PSTAT 100 Lab1

Lab 1: Dataframe Overview with R

R is a programming language widely used for statistical computing and data analysis. In this tutorial, we will focus on the `dplyr` package for data manipulation.

Objectives

This lab covers the following topics:

- **Dataframe basics**
  - Creating dataframes
  - Dataframe indexing and attributes
  - Adding, removing, and renaming variables
- **Operations on dataframes**
  - Slicing (selecting rows and columns)
  - Filtering (selecting rows that meet certain conditions)
- **Grouping and aggregation**
  - Summary statistics (mean, median, variance, etc.)
  - Grouped summaries
  - Chaining operations and style guidelines
  - Pivoting

```r
# Load libraries
library(dplyr)
library(ggplot2)
```

Creating DataFrames & Basic Manipulations

A **dataframe** in R is a table where each column has a specific type (numeric, character, etc.). It has two main indices:

- **Column index**: Labels for each column (usually strings).
- **Row index**: Ordinal numbers representing each row.

In this tutorial, we will cover:

1. Creating dataframes from scratch.
2. Retrieving dataframe attributes.
3. Indexing dataframes.
4. Adding, removing, and renaming columns.

We will use the example dataset: **`fruit_info`**, which contains information about different fruits.

Creating DataFrames from Scratch

In R, we use the `data.frame()` function to create a dataframe. Let's create a dataframe for the **`fruit_info`** dataset.

Syntax 1: Using a List

In R, we use the `data.frame()` function to create a dataframe. Here's an example of creating a dataframe using a list.

Example: Creating a DataFrame from a Dictionary

```r
# Creating a dataframe using a list
fruit_info <- data.frame(
  Fruit = c("Apple", "Banana", "Cherry", "Date", "Elderberry"),
  Color = c("Red", "Yellow", "Red", "Brown", "Purple"),
  Weight = c(150, 120, 10, 25, 5)
```

```
)

# Display the dataframe
print(fruit_info)
```

```
       Fruit  Color Weight
1      Apple    Red    150
2     Banana Yellow    120
3     Cherry    Red     10
4       Date  Brown     25
5 Elderberry Purple      5
```

Syntax 2: Row-wise Tuples

Another way to create a dataframe is by specifying the rows directly. We use `data.frame()` again but creating from **Row-wise Tuples**.

Example: Creating a DataFrame from Row-wise Tuples

```
# Creating a dataframe using row-wise input
fruit_info_rowwise <- rbind(
  c("Apple", "Red", 150),
  c("Banana", "Yellow", 120),
  c("Cherry", "Red", 10),
  c("Date", "Brown", 25),
  c("Elderberry", "Purple", 5)
)

# Converting it to a data frame and assigning column names
fruit_info_rowwise <- as.data.frame(fruit_info_rowwise,
                                    stringsAsFactors = FALSE)
colnames(fruit_info_rowwise) <- c("Fruit", "Color", "Weight")

# Display the dataframe
print(fruit_info_rowwise)
```

```
       Fruit  Color Weight
1      Apple    Red    150
2     Banana Yellow    120
3     Cherry    Red     10
```

```
4       Date  Brown     25
5 Elderberry Purple      5
```

Retrieving Attributes

Once the dataframe is created, you may want to check its attributes like the column names, the number of rows, and the number of columns.

Dataframe Attributes

- `nrow()`: Number of rows.
- `ncol()`: Number of columns.
- `colnames()`: Names of the columns.
- `str()`: Structure of the dataframe.

```r
# Checking the number of rows and columns
num_rows <- nrow(fruit_info)
num_cols <- ncol(fruit_info)
num_rows
```

```
[1] 5
```

```r
num_cols
```

```
[1] 3
```

```r
# Displaying column names
col_names <- colnames(fruit_info)
col_names
```

```
[1] "Fruit"  "Color"  "Weight"
```

```r
# Displaying the structure of the dataframe
str(fruit_info)
```

```
'data.frame':   5 obs. of  3 variables:
 $ Fruit : chr  "Apple" "Banana" "Cherry" "Date" ...
 $ Color : chr  "Red" "Yellow" "Red" "Brown" ...
 $ Weight: num  150 120 10 25 5
```

Indexing DataFrames

In R, you can access the data in a dataframe in various ways.

