DATA WRANGLING

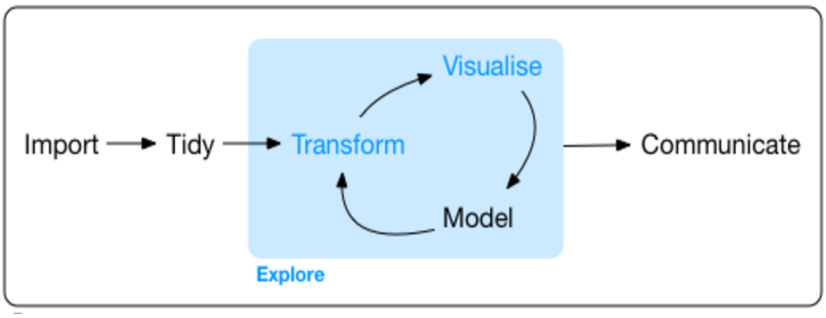
Research Methods in Psychology I & II ▪ Department of Psychology ▪ Colorado State University

BY THE END OF THIS SECTION YOU WILL:

1. Know how to use the dplyr verbs to wrangle and prepare data.
2. Know how to use the magrittr package to simplify and link R commands with pipes.
3. Know how to use the gather and spread functions to flip data from wide to long, and long to wide.
4. Know how to join dataframes by a common key or keys.

What is Data Wrangling?

To wrangle data is to clean and prepare data for analysis. Our goal is to get the data into a tidy shape that can be readily plotted and analyzed, and to prepare new forms of variables, or subsets of variables, in accordance with our research questions.



Wickham & Grolemund—R for Data Science

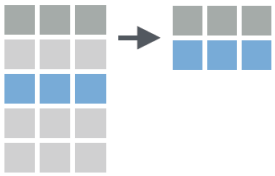
The basics of dplyr

dplyr is a R package that is part of the tidyverse — it's the workhorse for wrangling your data. We will start with five key functions that you will use with most projects:

1. Choose certain cases (rows) in your dataframe (e.g., all countries in Africa) with **filter().**
2. Choose certain variables (columns) in your dataframe (e.g., lifeExp & gdpPercap) with **select().**
3. Create or modify variables with **mutate().**
4. Order your cases (rows) with **arrange().**
5. Collapse many values down to a single summary with **summarize().**

filter()

Use filter() to subset cases (rows) based on their values. The first argument to the filter command is the name of the dataframe. Subsequent arguments denote the desired cases.



Let's work with our Gapminder Notebook. Imagine that we want to create a new dataframe that includes only the most recent year of estimates (2007). First, open up your MyClassActivities Project, and your Gapminder notebook. Run the chunks to load the libraries and import the data.

As the first argument to the function, we list the dataframe that we are going to filter

Then, we list the selection criteria. Notice the double equals for an equality check.

Name your new, subsetted dataframe →

```
gm2007 <- filter(gm, year == 2007)
```

Legal comparison operators:

- >, >= (greater than, greater than or equal to)
- <, <= (less than, less than or equal to)
- == (equal to)
- != (not equal to)
- | (or)
- %in% (can be used to specify a list of values to choose. For example: x %in% c(1, 2, 3) would choose all values of x equal to 1, 2, or 3.

```
# Wrangle the data
## Explore filter
### Make a subset dataframe with only data from 2007
```{r}

gm2007 <- filter(gm, year == 2007)

str(gm2007)

```
```

```
Classes 'tbl_df', 'tbl' and 'data.frame':       142 obs. of  6 variables:
 $ country  : chr  "Afghanistan" "Albania" "Algeria" "Angola" ...
 $ continent: chr  "Asia" "Europe" "Africa" "Africa" ...
 $ year     : int  2007 2007 2007 2007 2007 2007 2007 2007 2007 ...
 $ lifeExp  : num  43.8 76.4 72.3 42.7 75.3 ...
 $ pop      : int  31889923 3600523 33333216 12420476 40301927 20434176
8199783 708573 150448339 10392226 ...
 $ gdpPercap: num  975 5937 6223 4797 12779 ...
```

