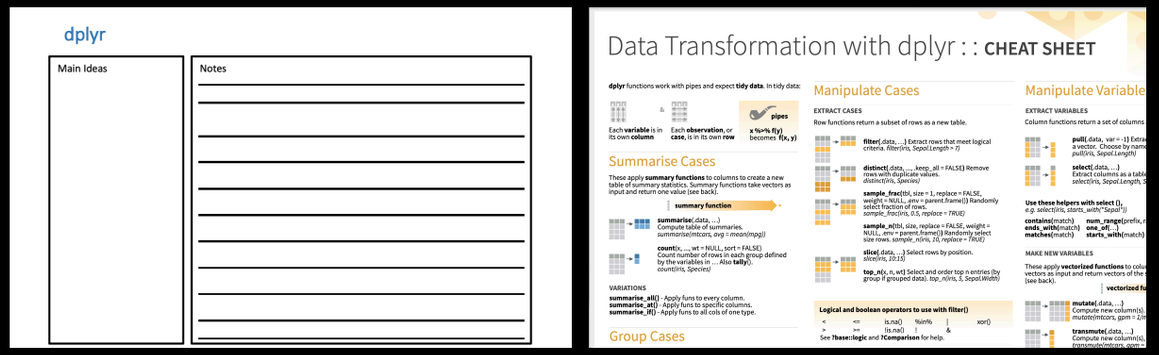
Transform Data with

I mentioned at the beginning that there are two main packages in the Tidyverse: ggplot2 and dplyr. We already learned ggplot2, and now let's turn out attention to dplyr.

Exercise: What do you already know?

- Split into groups of 2
- Ask you partner:
  - *What do you already know about dplyr?*
  - *What type of "data transformations" do you normally do?*
- 5 minutes



Notes form & Cheat Sheet

Please take out

There are two documents that I'd like you to take out before we start. The first is the notes form I gave you.

Your Turn

Open **02-dplyr-exercises.Rmd**.

babynames

Names of male and female babies born in the US from 1880 to 2015. 1.8M rows.

```
# install.packages("babynames")
library(babynames)
```

In the first half of this lesson we're going to focus on analyzing a single dataframe: babynames. It has information on the names of all babies born in the US between 1880 and 2015. It's in its own package, which you should have already installed: babynames.

babynames

year <dbl>	sex <chr>	name <chr>	n <int>	prop <dbl>
1880	F	Mary	7065	7.238433e-02
1880	F	Anna	2604	2.667923e-02
1880	F	Emma	2003	2.052170e-02
1880	F	Elizabeth	1939	1.986599e-02
1880	F	Minnie	1746	1.788861e-02
1880	F	Margaret	1578	1.616737e-02
1880	F	Ida	1472	1.508135e-02
1880	F	Alice	1414	1.448711e-02
1880	F	Bertha	1320	1.352404e-02
1880	F	Sarah	1288	1.319618e-02

1–10 of 1,858,689 rows Previous 1 2 3 4 5 6 ... 100 Next

Here we can see what the babynames dataframe looks like. <describe the columns>.

tibbles

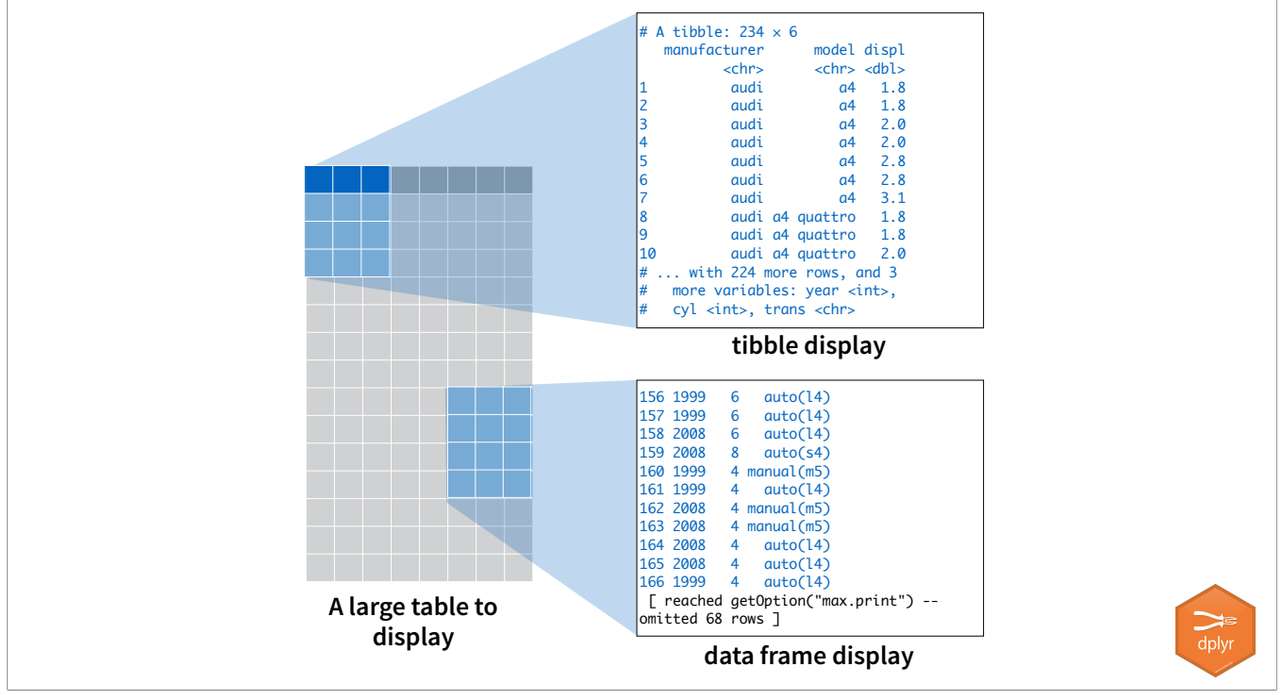
Earlier I mentioned that Hadley views the Tidyverse as a "dialect" of R. What does this really mean?
Well, one way you can see it is that the Tidyverse does not use regular "data.frames" to store data. Instead, they created a new data structure called "tibbles". Tibbles are basically the same as data.frames, but there are some differences.

tibbles

A type of data frame common throughout tidyverse packages.
Tibbles enhance data frames in three ways:

1. **Display** - When you print a tibble, R provides a concise view of the data that fits on one screen
2. **Subsetting** - [always returns a new tibble, [[and $ always return a new vector
3. **No partial matching** - You must use full column names when subsetting

This slide lists the 3 differences between data.frames and tibbles. The only difference that I've ever personally noticed is number 1: the output always prints on a single screen. A Tibble "is-a" data.frame.



You may have actually noticed this. But when you type the name of a large tibble on the console, you only see the first few rows and columns. If you look at a similar-sized data.frame, you see the output somewhere in the middle on your screen.

tibble

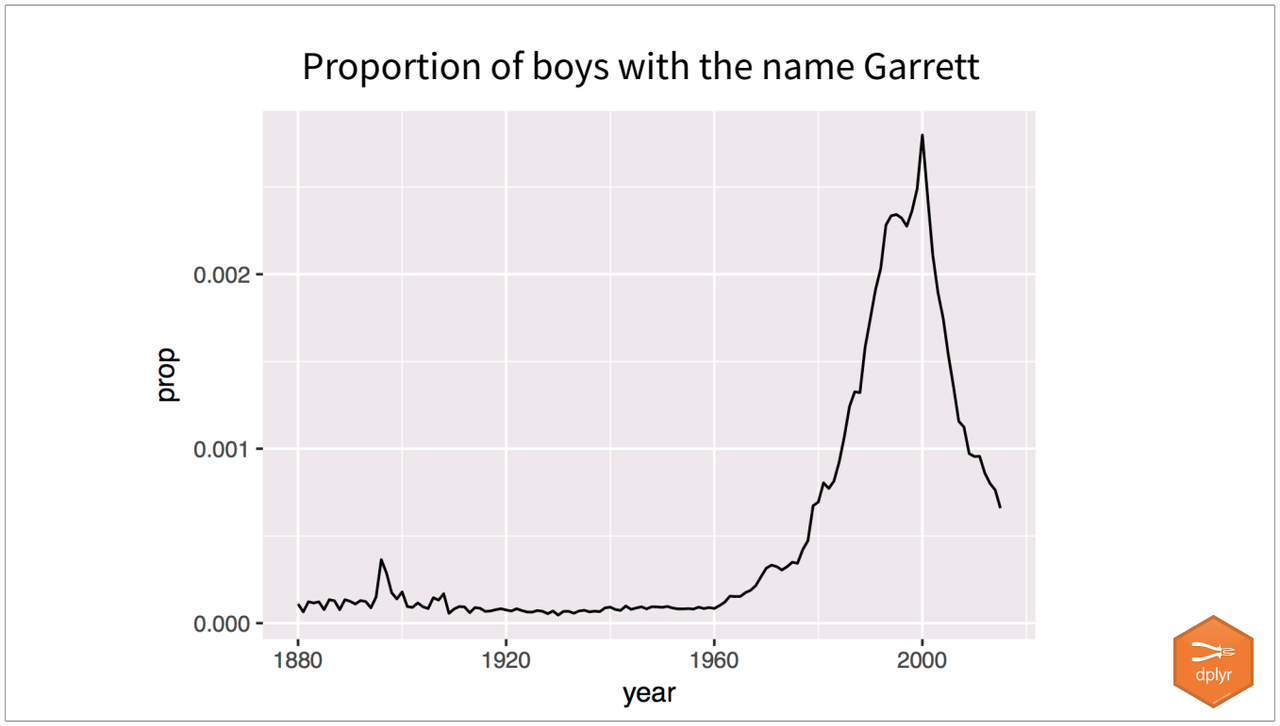
A package with several helper functions for tibbles:

- **as_tibble()** - convert a data frame to a tibble
- **as.data.frame()** - convert a tibble to a data frame
- **tribble()** - make a tibble (transversed)

```
tribble(
   ~x,  ~y,
    1, "a",
    2, "b",
    3, "c")
```

x	y
1	a
2	b
3	c

There are 3 functions for working with tibbles that you might want to know about. The odds are that you won't have to use these yourself (all tidyverse functions accept data.frames and use tibbles internally).



These slides were originally created by Garrett Grolemund, the lead instructor at RStudio. Not surprisingly, he was curious at how popular his name has been over time! Later today, you'll create a graph like this of your own name.

How to isolate?

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081
1881	M	William	8524	0.0787
1881	M	James	5442	0.0503
1881	M	Charles	4664	0.0431
1881	M	Garrett	7	0.0001
1881	M	Gideon	7	0.0001

year	sex	name	n	prop
1880	M	Garrett	13	0.0001
1881	M	Garrett	7	0.0001
…	…	Garrett	…	…

To make this graph from this data.frame (tibble), you need to be able to ***isolate*** *rows that just contain a given name. This is what we'll be working on today.*

dplyr

Now we have our motivating examples for dplyr - let's get learning it!

dplyr

A package that transforms data.

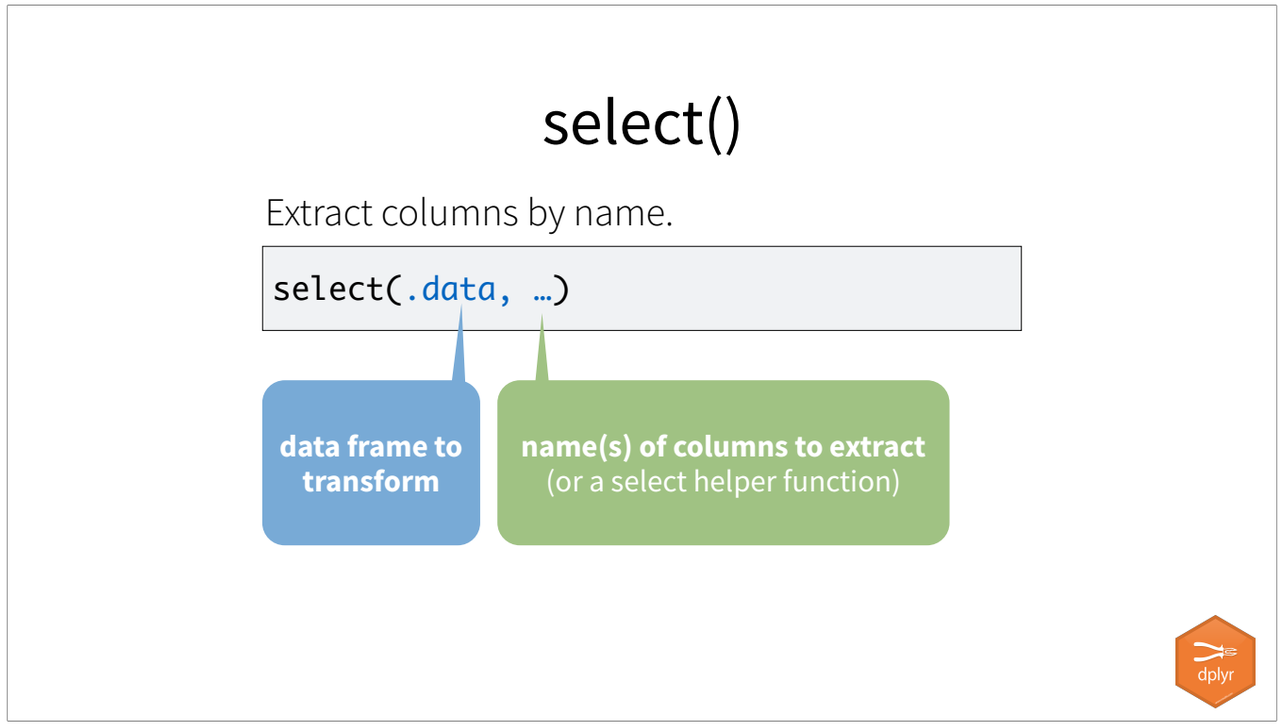
dplyr implements a *grammar* for transforming tabular data.

