na(stateleg)
```

To **forcibly delete** any row with at least one missing value, type

```
stateleg <- na.omit(stateleg)
```

In general, I don't recommend doing that. It's heavy handed and there are better approaches to handling missing data.

Some notes on missing data

Data may be missing for a lot of reasons. On a survey, respondents might say "don't know," may refuse to answer, or the question might not be applicable.

Sometimes surveys will place a marker value to indicate a missing value.(For example, using 998 for "don't know"). If you do not recode these values, then it messes up mathematical functions like averages.

"Missing values are contagious."

Any mathematical function that includes even one missing value will be missing.

$$1 + \text{NA} = \text{NA}, \qquad 1 + \text{NA} \neq 1$$

In other words, **don't confuse missing with 0!** 0 is information, and missing is a lack of information.

Missing values for continuous variables

We can use logical statements to deal with situations in which missing values for continuous variables are given numeric codes.

For example, in the data data, thermometer scores are continuous, coded **0 to 100 scales**. But if a respondent says "don't know", the score is marked as **998**. If we don't replace these values, all results with these variables are thrown off.

The logical statement `data$ft_obama == 998` returns a vector of logical values: `TRUE` if the `ft_obama` variable is 998, `FALSE` if not.

This command replaces these values with `NA`s:

```
data$ft_obama[data$ft_obama == 998] <- NA
```

Recoding values

Recoding values means replacing many values of a categorical variable with new values, simultaneously.

There are a few reasons why you may want to recode:

1. Change the labels of values. Make it easier to see and remember what each value means. Better to code as `"Male"`, `"Female"` than 1, 2.

2. Replace missing codes with `NA`.

3. Change the order of categories for display in graphs, or in case you want to treat the variable as ordinal and categories are out of order.

Labeling categorical values

Suppose you have a categorical variable with values 1, 2, 3, 4, **8 and 9**.

You look in the data's codebook (hopefully it has one) and see that

Category	Meaning
1	I speak Spanish primarily
2	I speak both Spanish and English equally
3	I speak English primarily but can speak Spanish
4	I can not speak Spanish
8	**refused**
9	**skipped**

We can replace the numeric values with their written meanings, without changing the way the categorical data is treated in R.

Using `fct_recode()`

There are many ways to replace numeric categories with their written meanings. The easiest method uses the `fct_recode()` function from the `forcats` package (one of the `tidyverse`).

Here's an example of how to use `fct_recode()`:

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

Let's break down the elements of this code:

Using `fct_recode()`

This function must be only ever used inside the `mutate()` function.

Type `mutate()`, then the data frame, then the name of the categorical variable you are editing, then an equal sign.

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

Parentheses will appear automatically and will indent correctly when you push enter. Leave the closing parentheses alone.

Using `fct_recode()`

Then type `fct_recode()`

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

The first argument of `fct_recode()` is the categorical variable, which must be of the factor class. If it is not (here it is numeric), use `as.factor()` to coerce the variable:

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

Using `fct_recode()`

The press enter, and on each new line write the new categorical text label, in quotes, equal to the old categorical label, also in quotes.

Remember, as with the `rename()` function: **new first, then old**.

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

This code works whether the old labels are numbers or text.

Using `fct_recode()`

Finally, for the categories you want to set to be **missing**, write the new category labels as `NULL`, with no quotes:

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

This code is more space-consuming than other approaches. But the advantage is that we can more easily keep track of the new and old categories, minimizing the risk of confusing which label goes with which number.

Using `fct_recode()`

One more nice thing that `fct_recode()` can do: combine categories.

To combine two old categories into the same new category, just use the same new category label for multiple old categories.

For example, suppose we want to group every category in which a person knows at least some Spanish as **Yes**, and the category in which a person knows no Spanish as **No**. We can write:

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "Yes" = "1",
                                               "Yes" = "2",
                                               "Yes" = "3",
                                               "No" = "4",
                                               NULL = "8",
                                               NULL = "9"))
```

Reordering categories

The categories of a `factor` variable have a built-in order. The order controls a few things:

1. The order of the categories appear in any table
2. The order the categories appear left-to-right in any graph
3. The meaning of the variable when used in a **regression model**

Sometimes the categories have a *natural* ordering: for example, we can arrange categories in order of how much Spanish a person speaks.

