CHAPTER 4

Exploratory Data Analysis with Tidyverse

OBJECTIVE

In this chapter, you will learn to conduct a data analysis project with R, using the `tidyverse` package. Of all R packages, `tidyverse` is the one you will likely use the most in your data analysis tasks. It provides many very powerful functions for reading or transforming large datasets (millions of records and thousands of variables), as well as for creating various summary statistics. After reading this chapter, I expect you to be able to take a dataset in Excel and create several types of summary tables using the R package `tidyverse`.

Contents

4.1 Introduction

The primary objective of this chapter is to introduce what I consider to be the basic tools any analyst needs to perform data analysis. Using these R tools, the analyst is expected to be more efficient than if the same analysis was done with Excel. Reproducibility, automation, precision and documentation are expected to be the 4 biggest gains you can expect to obtain using R. The price to pay for this gain in efficiency is to comply with the requirements of organizing all of your input data in a logical way before analysis can begin, and to learn some R commands. In R, you will not be looking at your data with your naked eyes before a specific calculation is done, as you would in Excel. Instead, R will interpret your commands and do the work for you. This would be impossible with a disorganized input dataset.

`Tidyverse` is one of the most popular packages among R analysts. It provides a plethora of tools for making data wrangling easier, more efficient and will be at the center of the discussions in this chapter. There is so much that you can do with `tidyverse` that it would be unrealistic and inefficient to present a comprehensive review of its capability. I want to help Excel users have a smooth transition to R as I did myself several years ago. After you gain experience with R, you may obtain more information on `tidyverse` by visiting the web pages https://www.tidyverse.org/ or https://tidyverse.tidyverse.org/.

Although `tidyverse` is often referred to as an R package, it is actually a collection of R packages that are popular among analysts. To use `tidyverse`, you need to install it on your machine, then load it to the current R session. Installing `tidyverse` on your machine is done by going to the RStudio console pane and typing the following command:

```
> install.packages("tidyverse")
```

Once installed, you can load `tidyverse` to the current environment either from Rstudio console, or within your R script with the following com-

mand: `library(tidyverse)`. This command automatically loads the core `tidyverse` packages, which is a collection of packages you will need in your every day analysis. I will occasionally mention these specific packages as we move along.

In section 4.2, you will learn the basics of dataset manipulation and the calculation of summary statistics. Section 4.3 addresses the important problem of combining different datasets in R. Advanced statistical techniques such as regression analysis, or the Analysis Of Variance (ANOVA) were briefly discussed in chapter 2, but are out of scope for this book.

4.2 Basic Analysis of R Datasets

One of the built-in datasets in R is named `USPersonalExpenditure` and consists of the United States personal expenditures (in billions of dollars) in the following 5 categories:

- "food and tobacco"
- "household operation"
- "medical and health"
- "personal care"
- "private education"

This data is available for the years 1940, 1945, 1950, 1955 and 1960. I am going to use this information to illustrate different techniques for manipulating R datasets with `tidyverse`.

> *You may want to know that typing the command* `> data()` *on RStudio console will display all built-in datasets in R. You can get a detailed description of a particular dataset such as* `USPersonalExpenditure` *by typing the command* `> ?USPersonalExpenditure` *on the RStudio console (do not omit the question mark "?" before typing the dataset name).*

If you type `> USPersonalExpenditure` on the console, you will obtain the following output:

Table 4.1 : R Built-in R Dataset `USPersonalExpenditure`

	1940	1945	1950	1955	1960
Food and Tobacco	22.200	44.500	59.60	73.2	86.80
Household Operation	10.500	15.500	29.00	36.5	46.20
Medical and Health	3.530	5.760	9.71	14.0	21.10
Personal Care	1.040	1.980	2.45	3.4	5.40
Private Education	0.341	0.974	1.80	2.6	3.64

I want to make a few comments about this dataset :

- Table 4.1 is an R data frame, and not a tibble[1]. But I previously recommended to always use tibbles. Therefore, I need to convert this data frame to a tibble, causing the loss of expense description. I will then need to create a new column of expense description to the new tibble.

- The column names of this data frame are numbers. For easy reference later on, I will replace all of them with y1940, y1945, y1950, y1955, y1960 so that the variable names start with a character, and not with a number.

