Data Wrangling Part 1

Cecilia Y. Sui and all other TAs

Contents

1 Tips on Writing Code

By default, RStudio will save your current workspace when you quit. While convenient, this can mean that you make one-off changes to your data and forget to save that command in your script. Starting with a fresh session every time you open RStudio means you'll learn to keep every step of your analysis in your script, and you'll know that you can get back to where you were by rerunning the script.

1.1 Coding Style

There are two very good reasons to try to write your code in a clear, understandable way:

- Other people might need to use your code to replicate your studies.
- You might need to use your code, a few weeks/months/years after you've written it.

It's possible to write R code that "works" perfectly, and produces all the results and output you want, but proves very difficult to make changes to when you have to come back to it (because a reviewer asked for one additional analysis, etc. which happens very often.)

1.1.1 Formatting Tips

You can improve the readability of your code a lot by following a few simple rules:

- Put spaces between and around variable names and operators (`=+-\*/`)
- Break up long lines of code
- Use meaningful variable names composed of 2 or 3 words" (avoid abbreviations unless they're very common and you use them very consistently)
- Indentation (2 or 4 spaces)

These rules can mean the difference between this:

```
lm1=lm(y~grp+grpTime,mydf,subset=sext1== "m")
```

and this:

```
male_difference = lm(DepressionScore ~ Group + GroupTimeInteraction,
                     data = interview_data,
                     subset = BaselineSex == "Male")
```

R will treat both pieces of code exactly the same, but for any humans reading, the nicer layout and meaningful names make it much easier to understand what's happening, and spot any errors in syntax or intent.

1.1.2 Keep a Consistent Style

Try to follow a consistent style for naming things, e.g. using ** snake_case **for all your variable names in your R code, and** TitleCase ** for the columns in your data. Either style is probably better than lowercase with no spacing allmashedtogether. DO NOT use `.` in names (`do.not.do.this`). It doesn't particularly matter what that style is, as long as you're consistent.

1.1.3 Write LOTS of Comments

As any other code, one of the best things you can do to make R code readable and understandable is write comments. R ignores lines that start with # so you can write whatever you want and it won't affect the way your code runs.

Comments that explain why something was done are great:

```
# reverse the score for question 3
data$DepressionQ3 = 4 - data$DepressionQ3
```

Comments that explain what is being done are also helpful. These comments might appear redundant for people who already understand R code, but can be super helpful for you to learn what is going on and for grading purposes as well.

```
# Calculate the mean of the anxiety scores
anxiety_mean = mean(data$AnxietyTotal)
```

You can also put comments that remind you to fix certain errors.

```
###### FIX HERE: This fails to converge #######
```

1.1.4 Google's R Style Guide

In QPM I, we would like you to follow the R Programming Style Guide by Google to make your R code easier to read, share, and verify. For more details, please check out this link: https://google.github.io/styleguide/Rguide.html

1.2 Organizing Your Files

Working directories are extremely useful for keeping your work organized. When you have hundreds of R files along with images and data, it is much easier to find things using directories instead of putting all the files in one folder. A directory is basically a folder that you have created to save all your work.

1.2.1 Get the working directory

"Each time you open R, it links itself to a directory on your computer, which R calls the working directory. This is where R will look for files when you attempt to load them, and it is where R will save files when you save them. The location of your working directory will vary on different computers. You can find out which directory by running the `getwd()` function, which stands for"get the working directory".

By default your working directory is set to be *the location of your current file*. For example, if you have created an R script as `~/Documents/Rcamp2026/test.R`, the working directory for the R script will be `~/Documents/Rcamp2026/`. Same for R markdown files.

```r
getwd()
```

```
## [1] "/Users/chanhyuk/Dropbox/courses/RCamp2026/Day2/Lecture"
```

1.2.2 Change the working directory

You can place data files straight into the folder that is your working directory, or you can move your working directory to where your data files are. You can move your working directory to any folder on your computer with the function `setwd()`. Just give `setwd()` the file path to your new working directory. This may be from the root directory (starting with / on a Mac), it may include a double-dot (..) to move locally up a folder from the current directory, and it may include a path from the current directory. You can set the working directory to a folder dedicated to whichever project you are currently working on. That way you can keep all of my data, scripts, graphs, and reports in the same place. For example:

```r
setwd("/Users/river/Documents/GitHub/RCamp2023/Day2/Lecture")
```

If the file path does not begin with your root directory, R will assume that it begins at your current working directory.

CAUTION When you are working on Rmarkdown, `setwd()` only **temporarily** changes working directory for *a code chunk*. If you want to set the working directory for the *whole R markdown document*, include or change `knitr::opts_knit$set(root.dir = '{PATH}')`. However, the best practice is just to create R script / R markdown file on the working directory and do not change it.

1.2.3 1.2.3 List files in the working directory

You can see what files are in your working directory with `list.files()`. If you see the file that you would like to open in your working directory, then you are ready to proceed. How you open files in your working directory will depend on which type of file you would like to open.

```r
list.files()
```

```
## [1] "booleans.png"     "data"             "dataset.csv"      "Day2_Lec2022.pdf"
## [5] "Day2_Lec2022.Rmd" "logicals.png"     "school_loc.csv"   "stuff.RData"
## [9] "stuff.RDS"
```

tips If you ever worked with UNIX terminal, this reminds you of `ls` command. R do have `ls()` command too, but that command will print out the list of variables.

1.2.4 Create a new directory

To create a directory in R, use the `dir.create(path)` method. The `dir.create()` method accepts a folder generated in the current working directory or specifies a path. Another way is to go to Session > Set Working

Directory.

```
# This creates a new directory inside the current directory at path = /Users/river/Documents/GitHub/RCa
dir.create("./pset2")
```

1.3 In-class exercises:

1. Get your current working directory.

2 Loading and Saving Data

In Day 1's lecture, we covered how to build a dataframe by hand-typing all the values in each row and column. In this section, we will explore the built-in datasets provided by base R. More importantly, we will cover how to load data from sources like csv files or excel sheets, and how to save the new dataframe that you have made modifications to.

2.1 Built-in Datasets

R comes with many data sets preloaded in the datasets package, which comes with base R. These data sets are not very interesting, but they give you a chance to test code or make a point without having to load a data set from outside R. You can see a list of R's data sets as well as a short description of each by running:

```
help(package = "datasets")
```

To use a data set, just type its name. Each data set is already presaved as an R object. For example:

```
iris
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 2            4.9         3.0          1.4         0.2     setosa
## 3            4.7         3.2          1.3         0.2     setosa
## 4            4.6         3.1          1.5         0.2     setosa
## 5            5.0         3.6          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 7            4.6         3.4          1.4         0.3     setosa
## 8            5.0         3.4          1.5         0.2     setosa
## 9            4.4         2.9          1.4         0.2     setosa
...
```

```
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
```

```
...
```

You can use `str()` function to view the structure of the data and `summary()` function to summarize the dataset. The function tells you the minimum, maximum and quartiles for each numeric column. If there are NAs in your dataframe, it also tells you how many NAs there are in each column.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
##        Species  
##  setosa    :50  
##  versicolor:50  
##  virginica :50  
##                 
##                 
## 
```

However, when doing your own research, you will need to load your own data into R from a wide variety of file formats. Before you can load any data files into R, you will need to tell R where your working directory is.

2.2 Loading Data from Sources

2.2.1 CSV Files

R provides several read functions for you to load data from files, which are shortcuts for read.table with different default arguments. The most commonly used one is `read.csv()`. You will probably use this function hundreds of times during our sequence of methods courses here at WashU. As suggested by its name, the function read.csv allows you to load in comma-separated-variable (CSV) files. For example:

```
school_loc <- read.csv("school_loc.csv")
# remove(school_loc)
```

2.2.2 HTML Links

Many data files are made available on the Internet at their own web address. If you are connected to the Internet, you can open these files straight into R with `read.csv()` (or other functions based on `read.table()`). You can pass a web address into the file name argument for any of R's data-reading functions. For example:

```
school_loc <- read.csv("https://data-nces.opendata.arcgis.com/datasets/a15e8731a17a46aabc452ea607f172c0
# View(school_loc)
# https://www.data.gov/
```

Just make sure that the web address links directly to the file and not to a web page that links to the file.

Usually, when you visit a data file's web address, the file will begin to download or the raw data will appear in your browser window.

2.2.3 Native R Data Formats

R provides two file formats of its own for storing data, `.RDS` and `.RData`. These are known to take up smaller spaces than csvs and faster to read and write. RDS files can store *a single R object*, and RData files can store *multiple R objects*.

You can open a RDS file with readRDS. For example, if the royal flush data was saved as poker.RDS, you could open it with:

```r
school_loc <- readRDS("stuff.RDS")
```

Opening RData files is even easier. Simply run the function load with the file:

```r
# be careful not to assign overlapping data name
# a <- 'text'
load("stuff.RData")
```

There's no need to assign the output to an object. The R objects in your RData file will be loaded into your R session *with their original names*. RData files can contain multiple R objects, so loading one may read in multiple objects. `load()` doesn't tell you how many objects it is reading in, nor what their names are, so it pays to know a little about the RData file before you load it.

If worse comes to worst, you can keep an eye on the environment pane in RStudio as you load an RData file. It displays all of the objects that you have created or loaded during your R session. Another useful trick is to put parentheses around your load command like so, `(load("file.RData"))`. This will cause R to print out the names of each object it loads from the file.

Both `readRDS()` and `load()` take a file path as their first argument, just like R's other read and write functions. If your file is in your working directory, the file path will be the file name.

2.3 Saving Data to Files

2.3.1 Native R Data Formats

You can save an R object like a data frame as either an RData file or an RDS file. RData files can store multiple R objects at once, but RDS files are the better choice because they foster reproducible code.

To save data as an RData object, use `save()` function. To save data as a RDS object, use `saveRDS()` function. In each case, the first argument should be the name of the R object you wish to save. You should then include a file argument that has the file name or file path you want to save the data set to. Remember to put file extensions!

For example, if you have three R objects, a, b, and c, you could save them all in the same RData file and then reload them in another R session:

```r
a <- 1
b <- 2
c <- 3
save(a, b, c, file = "stuff.RData")
load("stuff.RData")
```

However, if you forget the names of your objects or give your file to someone else to use, it will be difficult to determine what was in the file—even after you (or they) load it. The user interface for RDS files is much more clear. You can save only one object per file, and whoever loads it can decide what they want to call their new data. As a bonus, you don't have to worry about load overwriting any R objects that happened to have the same name as the objects you are loading:

```r
saveRDS(a, file = "stuff.RDS") # saving a to RDS file
d <- readRDS("stuff.RDS")
# readRDS("stuff.RDS")
```

Saving your data as an R file offers some advantages over saving your data as a plain-text file. R automatically compresses the file and will also save any R-related metadata associated with your object. This can be handy if your data contains factors, dates and times, or class attributes. You won't have to reparse this information into R the way you would if you converted everything to a text file.

On the other hand, R files cannot be read by many other programs, which makes them inefficient for sharing. They may also create a problem for long-term storage if you don't think you'll have a copy of R when you reopen the files.

2.3.2 CSV Files

You can also use `write.csv()` to save your data as a .csv file. The first argument is the R object that contains your data set. The file argument is the file name (including extension) that you wish to give the saved data. By default, the function will save your data into your working directory. However, you can supply a file path to the file argument. R will oblige by saving the file at the end of the file path. If the file path does not begin with your root directory, R will append it to the end of the file path that leads to your working directory.

For example, you can save the school location data frame to a subdirectory named data within your working directory with the command:

```r
# write.csv(r_object, file = filepath, row.names = FALSE)
write.csv(school_loc, "./data/school_loc.csv", row.names = FALSE)
```

Keep in mind that `write.csv()` cannot create new directories on your computer. Each folder in the file path must exist before you try to save a file with it.

The `row.names()` argument prevents R from saving the data frame's row names as a column in the plain-text file. You might have noticed that R automatically names each row in a data frame with a number.

2.4 In-class exercises:

1. Load the world total fertility rate dataset from dataset.csv. Name the dataframe TFR.
2. Create a new directory named data.
3. Save the dataframe as "newdata.csv" into the new directory that you just created.

3 Data Wrangling Part 1

In Day 1's lecture, we covered how to create a vector, a list, and most importantly a data frame. Now we need to know how to manipulate the individual values inside your data frame. You can select values within an R object with R's notation system.

3.1 Selecting Values

R has a notation system that lets you extract values from R objects. To extract a value or set of values from a data frame, write the data frame's name followed by a pair of hard brackets:

Between the brackets will go two indices separated by a comma. The indices tell R which values to return. R will use the first index to subset the rows of the data frame and the second index to subset the columns.

```r
# View(school_loc)
school_loc[ , ]

# str(school_loc)
# summary(school_loc)

# First few rows
head(school_loc)

# Last few rows
tail(school_loc)

# typeof(school_loc)
# class(school_loc)
```

You have a choice when it comes to writing indices. There are six different ways to write an index for R, and each does something slightly different. They are all very simple and quite handy, so let's take a look at each of them. You can create indices with: * Positive integers * Negative integers * Zero * Blank spaces * Logical values * Names

The simplest and most commonly used of these is positive integers.

3.1.1 Positive integers

R treats positive integers just like the `ij` notation used in linear algebra: `school_loc[i,j]` will return the value of dataframe that is in the ith row and the jth column. Notice that i and j only need to be integers in the mathematical sense.

```r
head(school_loc)
```

```
##           X        Y OBJECTID UNITID                              NAME
## 1 -86.56850 34.78337        1 100654            Alabama A & M University
## 2 -86.79935 33.50570        2 100663 University of Alabama at Birmingham
## 3 -86.17401 32.36261        3 100690                  Amridge University
## 4 -86.64045 34.72456        4 100706 University of Alabama in Huntsville
## 5 -86.29568 32.36432        5 100724            Alabama State University
## 6 -87.52959 33.20702        6 100733 University of Alabama System Office
##                             STREET       CITY STATE        ZIP STFIP  CNTY
## 1           4900 Meridian Street     Normal    AL      35762    01 01089
## 2 Administration Bldg Suite 1070 Birmingham    AL 35294-0110    01 01073
## 3                 1200 Taylor Rd Montgomery    AL 36117-3553    01 01101
## 4                301 Sparkman Dr Huntsville    AL      35899    01 01089
## 5           915 S Jackson Street Montgomery    AL 36104-0271    01 01101
## 6      500 University Blvd. East Tuscaloosa    AL      35401    01 01125
##             NMCNTY LOCALE      LAT       LON  CBSA                 NMCBSA
## 1    Madison County     12 34.78337 -86.56850 26620         Huntsville, AL
## 2  Jefferson County     12 33.50570 -86.79935 13820 Birmingham-Hoover, AL
## 3 Montgomery County     12 32.36261 -86.17401 33860         Montgomery, AL
## 4    Madison County     12 34.72456 -86.64045 26620         Huntsville, AL
## 5 Montgomery County     12 32.36432 -86.29568 33860         Montgomery, AL
## 6 Tuscaloosa County     12 33.20701 -87.52959 46220         Tuscaloosa, AL
##   CBSATYPE CSA                              NMCSA NECTA NMNECTA   CD  SLDL
## 1        1 290              Huntsville-Decatur, AL     N       N 0105 01019
## 2        1 142     Birmingham-Hoover-Talladega, AL     N       N 0107 01055
## 3        1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
## 4        1 290              Huntsville-Decatur, AL     N       N 0105 01006
## 5        1 388 Montgomery-Selma-Alexander City, AL     N       N 0107 01077
## 6        1   N                                    N     N       N 0107 01063
##    SLDU SCHOOLYEAR
## 1 01007  2020-2021
## 2 01018  2020-2021
## 3 01025  2020-2021
## 4 01002  2020-2021
## 5 01026  2020-2021
## 6 01021  2020-2021
```

```
# Extract value at the 1st row and the 2nd column
school_loc[1,2]
```

```
## [1] 34.78337
```

Notice that the row and column indices for R starts at 1, instead of 0 in some other programming languages. This is a more intuitive way of indexing things, but can be confusing if you go back and forth.

To extract more than one value, use *a vector* of positive integers. For example, you can return the first three elements in the first row with:

```
school_loc[1, c(1,2,3)]
```

```
##          X        Y OBJECTID
## 1 -86.5685 34.78337        1
```

R will return the values of `school_loc` that are in both the first row and the first, second, and third columns. Note that R won't actually remove these values from the dataframe. R will give you a new set of values which are copies of the original values. You can then save this new set to an R object with R's assignment operator:

```
newdf <- school_loc[1, c(1,2,3)]
# newdf = school_loc[1, c(1,2,3)] # You can do this, but avoid.
head(newdf)
```

```
##          X        Y OBJECTID
## 1 -86.5685 34.78337        1
```

If you repeat a number in your index, R will return the corresponding value(s) more than once in your "subset." This code will return the first row of `school_loc` twice:

```
school_loc[c(1,1), c(1,2,2,3)]
```

```
##            X        Y      Y.1 OBJECTID
## 1   -86.5685 34.78337 34.78337        1
## 1.1 -86.5685 34.78337 34.78337        1
```

R's notation system is not limited to data frames. "You can use the same syntax to select values in any R object," as long as you supply one index for each dimension of the object. So, for example, you can subset a vector (which has one dimension) with a single index:

```
vec <- c(2,4,6,8,10)
print(vec[1:3])
```

```
## [1] 2 4 6
```

If you select two or more columns from a data frame, R will return a new data frame:

```
school_loc[1:2,1:2]
```

```
##           X        Y
## 1 -86.56850 34.78337
## 2 -86.79935 33.50570
```

However, if you select a single column, R will return a vector:

```
school_loc[1:2, 1]
```

```
## [1] -86.56850 -86.79935
```

If you would prefer a data frame instead, you can add the optional argument `drop = FALSE` between the brackets. This method also works for selecting a single column from a matrix or an array.

```
school_loc[1:2, 1, drop = FALSE]
```

```
##           X
## 1 -86.56850
## 2 -86.79935
```

```
# school_loc[1:2, 1, drop = T]
```

3.1.2 Negative integers

Negative integers do the exact opposite of positive integers when indexing. R will return every element *except the elements in a negative index*. For example, `school_loc[-1, 1:3]` will return everything but the first row. `school_loc[-(2:52), 1:3]` will return the first row (and exclude everything else):

```
school_loc[-1, 1:3]
school_loc[-(2:52), 1:3]
school_loc[c(-1,-3), 1, drop=F]
```

Negative integers are a more efficient way to subset than positive integers if you want to include the majority of a data frame's rows or columns.

R will return an error if you try to pair a negative integer with a positive integer in the same index:

```
school_loc[c(-1, 1), 1]
# Error in xj[i] : only 0's may be mixed with negative subscripts
```

However, you can use both negative and positive integers to subset an object if you use them in different indexes (e.g., if you use one in the rows index and one in the columns index, like `school_loc[-1, 1]`).

```
school_loc[-1, 1, drop=F]
```

3.1.3 Zero

What would happen if you used zero as an index? Zero is neither a positive integer nor a negative integer, but R will still use it to do a type of subsetting. R will return nothing from a dimension when you use zero as an index. This "creates an empty object":

```
# data frame with 0 columns and 0 rows
x <- school_loc[0, 0, drop = F]
dim(x) # 0 0
```

```
## [1] 0 0
```

```
# creating same class, attributes, structure of objects
# class(school_loc)
# class(x)
# attributes(school_loc)
# attributes(x)
```

To be honest, indexing with zero is not very helpful and you will rarely need it.

3.1.4 Blank spaces

You can use a blank space to tell R to extract every value in a dimension. This lets you subset an object on one dimension but not the others, which is useful for extracting entire rows or columns from a data frame:

```
# This extracts the first row
school_loc[1, ]
# This extracts the first column
school_loc[ ,1, drop=F]
```

3.1.5 Logical values

If you supply a vector of `TRUE`s and `FALSE`s as your index, R will match each `TRUE` and `FALSE` to a row in your data frame (or a column depending on where you place the index). R will then return each row that corresponds to a `TRUE`.

It may help to imagine R reading through the data frame and asking, "Should I return the _i_th row of the data structure?" and then consulting the _i_th value of the index for its answer. For this system to work, your vector must be as long as the dimension you are trying to subset:

```
dim(school_loc) # [1] 7012   26
```

```
## [1] 7012   26
```

```
# You can use the multiplication symbol here as a shortcut to the FALSEs.

num_cols <- ncol(school_loc) # 26
head(school_loc)
```

```
##           X        Y OBJECTID UNITID                                 NAME
## 1 -86.56850 34.78337        1 100654            Alabama A & M University
## 2 -86.79935 33.50570        2 100663 University of Alabama at Birmingham
## 3 -86.17401 32.36261        3 100690                  Amridge University
## 4 -86.64045 34.72456        4 100706 University of Alabama in Huntsville
## 5 -86.29568 32.36432        5 100724            Alabama State University
## 6 -87.52959 33.20702        6 100733 University of Alabama System Office
##                              STREET       CITY STATE        ZIP STFIP  CNTY
## 1             4900 Meridian Street     Normal    AL      35762    01 01089
## 2 Administration Bldg Suite 1070 Birmingham    AL 35294-0110    01 01073
## 3                  1200 Taylor Rd Montgomery    AL 36117-3553    01 01101
## 4                 301 Sparkman Dr Huntsville    AL      35899    01 01089
## 5             915 S Jackson Street Montgomery    AL 36104-0271    01 01101
## 6       500 University Blvd. East Tuscaloosa    AL      35401    01 01125
##              NMCNTY LOCALE      LAT       LON  CBSA                  NMCBSA
## 1    Madison County     12 34.78337 -86.56850 26620          Huntsville, AL
## 2  Jefferson County     12 33.50570 -86.79935 13820 Birmingham-Hoover, AL
## 3 Montgomery County     12 32.36261 -86.17401 33860          Montgomery, AL
## 4    Madison County     12 34.72456 -86.64045 26620          Huntsville, AL
## 5 Montgomery County     12 32.36432 -86.29568 33860          Montgomery, AL
## 6 Tuscaloosa County     12 33.20701 -87.52959 46220          Tuscaloosa, AL
##   CBSATYPE CSA                              NMCSA NECTA NMNECTA   CD  SLDL
## 1        1 290              Huntsville-Decatur, AL     N       N 0105 01019
## 2        1 142     Birmingham-Hoover-Talladega, AL     N       N 0107 01055
## 3        1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
## 4        1 290              Huntsville-Decatur, AL     N       N 0105 01006
## 5        1 388 Montgomery-Selma-Alexander City, AL     N       N 0107 01077
## 6        1   N                                   N     N       N 0107 01063
##    SLDU SCHOOLYEAR
## 1 01007  2020-2021
## 2 01018  2020-2021
## 3 01025  2020-2021
## 4 01002  2020-2021
## 5 01026  2020-2021
## 6 01021  2020-2021
```

```
school_loc[1, c(TRUE, FALSE, TRUE, rep(FALSE, 23))]
```

```
##         X OBJECTID
## 1 -86.5685        1
```

```
# View(school_loc)
```

This system may seem odd—who wants to type so many `TRUE`s and `FALSE`s? But it will become very powerful in the next section on modifying values.

3.1.6 Names

Finally, you can ask for the elements you want by name—if your object has names. This is a common way to extract the columns of a data frame, since columns almost always have names:

```
# If you do not remember the exact names of each column,
# you can first use the function colnames() to list all the column names.
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

```
school_loc[1:10, c("X", "Y", "NAME", "SCHOOLYEAR")]
```

```
##            X        Y                                NAME SCHOOLYEAR
## 1  -86.56850 34.78337            Alabama A & M University  2020-2021
## 2  -86.79935 33.50570 University of Alabama at Birmingham  2020-2021
## 3  -86.17401 32.36261                  Amridge University  2020-2021
## 4  -86.64045 34.72456 University of Alabama in Huntsville  2020-2021
## 5  -86.29568 32.36432            Alabama State University  2020-2021
## 6  -87.52959 33.20702 University of Alabama System Office  2020-2021
## 7  -87.54598 33.21188           The University of Alabama  2020-2021
## 8  -85.94527 32.92478   Central Alabama Community College  2020-2021
## 9  -86.96470 34.80679             Athens State University  2020-2021
## 10 -86.17754 32.36736     Auburn University at Montgomery  2020-2021
```

3.2 In-class exercises:

Now that you know the basics of R's notation system, let's put it to use.