Sometimes the categories don't have a natural ordering, but it makes sense to choose a particular order because it makes a table or graph looks better, or to change the base category in a regression.

Reordering categories

To change the order, use the `fct_relevel()` function. It works a lot like `fct_recode()`, only instead of writing old categories, simply write the existing categories in the order you want.

For example:

```
anes <- mutate(anes, speakspanish = fct_relevel(speakspanish,
                                                "I can not speak Spanish",
                                                "I speak English primarily but can speak Spanish",
                                                "I speak both Spanish and English equally",
                                                "I speak Spanish primarily"))
```

We'll talk more about why it matters to change the order in more detail over the next few weeks.

Reordering categories

Also, note that both the `fct_recode()` and `fct_relevel()` functions can be called within the same call to `mutate()`:

```
anes <- mutate(anes, speakspanish = fct_recode(as.factor(speakspanish),
                                               "I speak Spanish primarily" = "1",
                                               "I speak both Spanish and English equally" = "2",
                                               "I speak English primarily but can speak Spanish" = "3",
                                               "I can not speak Spanish" = "4",
                                               NULL = "8",
                                               NULL = "9"),
                    speakspanish = fct_relevel(speakspanish,
                                               "I can not speak Spanish",
                                               "I speak English primarily but can speak Spanish",
                                               "I speak both Spanish and English equally",
                                               "I speak Spanish primarily"))
```

If you have multiple categorical variables to edit in this way, you can place all the calls to `fct_recode()` and `fct_relevel()` in the same `mutate()` command.

Combining categories across multiple variables

Sometimes a survey will store categorical data in multiple variables.

For example:

- **Variable 1**: are you a Democrat, Republican, or neither?
- **Variable 2**: (if Democrat/Republican) are you a strong Democrat/Republican?
- **Variable 3**: (if neither) do you lean towards one party or the other?

To combine categorical variables, use the `unite()` command.

```
data <- unite(data, pid, pid1d, pid1r, pidstr, pidlean)
```

It pastes categories from different variables together. THEN you can recode these pasted categories.

Managing string/character variables

To split up a string into a list of fragments of the string, use the `str_split()` command.

```
> str_split(response, pattern=" ")
[[1]]
 [1] "I"        "have"     "the"      "faintest" "idea."
 [6] "I"        "really"   "don't"    "watch"    "the"
[11] "news"
```

You can break the string on any kind of character you type with the `pattern` argument:

```
> str_split(response, pattern="'")
[[1]]
[1] "I have the faintest idea. I really don"
[2] "t watch the news"
```

Managing string/character variables

One annoying thing about how strings are coded sometimes is extraneous white space before and after the text. To remove **leading and trailing spaces**, use the `str_trim()` command:

```
> hello <- c("           whatsup?           ")
> hello
[1] "           whatsup?           "
> str_trim(hello)
[1] "whatsup?"
```

Managing string/character variables

One of the most useful ways to pull data from a string is to search for particular words. The `str_detect()` command returns TRUE if the word is found in the string, FALSE if not. The match must be exact, and it is case sensitive.

```
> str_detect(response, "news")
[1] TRUE
> str_detect(response, "have")
[1] TRUE
> str_detect(response, "haven't")
[1] FALSE
```

The `str_detect()` function can also take regular expressions instead of a single word. We'll cover that in more detail later in the semester.

Managing string/character variables

You can also search for multiple patterns at the same time with `str_detect()`, but the code is a bit strange. There are two ways to proceed.

First, you can separate different words or terms with the | symbol inside the quotes. This only works if there are **no spaces or carriage returns** next to any | symbol. So if you are looking for one of "larry", "curly", and "moe" in the text:

This works: `str_detect(response, "larry|curly|moe")`
You will get a variable that is TRUE if the text contains "larry" or "curly" or "moe"

Doesn't work:
`str_detect(response, "larry | curly | moe")`
You won't get an error but the TRUE and FALSE values won't be assigned correctly.

Managing string/character variables

If you have a **long list** of words, it's better to use the following technique:

1. Put all of the words you want to search for in a character vector object

2. Instead of a word in quotes, type

```
paste(object, collapse = "|")
```

where `object` is the vector you made in step 1.

Then the variable you create will be TRUE if any of the words in your vector are present.

