



NYC DATA SCIENCE
ACADEMY

Manipulating Data with “dplyr”

Data Science Bootcamp

OVERVIEW

- ❖ Introduction to dplyr
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

dplyr

- ❖ We have now covered the major data structures, built-in functions, and programming features of R. *In principle*, we can do anything. In practice, using R effectively involves learning a variety of packages to simplify your tasks.
- ❖ **dplyr** provides a set of functions, somewhat similar to database query-type operations, to efficiently manipulate data frames.
 - Filtering & rearranging: Built-in functions to do basic transformations.
 - Joins: When the analysis requires information from multiple data frames.
 - Groupwise operations: When analysis requires aggregation.

Basic dplyr Functions

- ❖ dplyr is based on surprisingly few operations which we will study in detail:
 - `filter` - Take subsets of rows
 - `select` - Take subsets of columns
 - `arrange` - Reorder rows
 - `mutate/transmute` - Add or replace columns
 - `summarise` - Compute aggregate values
 - `group_by` - Grouping operations that “split-apply-combine”
 - `joins` - Methods to combine multiple data frames

Baby Births Dataset

- ❖ We introduce the sample data sets we will use during class today.
births.csv contains counts of boys and girls born in each year.

```
births <- read.csv("births.csv",
                     stringsAsFactors = FALSE)
births[c(1:2, 131:132), ]
```

	year	sex	births
1	1880	boy	118405
2	1881	boy	108290
131	1880	girl	97606
132	1881	girl	98860

Baby Names Dataset

- ❖ **bnames.csv** is a much larger file giving information on babies with different names born in each year. For each year and child, the prop column gives the percentage of boys or girls given that name in that year. The soundex column gives the soundex code, which describes the pronunciation of that name (see en.wikipedia.org/wiki/Soundex).

```
bnames <- read.csv("bnames.csv",
                      stringsAsFactors = FALSE)
head(bnames, 5)
```

	year	name	prop	sex	soundex
1	1880	John	0.081541	boy	J500
2	1880	William	0.080511	boy	W450
3	1880	James	0.050057	boy	J520
4	1880	Charles	0.045167	boy	C642
5	1880	George	0.043292	boy	G620

Baby Names Dataset

- ❖ We might want to ask: How has the percentage of each name evolved since 1880? For example:

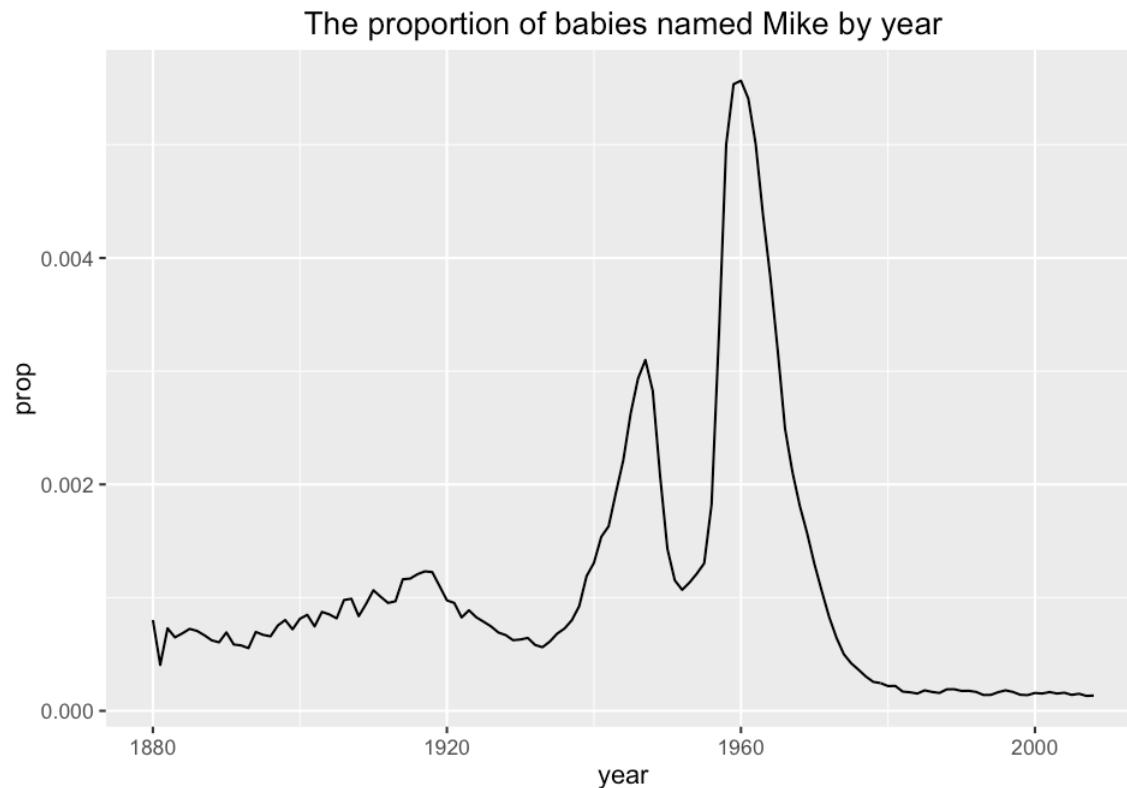
```
mike = bnames[bnames$name == "Mike", ]  
head(mike, 3)
```

	year	name	prop	sex	soundex
143	1880	Mike	0.000802	boy	M200
1231	1881	Mike	0.000406	boy	M200
2156	1882	Mike	0.000729	boy	M200

- ❖ The code above selects the rows with name “Mike”. We see that for each year the percentage of the name changes. This can be better seen with visualization.