1. Extract all values from the first ten rows from school_loc
2. Extract the 100th row from school_loc
3. Extract the 200th to 300th rows from school_loc, but only from the following columns: NAME, STREET, CITY, STATE, ZIP.

```r
school_loc[1:10,]
```

```
##            X        Y OBJECTID UNITID                              NAME
## 1  -86.56850 34.78337        1 100654            Alabama A & M University
## 2  -86.79935 33.50570        2 100663 University of Alabama at Birmingham
## 3  -86.17401 32.36261        3 100690                  Amridge University
## 4  -86.64045 34.72456        4 100706 University of Alabama in Huntsville
## 5  -86.29568 32.36432        5 100724            Alabama State University
## 6  -87.52959 33.20702        6 100733 University of Alabama System Office
## 7  -87.54598 33.21188        7 100751           The University of Alabama
## 8  -85.94527 32.92478        8 100760   Central Alabama Community College
## 9  -86.96470 34.80679        9 100812             Athens State University
## 10 -86.17754 32.36736       10 100830     Auburn University at Montgomery
##                          STREET           CITY STATE        ZIP STFIP  CNTY
## 1            4900 Meridian Street         Normal    AL      35762    01 01089
## 2  Administration Bldg Suite 1070     Birmingham    AL 35294-0110    01 01073
## 3                  1200 Taylor Rd     Montgomery    AL 36117-3553    01 01101
## 4                 301 Sparkman Dr     Huntsville    AL      35899    01 01089
## 5            915 S Jackson Street     Montgomery    AL 36104-0271    01 01101
## 6       500 University Blvd. East     Tuscaloosa    AL      35401    01 01125
## 7             739 University Blvd     Tuscaloosa    AL 35487-0100    01 01125
## 8               1675 Cherokee Rd Alexander City    AL      35010    01 01123
## 9                 300 N Beaty St         Athens    AL      35611    01 01083
## 10              7440 East Drive     Montgomery    AL 36117-3596    01 01101
##               NMCNTY LOCALE      LAT       LON  CBSA              NMCBSA
## 1     Madison County     12 34.78337 -86.56850 26620        Huntsville, AL
## 2   Jefferson County     12 33.50570 -86.79935 13820 Birmingham-Hoover, AL
## 3  Montgomery County     12 32.36261 -86.17401 33860        Montgomery, AL
## 4     Madison County     12 34.72456 -86.64045 26620        Huntsville, AL
## 5  Montgomery County     12 32.36432 -86.29568 33860        Montgomery, AL
## 6  Tuscaloosa County     12 33.20701 -87.52959 46220        Tuscaloosa, AL
## 7  Tuscaloosa County     12 33.21187 -87.54598 46220        Tuscaloosa, AL
## 8  Tallapoosa County     32 32.92478 -85.94527 10760    Alexander City, AL
## 9   Limestone County     31 34.80679 -86.96470 26620        Huntsville, AL
## 10 Montgomery County     12 32.36736 -86.17754 33860        Montgomery, AL
##    CBSATYPE CSA                             NMCSA NECTA NMNECTA   CD  SLDL
## 1         1 290              Huntsville-Decatur, AL     N       N 0105 01019
## 2         1 142     Birmingham-Hoover-Talladega, AL     N       N 0107 01055
## 3         1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
## 4         1 290              Huntsville-Decatur, AL     N       N 0105 01006
## 5         1 388 Montgomery-Selma-Alexander City, AL     N       N 0107 01077
## 6         1   N                                   N     N       N 0107 01063
## 7         1   N                                   N     N       N 0107 01063
## 8         2 388 Montgomery-Selma-Alexander City, AL     N       N 0103 01081
## 9         1 290              Huntsville-Decatur, AL     N       N 0105 01005
## 10        1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
##     SLDU SCHOOLYEAR
## 1  01007  2020-2021
## 2  01018  2020-2021
## 3  01025  2020-2021
## 4  01002  2020-2021
## 5  01026  2020-2021
## 6  01021  2020-2021
## 7  01021  2020-2021
## 8  01027  2020-2021
## 9  01001  2020-2021
## 10 01025  2020-2021
```

```
school_loc[c(1:10),]
```

```
##            X        Y OBJECTID UNITID                              NAME
## 1  -86.56850 34.78337        1 100654            Alabama A & M University
## 2  -86.79935 33.50570        2 100663 University of Alabama at Birmingham
## 3  -86.17401 32.36261        3 100690                  Amridge University
## 4  -86.64045 34.72456        4 100706 University of Alabama in Huntsville
## 5  -86.29568 32.36432        5 100724            Alabama State University
## 6  -87.52959 33.20702        6 100733 University of Alabama System Office
## 7  -87.54598 33.21188        7 100751           The University of Alabama
## 8  -85.94527 32.92478        8 100760   Central Alabama Community College
## 9  -86.96470 34.80679        9 100812             Athens State University
## 10 -86.17754 32.36736       10 100830     Auburn University at Montgomery
##                           STREET           CITY STATE        ZIP STFIP  CNTY
## 1            4900 Meridian Street         Normal    AL      35762    01 01089
## 2  Administration Bldg Suite 1070     Birmingham    AL 35294-0110    01 01073
## 3                  1200 Taylor Rd     Montgomery    AL 36117-3553    01 01101
## 4                 301 Sparkman Dr     Huntsville    AL      35899    01 01089
## 5            915 S Jackson Street     Montgomery    AL 36104-0271    01 01101
## 6       500 University Blvd. East     Tuscaloosa    AL      35401    01 01125
## 7             739 University Blvd     Tuscaloosa    AL 35487-0100    01 01125
## 8                1675 Cherokee Rd Alexander City    AL      35010    01 01123
## 9                  300 N Beaty St         Athens    AL      35611    01 01083
## 10               7440 East Drive     Montgomery    AL 36117-3596    01 01101
##               NMCNTY LOCALE      LAT       LON  CBSA              NMCBSA
## 1     Madison County     12 34.78337 -86.56850 26620        Huntsville, AL
## 2   Jefferson County     12 33.50570 -86.79935 13820 Birmingham-Hoover, AL
## 3  Montgomery County     12 32.36261 -86.17401 33860        Montgomery, AL
## 4     Madison County     12 34.72456 -86.64045 26620        Huntsville, AL
## 5  Montgomery County     12 32.36432 -86.29568 33860        Montgomery, AL
## 6  Tuscaloosa County     12 33.20701 -87.52959 46220        Tuscaloosa, AL
## 7  Tuscaloosa County     12 33.21187 -87.54598 46220        Tuscaloosa, AL
## 8  Tallapoosa County     32 32.92478 -85.94527 10760    Alexander City, AL
## 9   Limestone County     31 34.80679 -86.96470 26620        Huntsville, AL
## 10 Montgomery County     12 32.36736 -86.17754 33860        Montgomery, AL
##    CBSATYPE CSA                              NMCSA NECTA NMNECTA   CD  SLDL
## 1         1 290              Huntsville-Decatur, AL     N       N 0105 01019
## 2         1 142     Birmingham-Hoover-Talladega, AL     N       N 0107 01055
## 3         1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
## 4         1 290              Huntsville-Decatur, AL     N       N 0105 01006
## 5         1 388 Montgomery-Selma-Alexander City, AL     N       N 0107 01077
## 6         1   N                                  N     N       N 0107 01063
## 7         1   N                                  N     N       N 0107 01063
## 8         2 388 Montgomery-Selma-Alexander City, AL     N       N 0103 01081
## 9         1 290              Huntsville-Decatur, AL     N       N 0105 01005
## 10        1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
##     SLDU SCHOOLYEAR
## 1  01007  2020-2021
## 2  01018  2020-2021
## 3  01025  2020-2021
## 4  01002  2020-2021
## 5  01026  2020-2021
## 6  01021  2020-2021
## 7  01021  2020-2021
## 8  01027  2020-2021
## 9  01001  2020-2021
## 10 01025  2020-2021
```

3.3 Dollar Signs and Double Brackets

Two types of object in R obey an optional second system of notation. You can extract values from data frames and lists with the $ syntax. You will encounter the $ syntax again and again as an R programmer, so let's examine how it works.