These 2 objectives will be achieved after running Script 4.1. This script file is divided in 2 parts. Part 1 renames the data frame from its original name `USPersonalExpenditure` to a shorter an more convenient name `spending.df`. My data frame or tibble naming convention is to end the names with ".df" so I can easily identify data frames by their names. Next, I change the column names by left-padding the "y" character to the original numeric names. This yields the new data frame shown in Table 4.2 .

In part 2, I first create a vector variable named `expense`, which contains all row names of the `spending.df` data frame, which in turn is converted to a tibble named `xpense.df` using the `as_tibble` function (note that the

[1]Note that tibbles do not have row names and the description of expenses in the above table are only row names and not elements of a dataset column (no column name).

tibble only contains the 4 numeric columns). Finally, using the `add_column()` function, I add `expense` as a new column to the `xpense.df` tibble, right before the first variable `y1940`. The result is the complete tibble shown in Table 4.3 and which I will analyze in the next few sections.

Script 4.1. Script for converting the data frame `USPersonalExpenditure` to tibble

```
library(tidyverse)
#Part 1: Renaming the dataset and column labels
spending.df <- USPersonalExpenditure #rename dataset
clabels <- c("y1940","y1945","y1950","y1955","y1960")
colnames(spending.df) <- clabels

#Part 2: Convert dataframe to tibble and add new expense column
expense <- rownames(spending.df) #assign row names to a variable
xpense.df <- as_tibble(spending.df) #convert dataframe to tibble
xpense.df <- add_column(xpense.df,expense,.before = "y1940")

print(spending.df)
print(xpense.df)
```

End of Script

Table 4.2 : The `spending.df` data frame after renaming its column labels

	y1940	y1945	y1950	y1955	y1960
Food and Tobacco	22.200	44.500	59.60	73.2	86.80
Household Operation	10.500	15.500	29.00	36.5	46.20
Medical and Health	3.530	5.760	9.71	14.0	21.10
Personal Care	1.040	1.980	2.45	3.4	5.40
Private Education	0.341	0.974	1.80	2.6	3.64

Table 4.3 : The newly-created tibble `xpense.df`

	expense	y1940	y1945	y1950	y1955	y1960
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Food and Tobacco	22.2	44.5	59.6	73.2	86.8
2	Household Operation	10.5	15.5	29	36.5	46.2
3	Medical and Health	3.53	5.76	9.71	14	21.1
4	Personal Care	1.04	1.98	2.45	3.4	5.4
5	Private Education	0.341	0.974	1.8	2.6	3.64

4.2.1 *Exploring R Datasets*

Before you start analyzing a dataset, you must first inspect and explore it to become familiar with its content. Inspect and explore are 2 tasks, which are not near as important in Excel as they are in R. Remember that in Excel, you can manually select the part of your worksheet you want to process before applying a function to it. In R, you must provide instructions for the software to do it for you. Therefore, your input data must be better organized and you must know its structure very well. Otherwise, you will never be able to give the right instructions to R for accomplishing a specific task.

I typically use 3 R functions to perform a preliminary exploration of my dataset. These functions are the following:

- `> head(xpense.df,n=3)` returns the first 3 records of the data frame `xpense.df`. It is the head of the dataset, defined by the top 3 observations.

	expense	y1940	y1945	y1950	y1955	y1960
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	Food and Tobacco	22.2	44.5	59.6	73.2	86.8
2	Household Operation	10.5	15.5	29	36.5	46.2
3	Medical and Health	3.53	5.76	9.71	14	21.1

- `> tail(xpense.df,n=3)` returns the last 3 records of the data frame `xpense.df`. It is the tail of the dataset defined by the last 3 observations.

```
  expense             y1940 y1945 y1950 y1955 y1960
  <chr>               <dbl> <dbl> <dbl> <dbl> <dbl>
1 Medical and Health 3.53  5.76   9.71  14   21.1
2 Personal Care      1.04  1.98   2.45   3.4  5.4
3 Private Education  0.341 0.974  1.8    2.6  3.64
```