Isolating data

select() - extract **variables (columns)**
filter() - extract **cases (rows)**
arrange() - reorder **cases**

Earlier we talked about isolating rows that had Garrett's name. There are actually 3 ways to isolate data: columns, rows and (loosely) sorting / reording rows. We're going to start our study of dplyr by learning how to do these tasks with dplyr.

select()



Select is a function that takes a dataframe / tibble as the first parameter. Each additional parameter is the name of a column that you want returned.

select()

Extract columns by name.

```
select(babynames, name, prop)
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

name	prop
John	0.0815
William	0.0805
James	0.0501
Charles	0.0451
Garrett	0.0001
John	0.081

Here's an example of select in action. The first parameter is "babynames". Then we give it the names of two columns we want, and it returns a data.frame with just those columns selected.

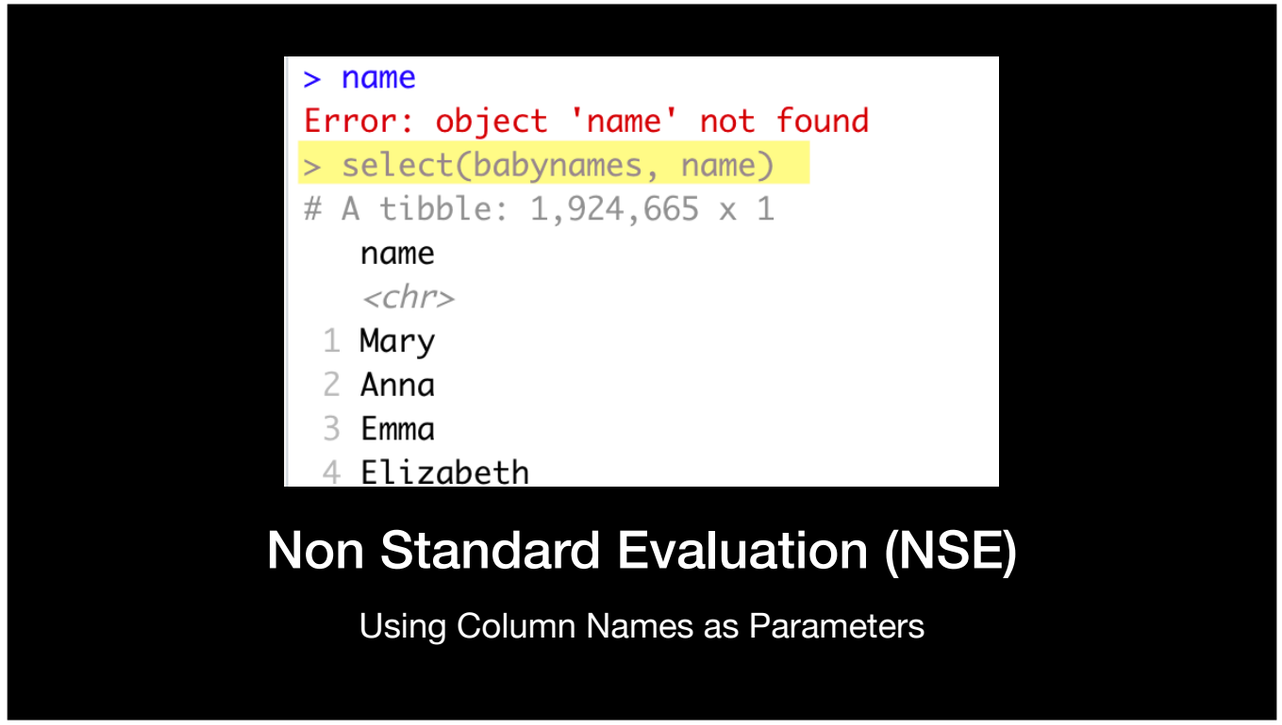
Your Turn 1

Open up **02-dplyr-exercises.Rmd**

Alter the code to select just the **n** column:

```
select(babynames, name, prop)
```

```
select(babynames, n)
#       n
#   <int>
# 1  7065
# 2  2604
# 3  2003
# 4  1939
# 5  1746
# …   …
```



I'd like to make a diversion to talk about a topic in the Tidyverse that you'll sometimes hear mentioned: Non Standard Evaluation (NSE).

In both ggplot2 and dplyr, you can use the name of a column as a parameter. This seems natural, but it's actually a bit magical. Because the name of the column is neither data (it's not a string - there are no quotes around it), nor a variable.

In the code above, you can see I get an error when I type just "name" or "prop" on the console. But yet dplyr knows exactly what to do with those names when I type them in the context of the "select" function.

All I need you to recognize now is that this is how the Tidyverse works - you can use column names in the functions - and it's a bit magical. And the name of the magic that makes it work is NSE.

select() helpers

: - Select range of columns

```
select(storms, storm:pressure)
```

- - Select every column but

```
select(storms, -c(storm, pressure))
```

starts_with() - Select columns that start with…

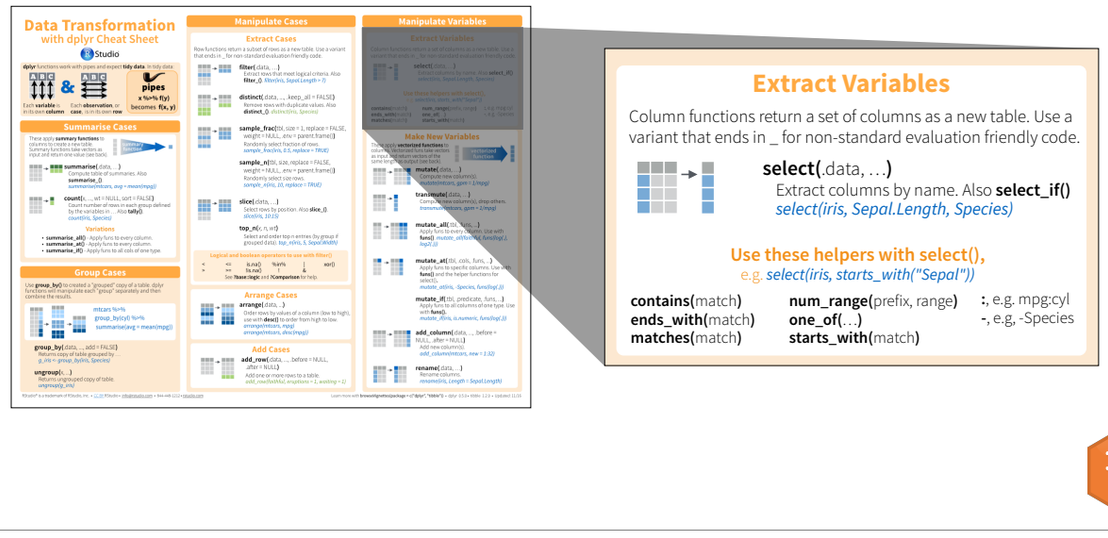
```
select(storms, starts_with("w"))
```

ends_with() - Select columns that end with…

```
select(storms, ends_with("e"))
```

One thing that makes dplyr so popular is that it's very *robust.* Besides just selecting a single column, there's ways to programmatically select just a few columns.

select() helpers



I hope that all of you have the cheatsheet printed out. I want to point out that this part of the cheatsheet has all of this reference material on it.

Quiz

Which of these is NOT a way to select the **name** and **n** columns together?

```
select(babynames, -c(year, sex, prop))
```

```
select(babynames, name:n)
```

```
select(babynames, starts_with("n"))
```

```
select(babynames, ends_with("n"))
```

Spent a minute thinking about - and experimenting in the console - to find the answer to this question

Quiz

Which of these is NOT a way to select the **name** and **n** columns together?

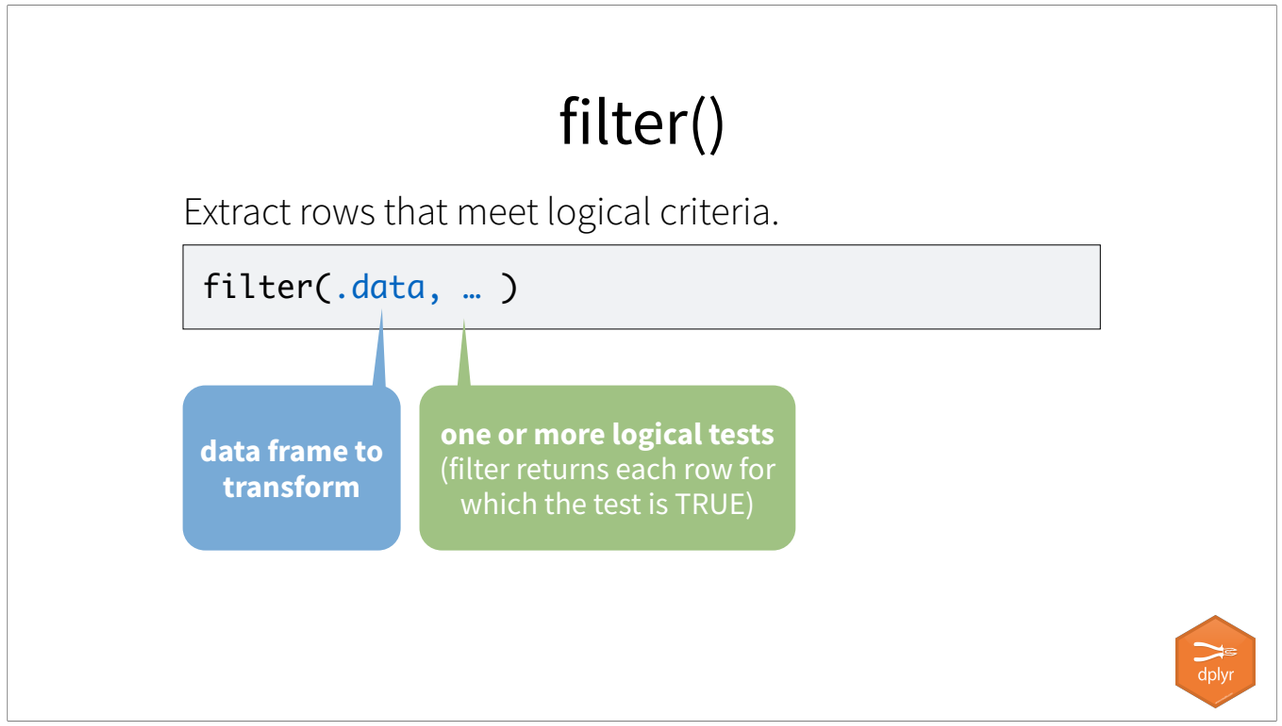
```
select(babynames, -c(year, sex, prop))
```

```
select(babynames, name:n)
```

```
select(babynames, starts_with("n"))
```

```
select(babynames, ends_with("n"))
```

filter()



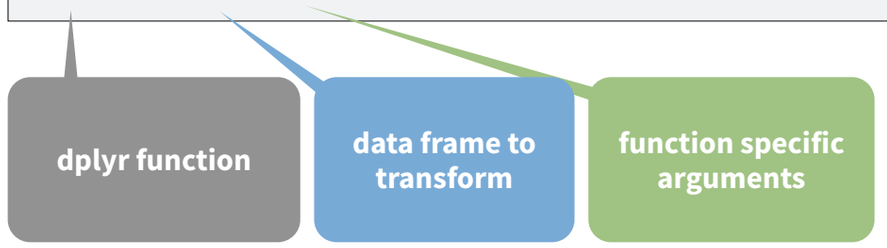
In "select" the first parameter was a dataframe and the next parameters were column names.

In "filter" the first parameter is (again) a dataframe, but the subsequent parameters are *logical tests.* If the tests are true, the row is returned.

common syntax

Each function takes a data frame / tibble as its first argument and returns a data frame / tibble.

```
filter(.data, … )
```



In fact, this pattern is true for all functions in dplyr. We'll see why this is important later, when we talk about pipes. But for now, just focus on the pattern: for each function, the first parameter is data, and then subsequent parameters are more specific things.

filter()

Extract rows that meet logical criteria.