Column Indexing

You can access columns by name or by column index.

```r
# Accessing columns by name
fruit_info$Fruit
```

```
[1] "Apple"      "Banana"     "Cherry"     "Date"       "Elderberry"
```

```r
# Accessing columns by index
fruit_info[[1]]
```

```
[1] "Apple"      "Banana"     "Cherry"     "Date"       "Elderberry"
```

Row Indexing

You can access rows by number or by condition.

```r
# Accessing rows by number
fruit_info[1, ]
```

```
  Fruit Color Weight
1 Apple   Red    150
```

```r
fruit_info[1:3, ]
```

```
   Fruit  Color Weight
1  Apple    Red    150
2 Banana Yellow    120
3 Cherry    Red     10
```

```r
fruit_info[c(2,4,6), ]
```

```
    Fruit  Color Weight
2  Banana Yellow    120
4    Date  Brown     25
NA   <NA>   <NA>     NA
```

```r
# Accessing rows by condition (Weight > 100)
fruit_info[fruit_info$Weight > 100, ]
```

```
   Fruit  Color Weight
1  Apple    Red    150
2 Banana Yellow    120
```

Adding, Removing, and Renaming Columns

Adding a Column

You can add a new column by assigning values to a new column name.

```r
# Adding a new column for price (just an example)
fruit_info$Price <- c(1.2, 0.5, 2.0, 3.5, 4.0)

# View updated dataframe
print(fruit_info)
```

```
       Fruit  Color Weight Price
1      Apple    Red    150   1.2
2     Banana Yellow    120   0.5
3     Cherry    Red     10   2.0
4       Date  Brown     25   3.5
5 Elderberry Purple      5   4.0
```

Removing a Column

You can remove a column using `select()` from the `dplyr` package.

```r
# Removing the Score column
library(dplyr)
```

```r
fruit_info <- select(fruit_info, -Price)

# View updated dataframe
print(fruit_info)
```

```
       Fruit  Color Weight
1      Apple    Red    150
2     Banana Yellow    120
3     Cherry    Red     10
4       Date  Brown     25
5 Elderberry Purple      5
```

Renaming Columns

Use `rename()` to change column names.

```r
# Renaming columns
fruit_info <- rename(fruit_info, Fruit_Name = Fruit)

# View updated dataframe
print(fruit_info)
```

```
  Fruit_Name  Color Weight
1      Apple    Red    150
2     Banana Yellow    120
3     Cherry    Red     10
4       Date  Brown     25
5 Elderberry Purple      5
```

Hand-on Exercise:

Question 1

(1) Adding a new column:

Using direct specification, add to the `fruit_info` table a new column called `rank1` containing integers 1, 2, 3, 4, and 5, which express your personal preference about the taste ordering for each fruit (1 is the tastiest; 5 is the least tasty). Make sure that the numbers utilized are unique - no ties are allowed.

YOUR ANSWER:

```
# replace with your codes
```

(2) Creating a modified dataframe:

Now, create a new dataframe `fruit_info_mod1` with the same information as `fruit_info`, but with an additional column called `rank2`. We'll start by making a copy of the `fruit_info` dataframe to create the new dataframe `fruit_info_mod1`.

YOUR ANSWER:

```
# replace with your codes
```

Question 2

(1) Adding `rank2` from `rank1` using indexing:

Using **indexing**, add a column called `rank2` to the `fruit_info_mod1` dataframe that contains the same values in the same order as the `rank1` column.

We will assign the values of `rank1` directly to the new `rank2` column, ensuring that the values are copied in the same order.

YOUR ANSWER:

```
# replace with your codes
```

When using the indexing `[]` approach, the `:` specifies that values are assigned to all rows of the data frame, so the array assigned to the new variable must be the same length as the data frame. What if we only assign values to certain rows?

(2) New column assignment with missing data:

For example, let's try adding a new column `rank3` and assign values only to the first two rows. Try running the cell below.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Partial Assignment**: We use row-based indexing (`1:2`) to assign values only to the first two rows of `rank3`. The remaining rows in `rank3` will have `NA` values by default because we didn't assign values to them.
- **Missing Values**: The unassigned rows will contain missing values (`NA`).