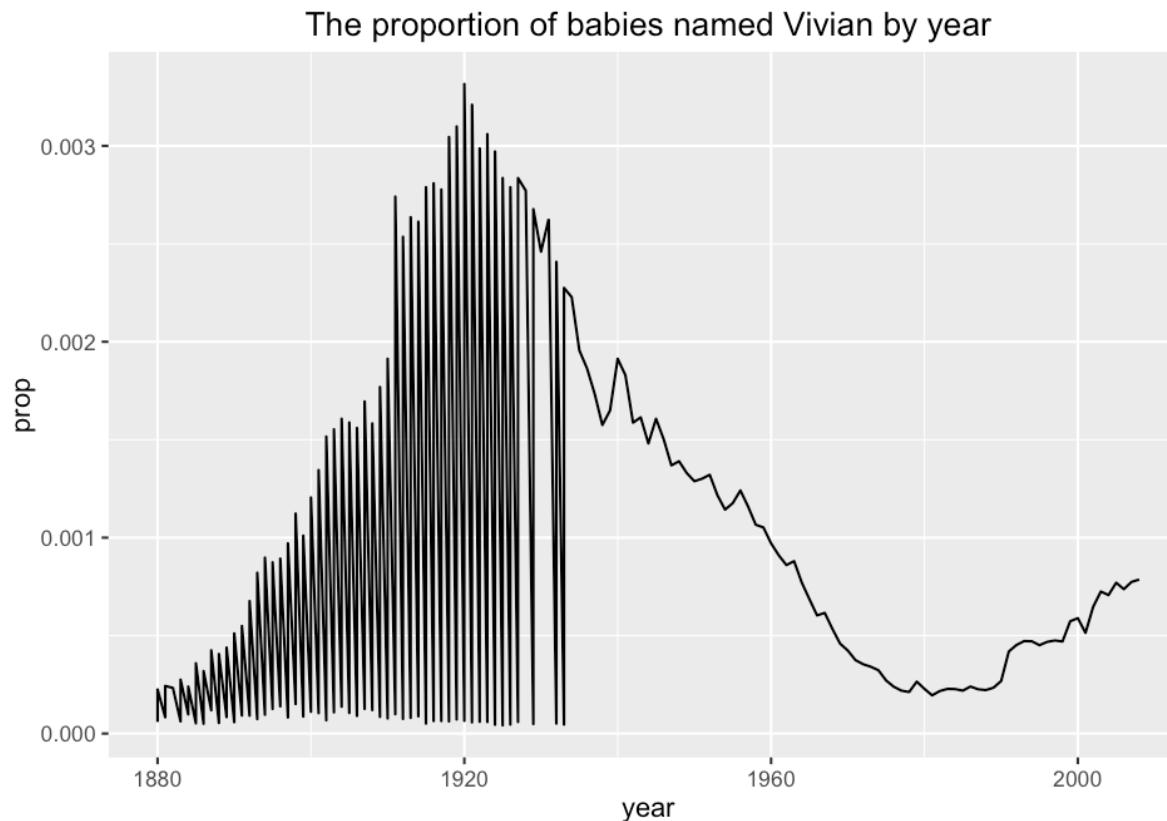
Baby Names Dataset

- ❖ The graph below is plotted with `ggplot2`, a package we will cover in a separate class. It plots how popular the name “Mike” was in each year. It reached its peak popularity in 1960.