```
filter(babynames, name == "Garrett")
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop
1880	M	Garrett	13	0.0001
1881	M	Garrett	7	0.0001
…	…	Garrett	…	…

And now we know a key function necessary to create the graph we saw earlier. We can create the dataframe we need by writing "filter(babynames, name=="Garrett")"

filter()

Extract rows that meet logical criteria.

```
filter(babynames, name == "Garrett")
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

= sets
(returns nothing)

== tests if equal
(returns TRUE or FALSE)

This is the single most common error when working with dplyr! You need two equal signs to do a logical test. If you only do 1, you will get an error.

Logical tests

?Comparison

`x < y`	Less than
`x > y`	Greater than
`x == y`	Equal to
`x <= y`	Less than or equal to
`x >= y`	Greater than or equal to
`x != y`	Not equal to
`x %in% y`	Group membership
`is.na(x)`	Is NA
`!is.na(x)`	Is not NA

Here's a list of all the possible logical tests you can do with dplyr.
-Draw special attention to !=, %in% and NA.

Your Turn 2

See if you can use the logical operators to manipulate our code below to show:

- All of the names where **prop** is greater than or equal to 0.08
- All of the children named “Sea”
- All of the names that have a missing value for **n**
  (Hint: this should return an empty data set).

This is a test to get you comfortable using logical operators with filter.
These are 3 separate tests.

```
filter(babynames, prop >= 0.08)
#    year   sex    name     n       prop
# 1  1880     M    John  9655 0.08154630
# 2  1880     M William  9531 0.08049899
# 3  1881     M    John  8769 0.08098299
```

```
filter(babynames, name == "Sea")
#    year   sex  name     n         prop
# 1  1982     F   Sea     5 2.756771e-06
# 2  1985     M   Sea     6 3.119547e-06
# 3  1986     M   Sea     5 2.603512e-06
# 4  1998     F   Sea     5 2.580377e-06
```

```
filter(babynames, is.na(n))
#  0 rows
```

Two common mistakes

1. Using `=` instead of `==`

```
filter(babynames, name = "Sea")
filter(babynames, name == "Sea")
```

2. Forgetting quotes

```
filter(babynames, name == Sea)
filter(babynames, name == "Sea")
```

filter()

Extract rows that meet *every* logical criteria.

```
filter(babynames, name == "Garrett", year == 1880)
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop
1880	M	Garrett	13	0.0001

In the last exercise I expected you to write 3 separate lines of code - 1 for each test.
Now I want to point out that you can combine multiple tests in a single call to filter. To do this, separate each test with a comma.

Boolean operators

?base::Logic

`a & b`	and
`a | b`	or
`xor(a,b)`	exactly or
`!a`	not
`a %in% c(a, b)`	one of (in)

There are other ways to combine statements though. With a comma, you essentially do a logical "and". But you can combine multiple tests without a comma, using the &. You can also use a | for or, and so on. Here's a list of all Boolean operators in R. They all work with dplyr.

filter()

Extract rows that meet *every* logical criteria.

```
filter(babynames, name == "Garrett" & year == 1880)
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop
1880	M	Garrett	13	0.0001

Here's an example of using & to find all rows in 1880 for Garrett.

Your Turn 3

Use Boolean operators to alter the code below to return only the rows that contain:

- Girls named Sea
- Names that were used by exactly 5 or 6 children in 1880
- Names that are one of Acura, Lexus, or Yugo

```
filter(babynames, name == "Sea" | name == "Anemone")
```

```
filter(babynames, name == "Sea", sex == "F")
#    year   sex  name     n         prop
# 1  1982     F   Sea     5 2.756771e-06
# 2  1998     F   Sea     5 2.580377e-06
```

```
filter(babynames, n == 5 | n == 6, year == 1880)
#    year   sex    name     n         prop
# 1  1880     F    Abby     6 6.147289e-05
# 2  1880     F  Aileen     6 6.147289e-05
# …     …     …       …     …            …
```

```
filter(babynames, name == "Acura" | name =="Lexus" |name =="Yugo"))
#    year   sex  name     n         prop
# 1  1990     F Lexus    36 1.752932e-05
# 2  1990     M Lexus    12 5.579156e-06
# …     …     …     …     …           …
```

Two more common mistakes

3. Collapsing multiple tests into one

```
filter(babynames, 10 < n < 20)
filter(babynames, 10 < n, n < 20)
```

4. Stringing together many tests (when you could use %in%)

```
filter(babynames, n == 5 | n == 6 | n == 7 | n == 8)
filter(babynames, n %in% c(5, 6, 7, 8))
```

```
filter(babynames, name == "Sea", sex == "F")
#    year   sex  name     n         prop
# 1  1982     F   Sea     5 2.756771e-06
# 2  1998     F   Sea     5 2.580377e-06
```

```
filter(babynames, n == 5 | n == 6, year == 1880)
#    year   sex    name     n         prop
# 1  1880     F    Abby     6 6.147289e-05
# 2  1880     F  Aileen     6 6.147289e-05
# …     …     …       …     …            …
```

```
filter(babynames, name %in% c("Acura", "Lexus", "Yugo"))
#    year   sex  name     n         prop
# 1  1990     F Lexus    36 1.752932e-05
# 2  1990     M Lexus    12 5.579156e-06
# …     …     …     …     …            …
```

Here we see an example where using %in% is clearer than using multiple "or"s.

arrange()

Now let's talk about arrange(), the final function for isolating data. Arrange lets you sort a dataframe by the values in columns. So it helps you "isolate" data by finding the max and min values.

arrange()

Order rows from smallest to largest values.

```
arrange(.data, …)
```

data frame to transform

one or more columns to order by (additional columns will be used as tie breakers)

Like all the dplyr functions, the first argument to arrange is a dataframe. And then you give one or more columns that you want to order by.

arrange()

Order rows from smallest to largest values.

```
arrange(babynames, n)
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

year	sex	name	n	prop
1880	M	Garrett	13	0.0001
1880	M	Charles	5348	0.0451
1880	M	James	5927	0.0501
1881	M	John	8769	0.081
1880	M	William	9532	0.0805
1880	M	John	9655	0.0815

Here's an example of arranging babynames by n. This example uses a subset of the babynames. You can see that when we arrange by n, Garrett goes up to the top, because n=13 for Garrett in 1880.

Your Turn 4

Arrange babynames by **n**. Add **prop** as a second (tie breaking) variable to arrange by.

Can you tell what the smallest value of **n** is?

Here is your first assignment using arrange.

```
arrange(babynames, n, prop)
#    year   sex        name     n         prop
#  1  2007     M       Aaban     5 2.259872e-06
#  2  2007     M      Aareon     5 2.259872e-06
#  3  2007     M       Aaris     5 2.259872e-06
#  4  2007     M         Abd     5 2.259872e-06
#  5  2007     M  Abdulazeez     5 2.259872e-06
#  6  2007     M   Abdulhadi     5 2.259872e-06
#  7  2007     M  Abdulhamid     5 2.259872e-06
#  8  2007     M  Abdulkadir     5 2.259872e-06
#  9  2007     M Abdulraheem     5 2.259872e-06
# 10  2007     M  Abdulrahim     5 2.259872e-06
# ... with 1,858,679 more rows
```

desc()

Changes ordering to largest to smallest.

```
arrange(babynames, desc(n))
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1881	M	John	8769	0.081
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001

This is probably the most important function for you to know about with ?arrange. Arrange always goes from lowest to highest. To change this, you need to wrap the column you want inside the function "desc".

Your Turn 5

Use **desc()** to find the names with the highest **prop**.

Then, use **desc()** to find the names with the highest **n**.

```
arrange(babynames, desc(prop))
#     year   sex    name     n       prop
#  1  1880     M    John  9655 0.08154630
#  2  1881     M    John  8769 0.08098299
#  3  1880     M William  9531 0.08049899
#  4  1883     M    John  8894 0.07907324
#  5  1881     M William  8524 0.07872038
#  6  1882     M    John  9557 0.07831617
#  7  1884     M    John  9388 0.07648751
#  8  1882     M William  9298 0.07619375
#  9  1886     M    John  9026 0.07582198
# 10  1885     M    John  8756 0.07551791
# ... with 1,858,679 more rows
```

```
arrange(babynames, desc(n))
#     year   sex    name     n       prop
#  1  1947     F   Linda 99680 0.05483609
#  2  1948     F   Linda 96211 0.05521159
#  3  1947     M   James 94763 0.05102057
#  4  1957     M Michael 92726 0.04238659
#  5  1947     M  Robert 91646 0.04934237
#  6  1949     F   Linda 91010 0.05184281
#  7  1956     M Michael 90623 0.04225479
#  8  1958     M Michael 90517 0.04203881
#  9  1948     M   James 88588 0.04969679
# 10  1954     M Michael 88493 0.04279403
# ... with 1,858,679 more rows
```

%>%

(Now also |>)

Now we're getting into one of the most important parts of the Tidyverse: the pipe operator. This has really revolutionized how many people approach programming in R. The Pipe operator is not strictly part of the dplyr package. It's part of its own separate package called magrittr. But dplyr uses it extensively, and is one reason why it has become so popular.

%>% became so popular that the R language eventually added in its own version of it, which is |>. Both work pretty much the same. But these slides are old, which is why they use %>%!

Steps

```
boys_2015 <- filter(babynames, year == 2015, sex == "M")
boys_2015 <- select(boys_2015, name, n)
boys_2015 <- arrange(boys_2015, desc(n))
boys_2015
```

1. Filter babynames to just boys born in 2015
2. Select the *name* and *n* columns from the result
3. Arrange the results so the n column is descending

To understand the pipe operator, let's start with a sample analysis: seeing the most popular male name in 2015. There are a few ways to do this in code. Here's 1 way.

Steps

```
boys_2015 <- filter(babynames, year == 2015, sex == "M")
boys_2015 <- select(boys_2015, name, n)
boys_2015 <- arrange(boys_2015, desc(n))
boys_2015
```

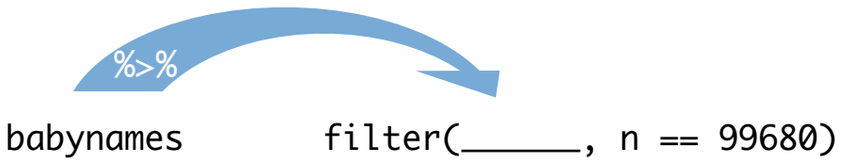
In this approach, we create a variable that will contain our final result. And we keep on storing / updating its value each step of the way.

Steps

```
arrange(select(filter(babynames, year == 2015,
  sex == "M"), name, n), desc(n))
```

Another way to do it, which is more confusing, is to do it all in 1 line. So the result of the filter is passed as the first parameter to the select. Which is in turn passed as the first parameter to the arrange.

The pipe operator %>%



Passes result on left into first argument of function on right.
So, for example, these do the same thing. Try it.

```
filter(babynames, n == 99680)
babynames %>% filter(n == 99680)
```

The pipe operator gives us a third way to do this analysis. Here's how it works: the pipe takes whatever is on its left hand side, and puts it into the first argument of the function on the right.

So you already know that filter(babynames, n==99680) will filter babynames to only rows that have n==99680. But the second example, babynames %>% filter(n==99680) does the same thing.

Pipes

```
babynames
boys_2015 <- filter(babynames, year == 2015, sex == "M")
boys_2015 <- select(boys_2015, name, n)
boys_2015 <- arrange(boys_2015, desc(n))
boys_2015
```

```
babynames %>%
  filter(year == 2015, sex == "M") %>%
  select(name, n) %>%
  arrange(desc(n))
```

In the case of a 1-line analysis, using the pipe isn't very useful. But when the analysis has multiple steps, the pipe can make things much easier to read.

People often read the %>% operator as "then". (go through both analyses).

Note that the pipe operator also reduces the amount of duplicate / extraneous text on the screen. You don't need to type or see "babynames" over and over again.

```
foo_foo <- little_bunny()
```

```
foo_foo %>%
  hop_through(forest) %>%
  scoop_up(field_mouse) %>%
  bop_on(head)
```

VS.

```
foo_foo2 <- hop_through(foo_foo, forest)
foo_foo3 <- scoop_up(foo_foo2, field_mouse)
bop_on(foo_foo3,  head)
```

This is a cute example that compares and contrasts the pipe operator and the imperative style of coding using a famous nursery rhyme.

Shortcut to type %>%

Cmd + Shift + M (Mac)

Ctrl + Shift + M (Windows)

Your Turn 6

Use **%>%** to write a sequence of functions that:

1. Filter babynames to just the girls that were born in 2015
2. Select the **name** and **n** columns
3. Arrange the results so that the most popular names are near the top.

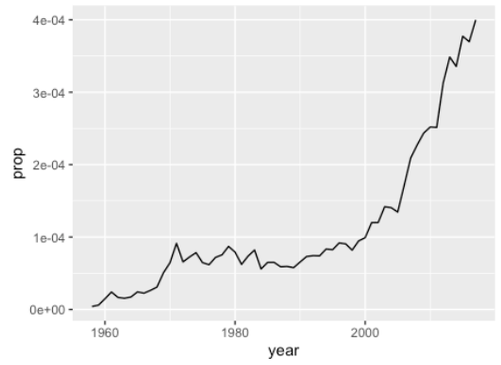
```
babynames %>%
  filter(year == 2015, sex == "F") %>%
  select(name, n) %>%
  arrange(desc(n))
#         name     n
#  1      Emma 20355
#  2    Olivia 19553
#  3    Sophia 17327
#  4       Ava 16286
#  5  Isabella 15504
#  6       Mia 14820
#  7   Abigail 12311
#  8     Emily 11727
#  9 Charlotte 11332
# 10    Harper 10241
# ... with 18,983 more rows
```

Exam

1. Filter babynames to just the rows that contain your **name** and your **sex**
2. Select just the columns that will appear in your graph (not strictly necessary, but useful practice)
3. Plot the results as a line graph with **year** on the x axis and **prop** on the y axis

This is a final exam for 3 things: the pipe operator, the dplyr functions for isolating data, and ggplot2.