Managing string/character variables

For example, to search for the names of UVA men's basketball players in a variable named `text`, type:

```
players <- c("Marco Anthony", "Francesco Badocchi",
             "Francisco Caffaro", "Kihei Clark",
             "Mamadi Diakite", "Kyle Guy", "Jay Huff",
             "De'Andre Hunter", "Ty Jerome",
             "Austin Katstra", "Braxton Key",
             "Jayden Nixon", "Jack Salt", "Kody Stattmann")

data <- mutate(data, mention_player =
                     str_detect(text, paste(players, collapse="|")))
```

The variable mention_player will be TRUE if `text` contains any of these names, and FALSE otherwise.

Managing date/time variables

There are lots of data in political science that involve dates and times. Unfortunately, there is NOT a consistent way that dates and times are recorded in data.

Problem 1: Sometimes a single date/time is recorded in many variables: year, month, day, hour, minute *might all be separate columns*.

Problem 2: Sometimes a date/time is recorded in one variable as a **messy and unusable block of text**.

How do we deal with each type of coding? How do we get R to recognize that the variable is a date or time?

Managing date/time variables

As with anything in R, there are many ways to do the same thing. Here we will use the `lubridate` package:

```
library(lubridate)
```

Problem 1: the date and time are stored in many variables:

```
year1 <- 1983; month1 <- 4; day1 <- 14
year2 <- 2018; month2 <- 2; day2 <- 22
```

We can use the `make_datetime()` command to create single variables from multiple variables. We specify which variables contain the year, month, day, etc.

```
date1 <- make_datetime(year=year1, month=month1, day=day1)
date2 <- make_datetime(year=year2, month=month2, day=day2)
```

We can also add the `hour`, `minute`, and `second` arguments if we have that information.

Managing date/time variables

Now `date1` and `date2` are recognized as dates:

```
> date1
[1] "1983-04-14 UTC"
> date2
[1] "2018-02-22 UTC"
```

The class of a date variable is called `POSIXct` and `POSIXt`. It's a technical name, but it just means "date and time".

```
> class(date1)
[1] "POSIXct" "POSIXt"
```

Managing date/time variables

Now that we have two date variables, we can use basic arithmetic functions with them. We can for example measure the length of time between the two dates:

```
> duration <- date2 - date1
> duration
Time difference of 12733 days
> as.numeric(duration)/365
[1] 34.88493
```

The `make_datetime()` command can work within the `mutate()` command to create new variables.

Managing date/time variables

Problem 2: pulling date and time information from a single, messy variable.

```
time1 <- "2018-02-22 10:05:28"
time2 <- "02-22-2018 10:05:28"
time3 <- "22-02-2018 10:05:28"
```

These variables contain the year, month, day, hour, minute, and second, all in one variable. They are also currently read as character. How do we get R to read these as **dates and times**, and to recognize the correct year, month, etc.?

The `ymd_hms()` command assumes that the data contains the year, month, day, hour, minute, and second in that order:

```
ymd_hms(time1)
[1] "2018-02-22 10:05:28 UTC"
```

Managing date/time variables

The `time2` variable is different: it lists the MONTH first, then the DAY, then the YEAR. If we use the `ymd_hms()` command, we parse incorrectly:

```
> ymd_hms(time2)
[1] NA
Warning message:
All formats failed to parse. No formats found.
```

The error occurs because it reads 2018 as the day, and that can't happen.

Instead we can use the `mdy_hms()` command:

```
> mdy_hms(time2)
[1] "2018-02-22 10:05:28 UTC"
```

Managing date/time variables

There are many of these commands, and they differ only in the order you write the letters **y**, **m**, and **d**.

These commands automatically recognize the different characters (hyphens, slashes, etc.) separate elements of the date, so no need to worry about specifying that.

Piping

Advanced R programmers know how to use a pipe.

Piping

A pipe is a way to connect consecutive R functions that manipulate the same data frame.

The code for a pipe is `%>%` at the end of one command. It means "**apply the next command to the data output by this command**." Pipes are implemented in the `magrittr` package, one of the packages included with `tidyverse`.

Using the pipe is completely optional. It saves some time and space, and it makes you look like an advanced programmer.

Piping

Here are some data cleaning commands you know:

```
data <- read_csv("data.csv")
data <- select(data, vote, age, sex, starts_with("party"))
data <- arrange(data, age)
data <- filter(data, party_ID=="republican")
data <- mutate(data, vote = fct_recode(vote,
                                       "Trump" = "1. D Trump",
                                       "Clinton" = "2. H Clinton",
                                       NULL = "3. G Johnson",
                                       NULL = "4. J Stein",
                                       NULL = "5. Other"))
```

Notice how every command begins by calling the data frame we want to alter.