To select a column from a data frame, write the data frame's name and the column name separated by a $. Notice that no quotes should go around the column name:

```
# For illustration purpose, I will just use a subset of the original school location dataset.
sub_school_loc <- school_loc[1:10,]
head(sub_school_loc)
```

```
##           X        Y OBJECTID UNITID                               NAME
## 1 -86.56850 34.78337        1 100654            Alabama A & M University
## 2 -86.79935 33.50570        2 100663 University of Alabama at Birmingham
## 3 -86.17401 32.36261        3 100690                  Amridge University
## 4 -86.64045 34.72456        4 100706 University of Alabama in Huntsville
## 5 -86.29568 32.36432        5 100724            Alabama State University
## 6 -87.52959 33.20702        6 100733 University of Alabama System Office
##                              STREET       CITY STATE        ZIP STFIP  CNTY
## 1           4900 Meridian Street     Normal    AL      35762    01 01089
## 2 Administration Bldg Suite 1070 Birmingham    AL 35294-0110    01 01073
## 3                  1200 Taylor Rd Montgomery    AL 36117-3553    01 01101
## 4                301 Sparkman Dr Huntsville    AL      35899    01 01089
## 5           915 S Jackson Street Montgomery    AL 36104-0271    01 01101
## 6      500 University Blvd. East Tuscaloosa    AL      35401    01 01125
##              NMCNTY LOCALE      LAT        LON  CBSA              NMCBSA
## 1    Madison County     12 34.78337 -86.56850 26620        Huntsville, AL
## 2  Jefferson County     12 33.50570 -86.79935 13820 Birmingham-Hoover, AL
## 3 Montgomery County     12 32.36261 -86.17401 33860        Montgomery, AL
## 4    Madison County     12 34.72456 -86.64045 26620        Huntsville, AL
## 5 Montgomery County     12 32.36432 -86.29568 33860        Montgomery, AL
## 6 Tuscaloosa County     12 33.20701 -87.52959 46220        Tuscaloosa, AL
##   CBSATYPE CSA                              NMCSA NECTA NMNECTA   CD  SLDL
## 1        1 290              Huntsville-Decatur, AL     N       N 0105 01019
## 2        1 142     Birmingham-Hoover-Talladega, AL     N       N 0107 01055
## 3        1 388 Montgomery-Selma-Alexander City, AL     N       N 0102 01074
## 4        1 290              Huntsville-Decatur, AL     N       N 0105 01006
## 5        1 388 Montgomery-Selma-Alexander City, AL     N       N 0107 01077
## 6        1   N                                   N     N       N 0107 01063
##    SLDU SCHOOLYEAR
## 1 01007  2020-2021
## 2 01018  2020-2021
## 3 01025  2020-2021
## 4 01002  2020-2021
## 5 01026  2020-2021
## 6 01021  2020-2021
```

```r
sub_school_loc$NAME
```

```
##  [1] "Alabama A & M University"            "University of Alabama at Birmingham"
##  [3] "Amridge University"                  "University of Alabama in Huntsville"
##  [5] "Alabama State University"            "University of Alabama System Office"
##  [7] "The University of Alabama"           "Central Alabama Community College"
##  [9] "Athens State University"             "Auburn University at Montgomery"
```

R will return all of the values in the column as a vector. This $ notation is incredibly useful because you will often store the variables of your data sets as columns in a data frame. From time to time, you'll want to run a function like mean or median on the values in a variable. In R, these functions expect a vector of values as input:

```r
mean(school_loc$X)
```

```
## [1] -90.35801
```

```r
median(school_loc$Y)
```

```
## [1] 38.54617
```

```r
max(school_loc$Y)
```

```
## [1] 71.3247
```

You can use the same $ notation with the elements of a list, if they have names. This notation has an advantage with lists, too. If you subset a list in the usual way, R will return a new list that has the elements you requested. This is true even if you only request a single element. For example:

```r
l <- list(numbers = c(1, 2), logical = TRUE, strings = c("a", "b", "c"))
l
```

```
## $numbers
## [1] 1 2
##
## $logical
## [1] TRUE
##
## $strings
## [1] "a" "b" "c"
```

```r
# And then subset it:
l[[1]] # access by index
```

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

```r
l$numbers
```

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

```r
l[1]
```

```
## $numbers
## [1] 1 2
```

```r
# typeof(l[1]) # list
# typeof(l$numbers) # double
```

The result is a smaller list with one element. That element is the vector `c(1, 2)`. This can be annoying because many R functions do not work with lists. For example, `sum(l[1])` will return an error. It would be horrible if once you stored a vector in a list, you could only ever get it back as a list:

```
sum(l[1])
# Error in sum(l[1]) : invalid 'type' (list) of argument
```

When you use the $ notation, R will return the selected values as they are, with no list structure around them:

```
l$numbers
```

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

```
l[1]
```

```
## $numbers
## [1] 1 2
```

```
l$strings
```

```
## [1] "a" "b" "c"
```

You can then immediately feed the results to a function:

```
sum(l$numbers)
```

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

If the elements in your list do not have names (or you do not wish to use the names), you can use two brackets, instead of one, to subset the list. This notation will do the same thing as the $ notation:

```
sum(l[[1]])
```

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

In other words, if you subset a list with single-bracket notation, R will return a smaller list. If you subset a list with double-bracket notation, R will return just the values that were inside an element of the list. You can combine this feature with any of R's indexing methods:

```
l
```

```
## $numbers
## [1] 1 2
##
## $logical
## [1] TRUE
##
## $strings
## [1] "a" "b" "c"
```

```
l["numbers"]
```

```
## $numbers
## [1] 1 2
```

```
typeof(l["numbers"])
```

```
## [1] "list"
```

```
l[["numbers"]]
```

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

```
typeof(l[["numbers"]])
```

```
## [1] "double"
```

```
sum(l[["numbers"]])
```

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

This difference is subtle but important. In the R community, there is a popular, and helpful, way to think about it. Imagine that each list is a train and each element is a train car. When you use single brackets, R selects individual train cars and returns them as a new train. Each car keeps its contents, but those contents are still inside a train car (i.e., a list). When you use double brackets, R actually unloads the car and gives you back the contents.

4 Modifying Values

In this section, you will learn how to change the actual values that are stored inside your data frame. This is all adding up to something special: complete control of your data. You can now store your data in your computer, retrieve individual values at will, and use your computer to perform correct calculations with those values.

4.1 Changing Values in Place

You can use R's notation system to modify values within an R object. First, describe the value (or values) you wish to modify. Then use the assignment operator <- to overwrite those values. R will update the selected values in the original object. Let's put this into action with a real example:

```
vec <- c(1:10)
vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
vec <- "text"
vec
```

```
## [1] "text"
```

Here's how you can select the first value of vec:

```
vec[1]
```

```
## [1] "text"
```

And here is how you can modify it:

```
vec
```

```
## [1] "text"
```

```
vec[1] <- 999
vec
```

```
## [1] "999"
```

```
vec[-1] <- 0
vec
```

```
## [1] "999"
```

You can replace multiple values at once as long as the number of new values equals the number of selected values:

```
vec <- 1:10
vec[c(1, 3, 5)] <- c(1000, 3000, 5000)
vec
```

```
##  [1] 1000    2 3000    4 5000    6    7    8    9   10
```

```
vec[4:6] <- vec[4:6] + 1
vec
```

```
##  [1] 1000    2 3000    5 5001    7    7    8    9   10
```

You can also create values that do not yet exist in your object. R will expand the object to accommodate the new values:

```
vec <- c(1:10)
vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
length(vec)
```

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

```
vec[12] <- 0
vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 NA  0
```

Notice the NA in the vector. Our original vector has 10 elements, but here we added a new value at index 12 without specifying the value for the 11th element. R will automatically fill in the unspecified elements with NA. When writing your own code, try to avoid doing this. We usually try to minimize the amount of NAs in our dataframe.

Now you can add new variables to your data set:

```
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR"
```

```
school_loc$indices <- 1:7012
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR" "indices"
```

```
# View(school_loc)
```

You can also remove columns from a data frame (and elements from a list) by assigning them the symbol NULL:

```
school_loc$newcol <- 0
# View(school_loc)
school_loc$newcol <- NULL
colnames(school_loc)
```

```
##  [1] "X"          "Y"          "OBJECTID"   "UNITID"     "NAME"
##  [6] "STREET"     "CITY"       "STATE"      "ZIP"        "STFIP"
## [11] "CNTY"       "NMCNTY"     "LOCALE"     "LAT"        "LON"
## [16] "CBSA"       "NMCBSA"     "CBSATYPE"   "CSA"        "NMCSA"
## [21] "NECTA"      "NMNECTA"    "CD"         "SLDL"       "SLDU"
## [26] "SCHOOLYEAR" "indices"
```

You can single out just the values by subsetting the columns dimension. You can also assign a new set of values to these old values. The set of new values will have to be the same size as the set of values that you are replacing.

```
# To illustrate subsetting values, we add the new variable back.
school_loc$newcol <- 1:7012
head(school_loc$newcol, 10)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
school_loc$newcol[c(1,3,5,7,9)] <- c(111, 333, 555, 777, 999)
head(school_loc$newcol, 10)
```

```
##  [1] 111   2 333   4 555   6 777   8 999  10
```

```
# If you would like to set them to the same value, simply do:
school_loc$newcol[c(2,4,6,8,10)] <- 1000
head(school_loc$newcol, 10)
```

```
##  [1]  111 1000  333 1000  555 1000  777 1000  999 1000
```

Notice that the values change in place. You do not end up with a new copy of the school_loc dataframe. The new values will appear inside the school_loc dataframe.