Always take a look at the subsetted dataframe and make sure it worked as intended!!!

filter() continued

```
filter1 <- filter(gm, year > 1970)
filter2 <- filter(gm, year == 2007 & continent == "Asia")
filter3 <- filter(gm, continent == "Americas" | continent == "Europe")
filter4 <- filter(gm, year %in% c(1952, 1957, 1962))
filter5 <- filter(gm, country %in% c("Panama","Costa Rica", "Indonesia", "Australia") & year != 1952)
```

What does each line of code do?

select()

Use select() to choose needed columns. The first argument to the filter command is the name of the dataframe, subsequent arguments denote the desired columns.

Imagine that we want to create a new dataframe that includes only country, year and lifeExp.

As the first argument to the function, we list the dataframe that we are going to select from

Then, we list the variables we want to select

Name your new, subsetted dataframe →

```
le <- select(gm, country, year, lifeExp)
```

Helper functions:

- starts_with("x1") selects all vars that start with x1
- ends_with("w1") selects all vars that end with w1
- contains("cog") selects all vars that contain the phrase cog
- num_range("y", 1:5) selects y1, y2, y3, y4, y5

Let's explore a few more ways to use select.

```
select1 <- select(gm, -lifeExp)
```

Use - (minus sign) to drop variables

```
select2 <- select(gm, continent:lifeExp)
```

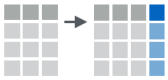
Use : to select everything between two variables

```
select3 <- select(gm, country, continent, gdpPercap, everything())
```

List a set of vars, and then "everything" to reorder

mutate()

We can use mutate to create new variables or to modify existing variables.



There are countless ways we can use mutate to form and modify variables. Let's look at a few that you will commonly use, and we'll learn more as we progress through the semester.

```
gm_add1 <- mutate(gm, gdp = gdpPercap*pop)
```

In this first example, we create a new dataframe called gm_add1 in which we mutate the dataframe called gm by adding a new variable called gdp. This new variable is computed by taking the existing variable called gdpPercap and multiplying it by the existing variable pop.

```
gm_add2 <- mutate(gm, live72 = ifelse(lifeExp >= 72, 1, 0),
                      live72.f = factor(live72, levels = c(0,1),
                                        labels = c("Life expectancy 72 years or less", "Life expectancy over 72 years")))
```

In this second example, we will create a new dataframe called gm_add2 in which we mutate the dataframe called gm by adding a new variable called live72 which differentiates between cases where the life expectancy is greater than or equal to 72 and those cases where life expectancy is less than 72.

This new variable is computed by using the ifelse function. The ifelse function supplies a test expression, in this case "lifeExp >= 72." Then the value to assign if the test expression is true is supplied (1), and a value to assign if the text expression is false is supplied (0)). So, for each row of data, the value of lifeExp will be evaluated. If lifeExp is greater than or equal to 72, a 1 will be assigned for the new variable called live72, if lifeExp is less than 72 (but not missing) then a 0 will be assigned for the new variable called live72.

Next, a factor based on live72, here we call it live72.f, is created. The first argument to the factor command is the variable that you want to base the factor on (live72), then you can optionally specify the levels of the factor (i.e., their order), and the corresponding names. The benefit of this is that R will treat this variable as a factor rather than a numeric variable when used in an analysis or plotted in a graph, and the names supplied in the label argument will appear (for example, in the legend if used as a grouping variable).

mutate() continued

We do not need a separate mutate command for each new variable, we can put them together as follows:

```
gm_new <- mutate(gm,
                 gdp = gdpPercap*pop,
                 live72 = ifelse(lifeExp >= 72, 1, 0),
                 live72.f = factor(live72, levels = c(0,1),
                                   labels = c("Life expectancy 72 years or less", "Life expectancy over 72 years")))
```

arrange()

We can use arrange to sort a dataframe.