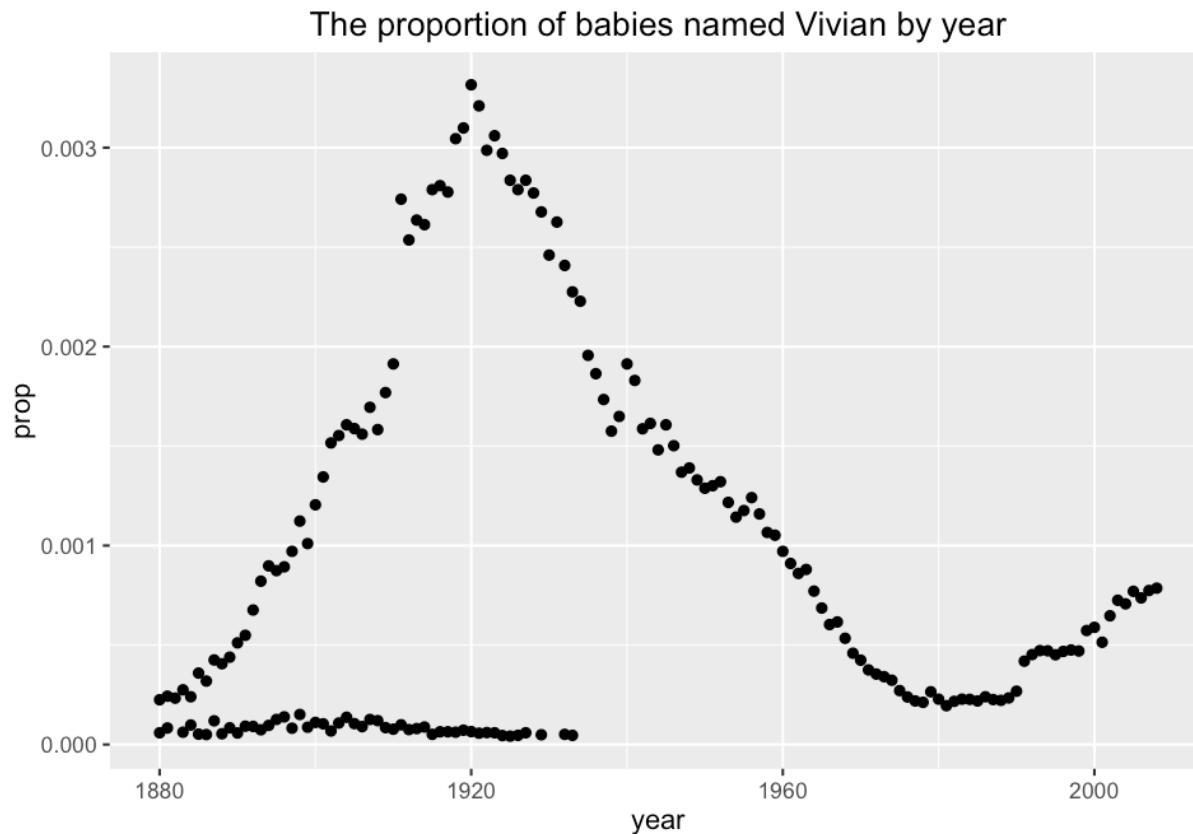
Baby Names Dataset

- ❖ We see a problem if we visualize the name Vivian. Why does the proportion seem to be oscillating so frequently?



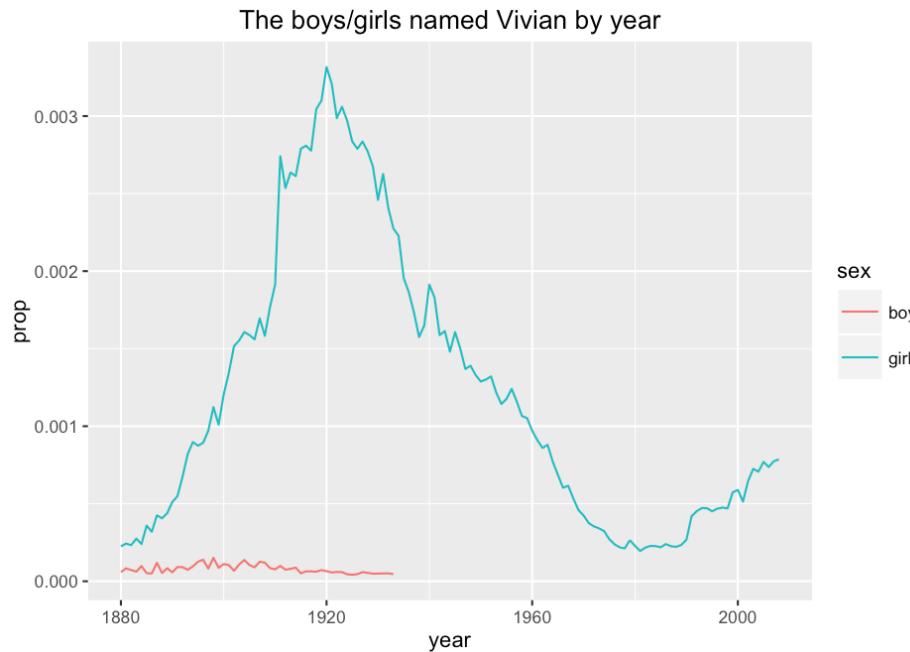
Baby Names Dataset

- ❖ Let's try sketching a scatterplot instead. What do you think causes the oscillation from the previous slide?



Baby Names Dataset

- ❖ From the scatter plot, it seems that there are actually *two* curves, instead of an oscillating one. It turns out that Vivian was once a fairly common boy's name. In the class today, we survey the tools that facilitate this kind of analysis in dplyr.



OVERVIEW

- ❖ Baby name data set
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

Built-in Functions

- ❖ We start with some built-in function in dplyr. Most of what we're doing here can be done with basic programming methods as well; however, dplyr provides a friendlier syntax as well as a better data storage method that improves efficiency. We will take a look at the functions:
 - filter
 - select
 - arrange
 - mutate/transmute
 - summarise
- ❖ We will see that the functions listed above all take a data frame as their first argument and the “instructions” in their second argument. They all return a data frame.