The same technique will work whether you store your data in a vector, matrix, list, or data frame. Just describe the values that you want to change with R's notation system, then assign over those values with R's assignment operator.

4.2 Logical Subsetting

Do you remember R's logical index system, logicals? To recap, you can select values with a vector of `TRUE`s and `FALSE`s. The vector must be the same length as the dimension that you wish to subset. R will return every element that matches a TRUE:

```
vec2 <- c(1,0,2,0,3,4,5,0,9)
vec2
```

```
## [1] 1 0 2 0 3 4 5 0 9
```

```
vec2[c(TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE, TRUE)] # filter
```

```
## [1] 1 5 9
```

At first glance, this system might seem impractical. Who wants to type out long vectors of `TRUE`s and `FALSE`s? No one. But you don't have to. You can let a logical test create a vector of `TRUE`s and `FALSE`s for you.

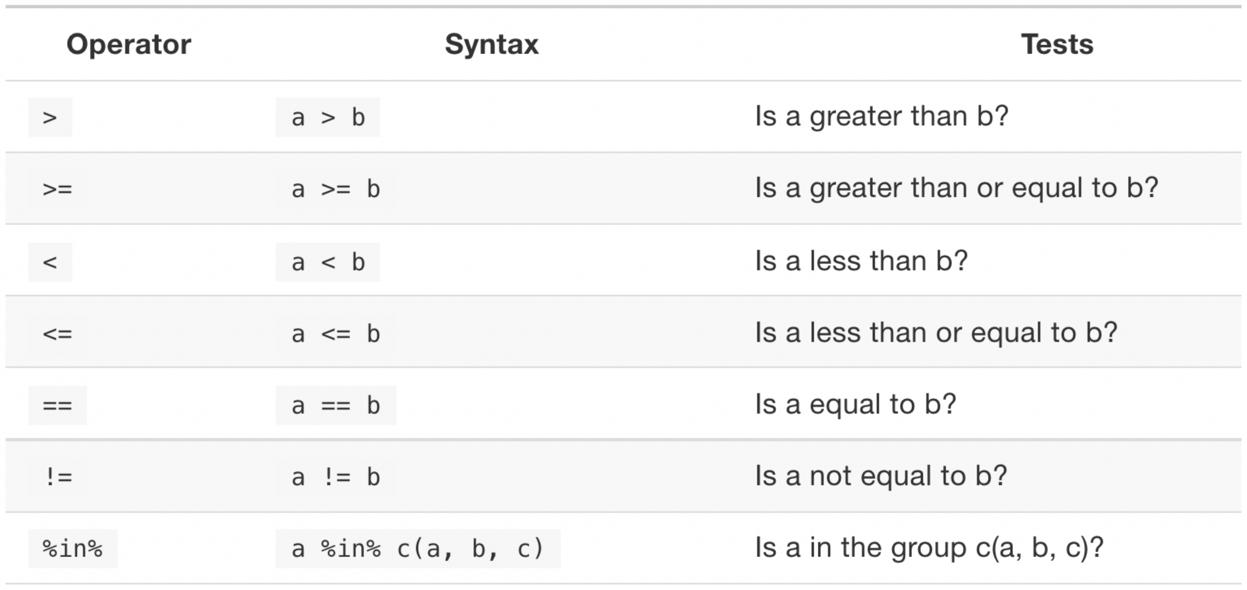
4.2.1 Logical Tests

A logical test is a comparison like "is one less than two?", $1 < 2$, or "is three greater than four?", $3 > 4$. R provides seven logical operators that you can use to make comparisons.

Each operator returns a TRUE or a FALSE. If you use an operator to compare vectors, R will do 'element-wise comparisons' — just like it does with the arithmetic operators:

```
1 < 2
```

```
## [1] TRUE
```



Operator	Syntax	Tests
`>`	`a > b`	Is a greater than b?
`>=`	`a >= b`	Is a greater than or equal to b?
`<`	`a < b`	Is a less than b?
`<=`	`a <= b`	Is a less than or equal to b?
`==`	`a == b`	Is a equal to b?
`!=`	`a != b`	Is a not equal to b?
`%in%`	`a %in% c(a, b, c)`	Is a in the group c(a, b, c)?

Figure 1: R's Logical Operators

```r
1 > c(0, 1, 2)
```

```
## [1]  TRUE FALSE FALSE
```

```r
c(1, 2, 3) == c(3, 2, 1)
```

```
## [1] FALSE  TRUE FALSE
```

`%in%` is the only operator that does 'not' do normal element-wise execution. `%in%` tests whether the value(s) on the left side are in the vector on the right side. If you provide a vector on the left side, %in% will not pair up the values on the left with the values on the right and then do element-wise tests. Instead, `%in%` will independently test whether each value on the left is somewhere in the vector on the right:

```r
1 %in% c(3, 4, 5)
```

```
## [1] FALSE
```

```r
c(1, 2) %in% c(3, 4, 5)
```

```
## [1] FALSE FALSE
```

```r
c(1, 2, 3) %in% c(3, 4, 5)
```

```
## [1] FALSE FALSE  TRUE
```

```r
c(1, 2, 3, 4) %in% c(3, 4, 5)
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```r
# works for strings
"a" %in% c("a", "b", "c")
```

```
## [1] TRUE
```

```r
"A" %in% c("a") # case sensitive
```

```
## [1] FALSE
```

```
"a" %in% c("abc") # %in% checks exact matches
```

```
## [1] FALSE
```

Notice that you test for equality with a double equals sign, `==`, and not a single equals sign, `=`, which is another way to write the assignment operator <-. It is easy to forget and use `a = b` to test if a equals b. Unfortunately, you'll be in for a nasty surprise. R won't return a `TRUE` or `FALSE`, because it won't have to: a will equal b, because you just ran the equivalent of `a <- b` where you manually set the value of a to be b.

You can compare any two R objects with a logical operator; however, logical operators make the most sense if you compare two objects of the same data type. If you compare objects of different data types, R will use its coercion rules to coerce the objects to the same type before it makes the comparison and the result might be hard to interpret.

We again use the school location dataset as our example.

```
colnames(school_loc)
# First extract the column "X"
school_loc$X
# Use the < operator to check whether each value is less than -100
school_loc$X < -100
# Use the sum() function to check how many TRUEs there are
# TRUE = 1 / FALSE = 0
sum(school_loc$X < -100)

# What does this do?
school_loc$Y[school_loc$X < -100]
# uses the logical vector to index or subset the Y column.
# extract all values from the Y column where the corresponding values in the X column are TRUE(less tha

# What does this do?
school_loc$Y[school_loc$X < -100] <- 0
# View(school_loc)
```

To summarize, you can use a logical test to select values within an object.

Logical subsetting is a powerful technique because it lets you quickly identify, extract, and modify individual values in your data set. When you work with logical subsetting, you do not need to know where in your data set a value exists. You only need to know how to describe the value with a logical test.

Logical subsetting is one of the things R does best. In fact, logical subsetting is a key component of vectorized programming, a coding style that lets you write fast and efficient R code.

4.3 In-class exercises:

Let's practice with the dataset TFR.