First, let's sort by one variable. List the dataframe name, and then the variables to sort by.

```
arrange(gm, year)
```

Sort by two variables.

```
arrange(gm, year, lifeExp)
```

Sort by two variables, but in descending order for the second.

```
arrange(gm, year, desc(lifeExp))
```

Save the sorted data as a new dataframe.

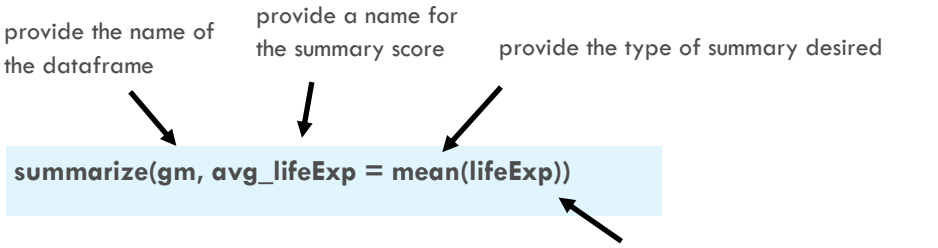
```
sorted_gm <- arrange(gm, year, desc(lifeExp))
```

summarize()

We can use summarize to get summaries of our variables. We'll explore some additional packages that are helpful for summarizing data when we begin to discuss data description, but summarize is a useful wrangling technique to learn and will come in handy in many occasions.

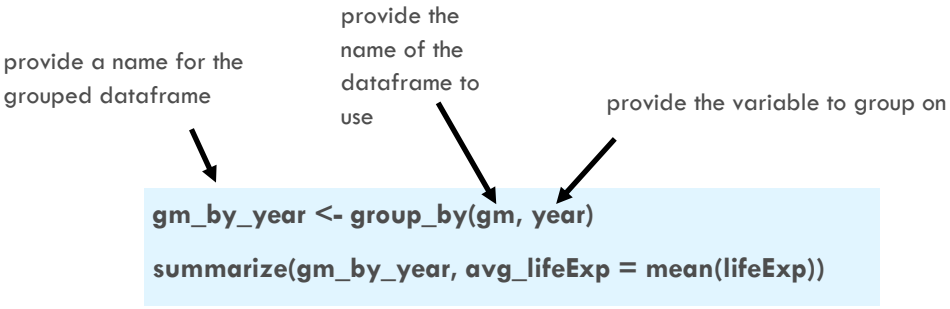
We can use summarize to get the overall mean of a variable in the dataframe.

NOTE: if there is missing data on the variable you want to summarize, you must include the following additional argument: na.rm = TRUE. This tells R to remove any missing cases and calculate the summary on the remainder.



avg_lifeExp <dbl>
59.47444

The utility of the summarize function becomes apparent when paired with the group_by function. We can use group_by and summarize to obtain summary information for subsets of our data. For example, let's obtain the average lifeExp by year.



year <int>	avg_lifeExp <dbl>
1952	49.05762
1957	51.50740
1962	53.60925
1967	55.67829
1972	57.64739
1977	59.57016
1982	61.53320
1987	63.21261
1992	64.16034
1997	65.01468

summarize() continued

We aren't limited to grouping by one variable—we can group by multiple variables. Let's group by continent and year.

```
gm_by_year_continent <- group_by(gm, continent, year)
summarize(gm_by_year_continent, avg_lifeExp = mean(lifeExp))
```

continent <chr>	year <int>	avg_lifeExp <dbl>
Africa	1952	39.13550
Africa	1957	41.26635
Africa	1962	43.31944
Africa	1967	45.33454
Africa	1972	47.45094
Africa	1977	49.58042
Africa	1982	51.59287
Africa	1987	53.34479
Africa	1992	53.62958
Africa	1997	53.59827