- `> summary(xpense.df)` returns basic summary statistics for each numeric variables in your dataset. The `summary()` function gives you a glimpse into the distribution of each variable. You get the smallest value (Min.), the first quartile, the median, the mean, the third quartile and the maximum value, as shown below.

```
   expense              y1940           y1945           y1950      
 Length:5           Min.   : 0.341   Min.   : 0.974   Min.   : 1.80  
 Class :character   1st Qu.: 1.040   1st Qu.: 1.980   1st Qu.: 2.45  
 Mode  :character   Median : 3.530   Median : 5.760   Median : 9.71  
                    Mean   : 7.522   Mean   :13.743   Mean   :20.51  
                    3rd Qu.:10.500   3rd Qu.:15.500   3rd Qu.:29.00  
                    Max.   :22.200   Max.   :44.500   Max.   :59.60  
     y1955           y1960      
 Min.   : 2.60   Min.   : 3.64  
 1st Qu.: 3.40   1st Qu.: 5.40  
 Median :14.00   Median :21.10  
 Mean   :25.94   Mean   :32.63  
 3rd Qu.:36.50   3rd Qu.:46.20  
 Max.   :73.20   Max.   :86.80  
```

4.2.2 *Slicing Datasets*

Exploring small portions of a bigger dataset is another effective way of learning about its content. R offers a variety of powerful functions you can use to extract specific information from your dataset. As a beginner, you need to focus on a few useful functions, and not make the mistake of attempting to know it all.

- **Selecting columns**

  A common task many Excel users perform is to hide columns not needed for a specific analysis or unhide what was previously hidden. This allows the analyst to have easy access to the most relevant data columns, without having to constantly scan a large number of columns or variables. To accomplish this task, R offers a far superior capability with the powerful `select()` function. To see this, consider the personal expense data of Table 4.3 for illustration purposes only[2].

  If you want to select specific columns such as `expense`, `y1955` and `y1960`, then you will type the following command:

  `select(xpense.df,expense,y1955,y1960)`,

  in a script or directly on RStudio console. This will yield the following data extract:

```
> select(xpense.df,expense,y1955,y1960)
# A tibble: 5 x 3
  expense             y1955 y1960
  <chr>               <dbl> <dbl>
1 Food and Tobacco     73.2 86.8
2 Household Operation  36.5 46.2
3 Medical and Health   14   21.1
4 Personal Care         3.4  5.4
5 Private Education     2.6  3.64
```

  Now, suppose that you want all columns from Table 4.3 , except `"y1940"`. You can do this with the command `select(xpense.df,-y1940)`. If you want to exclude `"y1940"` and `"y1945"` from the dataset, you use `select(xpense.df,-c(y1940,y1945))`.

  To fully appreciate how powerful the `select()` function is, suppose you only want to look into personal expenses in the fifties. The solution would

[2]The `select()` function will prove very useful when a dataset contains a large number of columns.

be to select all variables, which start with `"y195"`. This solution would be implemented as follows:

```
select(xpense.df, expense, starts_with("y195"))
```

After executing this command, the RStudio console will look like this:

```
> select(xpense.df,expense,starts_with("y195"))
# A tibble: 5 x 3
  expense             y1950 y1955
  <chr>               <dbl> <dbl>
1 Food and Tobacco    59.6   73.2
2 Household Operation 29     36.5
3 Medical and Health   9.71  14
4 Personal Care        2.45   3.4
5 Private Education    1.8    2.6
```

Now, suppose you only want to track personal expenses at the beginning of each decade. The solution would be select all variables, which end with a 0. This solution would be implemented as follows:

```
select(xpense.df,expense,ends_with("0"))
```

.

This command will produce the following output:

```
> select(xpense.df,expense,ends_with("0"))
# A tibble: 5 x 4
  expense              y1940 y1950 y1960
  <chr>                <dbl> <dbl> <dbl>
1 Food and Tobacco    22.2   59.6  86.8
2 Household Operation 10.5   29    46.2
3 Medical and Health   3.53   9.71 21.1
4 Personal Care        1.04   2.45  5.4
5 Private Education    0.341  1.8   3.64
```

An alternative way of restricting your analysis to personal expenditures in the fifties would be to retain all variables that contain the string of characters `"95"`. You would do it with the following command:

```
select(xpense.df,expense, contains("95")).
```

Another way of resolving this same problem using the `starts_with()` function is: `select(xpense.df, expense,starts_with("y195"))`. Note that you can also use column numbers with the `select()` function. For example, `select(xpense.df,1:3)` extracts the first 3 columns, whereas `select(xpense.df,c(1,3))` extracts columns 1 and 3.