(3) Checking missing entry:

We can detect missing values using the `is.na()` method in R, which will return a logical dataframe indicating whether an entry is missing.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Missing Data Check**: The `is.na()` function checks for missing (`NA`) values in the dataframe. It returns a logical dataframe where `TRUE` indicates missing values.
- **Visualization**: This is helpful for identifying rows or columns with missing data.

It's often more helpful to see if any column has missing values. You can do this by appending `any()` to the `is.na()` function, which will return `TRUE` if there are any missing values in the column.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Missing Data Detection**: We use `colSums(is.na(fruit_info_mod1))` to check for missing data in each column. This will give us the number of missing entries per column.
- **Helpful for Clean-Up**: This helps identify which columns may need further attention or data cleaning.

(4) Removing column(s) with missing entry:

Once we've finished with some columns, we might want to remove them. In R, we can use `select()` from the `dplyr` package or direct indexing to remove columns.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Removing Columns**: We used the `select()` function from `dplyr` to remove the `rank3` column from the dataframe. The negative sign (`-`) indicates that the column should be removed.

- **Data Cleaning**: This is a common operation when you no longer need certain columns in the dataset.

Question 3

(1) Removing `rank` columns:

In this task, we will remove all `rank` columns you created in `fruit_info_mod1`.

In R, we can use the `select()` function from the `dplyr` package to remove columns. The `select()` function does not modify the original dataframe directly, but instead returns a new dataframe with the specified columns removed. We will assign the result to `fruit_info_original`.

To remove all columns that start with the word “rank”, we will use the `starts_with()` function in `dplyr`.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Removing Columns**: We used the `select()` function from the `dplyr` package with the `-` operator to exclude columns starting with the word “rank”. The `starts_with()` function is used to identify these columns.
- **Returning a New Table**: This operation does not modify the original dataframe but creates a new one (`fruit_info_original`) with the `rank` columns removed.

(2) Creating a new dataframe with capitalized column names:

Now, let’s create a new dataframe `fruit_info_mod2` with the same information as `fruit_info_original`, but with all column names capitalized.

First, we’ll start by creating a copy of `fruit_info_original`.

YOUR ANSWER:

```
# replace with your codes
```

Then, we need to rename the columns of `fruit_info_mod2` so that they begin with capital letters by using `toupper()` function.

YOUR ANSWER:

```
# replace with your codes
```

Operations on Data Frames

With some basics in place, here you'll see how to perform subsetting operations on data frames that are useful for tidying up datasets.

- **Slicing**: Selecting columns or rows in chunks or by position.
- **Filtering**: Selecting rows that meet certain criteria.

We will illustrate these operations using a dataset comprising counts of baby names born in California each year from 1990 to 2018.

```
# import baby names data
# Note: put the csv file of the data into the same folder as this .qmd file
baby_names = read.csv('baby_names.csv')

# preview first few rows
head(baby_names)
```

```
  State Sex Year      Name Count
1    CA   F 1990   Jessica  6635
2    CA   F 1990    Ashley  4537
3    CA   F 1990 Stephanie  4001
4    CA   F 1990    Amanda  3856
5    CA   F 1990  Jennifer  3611
6    CA   F 1990 Elizabeth  3170
```

Question 4

(1) Checking dimensions:

You've already seen how to examine dimensions using dataframe attributes. Check the dimensions of the `baby_names` dataset and store them in `dimensions_baby_names`.

In R, we use the `dim()` function to get the dimensions of a dataframe.

YOUR ANSWER:

```
# replace with your codes
```

(2) Counting distinct years:

Count the number of occurrences of each distinct year in the `baby_names` dataset. Create a object `occur_per_year` that displays the number of occurrences, ordered by year.

How many years are represented in the dataset? Store your answer as `num_years`.

In R, we can use the `table()` function to count the occurrences of each value in a column, and then use `sort()` to arrange them by year.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Sorting by Year**: We first count the occurrences of each year with the `table()` function, which gives us a named vector. The `order()` function is then used to sort the vector by the year, which is achieved by converting the names of the `occur_per_year` (which are stored as character strings) into numeric values with `as.numeric()`.
- **Number of Years**: The `unique()` function is used to get the distinct years, and `length()` gives the number of unique years.

Slicing: Selecting Rows and Columns

In this section, we'll cover two primary ways to slice a dataframe:

- **Using index names** to specify rows and columns.
- **Using integer positions** to specify rows and columns.