```
babynames %>%
  filter(name == "Ari", sex == "M") %>%
  select(year, prop) %>%
  ggplot() +
  geom_line(mapping = aes(year, prop))
```



Here's my own answer.

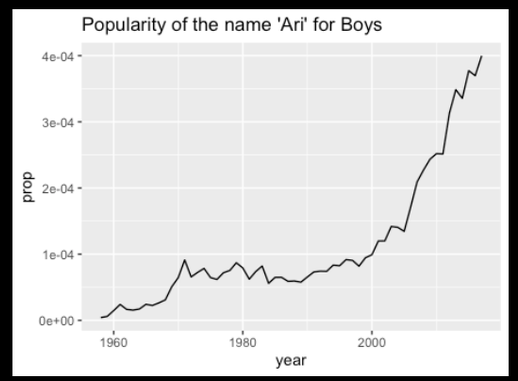
(Walk through the solution).

I now want to show you two tricks that can help you make this plot more professional: adding a title to the chart, and changing the y-axis scale so that instead of being in scientific notation, it shows full numbers.

Adding a Title

- Google *"How to add a title in ggplot2"*

```
p + ggitle("Popularity of ...")
```



In ggplot2, the title is just another layer. The function for that layer is "ggtitle", so we add a title by creating the chart like normal, and then adding "+ ggitle("popularity of ")"

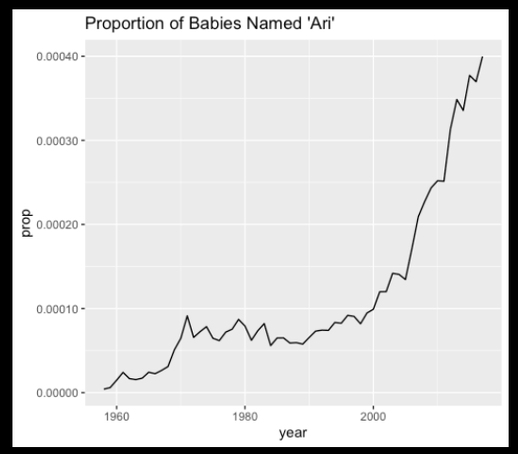
You can write this down on your notes form if you like. But probably more important to write down is this: Whenever you have a specific technical question about ggplot (like this), just google the question. A ton has been written about ggplot, and google will probably take you to a page that has a great answer.

Removing Scientific Notation

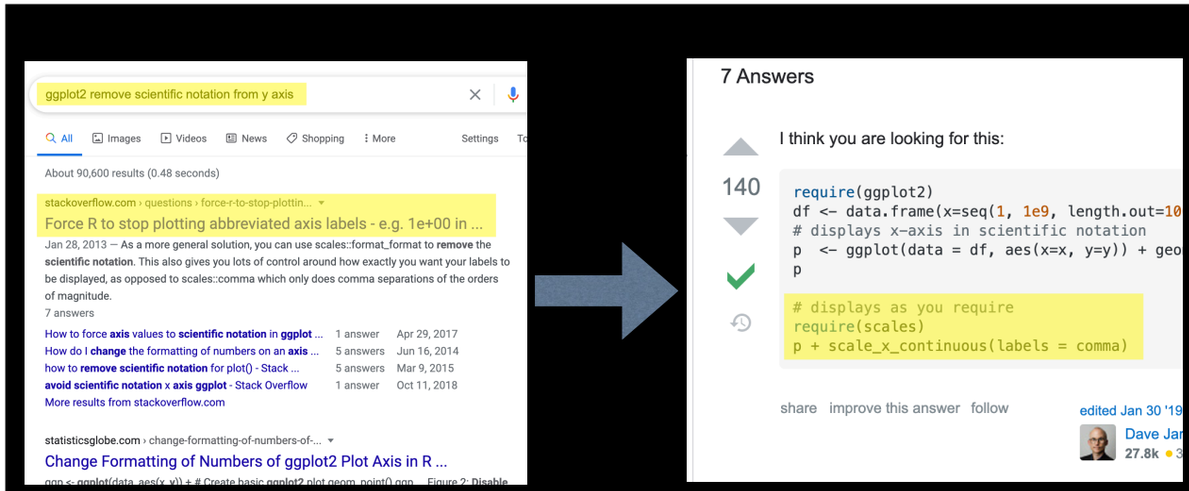
- Google *"ggplot2 remove scientific notation from y axis"*

```
library(scales)

p +
scale_y_continuous(labels=comma)
```



The same is true for removing scientific notation from the y-axis. This solution is more complicated, and I never remember how to do it, so I always google it. You need to load a separate library (scales), and then add a layer for the y-axis that overrides the default scale. Here it's scale_y_continuous, and the labels parameter is set to "comma"



The "Google -> Stack Overflow" Loop

"ggplot2 remove scientific notation from y axis"

Generally speaking, whenever you have a Tidyverse question, if you put it into google, you'll wind up at a site called StackOverflow. It's a technical Q&A site that you might have already heard about. But generally speaking, someone has probably already asked and answered your question.

What are the most popular names?

We've now seen how to calculate how an individual name has increased and decreased in popularity over time. But there are still important questions we can't answer, such as:

- The total number of children name "Ari" (this is different than the total number of rows)
- How many male and female babies were named "Ari", and how has that changed over time?

To answer these questions, we need to learn some new functions in dplyr

Deriving information

summarise() - summarise **variables**
group_by() - group **cases**
mutate() - create new **variables**

There are three functions in dplyr that can help you derive information about a dataframe: summarize, group_by and mutate. Note that you can spell summarize with either a "z" or an "s". Let's look at each of them in turn.

summarise()

summarise()

Compute table of summaries.

```
babynames %>% summarise(total = sum(n), max = max(n))
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

total	max
348120517	99686

Summarize creates a table that contains summary statistics. Here we're using the functions "sum" and "max" to count the total number of babies in the dataset, as well as the maximum number of babies with a given name, gender and year.

Summary functions

Take a vector as input.
Return a single value as output.

The dplyr cheat sheet has a list of popular summary functions. We're going to cover just a few of them here.

Your Turn 7

Use summarise() to compute three statistics *about the data*:

1. The first (minimum) year in the dataset
2. The last (maximum) year in the dataset
3. The total number of children represented in the data

```
babynames %>%
  summarise(first = min(year),
            last = max(year),
            total = sum(n))
#   first  last     total
# 1  1880  2017 340851912
```

Your Turn 8

Filter the rows where **name == "Khaleesi"**. Then use summarise() and a summary function to find:

1. The total number of children named Khaleesi
2. The first **year** Khaleesi appeared in the data

```
babynames %>%
  filter(name == "Khaleesi") %>%
  summarise(total = sum(n), first = min(year))
#   total first
# 1  1964  2011
```

First we filter the tibble to just have entries for the name "Khaleesi". Then we use "summarize" to generate two statistics about the data: the total number of babies, and the first year in the dataset.

n()

The number of rows in a dataset/group

```
babynames %>% summarise(n = n())
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

n
1924665

Most of the summary functions have very simple, descriptive names like "min" or "sum". But there two functions with less obvious names that you should be aware of.

The first is n(), which returns the total number of rows in a dataframe. You'll often see n=n(), which captures the value returned by the function n() into a variable named "n".

So here you can see that there are about 1.9M rows in this dataframe

n_distinct()

The number of distinct values in a variable

```
babynames %>% summarise(n = n(), nname = n_distinct(name))
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

n	nname
1924665	97310

There is also a function called "n_distinct", which returns the number of distinct elements in a column.
So while this dataframe has 1.9M rows, there are only about 97k distinct names.

Grouping cases

Next we'll talk about how to generate statistics for sub-groups of data within a data frame. This is one of the most important topics we'll cover today. In my opinion, group_by is what makes dplyr so powerful and popular.

group_by()

Groups cases by common values of one or more columns.

```
babynames %>%
  group_by(sex)
```

```
Source: local data frame [1,825,433 x 5]
Groups: sex [2]

   year   sex      name     n       prop
  <dbl> <chr>     <chr> <int>      <dbl>
1  1880     F      Mary  7065 0.07238359
```

In the babynames dataframe it seems natural to generate statistics separately for boys and girls. Right now we can calculate the most popular name overall. But it seems natural to want to calculate the most popular name separately for boys and girls.

The command "group_by" lets us do this. If we call "group_by" on a data frame, and do nothing else, you'll see this new line that says "groups".

group_by()

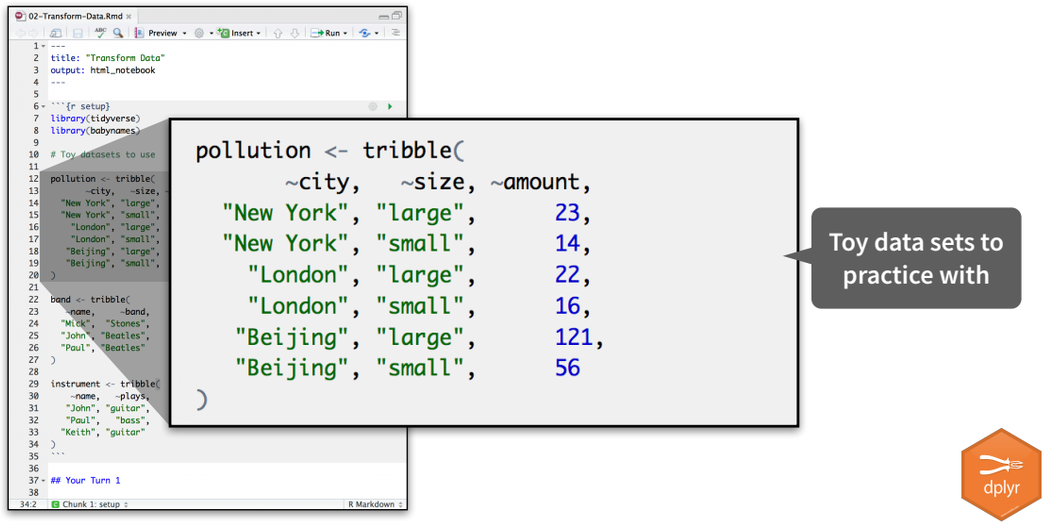
Groups cases by common values.

```
babynames %>%
  group_by(sex) %>%
  summarise(total = sum(n))
```

sex	total
F	167070477
M	170064949

Calling "group_by" by itself doesn't do very much. It's most often followed immediately by a call to summarize. Here we see that grouping by sex, and then summarizing by counting n, tells us the number of males and females in the dataframe.

Transform Data Notebook



Here we have a new toy dataset to work with. This dataset is called "pollution", and it has 3 columns: city, size and amount. We're going to start with this dataframe because it makes it easier to see how group_by works.

```
pollution <- tribble(
       ~city,   ~size, ~amount,
  "New York", "large",      23,
  "New York", "small",      14,
    "London", "large",      22,
    "London", "small",      16,
   "Beijing", "large",      121,
   "Beijing", "small",      56
)
```

pollution

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

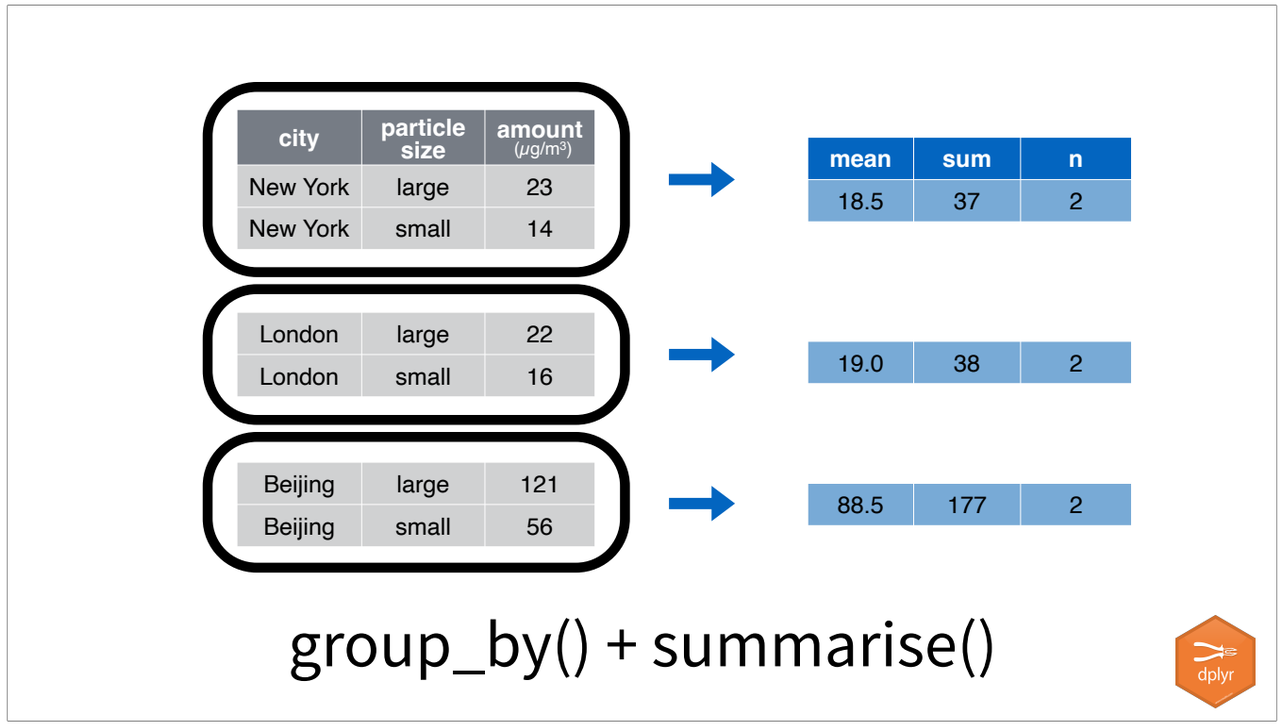
Here you can see the code to create the dataframe, and how the resulting tibble looks in R. Note that we have two readings for each city. The first reading for each city is for large particle pollution. The second reading for each city is for small particle pollution.