> *The `select()` function is very powerful, and can be used for selecting columns programmatically. That is, you can make R execute commands within the `select()` function by supplying other functions as parameters (e.g. `starts_with()` was used in the previous paragraph as parameter).*

Once again, consider the `xpense.df` dataset of Table 4.3 .

– **Selecting 1 Column.** Suppose you want to select from `xpense.df`, a single column such as `"y1945"`. Then you have 2 options. Either you select it as a one-column dataset, or as a numeric vector.

To create a one-column tibble, use `select(xpense.df,y1945)`, or `xpense.df["y1945"]`, or `xpense.df[3]`. These commands will all return the following :

```
> xpense.df["y1945"]
# A tibble: 5 x 1
   y1945
   <dbl>
1 44.5
2 15.5
3  5.76
4  1.98
5  0.974
```

One-variable or one-column datasets are uncommon in practice, since one often prefers the vector, which is the basic data structure in R. If you want to extract column `y1945` as a vector, you can do so with `xpense.df$y1945` or `xpense.df[["y1945"]]`, or `xpense.df[[3]].` Each of these commands will yield the following output:

```
[1] 44.500 15.500  5.760  1.980  0.974
```

Another tip you may want to know is that you can create a new dataset by removing a specific column or variable you do not want to see. For example `select(xpense.df,-y1945)` or `xpense.df[-3]` will create a version of the `xpense,df` dataset without the `y1945` variable.

– **Selecting Multiple columns**

If you want a two-column tibble, which is based on the variables `"y1945"` and `"y1955"`, then you would need to execute one of the following commands: `select(xpense.df,y1945,y1955)`, or `xpense.df[c("y1945","y1955")]`, or `xpense.df[c(3,5)].` The result will be the following dataset:

```
# A tibble: 5 x 2
   y1945 y1955
   <dbl> <dbl>
1 44.5    73.2
2 15.5    36.5
3  5.76   14
4  1.98    3.4
5  0.974   2.6
```

- **Selecting Rows**

Another task Excel users perform often is to hide specific rows that are not needed, or unhide previously hidden rows that have become relevant. `Tidyverse` offers a very useful `filer()` function. Suppose that in Table

4.3 , you want to select all expenses that exceeds 6.0 billion dollars in 1940 and 1950. This query can be performed as follows:

```
filter(xpense.df,y1950>6.0 & y1940>6.0)
```.

As it turned out, only the following 2 records meet the required condition:

```
> filter(xpense.df,y1950>6.0 & y1940>6.0)
# A tibble: 2 x 6
  expense             y1940 y1945 y1950 y1955 y1960
  <chr>               <dbl> <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco     22.2  44.5  59.6  73.2  86.8
2 Household Operation  10.5  15.5  29    36.5  46.2
```

You can filter records where personal expenditures were 29 billion dollars in 1950 by typing `filter(xpense.df,y1950==29)` in the RStudio console. Note that the logical "equal" operator in R is "==" as opposed to "=". The regular equal sign ("=") is interpreted in R as an assignment operator similar to `"<-"`.

Here is a list of the most commonly-used logical operators in R.

```
Operator    Description
-----------------------
    <       less than
    <=      less than or equal to
    >       greater than
    >=      greater than or equal to
    ==      exactly equal to
    !=      not equal to
    !x      Not x
    x | y   x OR y
    x & y   x AND y
 isTRUE(x)  test if X is TRUE
```

You can find a more detailed description of these operators in section A.2 of appendix A.

- **Sorting Datasets**

  Although there are several ways of sorting an R dataset, I often use the `tidyverse` approach. In `tidyverse`, sorting a tibble[3] amounts to `arranging` its rows in descending or ascending order by one or multiple columns used as sort keys.

  To sort your dataset, you will use the `arrange()` function. For example, you can sort the dataset of Table 4.3 (or arrange its rows) in ascending order (the default option) by the magnitude of the 1945 personal expenditures, using the following lines of code:

```
library(tidyverse)
s1945.df <- arrange(xpense.df,y1945)
print(s1945.df)
```

  Executing the above script will produce the following output in the RStudio console:

```
> s1945.df <- arrange(xpense.df,y1945)
> print(s1945.df)
# A tibble: 5 x 6
  expense              y1940  y1945 y1950 y1955 y1960
  <chr>                <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Private Education    0.341  0.974  1.8    2.6  3.64
2 Personal Care        1.04   1.98   2.45   3.4  5.4
3 Medical and Health   3.53   5.76   9.71  14   21.1
4 Household Operation 10.5   15.5   29     36.5 46.2
5 Food and Tobacco    22.2   44.5   59.6   73.2 86.8
```

  You can see that the script file has created a new dataset named `sq945.df`, which is sorted by `y1945` in ascending order. If you want to sort the `s1945.df` dataset by `y1945` in descending order, you would do it as follows:

[3]Remember that throughout this book, I assume all datasets to be tibbles, and use the words dataframe, dataset and tibble interchangeably.