These methods are very useful for subsetting data and inspecting different portions of a dataframe.

1. Slicing with Index Names

To slice a dataframe by index names (column and row names), you can use the `[]` operator in R.

Example: Single Index

You can select a specific entry by specifying both the row and column names.

```
# Selecting a single row and column by index name
baby_names[2, 'Name']
```

```
[1] "Ashley"
```

- **Single Index**: In this example, we select the second row and the `Name` column from the `baby_names` dataframe using row index 2 and column name `'Name'`.

2. Slicing with a List of Indices

You can also select multiple rows and specific columns by passing a **list** of indices or names.

Example: List of Indices

```
# Selecting multiple rows and specific columns by index names
baby_names[c(2, 3), c('Name', 'Count')]
```

```
       Name Count
2    Ashley  4537
3 Stephanie  4001
```

- **List of Indices**: We use `c(2, 3)` to select rows 2 and 3, and `c('Name', 'Count')` to select the columns `'Name'` and `'Count'`.

3. Slicing with Consecutive Indices

You can select a range of rows or columns using the colon (`:`) operator.

Example: Consecutive Indices

```
# Selecting rows 2 to 10 and columns from 'Year' to 'Count'
baby_names[2:10,] %>% select(Year:Count)
```

```
   Year      Name Count
2  1990    Ashley  4537
3  1990 Stephanie  4001
4  1990    Amanda  3856
5  1990  Jennifer  3611
6  1990 Elizabeth  3170
7  1990     Sarah  2843
8  1990  Brittany  2737
9  1990  Samantha  2720
10 1990  Michelle  2453
```

- **Consecutive Indices**: The `2:10` syntax selects rows 2 through 10, and `'Year':'Count'` selects all columns from `Year` to `Count`.
- **`select(Year:Count)`**: This approach uses `dplyr`'s `select()` function to select columns starting from `'Year'` and ending at `'Count'`. It properly handles the selection of consecutive columns by name, ensuring that no errors occur.

4. Slicing with Integer Positions

In R, you can also slice data by integer positions using the `[,]` operator. This is similar to using `.iloc[]` in Python.

Example: Single Position

To select a specific entry by position, you can specify row and column indices.

```r
# Selecting a specific entry by position (row 2, column 3)
baby_names[2, 3]
```

```
[1] 1990
```

- **Single Position**: The `baby_names[2, 3]` selects the entry in the second row and third column by position.

5. Slicing with a List of Positions

You can select multiple rows and columns by specifying their integer positions.

Example: List of Positions

```r
# Selecting multiple rows and columns by position
baby_names[c(2, 3), c(3, 4)]
```

```
  Year      Name
2 1990    Ashley
3 1990 Stephanie
```

- **List of Positions**: The `c(2, 3)` selects the second and third rows, and `c(3, 4)` selects the third and fourth columns by position.

6. Slicing with Consecutive Positions

You can also slice consecutive rows or columns by specifying a range of positions.

Example: Consecutive Positions

```r
# Selecting rows 2 through 11 and columns 2 through 5
baby_names[2:11, 2:5]
```

```
   Sex Year      Name Count
2    F 1990    Ashley  4537
3    F 1990 Stephanie  4001
4    F 1990    Amanda  3856
5    F 1990  Jennifer  3611
6    F 1990 Elizabeth  3170
7    F 1990     Sarah  2843
8    F 1990  Brittany  2737
9    F 1990  Samantha  2720
10   F 1990  Michelle  2453
11   F 1990   Melissa  2442
```

- **Consecutive Positions**: The `2:11` selects rows 2 through 11, and `2:5` selects columns 2 through 5 based on their integer positions.

7. Slicing Rows Based on Conditions

In R, you can also slice rows based on specific conditions by using logical indexing. For example, you can select rows where a certain column satisfies a condition.

Example: Selecting Rows with `Count` Greater than 100

```r
# Slice rows where 'Count' is greater than 100
head(baby_names[baby_names$Count > 100, ])
```

```
  State Sex Year      Name Count
1    CA   F 1990   Jessica  6635
2    CA   F 1990    Ashley  4537
3    CA   F 1990 Stephanie  4001
4    CA   F 1990    Amanda  3856
5    CA   F 1990  Jennifer  3611
6    CA   F 1990 Elizabeth  3170
```

- **Slicing with Condition**: We use logical indexing (`baby_names$Count > 100`) to select rows where the `Count` column is greater than 100. This condition returns a logical vector that is used to subset the rows.