We also aren't limited to just the mean — we can request a plethora of different summary types. For example, let's go back to the summary by year — but now request the mean, the standard deviation (sd), and the count. Note that for the count (n), the variable name is not listed. Other useful summaries are: median; IQR (i.e., the interquartile range), mad (i.e., median absolute deviation), min (i.e., minimum), max (i.e., maximum) and quantile (for example quantile(x, .75) will print the 75th percentile score for x).

```
gm_by_year <- group_by(gm, year)
summarize(gm_by_year, avg_lifeExp = mean(lifeExp), sd_lifeExp = sd(lifeExp), ncount = n())
```

year <int>	avg_lifeExp <dbl>	sd_lifeExp <dbl>	ncount <int>
1952	49.05762	12.22596	142
1957	51.50740	12.23129	142
1962	53.60925	12.09725	142
1967	55.67829	11.71886	142
1972	57.64739	11.38195	142
1977	59.57016	11.22723	142
1982	61.53320	10.77062	142
1987	63.21261	10.55629	142
1992	64.16034	11.22738	142
1997	65.01468	11.55944	142

magrittr and pipes

So far, we've focused primarily on writing a series of separate functions to perform different tasks. However, we can link together code by using pipes from the magrittr package. This is a package that automatically loads with the tidyverse. Pipes make your code more readable/understandable, and also can reduce the length of code (i.e., simplify code). The pipe operator looks like this %>%. As a short cut, when in RStudio you can hit CTRL + SHIFT + M on a PC or CMD + SHIFT + M on a MAC. Think of the pipe operator as saying: Take what's on the left side of the operator and feed it to what's on the right side of the operator.

Let's try a few examples. First, let's request the life expectancy for Indonesia across years.

```
gm %>%
  filter(country == "Indonesia") %>%
  select(year, lifeExp)
```

year <int>	lifeExp <dbl>
1952	37.468
1957	39.918
1962	42.518
1967	45.964
1972	49.203
1977	52.702
1982	56.159
1987	60.137
1992	62.681
1997	66.041

1-10 of 12 rows

If you want to save the results of this pipe, then simply provide a name. Now a dataframe called indo is created.

```
indo <- gm %>%
  filter(country == "Indonesia") %>%
  select(year, lifeExp)
```

magrittr and pipes continued

Let's go for a more complex example. Here we will obtain the mean, standard deviation and n cases for life expectancy for each continent at each year, but we exclude Oceania.

```
gm %>%
  filter(continent != "Oceania") %>%
  select(year, continent, country, lifeExp) %>%
  group_by(continent, year) %>%
  summarize(avg_lifeExp = mean(lifeExp), sd_lifeExp = sd(lifeExp), ncount = n())
```

continent <chr>	year <int>	avg_lifeExp <dbl>	sd_lifeExp <dbl>	ncount <int>
Africa	1952	39.13550	5.151581	52
Africa	1957	41.26635	5.620123	52
Africa	1962	43.31944	5.875364	52
Africa	1967	45.33454	6.082673	52
Africa	1972	47.45094	6.416258	52
Africa	1977	49.58042	6.808197	52
Africa	1982	51.59287	7.375940	52
Africa	1987	53.34479	7.864089	52
Africa	1992	53.62958	9.461071	52
Africa	1997	53.59827	9.103387	52

1-10 of 48 rows Previous 1 2 3 4 5 Next

Your turn. Please create a new dataframe called median_gdp, which includes year and the median gdpPercap for that year. Your goal is to obtain the dataframe below. Once you obtain the dataframe, use it to create a line chart to represent change in median gdp per cap across years.