`arrange(s1945.df,desc(y1945))`.

Note that the first argument of function `arrange()` function is the dataset name, followed by one or several names of columns by which the sorting will be done. These column names must be separated with commas.

- **The slice() family of functions**

  In the course of your analysis, you may want to extract the records associated with the 2 highest or lowest values of a specific variable, which could also be the current ranking of the records. Here is where the very useful `slice()` family of functions comes in. When selecting rows, consider the functions `head()`, `tail()`, `filter()`, and the `slice()` family of functions. One of these functions will likely resolve your problem.

  Once again, I want to use the dataset of Table 4.3 to illustrate the `slice()` family of functions.

  – `slice(xpense.df,n=c(1,4))` This will extract the first and 4th record of the `xpense.df` dataset. Executing this command produces the following outcome:

```
> slice(xpense.df,n=c(1,4))
# A tibble: 2 x 6
  expense          y1940 y1945 y1950 y1955 y1960
  <chr>            <dbl> <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco 22.2  44.5  59.6   73.2  86.8
2 Personal Care     1.04  1.98  2.45   3.4   5.4
```

  While the first parameter of function `slice()` is the dataset name, the second parameter `n=` will contain a vector of numbers representing the row numbers that must be extracted from the dataset. The function call `slice(xpense.df,n=3)` for example will return the third row of the `xpense.df` dataset.

- `slice_head(xpense.df,n=2)`, will return the top 2 rows in dataset `xpense.df`. This could be useful for datasets that are already sorted. This function is similar to the `head()` function.
- `slice_tail(xpense.df,n=2)`, will return the bottom 2 rows in dataset `xpense.df` and is similar to the `tail()` function. Again, this could be informative if the dataset is already sorted (or arranged).
- `slice_min(xpense.df,order_by = y1945,n=2)` will return the 2 rows associated with the 2 lowest y1945 values.

```
> slice_min(xpense.df,order_by = y1945,n=2)
# A tibble: 2 x 6
  expense           y1940 y1945 y1950 y1955 y1960
  <chr>             <dbl> <dbl> <dbl> <dbl> <dbl>
1 Private Education 0.341 0.974  1.8    2.6  3.64
2 Personal Care     1.04  1.98   2.45   3.4  5.4
```

Function `slice_min()` does not require the dataset to be sorted, yet can help you explore it with respect to the magnitude of the `y1945` variable.

- `slice_max(xpense.df,order_by = y1945,n=2)` will return the 2 rows associated with the 2 highest y1945 values, and does not require the dataset to be sorted. Executing it produces the following output:

```
> slice_max(xpense.df,order_by = y1945,n=2)
# A tibble: 2 x 6
  expense             y1940 y1945 y1950 y1955 y1960
  <chr>               <dbl> <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco     22.2  44.5  59.6  73.2  86.8
2 Household Operation  10.5  15.5  29    36.5  46.2
```

- `slice_sample(xpense.df,n=3)` will return 3 records, randomly selected (without replacement) from the `xpense.df` dataset. Here is an example of output you will obtain after calling this function:

```
> slice_sample(xpense.df,n=3)
# A tibble: 3 x 6
```

```
  expense              y1940  y1945 y1950 y1955 y1960
  <chr>                <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Personal Care        1.04   1.98   2.45   3.4  5.4
2 Household Operation 10.5   15.5   29     36.5 46.2
3 Private Education    0.341  0.974  1.8    2.6  3.64
```

- **Adding New Columns/Variables**

  You learned earlier in this section that you can use the `select()` function to eliminate columns you do not want to use. But can you add new columns that do not exist in your dataset? The answer is yes, you can achieve this by using the `mutate()` function.