8. Slicing Rows Based on Column Value Range

You can slice rows based on a range of values in a column, for example, selecting rows where the `Year` column is between two values.

Example: Selecting Rows for Years 2000 to 2005

```r
# Slice rows where the 'Year' is between 2000 and 2005
head(baby_names[baby_names$Year >= 2000 & baby_names$Year <= 2005, ])
```

```
      State Sex Year     Name Count
36417    CA   F 2000    Emily  2958
36418    CA   F 2000   Ashley  2831
36419    CA   F 2000 Samantha  2579
36420    CA   F 2000  Jessica  2484
36421    CA   F 2000 Jennifer  2263
36422    CA   F 2000   Alyssa  2005
```

- **Range Condition**: Here, we use a logical condition to select rows where the `Year` is between 2000 and 2005 (inclusive). The logical condition is used within the indexing brackets to filter rows.

Question 5:

Look up the name of a friend! Store the name as `friend_name`. Based on the `name` column, set specific conditions by using logical indexing to slice rows for the name of your friends, i.e., Amanda, and the `Name`, Count`,`Sex`, and`Year`columns **in that order**, and store the data frame as`friend_slice`.

Step 1: Define the Friend's Name(s)

We'll store the names of your friends as `friend_name`.

Step 2: Slice the Data Based on the Name

We'll use the `%in%` operator to match multiple names and filter rows based on the `Name` column. Then, we'll select the desired columns (`Count`, `Sex`, `Year`).

YOUR ANSWER:

```
# replace with your codes
```

Filtering

Filtering is the process of sifting out rows according to a criterion. In R, this can be accomplished using logical indexing, which creates a vector of `TRUE`s and `FALSE`s based on a comparison. Let's walk through a simple example:

Example: Filter Names Based on Occurrence Count

Let's say we want to filter out all names with fewer than 1000 occurrences. First, we can define a logical vector that checks the `Count` column.

```r
# Create a logical vector where TRUE means Count is greater than 1000
arr <- baby_names$Count > 1000

# View the logical array
head(arr, sum(arr)*0.05)
```

```
  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [25]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
 [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [85] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [97] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[109] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[121] FALSE FALSE FALSE FALSE FALSE
```

Once we have the logical array, we can apply it to filter the dataframe and select only the rows where `Count` is greater than 1000.

```r
# Filter the dataframe using the logical array
baby_names_filtered <- baby_names[arr, ]

# View the filtered dataframe
head(baby_names_filtered)
```

```
  State Sex Year      Name Count
1    CA   F 1990   Jessica  6635
2    CA   F 1990    Ashley  4537
3    CA   F 1990 Stephanie  4001
4    CA   F 1990    Amanda  3856
5    CA   F 1990  Jennifer  3611
6    CA   F 1990 Elizabeth  3170
```

Checking Dimensions Before and After Filtering

To get a sense of how many rows were filtered, we can compare the dimensions of the original and filtered dataframes.

```r
# Compare dimensions before and after filtering
print(dim(baby_names))
```

```
[1] 190762      5
```

```r
print(dim(baby_names_filtered))
```

```
[1] 2517    5
```

Notice that the filtered dataframe is much smaller than the overall dateframe – only about **2000** rows correspond to a name occurring more than 1000 times in a year for a gender.

Common Comparison Operators

Here are some commonly used comparison operators in R for filtering data:

Symbol	Usage	Meaning
`==`	`a == b`	Does `a` equal `b`?
`<=`	`a <= b`	Is `a` less than or equal to `b`?
`>=`	`a >= b`	Is `a` greater than or equal to `b`?
`<`	`a < b`	Is `a` less than `b`?
`>`	`a > b`	Is `a` greater than `b`?
`!`	`!p`	Negates `p` (logical NOT)
`&`	`p & q`	`p` AND `q`
`|`	`p | q`	`p` OR `q`

What if instead you wanted to filter using multiple conditions? Here's an example of retrieving rows with counts exceeding 1000 for only the year 2001:

Example: Filtering with Multiple Conditions

You can combine multiple conditions to filter the dataframe in more complex ways. For example, let's retrieve all names where `Count` exceeds 1000 and the `Year` is 2001:

```r
# Filter using two conditions
filtered_data <- baby_names[(baby_names$Year == 2001) & (baby_names$Count > 1000), ]

# View the filtered data
head(filtered_data, dim(filtered_data)[1]*0.1)
```

```
      State Sex Year     Name Count
40184    CA   F 2001    Emily  2928
40185    CA   F 2001   Ashley  2715
40186    CA   F 2001 Samantha  2535
40187    CA   F 2001  Jessica  2244
40188    CA   F 2001   Alyssa  2059
```

```
40189    CA    F 2001   Jennifer   2026
40190    CA    F 2001    Natalie   1724
40191    CA    F 2001 Elizabeth   1704
40192    CA    F 2001     Alexis   1700
```

- **Combining Conditions:** In this example, we combine two conditions using the `&` operator. We filter for rows where `Year` equals 2001 and `Count` is greater than 1000.

Question 6:

Select the **girl** names in **2010** that were given more than **3000 times**, and store them as `common_girl_names_2010`.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Multiple Conditions:** The logical expression `(baby_names$Year == 2010) & (baby_names$Sex == "F") & (baby_names$Count > 3000)` filters rows where the year is 2010, the sex is female (`"F"`), and the count is greater than 3000.

Grouping and Aggregation

Grouping and aggregation are useful in generating data summaries, which are often important starting points in exploring a dataset.

Aggregation

Aggregation literally means 'putting together' (etymologically the word means 'joining the herd'). In statistics and data science, this refers to data summaries like an average, a minimum, or a measure of spread such as the sample variance or mean absolute deviation. From a technical point of view, operations that take multiple values as inputs and return a single output are considered summaries — in other words, statistics.

Some of the most common aggregations are:

- sum
- product
- count
- number of distinct values
- mean
- median
- variance
- standard deviation
- minimum/maximum
- quantiles

R provides built-in functions that compute most of these summaries, such as:

- `.sum()`
- `.prod()`
- `.mean()`
- `.median()`
- `.var()`
- `.sd()` (standard deviation)
- `.unique()` (number of distinct values)
- `.min()` and `.max()`
- `.quantile()`

To illustrate these operations, let's filter out all names in 1995.

```
# Filter for names in 1995
names_95 <- baby_names[baby_names$Year == 1995, ]

# View the filtered dataset
head(names_95, dim(names_95)[1]*0.001)
```

```
      State Sex Year      Name Count
18605    CA   F 1995   Jessica  4620
18606    CA   F 1995    Ashley  2903
18607    CA   F 1995 Stephanie  2858
18608    CA   F 1995  Jennifer  2697
18609    CA   F 1995  Samantha  2425
18610    CA   F 1995     Emily  2341
```

Example: Summing the Total Count of Names in 1995

We can compute the total number of occurrences of names in 1995 by summing the `Count` column.

```
# Total count for 1995
total_count_1995 <- sum(names_95$Count)

# Print the result
total_count_1995
```

```
[1] 494580
```

Example: Calculating the Average Frequency of Names in 1995

Next, let's calculate the average frequency of names in 1995.

```
# Average count for a name in 1995
average_count_1995 <- mean(names_95$Count)

# Print the result
average_count_1995
```

```
[1] 81.18516
```

Question 7

Find how often the most common name in 1995 was given and store this as `names_95_max_count`. Use this value to filter `names_95` and find which name was most common that year. Store the filtered dataframe as `names_95_most_common_name`.

YOUR ANSWER:

```
# replace with your codes
```

Hints:

- **Finding the Maximum Count:** We use `max()` to find the maximum value in the `Count` column of the `names_95` dataframe, which represents the most common name's count.
- **Filtering for the Most Common Name:** We then filter the dataframe for rows where `Count` is equal to `names_95_max_count`, retrieving the most common name(s) from 1995.

Grouping

What if you want to know the most frequent male and female names in 1995? If so, you'll need to repeat the above operations **group-wise** by `Sex`.