OVERVIEW

- ❖ Baby name data set

- ❖ Built-in functions

- filter/select

- arrange, mutate/transmute

- summarise

- ❖ Joining data sets

- ❖ Groupwise operations

- group_by & summarise

Built-in Functions

- ❖ We first construct a toy sample:

```
df <- data.frame(  
  color = c("blue", "black", "blue", "blue", "black"),  
  value = 1:5)  
  
df  
  color value  
1  blue     1  
2 black     2  
3  blue     3  
4  blue     4  
5 black     5
```

- ❖ We'll want to import the dplyr package as well:

```
library(dplyr)
```

filter

- ❖ We then use the function `filter` to select the rows whose color is blue:

```
filter(df, color == "blue")
      color value
1  blue      1
2  blue      3
3  blue      4
```

- ❖ As we've seen before, this can also be done by ordinary slicing:

```
df[df$color == "blue", ]
```

The `filter` function has a simpler syntax; while the difference may be slight here, it will pay off as we do more complex things.

filter

- ❖ The first parameter of `filter()` is the data frame and the second is the filtering condition. (Note that we don't have to write "df\$" when referring to a column of df.)

```
filter(df, value %in% c(3, 4, 5))  
  color value  
1  blue     3  
2  blue     4  
3 black     5
```

- ❖ Logical operators like and (&) can be used as well:

```
filter(df, value %in% c(3, 4, 5) & color=='blue')  
  color value  
1  blue     3  
2  blue     4
```

filter

- ❖ Another example of logical operators using the or (|):

```
filter(df, value %in% c(1, 4) | color=="black")
  color value
  1  blue    1
  2 black    2
  3  blue    4
  4 black    5
```

- ❖ The second parameter is essentially broken down into a logical vector:

```
filter(df, c(T,F,F,F,T))
  color value
  1  blue    1
  2 black    5
```

The %>% Operator

- ❖ dplyr provides a handy operator that allows us to code in an intuitively.
- ❖ The %>% (“chaining”) operator passes the object into a function’s data frame argument, which is indicated by “.”. The period can be omitted if the data frame is the first argument of the function.
- ❖ The following three lines are identical:

```
filter(df, value %in% c(1, 4) | color=="black")
```

```
df %>% filter(., value %in% c(1, 4) | color=="black")
```

```
df %>% filter(df, value %in% c(1, 4) | color=="black")
```

select

- ❖ The `select()` function, like `filter()`, is used for slicing a data frame. `select` picks *columns* with column names (instead of picking *rows* with conditions).

```
df %>% select(., color)
      color
  1 blue
  2 black
  3 blue
  4 blue
  5 black
```

- ❖ You can select multiple columns:

```
df %>% select(., color, value)
      color value
  1 blue     1    #...some rows omitted...
```

select

- ❖ select can exclude columns by the “-” notation as well:

```
df %>% select(., -color)
      value
1       1
2       2
3       3
4       4
5       5
```

- ❖ We can specify the columns by indexes instead of names:

```
df %>% select(., 1)
      color
1   blue
2 black
3   blue    #...some rows omitted...
```

select

- ❖ When given integer arguments, `select` uses column numbers in the *original* data frame, not in any subsequent data frame. What do you think `select(df, -1, -1)` does?
- ❖ You might think this call first removes column 1 (`color`) and then removes the *new* column 1 (`value`), returning an empty data frame; this is not the case. The call `select(df, -1, -1)` is the same as the call `select(df, -1)`.
- ❖ To remove both columns, instead we can use the call `select(df, -1, -2)` or `select(df, -2, -1)`:

```
df %>% select(., -1, -2)
data frame with 0 columns and 5 rows
```

select

- ❖ It is possible to nest statements as well. What if we nested a selection statement with a negative index?

```
select(select(df, -1), -1)  
data frame with 0 columns and 5 rows
```

```
select(df, -1) %>%  
  select(., -1)  
data frame with 0 columns and 5 rows
```

- ❖ Why does this occur when it didn't work the same for the previous `select(df, -1, -1)` statement?

select

- ❖ It is important to note that selection is performed in a linear fashion, from left to right. Consider the statement below:

```
df %>% select(., -1, 1)

  value color
  1     1  blue
  2     2 black
  3     3  blue
  4     4  blue
  5     5 black
```

- ❖ First column 1 (color) is removed. At this point, there is only the column (value) left. Afterwards, column 1 from the original data frame is added back (color). Ultimately, we just ended up flipping the column order.

select

- ❖ What happens if we reverse the order of the indices?

```
df %>% select(., 1, -1)  
data frame with 0 columns and 5 rows
```

- ❖ This time, column 1 is selected first (color). At this point, column 2 (value) is removed and only the column 1 (color) is left in the data frame. Then, when we remove column 1, nothing is left. Thus, we ultimately ended up with an empty data frame.

select

- ❖ The function `select`, unlike `filter`, cannot take logical vectors; however, it provides some functions to select columns by certain conditions. These, and others, can be used as the second argument of `select`:
 - `starts_with(x, ignore.case=T)`: pick column names starting with x, case insensitive when `ignore.case` is (by default) set true.
 - `ends_with(x, ignore.case=T)`: pick column names ending with x.
 - `contains(x, ignore.case=T)`: pick column names containing x.
 - `matches(x, ignore.case=T)`: pick columns whose names match x, where x is a regular expression.
 - `one_of(name_1, name_2, ..., name_n)`: pick columns that have any of the names in the list. (The argument can also be a character vector.)