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

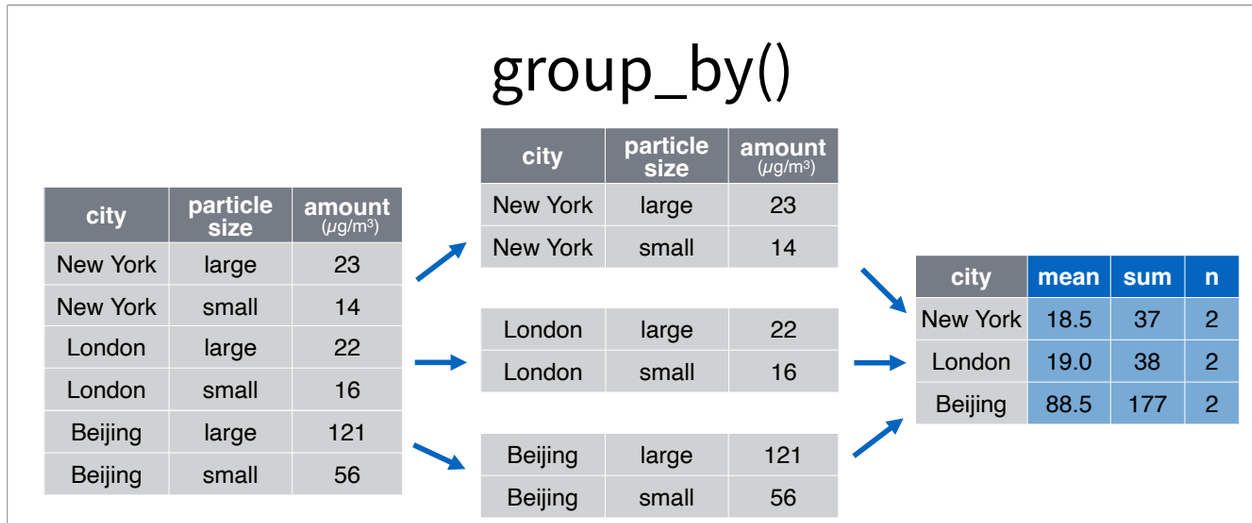
mean	sum	n
42	252	6

```
pollution %>%
 summarise(mean = mean(amount), sum = sum(amount), n = n())
```

I want to reiterate: we already know how to generate summary statistics for this dataframe. Here we're calculating the mean and sum of the pollution. Just for reference, we're also showing the number of rows.



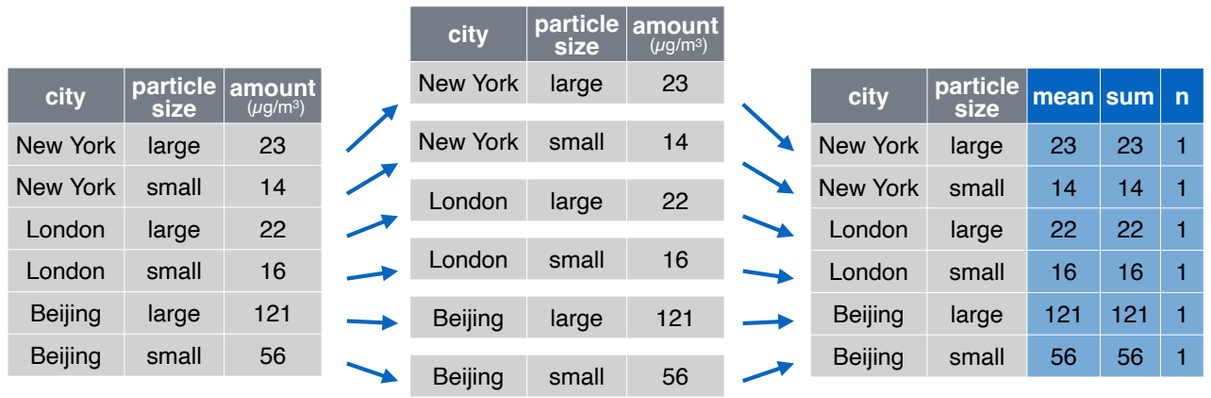
Calculating the total air pollution in the world is interesting. But it also makes sense that you'd want to calculate pollution on a per-city basis. That's what group_by + summarize let's you do.




```
pollution %>%
  group_by(city) %>%
  summarise(mean = mean(amount), sum = sum(amount), n = n())
```

When you do group_by + summarize, the calculations are computed per group. The final result that you get combines those per-group summary statistics. Here we see the actual code, and the result, that you get calculating the sum of per-city air pollution. Note that here, n=2 for all the rows in the final data frame.

group_by()



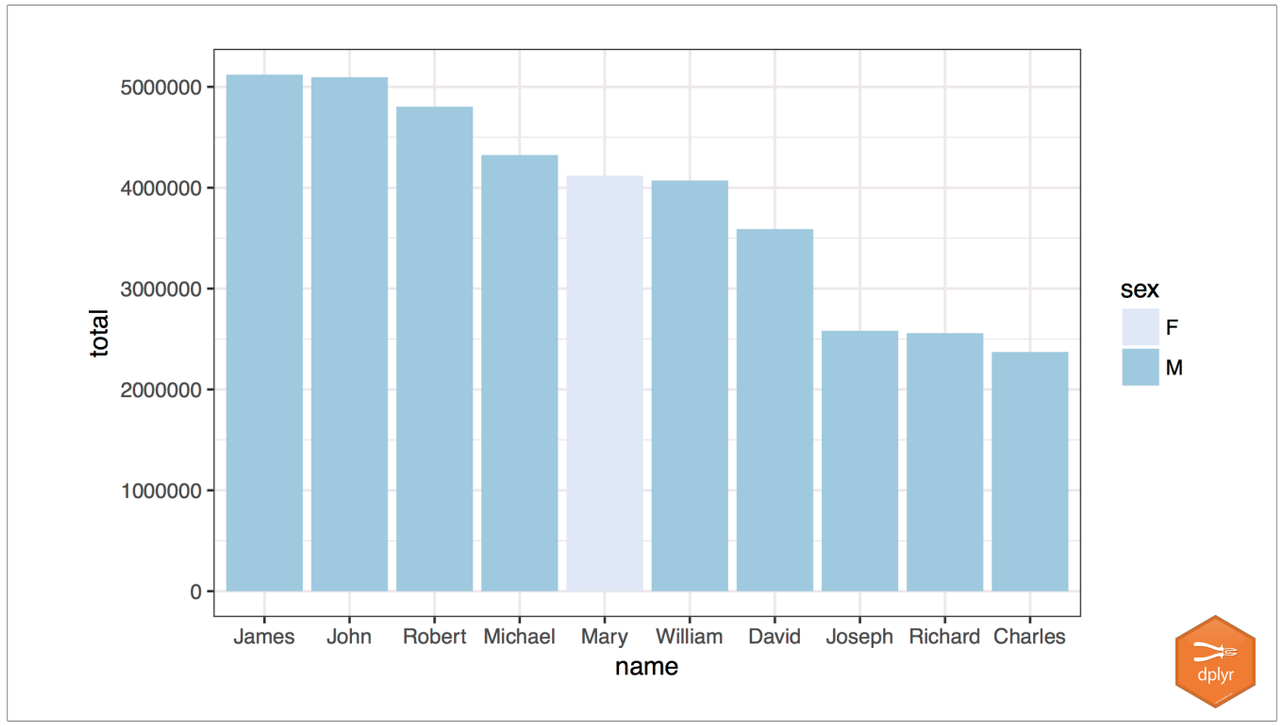

```
pollution %>%
  group_by(city, size) %>%
  summarise(mean = mean(amount), sum = sum(amount), n = n())
```

Note that each city has two readings: one for large particles, and one for small particles. So if we do a group_by for both city and size, we get a funny result. There's only one row in each group. So both the initial and final dataframe have 6 rows. And the mean and sum are the same number.

Your Turn 9

Use **group_by()**, **summarise()**, and **arrange()** to display the ten most popular names. Compute popularity as the total number of children of a single gender given a name.

```
babynames %>%
  group_by(name, sex) %>%
  summarise(total = sum(n)) %>%
  arrange(desc(total))
#       name   sex   total
#  1   James     M 5120990
#  2    John     M 5095674
#  3  Robert     M 4803068
#  4 Michael     M 4323928
#  5    Mary     F 4118058
#  6 William     M 4071645
#  7   David     M 3589754
#  8  Joseph     M 2581785
#  9 Richard     M 2558165
# 10 Charles     M 2371621
# ... with 105,376 more rows
```



Here you can see the results imagined as a pretty advanced ggplot. You can see that most of the most popular names are, surprisingly, male.

```
babynames %>%
  group_by(name, sex) %>%
  summarise(total = sum(n)) %>%
  arrange(desc(total)) %>%
  ungroup() %>%
  slice(1:10) %>%
  ggplot() +
    geom_col(mapping = aes(x = fct_reorder(name,
      desc(total)), y = total, fill = sex)) +
    theme_bw() +
    scale_fill_brewer() +
    labs(x = "name")
```

This is the code to generate that plot. I don't want to get into it in too much detail. But it's worth pointing out that it's common, when focusing on a single plot, to get very complicated and detailed in your code.

Your Turn 10

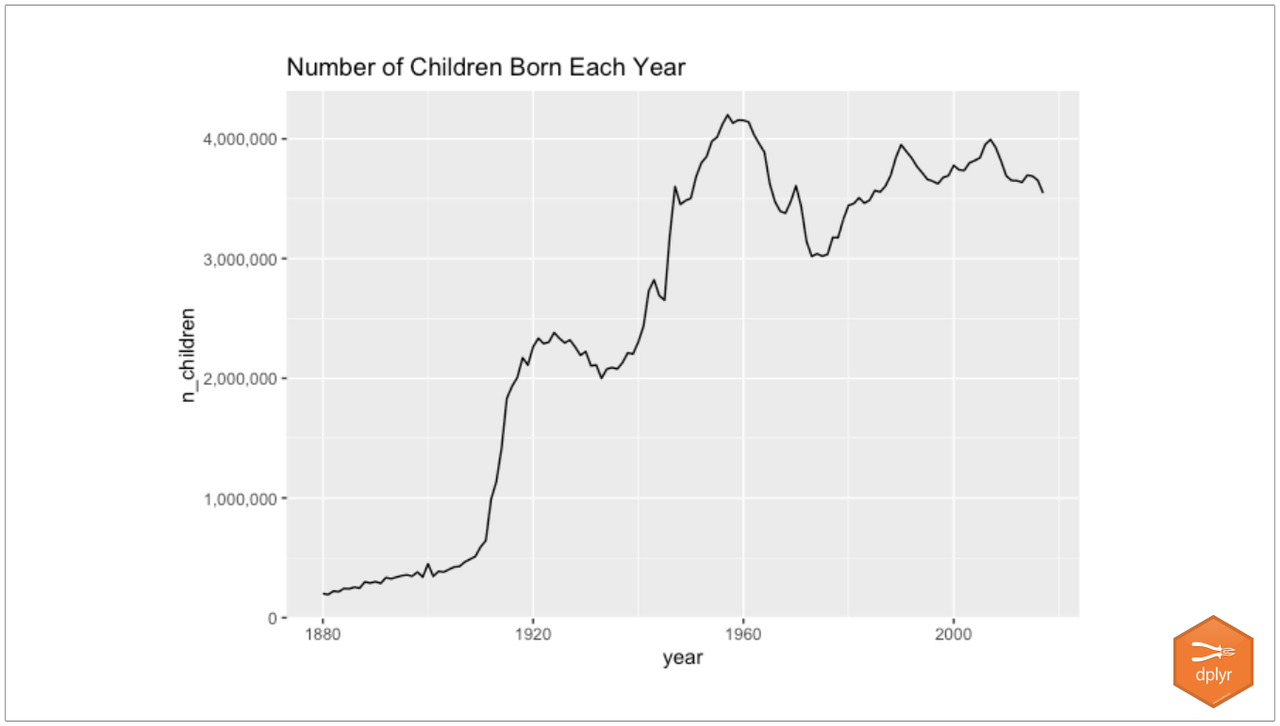
Use grouping to calculate **number of children born each year**.