In general, any variable in a dataframe can be used to define a grouping structure on the rows (or, less commonly, columns). After grouping, any dataframe operations will be executed **within** each group, but not across groups. This can be used to generate **grouped summaries**, such as the maximum count for boys and girls.

The `group_by()` function in R (part of the `dplyr` package) defines such a structure. Below is how we can group the `names_95` dataframe by **sex**.

```
# Group by sex
library(dplyr)
names_95_bysex <- names_95 %>% group_by(Sex)

# Preview the grouped dataframe
head(names_95_bysex,)
```

```
# A tibble: 6 x 5
# Groups:   Sex [1]
  State Sex    Year Name      Count
  <chr> <chr> <int> <chr>     <int>
1 CA    F      1995 Jessica    4620
2 CA    F      1995 Ashley     2903
3 CA    F      1995 Stephanie  2858
4 CA    F      1995 Jennifer   2697
5 CA    F      1995 Samantha   2425
6 CA    F      1995 Emily      2341
```

Example: Number of Individuals by Sex

To count the total number of individuals in the data for 1995, grouped by sex, we can use the `sum()` function.

```
# Number of individuals by sex
names_95_bysex %>% summarise(total_count = sum(Count))
```

```
# A tibble: 2 x 2
  Sex   total_count
  <chr>       <int>
1 F          234552
2 M          260028
```

- **Summing by Group:** The `summarise()` function computes the total number of individuals (by summing the `Count` column) for each group (male and female).

Example: Most Common Name by Sex

To find the most common boy and girl names in 1995, we can group the data by `Sex` and then use the `which.max()` function to get the row with the highest `Count` for each sex.

```
# Group by sex
names_95_bysex <- names_95 %>% group_by(Sex)

# Find the most common name by sex using which.max() to get the index of the highest count
most_common_names_bysex <- names_95_bysex %>%
  summarise(most_common_name = Name[which.max(Count)],
            most_common_count = max(Count))

# View the most common names by sex
```

```r
print(most_common_names_bysex)
```

```
# A tibble: 2 x 3
  Sex   most_common_name most_common_count
  <chr> <chr>                        <int>
1 F     Jessica                       4620
2 M     Daniel                        5003
```

Notes:

- **Grouping by Sex:** We use `group_by(Sex)` to group the data by sex (male or female).
- **Finding Most Common Name:** `which.max(Count)` finds the index of the row with the maximum `Count` in each group. We then use this index to retrieve the name (Name) and its count (Count).
- **Summarising:** The `summarise()` function creates a summary dataframe that shows the most common name and its count for each sex.

Question 8:

Are there more girl names or boy names in 1995? Use the grouped dataframe `names_95_bysex` with the `count()` aggregation to find the total number of names for each sex. Store the female and male counts as `girl_name_count` and `boy_name_count`, respectively.

YOUR ANSWER:

```r
# replace with your codes
```

Hint:

- **Counting Names:** We use `filter()` to separate girl names (`Sex == "F"`) and boy names (`Sex == "M"`), and then `summarise(... = sum(Count))` to count the number of names in each group. We then print the counts for both boys and girls.

Chaining Operations

You have already seen examples of this, but R operations can be chained together in sequence. For example, `names_95$Count %>% max()` is a chain with two steps: first select the `Count` column (`$Count`); then compute the maximum (`max()`).

Grouped summaries are often convenient to compute in a chained fashion, rather than by assigning the grouped dataframe a new name and performing operations on the resulting object. For example, finding the total number of boys and girls recorded in the 1995 data can be done with the following chain:

```
# Streamlined calculation for total counts by sex
names_95 %>% group_by(Sex) %>% summarise(total_count = sum(Count))
```

```
# A tibble: 2 x 2
  Sex   total_count
  <chr>       <int>
1 F          234552
2 M          260028
```

> **Notes:**
>
> - **Chaining:** We first group the data by `Sex` with `group_by(Sex)`, then use `summarise()` to compute the sum of the `Count` column for each group.
> - **Efficiency:** This is a more concise and readable way to perform operations compared to creating **intermediate** variables.