select

- ❖ We demonstrate the usage of the functions from the previous slide:

```
df %>% select(., starts_with('c'))  
  color  
1 blue  
2 black  
3 blue  
4 blue  
5 black  
  
df %>% select(., contains('e'), starts_with('c'))  
  value color  
1     1   blue  
2     2 black  
3     3   blue  
4     4   blue  
5     5 black
```

select

- ❖ Another benefit of `select` is that it can be used to rename a column. While selecting, if we specify *new-column-name* = *original-column-name* in the second parameter of `select`, we will rename the selected column in the resulting data frame:

```
df %>% select(., COLOR=color)
      COLOR
1  blue
2 black
3  blue
4  blue
5 black
```

select and rename

- ❖ If we desire to just rename columns without selecting a subset of columns, we could simply use the `rename()` function. The syntax is similar, but the function just renames and returns the whole (new) data frame:

```
df %>% rename(., COLOR = color, Something.New = value)  
  
      COLOR Something.New  
1    blue            1  
2  black            2  
3    blue            3  
4    blue            4  
5  black            5
```

select and rename

- ❖ We remark here that `select()` and `rename()` are both non-mutating functions. Remember, they do not change the original data frame, but instead produce a *new* data frame. Even after doing all of the previous operations, we can see that the original data frame is unchanged:

```
df
  color  value
1  blue    1
2 black    2
3  blue    3
4  blue    4
5 black    5
```

- ❖ If we would like to change `df`, we would have to use assignment:

```
df = df %>% rename(., COLOR = color)
```

OVERVIEW

- ❖ Baby name data set
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

arrange

- ❖ The function `arrange()` positions the rows of the data frame in ascending order according to the values in the specified column:

```
df %>% arrange(., value)
      color value
1  blue     1
2 black     2
3  blue     3
4  blue     4
5 black     5
```

arrange

- ❖ If we would like the column sorted in descending order, we can apply the function `desc()` to the column name:

```
df %>% arrange(., desc(value))
      color value
1 black      5
2 blue       4
3 blue       3
4 black      2
5 blue       1
```

arrange

- ❖ `arrange` can also be used to order strings alphabetically as well:

```
df %>% arrange(., color)
      color value
1 black     2
2 black     5
3 blue      1
4 blue      3
5 blue      4
```

mutate

- ❖ The function `mutate()` can be used to create a new column by deriving its value from an old column. For example, the code below doubles the numbers in `value` and also creates a new column called `double`.

```
df %>% mutate(., double = value*2)
  color value double
1  blue     1      2
2 black     2      4
3  blue     3      6
4  blue     4      8
5 black     5     10
```

mutate

- ❖ `mutate` can create multiple columns all at once:

```
df %>% mutate(., double = 2 * value, quadruple = 4 * value)
```

	color	value	double	quadruple
1	blue	1	2	4
2	black	2	4	8
3	blue	3	6	12
4	blue	4	8	16
5	black	5	10	20

transmute

- ❖ It is possible to only include newly created columns and omit the old ones in the resulting data frame. To do so, use the same syntax as `mutate()`, but instead call the `transmute()` function:

```
df %>% transmute(., double = 2 * value, quadruple = 4 * value)
```

```
double quadruple
1      2          4
2      4          8
3      6         12
4      8         16
5     10         20
```

OVERVIEW

- ❖ Baby name data set
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

Summarise

- ❖ We often need to extract summary information about a column. For example, we might want to know the mean and the standard deviation of a feature.
- ❖ The function `summarise()` returns a new data frame whose columns are aggregated from a column in the original data frame. The first argument is the data frame; the second is the new column name and the aggregation expression that defines its creation:

```
df %>% summarise(., total = sum(value))  
      total  
1     15
```

Summarise

- ❖ `summarise()` can also create more than one column at a time:

```
df %>% summarise(., total = sum(value),  
                    avg = mean(value))  
total avg  
1      15    3
```

- ❖ We can perform computations within the `summarise()` function as well:

```
df %>% summarise(., new_col=mean(value)/sd(value))  
new_col  
1 1.897367
```

Summarise

- ❖ Below are some of the aggregating functions that can be used in `summarise()`:
 - Built-in aggregating functions which take a vector and return a scalar value: `min()`, `max()`, `mean()`, `sum()`, `sd()` and `median()`.
 - `n()`: returns the number of observations.
 - `n_distinct(x)`: returns the number of unique values in the column `x`.
 - `first(x)` or `last(x)`: returns the first or last observations in the column `x`.
 - `nth(x, n)`: returns the `n`th observation in column `x`.

Summarise

❖ Examples:

- `n_distinct()`: In our data frame `df`, there are two different colors.

```
df %>% summarise(., n_distinct(color))
  n_distinct(color)
  1           2
```

- `nth()`: The fourth element in the `value` column is 4:

```
df %>% summarise(., nth(value, 4))
  nth(value, 4)
  1           4
```

Exercise 1: filter

- ❖ Find all of the names that are in the same soundex as your name (or your favorite name, in case you don't have a common English name).
- ❖ Select the rows of all girls born in 1900 or 2000.
- ❖ How many times did a name reach a proportion greater than 0.01 after the year 2000?

Exercise 2: select

- ❖ From bnames, select the columns whose names start with, ‘y’, ‘s’ or ‘p’.
 - *Note:* Because bnames is very large, be sure to assign the result of this (and subsequent) exercises to a variable. Then you can look at it using head or str.
- ❖ Create a data frame that is the same as bnames without the column “year”.
- ❖ Select the columns “year” and “name” from bnames. Use only one condition to achieve this.