1. Find the dimension of TFR. How many observations and variables?
2. Add a new column called index to TFR, where the values are the indices for the rows.
3. Select the first row of TFR.
4. Select the first column of TFR.
5. Get the last row of TFR.
6. How many values in the column TFR are greater than 8?
7. How many values in the column ChildBearing are over 25?
8. What does this code do?

```
TFR$ChildBearing[TFR$ChildBearing > 25]
```

9. Set all values in the column ChildBearing that are less than 25 to
   0.

```
TFR <- read.csv("dataset.csv")
sum(TFR$TFR > 8)
```

```
## [1] 67
```

```
nrow(TFR[TFR$TFR > 8,])
```

```
## [1] 67
```

```
typeof(TFR[TFR$TFR > 8,])
```

```
## [1] "list"
```

```
head(TFR[TFR$TFR >8, ])
```

```
##      Country Uncode Year    TFR InfMRateCME InfMRateUN U5MRateCME U5MRateUN
## 5215    Iraq    368 1950 8.3980       246.2   280.6580      364.3  407.6411
## 5216    Iraq    368 1951 8.1784       231.0   269.5758      343.1  391.3986
## 5625  Jordan    400 1964 8.0340        87.1   100.3606      123.8  149.2374
## 5626  Jordan    400 1965 8.0570        82.8    95.5380      117.1  140.3478
## 5627  Jordan    400 1966 8.0530        78.9    91.9096      110.9  133.8606
## 5628  Jordan    400 1967 8.0330        75.2    88.2812      105.1  127.3735
##      LifeExpB MtoFbirth   MtoF04  Pop1564 Pop1564Female    GDPpc GDPpcGrowth
## 5215  30.9550     1.065 1.078620 59.78318      60.41158       NA          NA
## 5216  32.3510     1.065 1.067612 59.09168      59.80392       NA          NA
## 5625  55.7927     1.054 1.079269 51.52381      51.79283 3744.551   0.1159344
## 5626  56.5522     1.054 1.073565 51.33929      51.86567 3991.914   0.0639687
## 5627  57.3033     1.054 1.069025 50.94963      51.03448 3890.935  -0.0256205
## 5628  58.0440     1.054 1.066867 50.90909      51.18483 4012.081   0.0306606
##      Yschooling YschoolF1549 GenrollPrim ChildBearing CountryCode
## 5215      0.240    0.1059529          NA      28.6970         IRQ
## 5216      0.258    0.1174685          NA      28.6970         IRQ
## 5625      2.882    1.8347658          NA      30.0230         JOR
## 5626      3.000    1.9325727          NA      30.0870         JOR
## 5627      3.094    2.0368810          NA      30.1594         JOR
## 5628      3.188    2.1411893          NA      30.2318         JOR
```

```
typeof(TFR$TFR > 8)
```

```
## [1] "logical"
```

```
nrow(TFR[TFR$TFR > 8,])
```

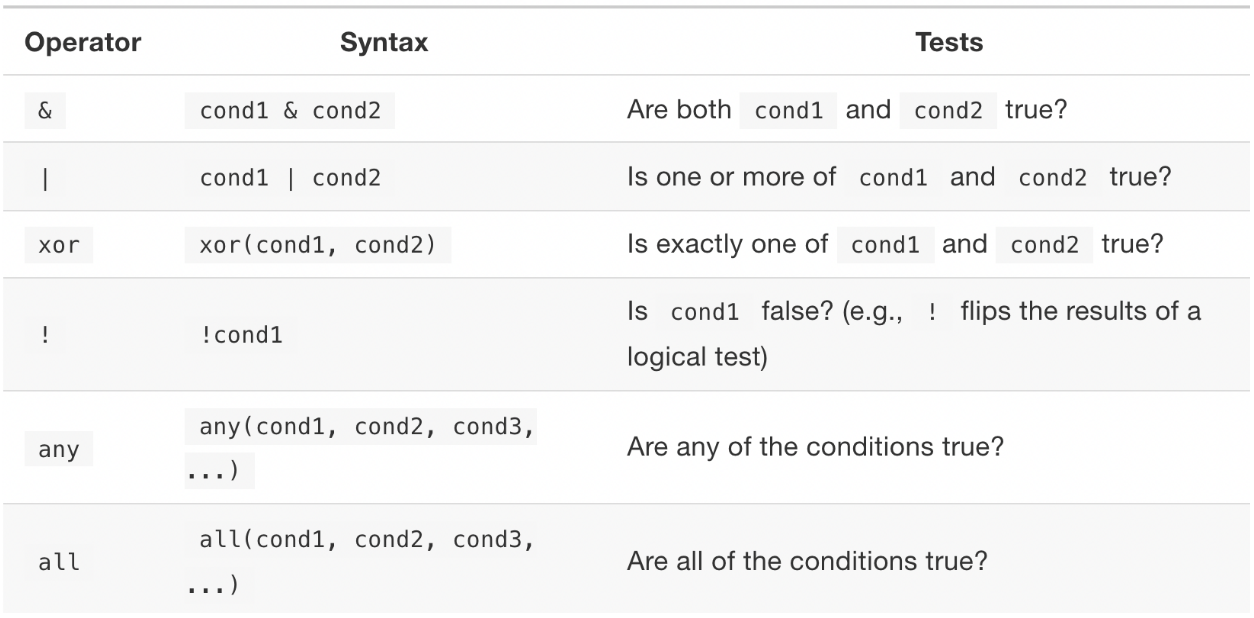
```
## [1] 67
```

4.3.1 Boolean Operators

Boolean operators are things like and (&) and or (|). They "collapse the results of multiple logical tests into a single TRUE or FALSE". R has six boolean operators.

To use a Boolean operator, place it between two complete logical tests. R will execute each logical test and then use the Boolean operator to combine the results into a single TRUE or FALSE. If you do not supply a complete test to each side of the operator, R will return an error.

When used with vectors, Boolean operators will follow the same element-wise execution as arithmetic and logical operators:



Operator	Syntax	Tests
`&`	`cond1 & cond2`	Are both `cond1` and `cond2` true?
`|`	`cond1 | cond2`	Is one or more of `cond1` and `cond2` true?
`xor`	`xor(cond1, cond2)`	Is exactly one of `cond1` and `cond2` true?
`!`	`!cond1`	Is `cond1` false? (e.g., `!` flips the results of a logical test)
`any`	`any(cond1, cond2, cond3, ...)`	Are any of the conditions true?
`all`	`all(cond1, cond2, cond3, ...)`	Are all of the conditions true?

Figure 2: Boolean Operators

```r
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 4)
a == b
```

```
## [1] TRUE TRUE TRUE
```

```r
b == c
```

```
## [1]  TRUE  TRUE FALSE
```

```r
a == b & b == c
```

```
## [1]  TRUE  TRUE FALSE
```

```r
all(a == b & b == c)
```

```
## [1] FALSE
```

```r
any(a == b & b == c)
```

```
## [1] TRUE
```

```r
# (TRUE TRUE TRUE) & (TRUE  TRUE FALSE)
```

4.4 In-class exercises:

If you think you have the hang of logical tests, try converting these sentences into tests written with R code. To help you out, I've defined some R objects after the sentences that you can use to test your answers:

1. Is w positive?
2. Is x greater than 10 and less than 20?
3. Is object y the word February?
4. Is every value in z a day of the week?
5. What does this expression evaluate to and why?

```r
w <- c(-1, 0, 1)
x <- c(5, 15)
y <- "February"
z <- c("Monday", "Tuesday", "Friday")
(TRUE + TRUE) * FALSE
```

```
## [1] 0
```