Example: Filtering and Grouping in One Chain

We can take this even one step further and also perform the filtering in sequence as part of the chain:

```
# Filter by year and then group by sex
baby_names %>%
  filter(Year == 1995) %>%
  group_by(Sex) %>%
  summarise(total_count = sum(Count))
```

```
# A tibble: 2 x 2
  Sex   total_count
```

```
  <chr>        <int>
1 F           234552
2 M           260028
```

Example: Average Counts by Sex and Year

Let's compute the average counts of boy and girl names for each year from 1990 to 1995. This can be accomplished by grouping by both `Year` and `Sex`:

```r
# Average counts by sex and year
baby_names %>%
  filter(Year <= 1995) %>%
  group_by(Year, Sex) %>%
  summarise(average_count = mean(Count))
```

```
# A tibble: 12 x 3
# Groups:   Year [6]
    Year Sex   average_count
   <int> <chr>         <dbl>
 1  1990 F              70.1
 2  1990 M             115.
 3  1991 F              70.4
 4  1991 M             115.
 5  1992 F              68.7
 6  1992 M             111.
 7  1993 F              66.3
 8  1993 M             108.
 9  1994 F              66.4
10  1994 M             103.
11  1995 F              64.9
12  1995 M             105.
```

Example: Pivoting the Data

We can further pivot the table into a **wide** format by adding a few extra steps in the chain: changing the indices to columns, and specifying which column should be the new row index, which should be the new column index, and which values should populate the table.

```r
# Pivot the data
library(tidyr)
```

```r
baby_names %>%
  filter(Year <= 1995) %>%
  group_by(Year, Sex) %>%
  summarise(average_count = mean(Count)) %>%
  pivot_wider(names_from = Year, values_from = average_count)
```

```
# A tibble: 2 x 7
  Sex   `1990` `1991` `1992` `1993` `1994` `1995`
  <chr>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
1 F       70.1   70.4   68.7   66.3   66.4   64.9
2 M      115.   115.   111.   108.   103.   105.
```

Notes:

- **Pivoting:** The `pivot_wider()` function in `tidyr` packaeg reshapes the data, making Year the column names and the values of `average_count` the cell values, effectively converting the data into a **wider** format.

Style Comment: break long chains over multiple lines with indentation

The above chain is too long to be readable in one line. To balance the readability of codes with the efficiency of chaining, it is good practice to break long chains over several lines, with appropriate indentations.

- **Separate comparisons by spaces** (`a < b` as `a < b`).
- **Split chains longer than 30-40 characters** over multiple lines.
- **Split lines between delimiters** `(, )`.
- **Increase indentation for lines between delimiters**.]
- **For chained operations**, try to get each step in the chain shown on a separate line.
- **For functions with multiple arguments**, split lines so that each argument is on its own line.

Question 9:

Write a chain with appropriate style to display the (first) most common boy and girl names in each of the years 2005-2015. Do this in two steps:

1. First, filter `baby_names` by `Year`, then group by `Year` and `Sex`, and find the indices of the first occurrence of the largest counts. Store these indices as `ind`.
2. Then, use the stored indices to slice `baby_names` so as to retrieve the rows corresponding to each most frequent name each year and for each sex; then pivot this table so that the columns are **years**, the rows are **sexes**, and the entries are **names**. Store this as `pivot_names`.

YOUR ANSWER:

```
# replace with your codes
```

Hints:
Step 1:

- **Filtering**: We filter the data to include only the years 2005-2015.
- **Grouping**: We group by `Year` and `Sex`, then use `mutate()` to create a new column `most_common_index` which stores the index of the row with the highest `Count` for each `Year` and `Sex` combination.
- **Ungrouping**: After using `mutate()`, we call `ungroup()` to remove the grouping structure since it's no longer needed for further operations.

Step 2:

- **Filtering for Most Frequent Names**: We filter the `baby_names` dataframe to keep only the rows where the `Year` and `Sex` combinations are present in the `ind` dataframe (where we calculated the most common name's index).
- **Pivoting**: We use `pivot_wider()` to transform the data such that the columns represent `Year`, the rows represent `Sex`, and the values are the most common `Name`.

Submission

1. Save the notebook.
2. Restart the kernel and run all cells. (**CAUTION**: if your notebook is not saved, you will lose your work.)

3. Carefully look through your notebook and verify that all computations execute correctly. You should see **no errors**; if there are any errors, make sure to correct them before you submit the notebook.

4. Download the notebook as an `.qmd` file. This is your backup copy.

5. Export the notebook as PDF and upload to Canvas.