Gapminder Notebook.Rmd* median_gdp

Filter

	year	med_gdpPercap
1	1952	1968.528
2	1957	2173.220
3	1962	2335.440
4	1967	2678.335
5	1972	3339.129
6	1977	3798.609
7	1982	4216.228
8	1987	4280.300
9	1992	4386.086
10	1997	4781.825
11	2002	5319.805
12	2007	6124.371

gather() and spread() — making data long and wide

Occasionally, our data aren't in the proper orientation to facilitate our analysis. In these cases, it will be necessary for us to change the orientation.

spread()

In the gapminder dataset, the data are in a long format. That is, each country has multiple lines of data. Let's imagine that rather than having a single variable called LifeExp that is indexed by year — we want to have a separate variable of LifeExp for each year. In this case we want to go from a long dataset to a wide dataset. We can use the spread() function in tidyr to accomplish this. The first step is to determine the variable that you want to take from long to wide, that is lifeExp for us; this is called the "value." Next, identify the variable that indexes the repeated measures, that is year for us; this is called the "key."

key value

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.801	8425333	779.4453
Afghanistan	Asia	1957	30.332	9240934	820.8530
Afghanistan	Asia	1962	31.997	10267083	853.1007
Afghanistan	Asia	1967	34.020	11537966	836.1971
Afghanistan	Asia	1972	36.088	13079460	739.9811
Afghanistan	Asia	1977	38.438	14880372	786.1134
Afghanistan	Asia	1982	39.854	12881816	978.0114
Afghanistan	Asia	1987	40.822	13867957	852.3959
Afghanistan	Asia	1992	41.674	16317921	649.3414
Afghanistan	Asia	1997	41.763	22227415	635.3414
Afghanistan	Asia	2002	42.129	25268405	726.7341
Afghanistan	Asia	2007	43.828	31889923	974.5803
Albania	Europe	1952	55.230	1282697	1601.0561
Albania	Europe	1957	59.280	1476505	1942.2842
Albania	Europe	1962	64.820	1728137	2312.8890
Albania	Europe	1967	66.220	1984060	2760.1969
Albania	Europe	1972	67.690	2263554	3313.4222

Using the pipe operator, we indicate that we want to create a new data-frame called gm_wide, based on the gm data-frame.

```
gm_wide <- gm %>%
  select(country, year, lifeExp) %>%
  spread(key = year, value = lifeExp)
```

Next select the columns necessary to spread the data, we need country (so that we know which row represents which country), year (this serves as the key), and lifeExp (this serves as the value).

Finally, we spread—just denote the key and the value.

Filter

country	1952	1957	1962	1967	1972	1977	1982	1987	1992	1997	2002	2007
Afghanistan	28.801	30.33200	31.99700	34.02000	36.08800	38.43800	39.854	40.822	41.674	41.763	42.129	43.828
Albania	55.230	59.28000	64.82000	66.22000	67.69000	68.93000	70.420	72.000	71.581	72.950	75.651	76.423

Now the data are in a wide format. However, the variable names of these new variables are not ideal — it would be preferable for them to denote that the value is lifeExp. This is an easy fix.

spread() continued

I have added an intermediate mutate step to modify the values in the variable year. Using the paste function, I request that the prefix lifeExp be add to the value, separated (sep) by an underscore.

```
gm_wide <- gm %>%
  select(country, year, lifeExp) %>%
  mutate(year = paste('lifeExp', year, sep="_")) %>%
  spread(key = year, value = lifeExp)
```

key value

	country	year	lifeExp
1	Afghanistan	lifeExp_1952	28.801
2	Afghanistan	lifeExp_1957	30.332
3	Afghanistan	lifeExp_1962	31.997
4	Afghanistan	lifeExp_1967	34.020
5	Afghanistan	lifeExp_1972	36.088
6	Afghanistan	lifeExp_1977	38.438