Exercise 3: arrange

- ❖ Reorder the dataset bnames by prop in descending order.
- ❖ In what year was your name most popular among the years?

Exercise 4: mutate

- ❖ For the data frame bnames, create a column “percentage”, which derives from “prop” by changing the proportion to a percentage.

Exercise 5: summarise

- ❖ Create a summary that displays the min, mean, and max prop for your name, from the data frame bnames.

OVERVIEW

- ❖ Baby name data set
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

Joining Data Sets

- ❖ For better allocation of resources, related data is often stored in separate locations. Let's examine the baby datasets once again:
 - Recall that the `births` dataset contains the variables of `year`, `sex`, and `births`.
 - On the other hand, the `bnames` dataset contains the variables of `year`, `name`, `prop`, `sex`, and `soundex`.
- ❖ In an analysis, we might need the information from both data frames at once. We can combine these data frames with a **join**.
 - If you know SQL, you are familiar with the notion of joining two tables. As in SQL, `dplyr` has several kinds of joins: *inner join*, *outer join*, *left outer join*, and *right outer join*; `dplyr` also has *semi-join* and *anti-join*.

Joins

- ❖ The basic idea of a join: We have two data frames $df1$ and $df2$ with one or more columns in common; these columns contain the same type of data, and may even have the same name. Call these columns the *join columns*.
 - For births and bnames, the join columns are year and sex.
- ❖ Take any row from $df1$ and any row from $df2$. If they have the same values in the join column(s), then combine those two rows into one and put the result into a new data frame.
 - Repeat this process for every pair of rows from $df1$ and $df2$.
- ❖ The different types of joins mostly differ on how they deal with the situation where a row in one data frame has *no* matching row in the other.

Joins

- ❖ Let's see what happens with an *inner join*. We have a table that matches buildings with cities, and another that matches cities with countries. We want to match buildings with countries:

Building	City
Empire State	New York
Eiffel Tower	Paris
Notre Dame	Paris

City	Country
New York	USA
Paris	France

- ❖ The join compares each pair of rows to find where the values in the City column match. The result is as follows:

Building	City	Country
Empire State	New York	USA
Eiffel Tower	Paris	France
Notre Dame	Paris	France

Joining Data Sets

- ❖ Let's construct a couple sample data sets to illustrate different joining methods:

```
x = data.frame( name = c("John", "Paul", "George",
                           "Ringo", "Stuart", "Pete"),
                 instrument = c("guitar", "bass", "guitar",
                               "drums", "bass", "drums"),
                 stringsAsFactors = FALSE)

x
  name instrument
1  John      guitar
2  Paul       bass
3 George     guitar
4 Ringo      drums
5 Stuart     bass
6 Pete       drums
```

Joining Data Sets

- ❖ Here's another data frame y:

```
y <- data.frame( name = c("John", "Paul", "George",
                           "Ringo", "Brian"),
                  band = c(TRUE, TRUE, TRUE,
                           TRUE, FALSE),
                  stringsAsFactors = FALSE)

y
  name   band
1 John   TRUE
2 Paul   TRUE
3 George TRUE
4 Ringo  TRUE
5 Brian  FALSE
```

- ❖ Note that x and y both have a column called name.

Inner Join

- ❖ The function `inner_join()` is the basic and original join. As described previously, it takes every pair of rows from the left and right data frames; if the values in the join column are the same, the combined row is kept in the resulting data frame:

```
inner_join(x, y, by = "name")
  name instrument band
  1  John      guitar TRUE
  2  Paul       bass  TRUE
  3 George     guitar TRUE
  4 Ringo      drums TRUE
```

- ❖ Since the rows in `x` with names Stuart and Pete don't match any rows in `y`, they do not show up in the result.

Left Join

- ❖ The function `left_join()` differs from the inner join in that every row from the left data frame is included in the result, regardless of whether there is a match in the right data frame. How?
- ❖ If a row in the left data frame is not matched by any row in the right, it is still put into the result; however, the corresponding columns from the right data frame appear to have NAs instead of actual values:

```
left_join(x, y, by = "name")
  name instrument band
  1  John      guitar TRUE
  2  Paul       bass  TRUE
  3 George     guitar TRUE
  4 Ringo      drums TRUE
  5 Stuart     bass   NA
  6 Pete       drums NA
```

Left Join

- ❖ Let's see how this works:



- ❖ Since Stuart and Pete don't match anything, they are included in the result with NA.
- ❖ Note that Brian isn't included in the final result. This is because we are performing a *left* join.