  Consider the dataset of Table 4.3 , and suppose that you want to add 2 columns to it. The first column is the mean personal expenditure of the last 2 years 1955 and 1960. The second column is a record id assigned to each record, sequentially from 1 to 5 in increments of 1. This task can be accomplished with the following command:

  ```
  mutate(xpense.df, av55_60=(y1955+y1960)/2, rec_id=1:5)
  ```

  Executing this command will yield the following outcome on RStudio console:

```
> mutate(xpense.df,av55_60=(y1955+y1960)/2,rec_id=1:5)
# A tibble: 5 x 8
expense             y1940  y1945 y1950 y1955 y1960 av55_60 rec_id
<chr>               <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl>   <int>
Food and Tobacco    22.2   44.5   59.6   73.2 86.8   80        1
Household Operation 10.5   15.5   29     36.5 46.2   41.4      2
Medical and Health   3.53   5.76   9.71  14   21.1   17.6      3
Personal Care        1.04   1.98   2.45   3.4  5.4    4.4      4
Private Education    0.341  0.974  1.8    2.6  3.64   3.12     5
```

> *Note that the `add_column()` function used in Script 4.1 can also add new columns/variables to a dataset. However, this function can only access objects in the workspace and cannot access other variables in the dataset like the `mutate()` function.*

- **Renaming Columns**

  `tidyverse` allows you to rename your dataset columns using the 2 functions `rename()` and `rename_with()`. On RStudio console, the command `> rename(xpense.df,year1940=y1940)` creates a new dataset from `xpense.df`, after changing the old variable name `y1940` to the new one `year1940`. If you execute this command, you will obtain the following outcome:

```
> rename(xpense.df,year1940=y1940)
# A tibble: 5 x 6
  expense             year1940  y1945 y1950 y1955 y1960
  <chr>                  <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco      22.2   44.5   59.6   73.2 86.8
2 Household Operation   10.5   15.5   29     36.5 46.2
3 Medical and Health     3.53   5.76   9.71  14   21.1
4 Personal Care          1.04   1.98   2.45   3.4  5.4
5 Private Education      0.341  0.974  1.8    2.6  3.64
```

  If you want to save this newly-crated dataset then assign the return of the `rename()` function to an R object as follows:

```
> new_xpense.df <- rename(xpense.df,year1940=y1940)
```

  The `rename_with()` function on the other hand, makes it easier to rename variables programmatically, and can be useful for renaming not one variable, but a series of variables. For example, if you want to change all column names in the `xpense.df` to uppercase letters, you can do it as

follows: `> rename_with(xpense.df,toupper)`. If you only want to rename all variables that start with letter `"y"`, it can be done with the command `> rename_with(xpense.df,toupper,starts_with("y"))`. If you execute these 2 commands, you will get the following 2 outputs:

```
> rename_with(xpense.df,toupper)
# A tibble: 5 x 6
  EXPENSE              Y1940  Y1945 Y1950 Y1955 Y1960
  <chr>                <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco    22.2   44.5   59.6   73.2 86.8
2 Household Operation 10.5   15.5   29     36.5 46.2
3 Medical and Health   3.53   5.76   9.71  14   21.1
4 Personal Care        1.04   1.98   2.45   3.4  5.4
5 Private Education    0.341  0.974  1.8    2.6  3.64


> rename_with(xpense.df,toupper,starts_with("y"))
# A tibble: 5 x 6
  expense              Y1940  Y1945 Y1950 Y1955 Y1960
  <chr>                <dbl>  <dbl> <dbl> <dbl> <dbl>
1 Food and Tobacco    22.2   44.5   59.6   73.2 86.8
2 Household Operation 10.5   15.5   29     36.5 46.2
3 Medical and Health   3.53   5.76   9.71  14   21.1
4 Personal Care        1.04   1.98   2.45   3.4  5.4
5 Private Education    0.341  0.974  1.8    2.6  3.64
```

As you can see, R provides so many powerful functions for managing your datasets that with a single line of code, you can achieve results that would take a long time with Excel.

4.2.3 *Using the Pipe Operator*

So far, you have learned how to use various functions to explore your R datasets. However, as you move on to conduct important projects, you must be organized and must develop a good work ethic, or you might end up with

a script file too cluttered to be readable. It will be error-prone and difficult to maintain.