6. Use logical operators to output only those rows of data in TFR where column Year is between 1950 and 1955 inclusively.
7. Use logical operators to output only those rows of data where column TFR is equal to 7.45 and column Year is greater than 1960.
8. Use logical operators to output only the even rows of the dataframe.
9. Use logical operators and change every 4th element in column LifeExpB to 0.

```r
data <- TFR
# 6
head(data[data$Year >= 1950 & data$Year <= 1955,])
```

```
##       Country Uncode Year  TFR InfMRateCME InfMRateUN U5MRateCME U5MRateUN
## 1 Afghanistan      4 1950 7.45          NA   304.5940         NA  438.0103
## 2 Afghanistan      4 1951 7.45          NA   299.6836         NA  431.6065
## 3 Afghanistan      4 1952 7.45          NA   294.7732         NA  425.2027
## 4 Afghanistan      4 1953 7.45          NA   289.8628         NA  418.7989
## 5 Afghanistan      4 1954 7.45          NA   284.9524         NA  412.3951
## 6 Afghanistan      4 1955 7.45          NA   280.0420         NA  405.9913
##   LifeExpB MtoFbirth    MtoF04  Pop1564 Pop1564Female GDPpc GDPpcGrowth
## 1  26.0690      1.06 0.9523352 56.08875      54.25678    NA          NA
## 2  26.5736      1.06 0.9524428 55.82908      54.06208    NA          NA
## 3  27.0782      1.06 0.9728808 55.74039      54.08136    NA          NA
## 4  27.5828      1.06 0.9952082 55.71038      54.13613    NA          NA
## 5  28.0874      1.06 1.0122050 55.71779      54.22245    NA          NA
## 6  28.5920      1.06 1.0222970 55.68319      54.21412    NA          NA
##   Yschooling YschoolF1549 GenrollPrim ChildBearing CountryCode
## 1      0.270   0.08679564          NA       29.835         AFG
## 2      0.278   0.08758149          NA       29.835         AFG
## 3      0.286   0.08836733          NA       29.835         AFG
## 4      0.294   0.08915317          NA       29.835         AFG
## 5      0.302   0.08993901          NA       29.835         AFG
## 6      0.310   0.09072485          NA       29.835         AFG
```

```r
# 7
head(data[data$TFR == 7.45 & data$Year > 1960,])
```

```
##        Country Uncode Year  TFR InfMRateCME InfMRateUN U5MRateCME U5MRateUN
## 12 Afghanistan      4 1961 7.45       240.5   251.1132      356.5  368.4389
## 13 Afghanistan      4 1962 7.45       236.3   246.7364      350.6  362.9053
## 14 Afghanistan      4 1963 7.45       232.3   242.3596      345.0  357.3718
## 15 Afghanistan      4 1964 7.45       228.5   237.9828      339.7  351.8383
## 16 Afghanistan      4 1965 7.45       224.6   233.6060      334.1  346.3048
## 17 Afghanistan      4 1966 7.45       220.7   229.9888      328.7  341.2930
##    LifeExpB MtoFbirth   MtoF04  Pop1564 Pop1564Female GDPpc GDPpcGrowth
## 12  32.7774      1.06 1.017161 54.74086      53.69702    NA          NA
## 13  33.2199      1.06 1.016234 54.53767      53.62031    NA          NA
## 14  33.6579      1.06 1.021365 54.46916      53.74811    NA           NA
## 15  34.0929      1.06 1.027291 54.43051      53.94208    NA           NA
## 16  34.5254      1.06 1.030376 54.34323      53.93676    NA           NA
## 17  34.9574      1.06 1.033343 54.02503      53.72803    NA           NA
##    Yschooling YschoolF1549 GenrollPrim ChildBearing CountryCode
## 12      0.380    0.1088078          NA       29.835         AFG
## 13      0.390    0.1101457          NA       29.835         AFG
## 14      0.400    0.1114836          NA       29.835         AFG
## 15      0.410    0.1128215          NA       29.835         AFG
## 16      0.420    0.1141593          NA       29.835         AFG
## 17      0.474    0.1291685          NA       29.835         AFG
```

```
# 8
head(data[c(F,T),])
```

```
##        Country Uncode Year  TFR InfMRateCME InfMRateUN U5MRateCME U5MRateUN
## 2  Afghanistan      4 1951 7.45          NA   299.6836         NA  431.6065
## 4  Afghanistan      4 1953 7.45          NA   289.8628         NA  418.7989
## 6  Afghanistan      4 1955 7.45          NA   280.0420         NA  405.9913
## 8  Afghanistan      4 1957 7.45          NA   270.2212         NA  393.1837
## 10 Afghanistan      4 1959 7.45          NA   260.4004         NA  380.3761
## 12 Afghanistan      4 1961 7.45       240.5   251.1132      356.5  368.4389
##    LifeExpB MtoFbirth    MtoF04  Pop1564 Pop1564Female GDPpc GDPpcGrowth
## 2   26.5736      1.06 0.9524428 55.82908      54.06208    NA          NA
## 4   27.5828      1.06 0.9952082 55.71038      54.13613    NA          NA
## 6   28.5920      1.06 1.0222970 55.68319      54.21412    NA          NA
## 8   29.6012      1.06 1.0493490 55.25483      53.91750    NA          NA
## 10  30.6104      1.06 1.0313490 55.02604      53.76849    NA          NA
## 12  32.7774      1.06 1.0171610 54.74086      53.69702    NA          NA
##    Yschooling YschoolF1549 GenrollPrim ChildBearing CountryCode
## 2       0.278   0.08758149          NA       29.835         AFG
## 4       0.294   0.08915317          NA       29.835         AFG
## 6       0.310   0.09072485          NA       29.835         AFG
## 8       0.334   0.09742289          NA       29.835         AFG
## 10      0.358   0.10412092          NA       29.835         AFG
## 12      0.380   0.10880781          NA       29.835         AFG
```

```
# 9
data$LifeExpB[c(F,F,F,T)] <- 0

# %%
```

4.5 Missing Information (NA)

Missing information problems happen frequently in data science. Usually, they are more straightforward: you don't know a value because the measurement was lost, corrupted, or never taken to begin with. R has a way to help you manage these missing values.

The NA character is a special symbol in R. It stands for "not available" and can be used as a placeholder for missing information. R will treat NA exactly as you should want missing information treated. For example, what result would you expect if you add 1 to a piece of missing information?

```
1 + NA
```

```
## [1] NA
```

R will return a second piece of missing information. It would not be correct to say that 1 + NA = 1 because there is a good chance that the missing quantity is not zero. You do not have enough information to determine the result.

What if you tested whether a piece of missing information is equal to 1?

```
NA == 3.14
```

```
## [1] NA
```

Again, your answer would be something like “I do not know if this is equal to one,” that is, NA. Generally, NAs will propagate whenever you use them in an R operation or function. This can save you from making errors based on missing data.

4.5.1 Inside functions: `na.rm` argument

Missing values can help you work around holes in your data sets, but they can also create some frustrating problems. Suppose, for example, that you’ve collected 1,000 observations and wish to take their average with R’s mean function. If even one of the values is NA, your result will be NA:

```
c(NA, 1:50)
```

```
##  [1] NA  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## [26] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
## [51] 50
```

```
mean(c(NA, 1:50))
```

```
## [1] NA
```

Understandably, you may prefer a different behavior. Most R functions come with the optional argument, `na.rm`, which stands for NA remove. R will ignore NAs when it evaluates a function if you add the argument `na.rm = TRUE`:

```
mean(c(NA, 1:50), na.rm = TRUE)
```

```
## [1] 25.5
```

```
mean(c(1:50)) # which returns the same value as above
```

```
## [1] 25.5
```

4.5.2 Test for NA: `is.na()`

On occasion, you may want to identify the NAs in your data set with a logical test, but that too creates a problem. How would you go about it? If something is a missing value, any logical test that uses it will return a missing value, even this test:

```
NA == NA
```

```
## [1] NA
```

Which means that tests like this won’t help you find missing values:

```
c(1, 2, 3, NA) == NA
```

```
## [1] NA NA NA NA
```

But don’t worry too hard; R supplies a special function that can test whether a value is an NA. The function is sensibly named `is.na()`:

```
is.na(NA)
```

```
## [1] TRUE
```

```r
vec3 <- c(1, 2, 3, NA)
is.na(vec3)
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```r
# What does this do?
school_loc$Y[school_loc$X < -100] <- NA
# View(school_loc)
```

4.5.3 Omit the Entire Rows: `na.omit()`

You can also remove all rows containing NA's in any column by using the `na.omit()` function. For example,

```r
dim(school_loc)
summary(school_loc) # 7012 obs.
new_df <- na.omit(school_loc)
dim(new_df)
summary(new_df) # 5470 obs
```

4.6 In-class exercises:

1. What is the length of X?

```r
X <- c (123,0,NA,8,NA,200)
na.omit(X)
```

```
## [1] 123   0   8 200
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "omit"
```

```r
X
```

```
## [1] 123   0  NA   8  NA 200
```

2. Can you find all occurrences of NA in X? How can you find the total number of NAs in X?
3. Can you remove all occurrences of NA in X?
4. Can you replace all occurrences of NA with 88?
5. We will use the TFR dataset again. Create a new dataframe TFR2 where you drop all NA's.
6. Find the dimension of TFR2. How many observations are left?
7. Create a new dataframe TFR3 where you remove all rows with NA values in the GDPpc column.

```r
TFR3 <- TFR[!is.na(TFR$GDPpc), ]
# View(TFR3)
```