Plot the results as a line graph.

```
library(scales)

babynames %>%
    group_by(year) %>%
    summarise(n_children = sum(n)) %>%
    ggplot() +
        geom_line(mapping = aes(x = year, y = n_children)) +
        scale_y_continuous(label=comma) +
        ggtitle("Number of Children Born Each Year")
```

Because we want a number per year, we have to first group_by year. And then, for each year, we sum up n to get the number of children.

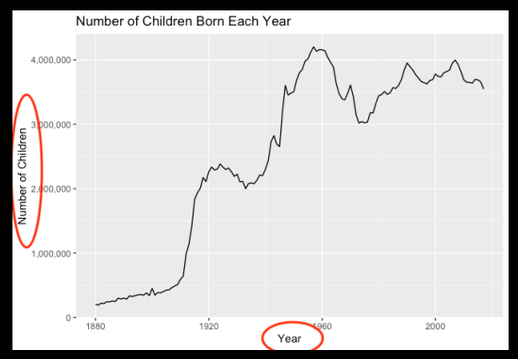


This is the graph the code produced. I want to point out a problem: ggplot2 uses column names for the axis. But "n_children" is an awkward label. Let's look at how to override the default.

Custom Scale Labels

- Google *"ggplot2 change axis labels"*

```
p +
  labs(x = "Year", y = "Number of Children")
```



Again, I want to reiterate that google is a great place to ask these questions. There is a function called "labs" (for labels) that you can use to override the default labels. Like everything else with ggplot2, you use a + to "add" it to the graph.

ungroup()

Removes grouping criteria from a data frame.

```
babynames %>%
  group_by(name, sex) %>%
  summarise(total = sum(n)) %>%
  arrange(desc(total))
#       name   sex   total
#  1   James     M 5120990
#  2    John     M 5095674
#  3  Robert     M 4803068
#  4 Michael     M 4323928
#  5    Mary     F 4118058
```

The opposite of group_by is a function called "ungroup". Note that "ungroup" is called, automatically, each time you call "summarize".

ungroup()

Removes grouping criteria from a data frame.

```
babynames %>%
  group_by(name, sex) %>%
  ungroup() %>%
  summarise(total = sum(n)) %>%
  arrange(desc(total))
#       total
# 1 340851912
```

This is a function that you rarely need to call on its own. Almost always, in practice, you call group_by right before you call summarize. I'm mentioning it mostly for convenience. But here's an example of calling ungroup manually. Here total is the total number of children, despite us first having a group_by.

mutate()

Mutate is a function that lets us add a column to a dataframe. This is especially useful if we want to add a new column that has a number derived from another column.

mutate()

Create new columns.

```
babynames %>%
  mutate(percent = round(prop*100, 2))
```

babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop	percent
1880	M	John	9655	0.0815	8.15
1880	M	William	9532	0.0805	8.05
1880	M	James	5927	0.0501	5.01
1880	M	Charles	5348	0.0451	4.51
1880	M	Garrett	13	0.0001	0.01
1881	M	John	8769	0.081	8.1

Here's an example of using mutate. While it's nice to know the proportion of babies with a certain name, you might also want to know the percent. We can calculate this by calling mutate, and then setting percent=round(prop*100, 2)). Note that this doesn't change the prop column at all.

mutate()

Create new columns.

```
babynames %>%
  mutate(percent = round(prop*100, 2), nper = round(percent))
```

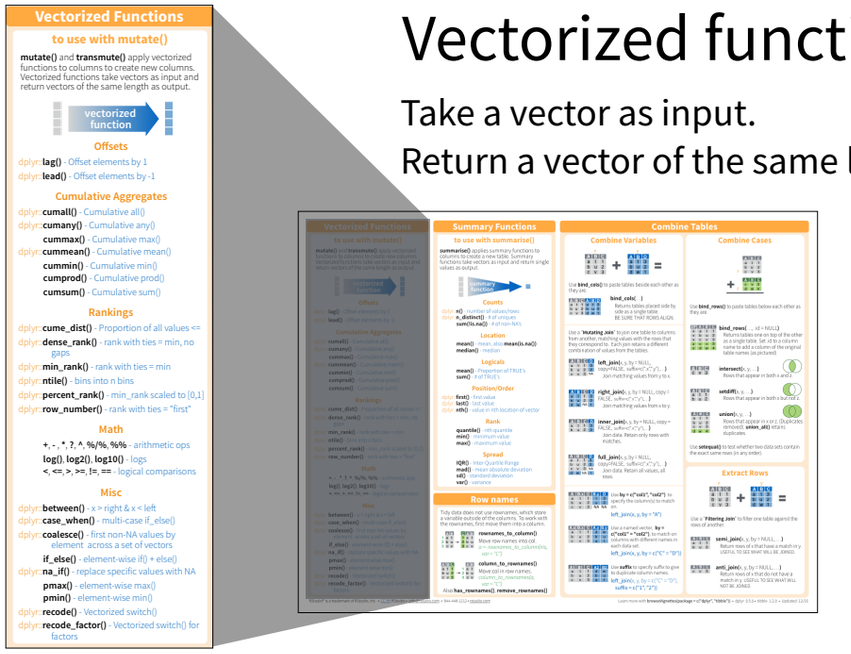
babynames

year	sex	name	n	prop
1880	M	John	9655	0.0815
1880	M	William	9532	0.0805
1880	M	James	5927	0.0501
1880	M	Charles	5348	0.0451
1880	M	Garrett	13	0.0001
1881	M	John	8769	0.081

→

year	sex	name	n	prop	percent	nper
1880	M	John	9655	0.0815	8.15	8
1880	M	William	9532	0.0805	8.05	8
1880	M	James	5927	0.0501	5.01	5
1880	M	Charles	5348	0.0451	4.51	5
1880	M	Garrett	13	0.0001	0.01	0
1881	M	John	8769	0.081	8.1	8

We can add multiple columns at a time with mutate by separating the column names with a comma. Here we add a second column, nper, that's the rounded version of percent.



Vectorized functions

Take a vector as input.

Return a vector of the same length as output.

As you can see, we often want to use vectorized functions with mutate - that means functions that act on an entire column. Here's a list of common functions that people often use with mutate.

Your Turn 11

Dataframe **weights** lists the name and weight (in pounds) of professional martial artists. Add a column **kgs** that lists their weight to kilograms.

Hint: multiply the **lbs** column by .45 to get the number of kilograms

```
> weights
# A tibble: 5 x 2
  name     lbs
  <chr>  <dbl>
1 Stipe    255
2 Jan      205
```

```
weights %>%
  mutate(kgs = lbs *.45)
```

```
> weights %>%
+     mutate(kgs = lbs *.45)
# A tibble: 5 x 3
  name     lbs   kgs
  <chr>  <dbl> <dbl>
1 Stipe    255 115.
2 Jan      205  92.2
3 Israel   185  83.2
4 Karamu   170  76.5
5 Khabib   150  67.5
```

We mutate the dataframe to say that the new column, **kgs**, is equal to the value of each value in the **lbs** columnn times .45.

Recap: Single table verbs

Extract variables / columns with **select()**

Extract cases / rows with **filter()**

Arrange rows, with **arrange()**.

Make tables of summaries with **summarise()**.

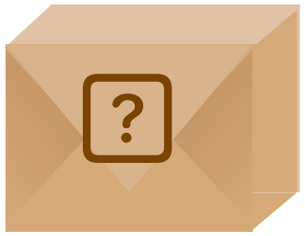
Make new variables, with **mutate()**.

Here's a summary of everything we've learned so far.

Joining Datasets

You now know pretty much everything you need to know about how to manipulate and derive information from dataframes / tibbles. But the last thing you need to know how to do is join multiple dataframes together. That's what we're going to cover now.

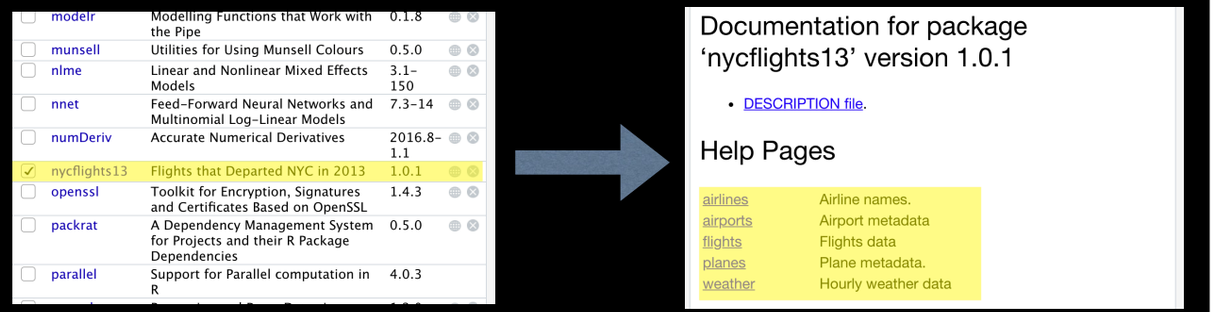
nycflights13



Data about every flight that departed La Guardia, JFK, or Newark airports in 2013

```
# install.packages("nycflights13")
library(nycflights13)
```

To study joins, we're going to work with a new dataset: nycflights13. This has information about each flight that departed New York City in 2013.



Package Description

5 tables

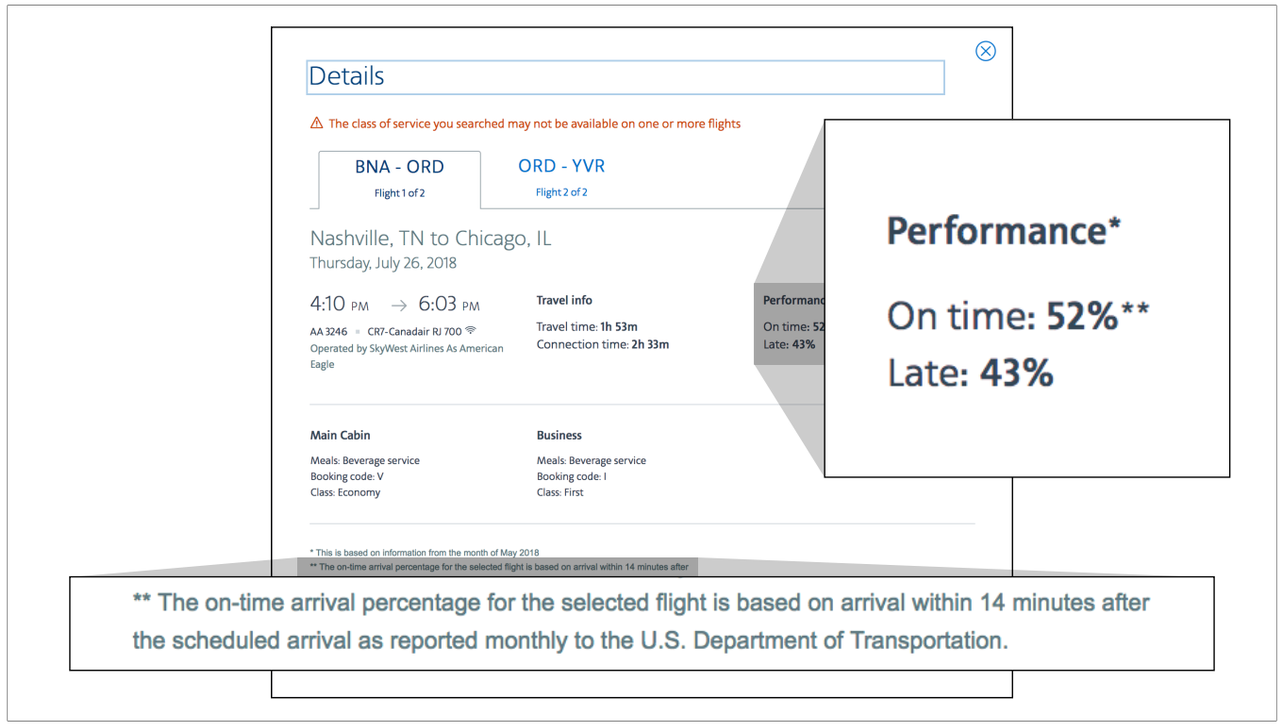
If you click on the package name in RStudio, you'll see the package's help file. It says that there are 5 tables in this package. The main table is the flights table, that has a list of each flight. But there are also specific tables that have detailed information on the airlines, airports, planes and weather.