Right Join

- ❖ The function `right_join()` works in much the same way. The only difference is that rows in the right data frame are all kept, regardless of matches in the left data frame. NA values are placed accordingly:

```
right_join(x, y, by = "name")
  name instrument band
1   John      guitar TRUE
2   Paul       bass  TRUE
3 George      guitar TRUE
4 Ringo      drums  TRUE
5 Brian        <NA> FALSE
```

Full Join

- ❖ The *full join* (aka *outer join* or *full outer join*) combines the processes of both the left and right joins into one process. Thus, all unmatched records are retained from both data frames. The function is called `full_join()`:

```
full_join(x, y, by = "name")
  name instrument band
  1  John      guitar TRUE
  2  Paul       bass  TRUE
  3 George     guitar TRUE
  4 Ringo      drums TRUE
  5 Stuart     bass   NA
  6 Pete       drums NA
  7 Brian      <NA> FALSE
```

Semi Join

- ❖ The function `semi_join()` is most similar to an `inner_join()` in that it keeps only the rows showing up in both of the data frames. The difference is that it retains only the columns in the left data frame:

```
semi_join(x, y, by = "name")
      name instrument
  1  John      guitar
  2  Paul      bass
  3 George     guitar
  4 Ringo     drums
```

- ❖ We could have used `filter()` if we referenced the `name` columns from both data frames as follows:

```
filter(x, name %in% y$name)
```

Anti Join

- ❖ The function `anti_join()` is similar to the `semi_join()` in that it keeps only the columns of the left data frame; however, it retains only the rows that do *not* have a match in the right data frame:

```
anti_join(x, y, by = "name")
      name instrument
  1  Pete        drums
  2 Stuart       bass
```

Join & Mutate Example Application

- ❖ We want to calculate the number of children with a particular name born in each year. First we create a data frame bnames2. Notice that the by parameter specifies a vector; in this example, rows are identified if both year and sex are the same when we set the by parameter to a vector:

```
bnames2 = inner_join(bnames, births,  
                      by = c("year", "sex"))
```

- ❖ Next, we add a new column n to bnames2 whose value is the desired number. The function round() makes sure we have an integer. (It doesn't really make sense to have 1.5 babies.):

```
bnames2 = bnames2 %>% mutate(., n = round(prop * births))
```

OVERVIEW

- ❖ Baby name data set
- ❖ Built-in functions
 - filter/select
 - arrange, mutate/transmute
 - summarise
- ❖ Joining data sets
- ❖ Groupwise operations
 - group_by & summarise

Groupwise Operations

- ❖ Analysis often reveals different insights when applied to different subsets of data. For example, in the baby names dataset, the total number of “Vivians” differs from the total number of “Vivians” within each year.
- ❖ In the last application, we created a column representing the total number of children with a particular name born in each year. If we sum the “Vivians,” we see that there were 183,011 total born from 1880 to 2008 (the whole dataset):

```
vivian = filter(bnames2,  
                 name == "Vivian")  
  
sum(vivian$n)  
  
[1] 183011
```

Groupwise Operations

- ❖ This is likely to be different from the total number of “Vivians” in 2008 only:

```
vivian2 = filter(bnames2, name == "Vivian",
                  year == 2008)
sum(vivian2$n)
```

```
[1] 1629
```

- ❖ If we want to find the number in each year, must we type and run the same code for each year? What if we want to do the same for every name too? That will take us forever!
- ❖ Groupwise operations help in such a scenario by creating groups and aggregating values over each group.

Groupwise Operations

- ❖ Recall our sample data frame df, which we will use to illustrate the groupwise operations:

df		
	color	value
1	blue	1
2	black	2
3	blue	3
4	blue	4
5	black	5

group_by

- ❖ We see two different groups in the color column: blue and black. Let's use the `group_by()` function to create a "grouped data frame".
 - `group_by()` takes two or more arguments: (1) the data frame, and (2) the column(s) whose values will be used for grouping.
 - Its output has several types: in addition to "`data.frame`", it has types "`tbl`", "`tbl_df`" and "`grouped_df`". The important one here is "`grouped_df`," which contains the grouping information:

```
by_color = df %>% group_by(., color)
class(by_color)
[1] "grouped_df" "tbl_df"     "tbl"       "data.frame"
```

group_by

- ❖ When a "grouped_df" object is printed, R shows the source data frame and any grouping factors:

```
by_color = df %>% group_by(., color)
By_color

Source: local data frame [5 x 2]
Groups: color [2]

  color value
1  blue     1
2 black     2
3  blue     3
4  blue     4
5 black     5
```

Summarise

- ❖ We can aggregate the values of the "grouped_df" object within each group by applying the `summarise()` function and the aggregating functions we introduced before:

```
summarise(by_color, total = sum(value))
```

Source: local data frame [2 x 2]

	color	total
1	black	7
2	blue	8

group_by and summarise

- ❖ Now we are ready to answer our earlier question: How many babies with a particular name were born in each year? Let's group the data by both name and year:

```
group_by(bnames2, name, year)
```

Source: local data frame [258,000 x 7]

Groups: name, year

	year	name	prop	sex	soundex	births	n
1	1880	John	0.081541	boy	J500	118405	9655
2	1880	William	0.080511	boy	W450	118405	9533
3	1880	James	0.050057	boy	J520	118405	5927
4	1880	Charles	0.045167	boy	C642	118405	5348
5	1880	George	0.043292	boy	G620	118405	5126

group_by and summarise

- ❖ Summarising the grouped data frame, we compare “Vivian” in 1895 in the summary and in the original data frame. We see that multiple rows in the original data frame were aggregated (summed) in the summary:

```
summary = summarise(group_by(bnames2, name, year), total=sum(n))
filter(summary, name=='Vivian', year==1895)
  name year total
1 Vivian 1895    232

filter(bnames2, name=='Vivian', year==1895)
  year   name     prop   sex soundex births   n
1 1895 Vivian 0.000126 boy    V150 126650   16
2 1895 Vivian 0.000874 girl   V150 247112  216
```

group_by and summarise

- ❖ Printing out the summary object, we see:

```
summary
```

```
Source: local data frame [244,078 x 3]  
Groups: name
```

	name	year	total
1	Aaden	2008	959
2	Aaliyah	1994	1449
3	Aaliyah	1995	1254
4	Aaliyah	1996	831
5	Aaliyah	1997	1738
6	Aaliyah	1998	1398
..