> *I can tell from experience, that if you are going to make your R journey successful, you must know what the powerful* Pipe Operator *is, and how to use it effectively.*

The pipe operator is denoted by `%>%` . Unless you are a computer scientist, this is likely the first time you are seeing this weird combination of symbols. To see how useful it is, consider the dataset of Table B.1 in appendix B, containing US 2020 quarterly Gross Domestic Product (GDP) statistics broken down by state, and whose extract is shown in Table 4.4 below[4]. Let this dataset be called `state.df`.

Table 4.4 : 2020 US Quarterly Gross Domestic Product (GDP) by State - Extract of Table B.1 of Appendix B

```
# A tibble: 51 x 7
   region      state           Q1GDP  Q2GDP  Q3GDP  Q4GDP  pop2019
   <chr>       <chr>           <dbl>  <dbl>  <dbl>  <dbl>    <dbl>
 1 New England Connecticut    284418  258045 278963 284265 3558382
 2 New England Maine           70824   64445  70412  71406 1342845
 3 New England Massachusetts  597756  545422 591547 595183 6887571
 4 New England New Hampshire   90373   81024  88842  90245 1358406
 5 New England Rhode Island    61799   56825  61273  62327 1054930
 6 New England Vermont         34378   30804  34028  34529  623417
 7 Mideast     Delaware        77068   71628  76380  78070  970344
 8 Mideast     District of C  146252  138511 145338 148118  701570
 9 Mideast     Maryland       421334  387137 414207 420021 6010629
10 Mideast     New Jersey     638654  577108 624411 634144 8872796
# ... with 41 more rows
```

Assume that you want to conduct an analysis that focuses on quarter 4 GDP data and on Mideast states. In other words, the final analytic file (named `mideast.df`) you need for analysis is the following:

[4]You may download the entire dataset using the link: https://bit.ly/3zVi72k

```
# A tibble: 6 x 3
  state                gdpQ4y2020  pop2019
  <chr>                     <dbl>    <dbl>
1 Delaware                  78070   970344
2 District of Columbia     148118   701570
3 Maryland                 420021  6010629
4 New Jersey               634144  8872796
5 New York                1777389 19428617
6 Pennsylvania             793893 12795687
```

The very first problem you must tackle is to figure out how to go from the main dataset of 51 records and 7 variables, to the smaller dataset of 6 records and 3 variables. To resolve this problem, you need the `select()` function to keep the columns you want, and the `filter()` function to keep the rows associated with the Mideast region. Here are 3 possible ways you can implement this in R:

- **Solution 1: Intermediate Datasets**

```
rowx.df <- filter(state.df,region=="Mideast")
mideast.df <- select(rowx.df,state,Q4GDP,pop2019)
print(mideast.df)
```

- **Solution 2: Nested Functions**

```
mideast.df <- select(filter(state.df,region=="Mideast"),
                     state,Q4GDP,pop2019)
print(mideast.df)
```

- **Solution 3: The Pipes**

```
mideast.df <- state.df %>%
  filter(region=="Mideast") %>%
  select(state,Q4GDP,pop2019)
print(mideast.df)
```

Solution 1 requires that you explicitly create and name intermediate datasets for each action you perform. Creating and naming a large number of intermediate datasets that are never going to be used again slows down the analysis process in addition to cluttering your workspace with new R objects you do not need to create. In complex analysis projects where different functions are called on many occasions, this approach makes script files more difficult to read, as important datasets are mixed with intermediate datasets in the workspace.

Solution 2 does not use intermediate files explicitly, which is interesting. However, all functions used are nested, which has a negative impact on the readability of your script file. Actually, when you take a look at solution 2, you immediately see how difficult it is to quickly understand what is going on. Therefore, maintaining such a script can be challenging.

The use of pipes in solution 3 resolves both the nesting as well as the intermediate dataset problems. With the pipe operator `%>%` , the object on the left is used as the first argument in the function on the right. That is `x%>% f(y)` translates to `f(x,y)`. Thus, if `f(x,y)` is needed as first argument in another function `h()`, then instead of writing `h(f(x,y),z)`, you will write `x%>% f(y)%>% h(z)`, which is much more readable, as the operations are listed sequentially.

> *Whenever you use pipes, the first argument you would normally use in any function must be omitted. Instead, that first argument would appear on the left side of the pipe operator. You will find script files based on pipes to be much easier to maintain.*

From now on, I will use the pipe operator frequently to make script files more readable. I strongly invite you to get use to piping, as it is one of the major innovations of `tidyverse`. Not using it will make it more difficult to conduct complex analysis projects with R.