Flights

```
View(flights)
```

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
2013	1	1	517	515	2	830	819	11	UA	1545	N14228	EWR	IAH	227	1400	5	15	2013-01-01 05:00:00
2013	1	1	533	529	4	850	830	20	UA	1714	N24211	LGA	IAH	227	1416	5	29	2013-01-01 05:00:00
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00
2013	1	1	544	545	-1	1004	1022	-18	B6	725	N804JB	JFK	BQN	183	1576	5	45	2013-01-01 05:00:00
2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116	762	6	0	2013-01-01 06:00:00
2013	1	1	554	558	-4	740	728	12	UA	1696	N39463	EWR	ORD	150	719	5	58	2013-01-01 05:00:00
2013	1	1	555	600	-5	913	854	19	B6	507	N516JB	EWR	FLL	158	1065	6	0	2013-01-01 06:00:00
2013	1	1	557	600	-3	709	723	-14	EV	5708	N829AS	LGA	IAD	53	229	6	0	2013-01-01 06:00:00
2013	1	1	557	600	-3	838	846	-8	B6	79	N593JB	JFK	MCO	140	944	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	753	745	8	AA	301	N3ALAA	LGA	ORD	138	733	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	849	851	-2	B6	49	N793JB	JFK	PBI	149	1028	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	853	856	-3	B6	71	N657JB	JFK	TPA	158	1005	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	924	917	7	UA	194	N29129	JFK	LAX	345	2475	6	0	2013-01-01 06:00:00
2013	1	1	558	600	-2	923	937	-14	UA	1124	N53441	EWR	SFO	361	2565	6	0	2013-01-01 06:00:00
2013	1	1	559	600	-1	941	910	31	AA	707	N3DUAA	LGA	DFW	257	1389	6	0	2013-01-01 06:00:00
2013	1	1	559	559	0	702	706	-4	B6	1806	N708JB	JFK	BOS	44	187	5	59	2013-01-01 05:00:00
2013	1	1	559	600	-1	854	902	-8	UA	1187	N76515	EWR	LAS	337	2227	6	0	2013-01-01 06:00:00
2013	1	1	600	600	0	851	858	-7	B6	371	N595JB	LGA	FLL	152	1076	6	0	2013-01-01 06:00:00

This is what the flights dataframe looks like. It has a lot of information on each flight. What I want to point out now is that the first several columns deal with departure and arrival times, as well as departure and arrival delays. When you look at the carrier column, though, you'll see that it's a code. I can guess what carrier "UA" and "AA" are, but I have no idea what carrier "B6" is.



Here's an example of an analysis we can do with the flights table. When you go to buy a ticket, you will see information like this: the "on-time arrival percentage". It's defined at the percentage of times that a given flight arrives within 14 minutes after the scheduled arrival time. The flights table gives us enough information to calculate this.

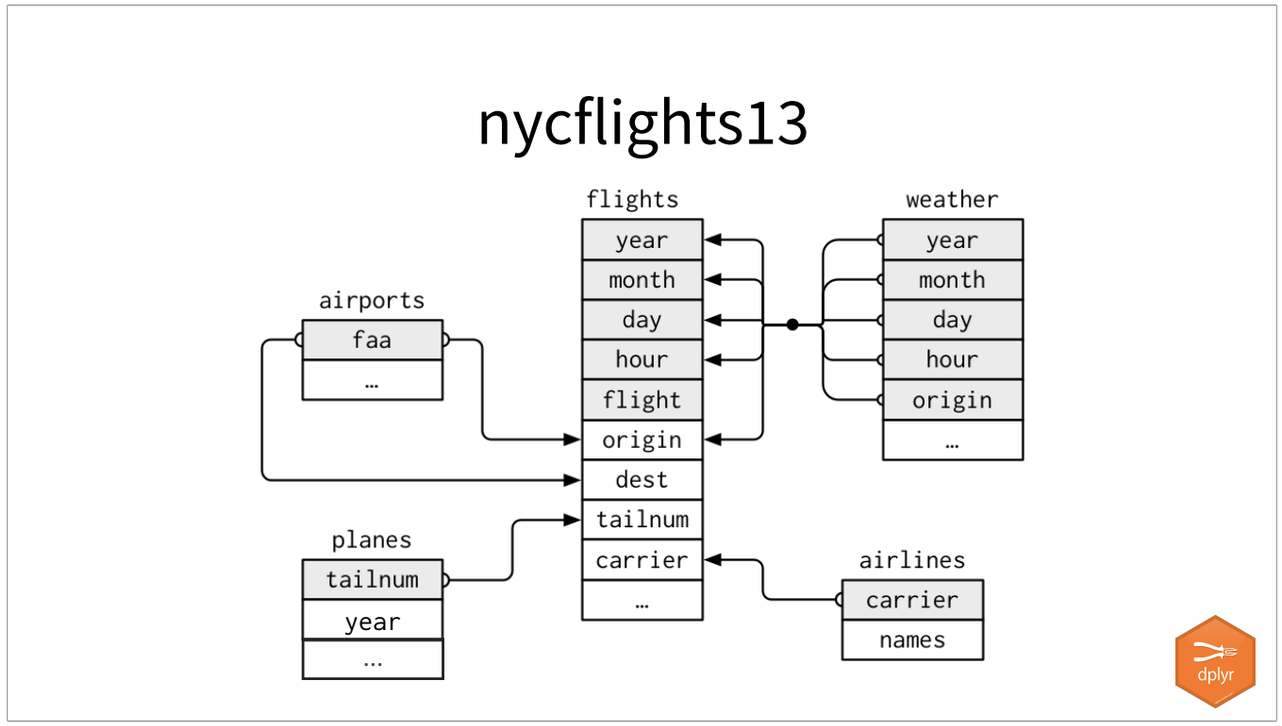
Carriers (airlines)

```
View(airlines)
```

	carrier	name
1	9E	Endeavor Air Inc.
2	AA	American Airlines Inc.
3	AS	Alaska Airlines Inc.
4	B6	JetBlue Airways

If you look at the airlines table you'll see that it has two columns: carrier and name. And here we can see that carrier B6 is JetBlue Airways.

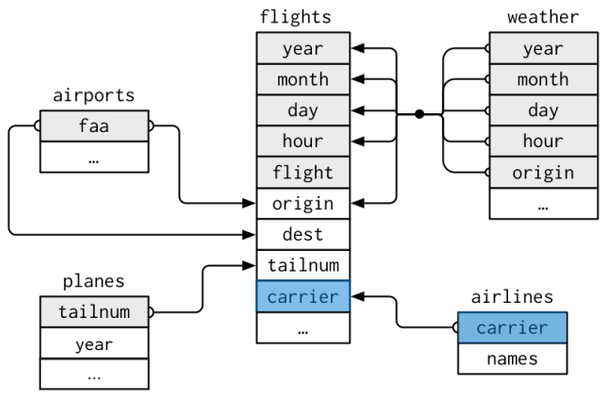
When we do our analysis on the flights table, we'll want to replace (or augment) the carrier codes with the name column from this table.



This is the schema for the entire nycflights13 database. You've already seen a few of the tables. What I want you to focus on here is how the tables share information. It's this shared information that lets you join them together.

nycflights13

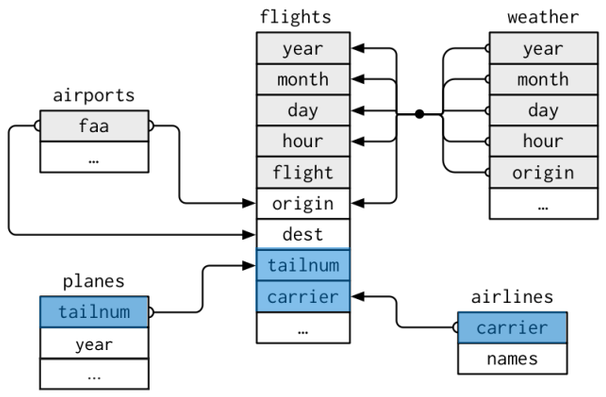
What airline had the longest delays?



For example, if we want to calculate the carrier with the longest delay, we can do that with just the flights table. But if we want to know the name of that carrier, we'll have to join the resulting table with the airlines table.

nycflights13

What airline used the newest planes?



Similarly, if we want to know the name of the airline with the newest planes, we'll have the join the flights table with the planes table (using the tailnum column). Then we'll have to join with the airlines table, to get the name of the carrier.

Types of joins

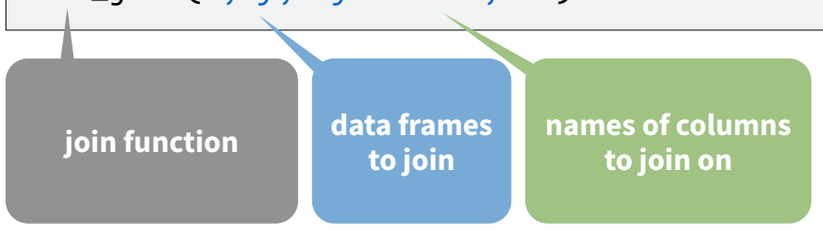
You now conceptually understand how joins work and when you'd want to use them. Now let's look at how they're done with dplyr.

What I want you to know up front is that there isn't a single "join" function in dplyr. Rather, there are 4 different functions that do joins. At some level, they all do the same thing. But there are important nuances between them. And that's what we're going to look at now.

common syntax

Each join function returns a data frame / tibble.

```
left_join(x, y, by = NULL, … )
```



Each of the 4 joins is done with a single function. The first two arguments are always data frames. And then there is a "by" parameter that takes the names of the columns to join on.
"by" defaults to NULL, which means that it will join on all columns in x and y that have the same column name.

Toy data

```
band <- tribble(
   ~name,     ~band,
  "Mick",  "Stones",
  "John", "Beatles",
  "Paul", "Beatles"
)
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

```
instrument <- tribble(
    ~name,   ~plays,
   "John", "guitar",
   "Paul",   "bass",
  "Keith", "guitar"
)
```

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

Before we look at joining on the flights dataset, let's go through some examples with toy data. You should have this code in your Notebook. <walk thru the tables, pointing out the similarities and differences>.

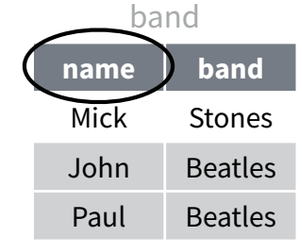
Toy data

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar



To see all the information about a musician at once, we'd need to join the two tables. The column that we'd want to join on is "name". But there's a complication here. Mick is in the band table but not the instrument table. And Keith is in the instrument table but not the band table. How should we handle this when we join? John and Paul are easier cases, because they appear in both tables.

Earlier I mentioned that there are several variations on joining in dplyr. These variations all have to do with how we handle the case when data is in one table but not the other.

inner

```
band %>% inner_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass

The most common type of join is the inner join. Inner joins say "only include rows that have a match in both tables". Because only John and Paul are in both tables, they're the only rows that are returned by inner_join

left

```
band %>% left_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass

The function left_join will always return all rows in the left table. It is very similar to inner_join, in that it returns rows that have a match in both tables. But it differs in that it also returns rows that don't have a match. So here the final table has Mick, even though he doesn't have a match in the instrument trable.

For rows that don't have a match, their values in the new table are filled in with NAs.

right

```
band %>% right_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

Right joins are much less commonly used than inner and left joins. But they work analogously to left_joins. They always return all rows in the right hand table.

If there's a match in the left-hand table, it returns that value. Otherwise it returns an NA. So here Mick is not returned because he's not in the right hand table. And Keith's "band" value is NA.

full

```
band %>% full_join(instrument, by = "name")
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

The full_join is the type of join I have used the least. It includes all rows from both tables, regardless of whether or not they have a match in the other table.

Your Turn 12

Which airline had the most flights?

```
flights %>%
   group_by(_____) %>%
   summarize(_____) %>%
   arrange(_____) %>%
   left_join(_____)
```

1. Start with a normal "group by / summarize / arrange"

2. Then do a "left_join" to get the carrier"

Here's your first assignment doing joins. I want you to find the airline that had the most flights. I want you to solve this problem in two steps. First: ignore the airline name and answer the question just using the carrier. You can do this using group_by, summarize and arrange, which you already know.

Once you're satisfied with this answer, then add in the left_join, which is new.

```
flights %>%
   group_by(carrier) %>%
   summarize(n=n()) %>%
   arrange(desc(n)) %>%
   left_join(airlines)
```

```
  carrier     n name
  <chr>   <int> <chr>
1 UA      58665 United Air Lines Inc.
2 B6      54635 JetBlue Airways
3 EV      54173 ExpressJet Airlines Inc.
4 DL      48110 Delta Air Lines Inc.
5 AA      32729 American Airlines Inc.
```

What I want you to realize here is that the join is just one extra line. Group_by, summarize and arrange - when combined with a pipe - are really the nuts and bolts of dplyr. They're often used together.

Your Turn 13

Which airlines had the largest arrival delays?

```
flights %>%
  drop_na(arr_delay) %>%
  ...
```

1. Drop rows with NA delays
2. Calculate average delay with group_by / summarize / arrange
3. Use left_join to get airline name

I now want you to calculate which airline had the largest average delay. To do this you need to use one special new function - drop_na - which will drop rows with a missing delay. Follow the same pattern as before.