Piping

The exact same commands can be run with the following code:

```
data <- read_csv("data.csv")
data <- data %>%
     select(vote, age, sex, starts_with("party")) %>%
     arrange(age) %>%
     filter(party_ID=="republican") %>%
     mutate(vote = fct_recode(vote,
                              "Trump" = "1. D Trump",
                              "Clinton" = "2. H Clinton",
                              NULL = "3. G Johnson",
                              NULL = "4. J Stein",
                              NULL = "5. Other"))
```

Notice that the `select()`, `arrange()`, `filter()`, and `mutate()` functions no longer need the data frame listed first.

Pipes can be used with **any function that takes a data frame as the first argument**.

Using `group_by()`

Sometimes it is necessary to do calculations within groups of observations. Consider these example data (saved as object `table1` in R once you load `tidyverse`):

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

We might want to create a variable with the total number of cases per year, or the average population by country.

Using group_by()

The `group_by()` function doesn't do anything by itself, but it allows you to use `mutate()` or `summarize()` to perform calculations in groups.

Within `group_by()`, specify the variable or variables that denote the groups.

`mutate()` performs within-group calculations within the existing data frame.

`summarize()` deletes rows, and leaves you with **one row per group**.

To turn off within-group calculations, use the `ungroup` function.

Using group_by()

To calculate the total number of cases per year in `table1` use this code:

```
table2 <- table1 %>%
     group_by(year) %>%
     mutate(totalcases = sum(cases))
```

The data now look like this:

country	year	cases	population	totalcases
Afghanistan	1999	745	19987071	250740
Afghanistan	2000	2666	20595360	296920
Brazil	1999	37737	172006362	250740
Brazil	2000	80488	174504898	296920
China	1999	212258	1272915272	250740
China	2000	213766	1280428583	296920

Using group_by()

To do this calculation leaving only one row per year, use `summarize()` instead of `mutate()`:

```
table2 <- table1 %>%
     group_by(year) %>%
     summarize(totalcases = sum(cases))
```

The data now look like this:

year	totalcases
1999	250740
2000	296920

Using group_by()

To calculate the average population for each country in table1 use this code:

```
table2 <- table1 %>%
     group_by(country) %>%
     mutate(avg.pop = mean(population))
```

The data now look like this:

country	year	cases	population	avg.pop
Afghanistan	1999	745	19987071	20291216
Afghanistan	2000	2666	20595360	20291216
Brazil	1999	37737	172006362	173255630
Brazil	2000	80488	174504898	173255630
China	1999	212258	1272915272	1276671928
China	2000	213766	1280428583	1276671928

Using group_by()

To do this calculation leaving only one row per country, use `summarize()` instead of `mutate()`:

```
table2 <- table1 %>%
     group_by(country) %>%
     summarize(avg.pop = mean(population))
```

The data now look like this:

country	avg.pop
Afghanistan	20291216
Brazil	173255630
China	1276671928

Using group_by()

ungroup turns off within-group calculations. To calculate the percent of cases that each country accounts for:

```
table2 <- table1 %>%
     group_by(year) %>%
     mutate(totalcases = sum(cases)) %>%
     ungroup %>%
     mutate(percent_of_cases = 100*cases/totalcases)
```

The data now look like this:

country	year	cases	population	totalcases	percent
Afghanistan	1999	745	19987071	250740	0.297
Afghanistan	2000	2666	20595360	296920	0.898
Brazil	1999	37737	172006362	250740	15.05
Brazil	2000	80488	174504898	296920	27.11
China	1999	212258	1272915272	250740	84.65
China	2000	213766	1280428583	296920	71.99

Using count()

The count() function counts the number of rows within each group. It's not necessary to use group_by() when calling count().

Suppose we have data with terrorist attacks, with variables year and country. To create a dataset with a count of the **total attacks by country**, sorted from most to least, type

```
attacks2 <- attacks %>%
        count(country, sort=TRUE)
```

To create a dataset with a count of the **total attacks by in each country and year**, sorted from most to least, type

```
attacks2 <- attacks %>%
        count(country, year, sort=TRUE)
```