ECON 590 Class 1 Activity

Taylor Mackay

2023-01-21

Overview

The goal of this first activity to familiarize ourselves with `RStudio`. We'll learn how to:

- Install and load packages (collections of `R` commands or functions) needed for data processing
- Load a sample data set and explore it visually
- Perform basic data manipulation tasks using `tidyverse`
- Calculate summary statistics for our data
- Report our results using both the console and RMarkdown `.rmd`files

NOTE: I've split up this file so that the practice problems are in a separate file – see `ECON-590-class-1-activity-solutions` on Canvas or GitHub for the practice problems.

Setting the Stage

`ECON-590-class-1-activity.rmd` is a `markdown` file `-` that means you can include regular, formatted text in addition to code and output in the same file.

- You can run code "in-line" so that results show immediately below your code – just select the line(s) of code you want to run and press `ctrl + Enter`
  - You can also copy and paste the line you want to run into the "Console" window below
  - When you're first experiment with running `R` code and commands, you might find yourself using the Console line a lot – remember that any code you use for analysis should always be saved in an `R` file (either a `.R`-formatted script or an `.Rmd`-formatted markdown file)
- You can also click "Knit" (or press `CTRL + SHIFT + k`) to have an HTML document created with both the written text from the assignment and all your code + output
  - The resulting file should open in your web browser and will be saved as an `.html` file wherever you saved your `markdown` file

Some `R` Basics

In `R`, we use `<-` as an *assignment operator* – it essentially means the same thing as `=` . For instance, entering `x <- 1` creates a new object named `x` that stores the numeric value 1. As a shortcut, you can press `ALT + -` instead of having to type out `<-`.

Take a look at the code below to get a sense of how this process works

```
# In R, any line that starts with a # is a comment -- this is how we'll explain
# what we're doing in a given section of code. Let's start by creating an object
# named "a" that stores the number 5

a <- 5

# Now that we've run the code above, we can just type "a" into the command line,
# and R will return the value 5

a
```

```
## [1] 5
```

```
# We can also use "a" in an equation like the following

a + 5
```

```
## [1] 10
```

```
# You can also create vectors of data in the same way

b.vector <- c(1, 2, 3)

b.vector
```

```
## [1] 1 2 3
```

```
# Take a look at the "Environment" pane on the right. Where are "a" and
# "b.vector" saved?
```

Writing Readable Code

Documenting your code means explaining what you're doing so that someone can read through what you've written and understand what the code does. It is *essential* for producing clear, usable.

- The ***key difference*** between the programming you might have done in class and what you'll have to do professionally is ***readability*** – making sure that other people can understand your code.
- We'll talk about how to write clear, readable code throughout this course, but the first step is to include comments in your code (see the code section above for an example).

While comments are an essential first step when learning to write code, once you've been programming for a while (or you're working on a big picture project), you'll want to take a more systematic approach to documentation. One major reason is that its easy for in-line comments to get "buried" when you have lots of code and not be updated every time a change is made to your code. Other ways of documenting code can include things like README files, examples or sample output of important functions you've defined, etc.

Installing and Loading Packages in `R`

"Basic `R`" comes with lots of commands installed already, but one of the benefits of `R` is the wide array of other programs that we can use. Packages in `R` are just collections of commands or functions that we can use to conduct data analysis tasks that aren't included in Base `R`.

Running the code below will install the data processing package `tidyverse` and load it so that you're able to use the commands included with it.

```r
# Start by installing the program

# install.packages("tidyverse")

# Load the program so we're able to use the commands included with the package

library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.1.1
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.4     v purrr   0.3.4
## v tibble  3.1.2     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Load a Sample Data Set

`R` is very flexible in how you can access and load data sets. As we go through the course, we'll cover a range of ways of loading data in `R`. In the example below, we'll load a `.csv`-formatted file that I've saved to GitHub.

- The function `read.csv()` tells `R` we want to load a `.csv` file
- The function `url()` tells `R` to go to the URL entered in quotes and look for information to retrieve - this is a handy way of accessing data that's not stored on your local computer.

```r
# Load .csv file stored via course GitHub

sample.data <- read.csv(url("https://raw.githubusercontent.com/mackaytc/econ-590-resources/main/data/AC
```

Sample Data Description

Our sample data set `sample.data` contains data from the American Community Survey (ACS). Each of the 4,000 rows in the data set corresponds to a particular person who responded to the survey. We'll refer to each row as an ***observation***. Each observation in this data set is then a person. Each column of the data set corresponds to a particular ***variable*** which stores information about each person in the survey.

This sort of data is known as ***"repeated cross-sectional"*** data – for each year in our sample, we have survey information from a *cross-section* or sample of the full population of the United States. Because the survey is repeated over multiple years (sampling new respondents each year), we have a *repeated* cross-sectional data set.

Key variables include:

- `year`: Tells you the year in which this person or observation took the ACS
- `state`: Tells you the state in which that person lived when they took the ACS
- `incwage`: Tells you the annual wage and salary information for that person

Working with Data using `dplyr`

Now we want to demonstrate several data cleaning tasks using `dplyr` functions. We'll start by removing variables from our data set, then demonstrate how to create a ***subset*** of the data (a limited set of observations from our broader data set, selected based on some critieria).

```
# We want to make the data set a bit smaller and more manageable for today.
# Let's start by using the select command to keep only "year", "statefip", and
# "incwage" variables

sample.data <- select(sample.data, year, state, incwage)

# The head() command will show us a snapshot of what the data set looks like

head(sample.data)
```

```
##   year        state incwage
## 1 2004     new york   10000
## 2 2004      florida   35000
## 3 2004      florida       0
## 4 2004 north dakota     300
## 5 2004      alabama     500
## 6 2004      indiana       0
```

```
# Suppose we wanted to subset the data so that we only kept observations that
# lived in New York. To do this, we can use the filter() function to create a
# new object "sample.data.NY" that stores our subsetted data.

sample.data.NY <- filter(sample.data, state == "new york")

# Take a look at the new data set stored in the "Environment" window. Are all
# the observations from New York?

head(sample.data.NY)
```

```
##   year    state incwage
## 1 2004 new york   10000
## 2 2004 new york  999999
## 3 2004 new york   10000
## 4 2004 new york       0
## 5 2004 new york  999999
## 6 2004 new york    2500
```

```r
# We can also use filter() to select observations by numeric values. This time,
# we won't create a new object - we'll just output the result using head(). We
# can use the n = 10 option to show 10 rows of the subsetted data.

head(filter(sample.data, incwage > 20000), n = 10)
```

```
##    year          state incwage
## 1  2004        florida   35000
## 2  2004           iowa   30000
## 3  2004       delaware   45000
## 4  2004         oregon   40500
## 5  2004        arizona   69000
## 6  2004       michigan   63000
## 7  2004   pennsylvania  999999
## 8  2004        arizona   38500
## 9  2004           ohio  999999
## 10 2004 south carolina  166000
```