```
flights %>%
  drop_na(arr_delay) %>%
  group_by(carrier) %>%
  summarize(avg_arr_delay = mean(arr_delay)) %>%
  arrange(desc(avg_arr_delay)) %>%
  left_join(airlines)
```

	carrier	avg_arr_delay	name
	<chr>	<dbl>	<chr>
1	F9	21.9	Frontier Airlines Inc.
2	FL	20.1	AirTran Airways Corporation
3	EV	15.8	ExpressJet Airlines Inc.
4	YV	15.6	Mesa Airlines Inc.
5	OO	11.9	SkyWest Airlines Inc.

There was 1 new thing with this exercise: I showed you how to drop rows with drop_na. I wouldn't focus on this too much. I encourage you to think about it as one of many little details in dplyr that you'll pick up as you use it more. The main thing, as always, is the "group_by / summarize / arrange" pattern. And then, at the end, we do a left-join with airlines to get the airline name.

Toy data

```
band <- tribble(
   ~name,     ~band,
  "Mick",  "Stones",
  "John", "Beatles",
  "Paul", "Beatles"
)
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

```
instrument2 <- tribble(
    ~artist,   ~plays,
   "John", "guitar",
   "Paul",   "bass",
  "Keith", "guitar"
)
```

instrument2

artist	plays
John	guitar
Paul	bass
Keith	guitar

Now let's look at a very common problem with joins. When the values in the columns match, but the names do not. Here we have the same "band" table as before. But the table we're joining to now is called "instrument2". And instead of having a "name" column, it has an "artist" column. The data is the same, but the name of the columns are now different: "name" vs. "artist".

What if the names do not match?

Use a named vector to match on variables with different names.

```
band %>% left_join(instrument2, by = c("name" = "artist"))
```

A named vector

The name of the element = the column name in the first data set

The value of the element = the column name in the second data set

The way that you handle joining columns that have different names is with the "by" argument. By default, "by" is NULL. That means that it will join all columns that match.
Here we're giving "by" what's called a "named vector". The LHS has the name of the column in the lefthand table. The RHS has the name of the column in the righthand table.

common syntax - matching names

```
band %>% left_join(instrument2, by = c("name" = "artist"))
```

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument2

artist	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass

And here's what happens when we do the left-join with *by* "*name*"="*artist*". Name is joined with Artist, so we have Mick in the returned table, even though Mick is not in the instrument2 table. John and Paul match, so they are returned. And Keith is not in the band table at all, so he is not returned.

Airport names

```
View(flights["dest"])
```

	dest
1	IAH
2	IAH
3	MIA
4	BQN
5	ATL
6	ORD

```
View(airports)
```

faa	name
04G	Lansdowne Airport
06A	Moton Field Municipal Airport
06C	Schaumburg Regional
06N	Randall Airport
09J	Jekyll Island Airport
0A9	Elizabethton Municipal Airport

Now let's look at a case in our own data where we have columns that don't have the same name.

In the flights table we have a *dest* column that has airport codes. In the *airports* table we have a column called *faa* that has airport codes as well. Like the airlines table earlier, this table also has a *name* column that has the proper name of the airport.

common syntax - matching names

```
airports %>% left_join(flights, by = c("faa" = "dest"))
```

faa <chr>	name <chr>
04G	Lansdowne Airport
06A	Moton Field Municipal Airport
06C	Schaumburg Regional
06N	Randall Airport
09J	Jekyll Island Airport
0A9	Elizabethton Municipal Airport
0G6	Williams County Airport
0G7	Finger Lakes Regional Airport

dest <chr>	air_time <dbl>
IAH	227
IAH	227
MIA	160
BQN	183
ATL	116
ORD	150
FLL	158
IAD	53

Here's the syntax for joining the tables. We start with the airports table, then do a left_join with the flights table, and then specify the "by" parameter to be the named vector where "faa" = "dest"

Your Turn 14

Use flights and airports to compute the average **arr_delay** by destination airport. Order by average delay, worst to best.

Select just 3 columns from the output: the airport code, the airport name, and the average arrival delay

```
flights %>%
  drop_na(arr_delay) %>%
  group_by(dest) %>%
  summarize(mean_arr_delay = mean(arr_delay)) %>%
  arrange(desc(mean_arr_delay)) %>%
  left_join(airports, by=c("dest"="faa")) %>%
  select(dest, name, mean_arr_delay)
```

dest	name	mean_arr_delay
<chr>	<chr>	<dbl>
1 CAE	Columbia Metropolitan	41.8
2 TUL	Tulsa Intl	33.7
3 OKC	Will Rogers World	30.6

Here's my solution. I want to reiterate that you already knew how to do most of this. Again, we have a special line of code at the top to remove the NAs. But then it's our old friend "group_by, summarize, arrange". To get the additional information, we then do a left_join. And I want to point out that, unlike our other examples - and unlike SQL - the select is the last line of the program. (It's also the first line after the join). I'm pointing this out just to say that the various commands can come in any order.

Recap: Two table verbs

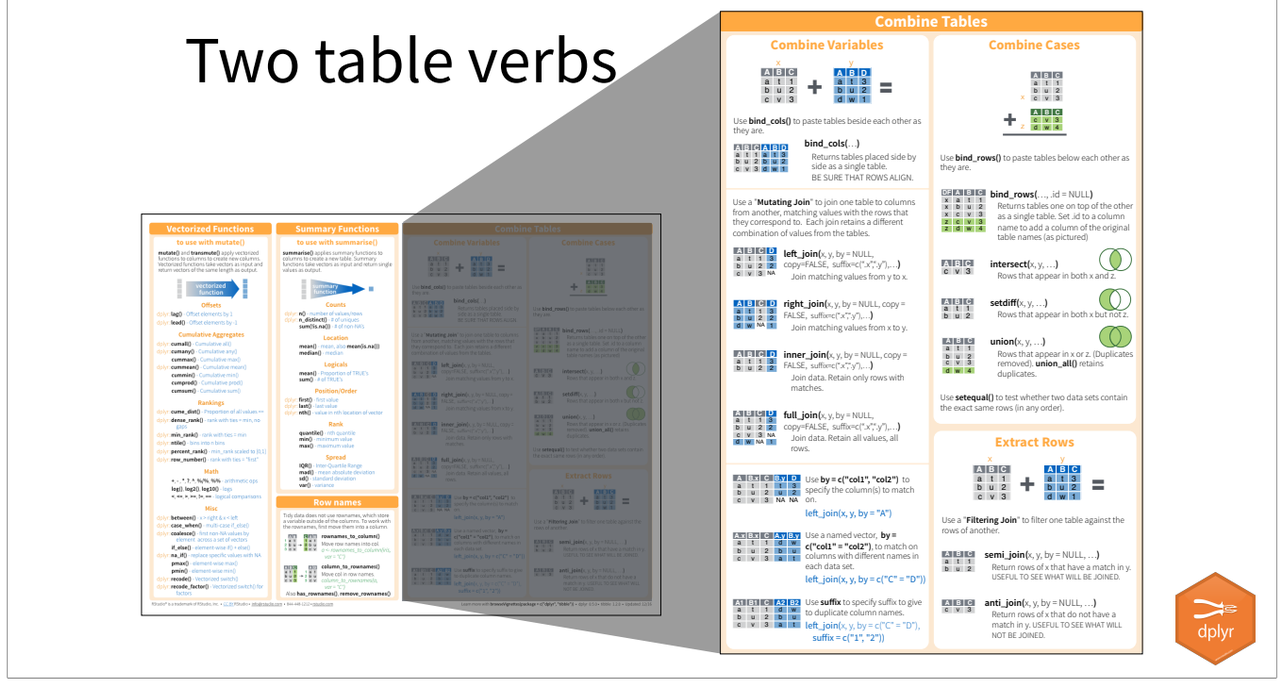
left_join() retains all cases in **left** data set

right_join() retains all cases in **right** data set

full_join() retains all cases in **either** data set

inner_join() retains only cases in **both** data sets

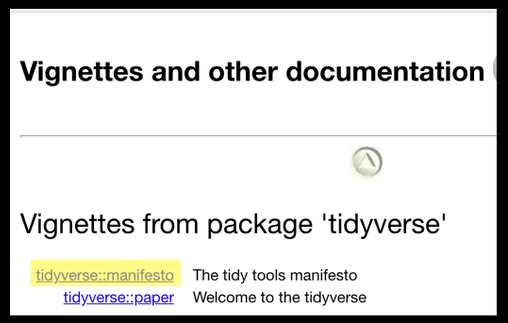
That's the last exercise I had for joins. Just to reiterate, there are 4 main types of joins.

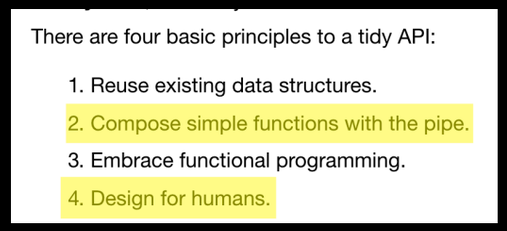


In the dplyr cheat sheet, you'll see that there's a whole section titled "combine tables". That details the material that we just covered.

Tidy tools

That wraps up everything I wanted to say about dplyr. As you might recall, at the start of the course I said that dplyr and ggplot are the most important parts of the tidyverse. ggplot was created first, before the tidyverse was even coined. When dplyr was created, it led to a desire to think of what linked these packages, and would be a principle of future work. I'd like to talk about that now.





The Tidy Tools Manifesto

Really

When dplyr came out Hadley wrote what he calls "The Tidy Tools Manifesto". If you read the Manifesto, Hadley is very up front that he's sort of making things up as he goes along. He has a strong instinct what he's doing, but he isn't sure that he can fully put it into words yet. This isn't a bad thing, and his honesty and humility are reasons why he's so popular.

You can read the Manifesto by clicking on the Tidyverse package in the RStudio Packages window, and then clicking on the "Vignettes" button. I want to focus on two parts of it.

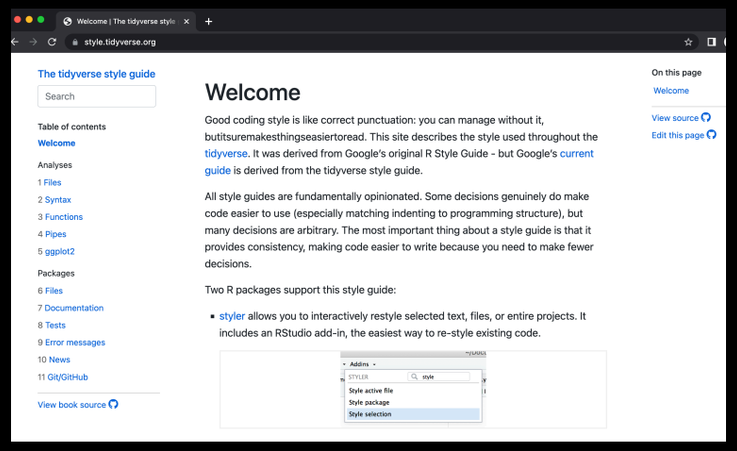
Anything in the Tidyverse should work with the pipe. And the packages shoud be designed for humans.

Designed for Humans:
Functions do one thing, and they do it well

filter() - extract **cases**
arrange() - reorder **cases**
group_by() - group **cases**
select() - extract **variables**
mutate() - create new **variables**
summarise() - summarise **variables** / create **cases**

My background is in software engineering. Once I started working in R, I was amazed at how ... most R programmers weren't "real" programmers. Hadley has really led to a quantum leap in improving the quality of code on CRAN. A major part of that is simply having clear, concise functions that do exactly one thing.

Filter, arrange, select, etc. - they all do just one thing.



Following Conventions

Tidyverse Style Guide

Another thing that the Tidyverse does very well is that it has explicit conventions, and it follows them. For example, at the start I mentioned that ?View and ?summary disagree on capitalization. The Tidyverse has an explicit style guide (google "tidyverse style guide") and it says, among other things, that they only use lower case letters, use an _ to separate words, etc. Dan and I have largely adopted this style guide for our own work in MB.

Compose simple functions with the pipe


```
babynames        mutate(_____, percent = prop * 100)
```

Each dplyr function takes a data frame as its first argument and returns a data frame. As a result, you can directly pipe the output of one function into the next.

You might have noticed that ggplot uses a + and not the pipe. This is because the pipe actually comes from another package, called magrittr, that didn't exist when ggplot2 was created!

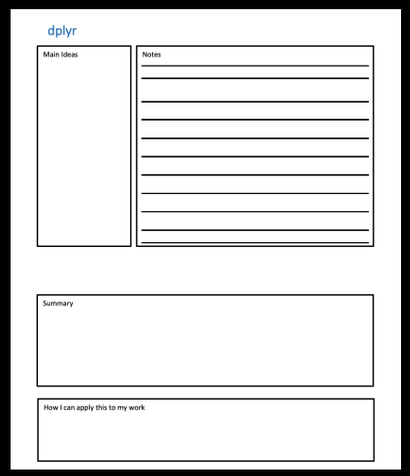
ggplot2 was first released in 2007, and magrittr was first released in 2014. dplyr was also created in 2014, and it adopted magrittr's pipe fairly early on in its development (version 0.2.0).

Piping, I think, is a major reason why dplyr became so popular. And that's why I think it's become a central feature of the tidyverse.

As a reminder, newer versions of the R language support piping with |> - you know longer need the magrittr package.

Closing Exercise

- Fill out the "Summary" section
- Fill out the "How I can apply to to my work"
- Share with your neighbor!



Any Questions?

This concludes the introdution to the Tidyverse that I wanted to give. Are there any questions?