Filter

country	lifeExp_1952	lifeExp_1957	lifeExp_1962	lifeExp_1967	lifeExp_1972	lifeExp_1977	lifeExp_1982	lifeExp_1987	lifeExp_1992	lifeExp_1997	lifeExp_2002	lifeExp_2007
Afghanistan	28.801	30.33200	31.99700	34.02000	36.08800	38.43800	39.854	40.822	41.674	41.763	42.129	43.828
Albania	55.230	59.28000	64.82000	66.22000	67.69000	68.93000	70.420	72.000	71.581	72.950	75.651	76.423

Now the data are in a long format, and the variables names for the lifeExp variables are informative.

gather()

We can go from wide to long with the gather function. Let's put the gm_wide dataframe we just created back to a long dataframe. For the gather function, we need to list the variables that we want to flip from wide to long— in this example this is the life expectancy variables for each year. Because they are lined up in the dataframe, we can use the : shortcut, which denotes all variables in the dataframe starting with lifeExp_1952 and ending with lifeExp_2007. If they weren't lined up, you would list them with a comma in between. Then, we list the key. This is the name that you want to give the indexing variable — in our case that's lifeExp_year. Finally, the name you want to give the corresponding variable that you are taking from wide to long — in our case that is lifeExp.

```
gm_long <- gm_wide %>%
  gather(lifeExp_1952:lifeExp_2007, key = "lifeExp_year", value = "lifeExp")
```

Variables to be flipped

Name for the new indexing variable

Name for the new variable that you are flipping

Filter

	country	lifeExp_year	lifeExp
1	Afghanistan	lifeExp_1952	28.80100
2	Albania	lifeExp_1952	55.23000
3	Algeria	lifeExp_1952	43.07700
4	Angola	lifeExp_1952	30.01500
5	Argentina	lifeExp_1952	62.48500
6	Australia	lifeExp_1952	69.12000

There are two undesirable aspects of this flipped data that we can remedy. First, the indexing column (lifeExp_year) has a prefix, it would be more desirable to take it back to the numerical value. Second, the data aren't sorted by country. We can easily fix these things with a few additional function calls. First, we can use the separate function to separate the lifeExp_year value into two new variables. By default, the separate function will make the split at a non-alphanumeric character, which works for us because the 2 parts are separated by an underscore. The arguments to the separate function are the variable to separate, then you list "into =" and the two new names of the variables. I will call the first one tempdelete because this will just hold the string lifeExp. The second will be called year, this is what we will keep. Next, I use the select function to drop tempdelete. And finally, I sort the data by country and year using the arrange function.

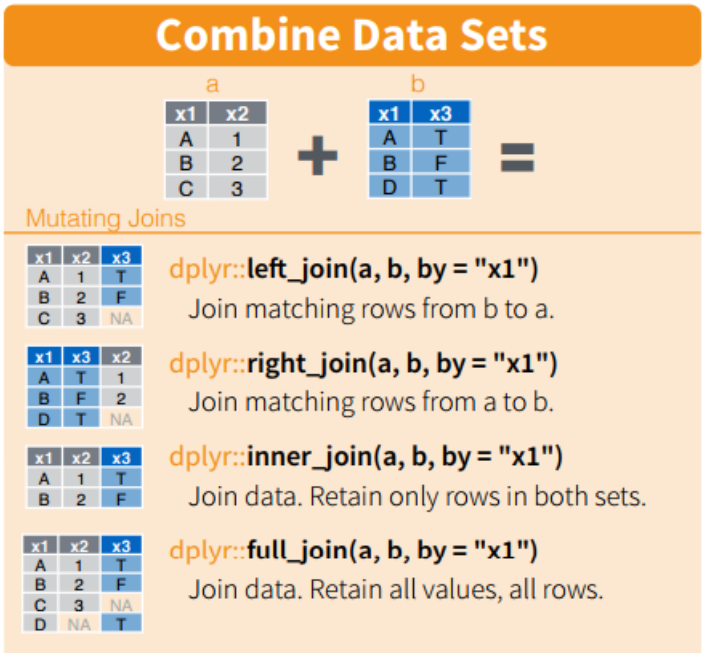
```
gm_long <- gm_wide %>%
  gather(lifeExp_1952:lifeExp_2007, key = "lifeExp_year", value = "lifeExp") %>%
  separate(lifeExp_year, into = c("tempdelete", "year")) %>%
  select(-tempdelete) %>%
  arrange(country, year)
```