Cleaner code with the %>% operator

- ❖ We can use the chaining operator to apply the dplyr functions in a more intuitive order:

```
bnames2 %>%  
  group_by(name, year) %>%  
  summarise(total=sum(n))
```

Source: local data frame [244,078 x 3]

Groups: name

		name	year	total
1	Aaden	2008	959	
2	Aaliyah	1994	1449	
3	Aaliyah	1995	1254	
4	Aaliyah	1996	831	
5	Aaliyah	1997	1738	
...

dplyr: start to finish

- ❖ Using `bnames` and `births`, how many Vivian's were born in each of the top 5 most popular years for the name?

```
bnames %>%  
  inner_join(., births, by=c('year', 'sex')) %>%  
  mutate(., n=round(prop * births)) %>%  
  group_by(., name, year) %>%  
  summarise(., total=sum(n)) %>%  
  filter(., name=='Vivian') %>%  
  arrange(., desc(total)) %>%  
  top_n(5)
```

Exercise 6: Join the Data Sets

- ❖ We want to calculate the number of children with a particular name born in each year.
 - Create a data frame bnames2 with the code below, replacing “*your_join*” by the join function you decide to use. Notice that the *by* parameter specifies a vector; how do you think it works?

```
bnames2 = your_join(bnames, births, by=c("year", "sex"))
```

- Add a new column to bnames2 whose value is the desired number. Name the column “n”.

Exercise 7: group_by and summarise

- ❖ We used a join function to construct bnames2.
 - Recover the births data frame from bnames2 with `group_by()` and `summarise()`. Call it `births2`. Hint: Look at the `births` data frame; what information does it provide and how we can obtain this information from `bnames2`?
 - If the result you obtain is not exactly the same as the original `births` data frame, what do you think caused the difference?

Exercise 8: Removing duplicates

- ❖ Recall our sample data frame, df

df		
	color	value
1	blue	1
2	black	2
3	blue	3
4	blue	4
5	black	5

- First, add a duplicate row to df and save the new data frame as df2. (Hint: check the structure of df2)
- Ignoring duplicates, compute the total sum of values for each color in df2.

Appendix

- ❖ Appendix of Exercise Solutions

Solution 1: filter

```
vivian <- bnames %>%
  filter(., name == "Vivian")
vivian$soundex[1]
[1] V150  # Find the soundex of your name first
filter(bnames, soundex=="V150")
```

```
Bnames %>%
  filter(bnames, sex=="girl"&(year==1900|year==2000))
```

```
nrow(filter(bnames, year > 2000 & prop > 0.01))
```

Solution 2: select

```
bnames %>%  
  select(., starts_with('y'),  
         starts_with('s'),  
         starts_with('p'))
```

```
select(bnames, -starts_with('y')) # either can be used  
select(bnames, -year)           # in this case
```

```
bnames %>% select(., contains('a'))
```

Solution 3: arrange

```
bnames %>%  
  arrange(., desc(prop))
```

```
bnames %>%  
  filter(., name=='Vivian') %>%  
  arrange(., desc(prop)) %>%  
  head(., 1)  
  year    name      prop   sex soundex  
1 1920 Vivian 0.003316 girl    V150
```

Solution 4: mutate

```
bnames %>%  
  mutate(., percentage=100*prop)
```

Solution 5: summarise

```
bnames %>%  
  filter(., name=='Vivian') %>%  
  summarise(., min(prop),  
            mean(prop),  
            max(prop))
```

Solution 6: Join the Data Sets

- ❖ We give the solution here because we will need to use bnames2.
 - Rows are identified if both year and sex are the same when we set the by parameter to a vector.

```
bnames2 = inner_join(bnames, births,  
                      by = c("year", "sex"))
```

- The function round() makes sure we have an integer. It doesn't really make sense to have 1.5 babies.

```
bnames2 = mutate(bnames2, n = round(prop * births))
```

Solution 7: group_by and summarise

- ❖ We used a join function to construct bnames2.

```
bnames2 %>%  
  group_by(., year, sex) %>%  
  summarise(births=first(births))  
  # Since within a year, the number of boys/girls  
  # born is independent of their name, we can just  
  # pick the first one.
```

- Because there is no data about 2009 in bnames, so the rows in 2009 from births are discarded by inner_join.

Solution 8: Removing duplicates

- ❖ Recall our sample data set, df

```
df2 <- df %>% rbind(., c("black", 5))
str(df2) #what's still awry?
```

```
df2 %>%
  transmute(., color, value=as.numeric(value)) %>%
  distinct() %>%
  group_by(color) %>%
  summarise(total=sum(value))
```