Filter

	country	year	lifeExp
1	Afghanistan	1952	28.801
2	Afghanistan	1957	30.332
3	Afghanistan	1962	31.997
4	Afghanistan	1967	34.020
5	Afghanistan	1972	36.088
6	Afghanistan	1977	38.438
7	Afghanistan	1982	39.854
8	Afghanistan	1987	40.822
9	Afghanistan	1992	41.674
10	Afghanistan	1997	41.763
11	Afghanistan	2002	42.129
12	Afghanistan	2007	43.828
13	Albania	1952	55.230
14	Albania	1957	59.280

Joining (i.e., merging) dataframes

As you analyze data, you will often have the need to join together dataframes. To get an intuitive sense of how joins work, consider the figure below, which depicts two dataframes called a and b. The column x1 represents the "key" variable (people often refer to this as the ID) in each dataframe. Our desire is to match each row in the a dataframe with the corresponding row in the b dataframe based on this key.



There are several different methods for joining dataframes. Let's start with an **inner join**, which matches pairs of observations with the same key. In this type of join, non-matching rows are discarded.

Now let's consider **outer joins**. Unlike an inner join, an outer joins does not automatically drop non-matching rows. There are three types of outer joins, depending on which non-matching rows you want to keep. Notice that missing observations are denoted as NA — this is R's default missing data indicator.

1. A left join keeps all rows in the a dataframe (it will be the first dataframe that you list).
2. A right join keeps all rows in the b dataframe (it will be the second dataframe that you list).
3. A full join keeps all rows in the a and b dataframes.

Joining (i.e., merging) dataframes continued

Import a dataframe
```{r}

a <- read_csv("a.csv")
a

```

Import b dataframe
```{r}

b <- read_csv("b.csv")
b

```

Perform an inner join
```{r}

inner <- inner_join(a, b, by = "x1")
inner

```

x1 <chr>	x2 <int>	x3 <lgl>
A	1	TRUE
B	2	FALSE

2 rows

Perform a left join
```{r}

left <- left_join(a, b, by = "x1")
left

```

x1 <chr>	x2 <int>	x3 <lgl>
A	1	TRUE
B	2	FALSE
C	3	NA

Perform right join
```{r}

right <- right_join(a, b, by = "x1")
right

```

x1 <chr>	x2 <int>	x3 <lgl>
A	1	TRUE
B	2	FALSE
D	NA	TRUE

3 rows

Perform a full join
```{r}

full <- full_join(a, b, by = "x1")
full

```

x1 <chr>	x2 <int>	x3 <lgl>
A	1	TRUE
B	2	FALSE
C	3	NA
D	NA	TRUE

4 rows

Joining dataframes with multiple keys

Sometimes we have more than one key that uniquely identifies each row — this is the case for the gapminder dataframe. Consider the example below. I first break apart the gm dataframe into two — one that contains lifeExp and one that contains gdpPercap. Then, I join them back together, indicating that the match needs to consider both country and year. Note that because there is no missing data and the exact same rows are in each dataframe, then I can choose any of the join methods. For more complex examples of joining data, see R for Data Science: http://r4ds.had.co.nz/.

```
# import data
gm <- read_csv(file="gapminder.csv")


# break apart
gm1 <- select(gm, country, year, lifeExp)
gm2 <- select(gm, country, year, gdpPercap)


# rejoin
gm3 <- inner_join(gm1, gm2, by = c("country", "year"))
```

Use the combine/concatenate function to indicate that multiple variables are needed to uniquely identify each row.